# Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOCA Common</td>
<td>5</td>
</tr>
<tr>
<td>DOCA Core</td>
<td>5</td>
</tr>
<tr>
<td>Sync Event</td>
<td>67</td>
</tr>
<tr>
<td>Mmap Advise</td>
<td>97</td>
</tr>
<tr>
<td>DOCA Log</td>
<td>106</td>
</tr>
<tr>
<td>DOCA Flow</td>
<td>111</td>
</tr>
<tr>
<td>DOCA Flow Connection Tracking</td>
<td>240</td>
</tr>
<tr>
<td>DOCA Flow Tune Server</td>
<td>273</td>
</tr>
<tr>
<td>DPA Subsystem</td>
<td>290</td>
</tr>
<tr>
<td>DPA Development</td>
<td>293</td>
</tr>
<tr>
<td>DOCA DPA</td>
<td>365</td>
</tr>
<tr>
<td>DOCA PCC</td>
<td>429</td>
</tr>
<tr>
<td>DOCA DMA</td>
<td>444</td>
</tr>
<tr>
<td>DOCA Comch</td>
<td>460</td>
</tr>
<tr>
<td>DOCA Comch – New</td>
<td>460</td>
</tr>
<tr>
<td>DOCA Comm Channel – Deprecated</td>
<td>498</td>
</tr>
<tr>
<td>DOCA UROM</td>
<td>533</td>
</tr>
<tr>
<td>DOCA RDMA</td>
<td>579</td>
</tr>
<tr>
<td>DOCA Ethernet</td>
<td>639</td>
</tr>
<tr>
<td>DOCA GPUNetIO</td>
<td>684</td>
</tr>
<tr>
<td>DOCA App Shield</td>
<td>757</td>
</tr>
<tr>
<td>Feature</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>DOCA Compress</td>
<td>793</td>
</tr>
<tr>
<td>DOCA SHA</td>
<td>817</td>
</tr>
<tr>
<td>DOCA Erasure Coding</td>
<td>834</td>
</tr>
<tr>
<td>DOCA AES-GCM</td>
<td>863</td>
</tr>
<tr>
<td>DOCA Rivermax</td>
<td>880</td>
</tr>
<tr>
<td>DOCA Telemetry Exporter</td>
<td>894</td>
</tr>
<tr>
<td>DOCA Device Emulation</td>
<td>916</td>
</tr>
<tr>
<td>DOCA DevEmu PCI</td>
<td>917</td>
</tr>
<tr>
<td>DOCA DevEmu PCI Generic</td>
<td>931</td>
</tr>
<tr>
<td>DOCA DevEmu Virtio</td>
<td>976</td>
</tr>
<tr>
<td>DOCA DevEmu Virtio-FS</td>
<td>983</td>
</tr>
</tbody>
</table>
This section describes in details how to use each DOCA library, its APIs, and different aspects related to that library.

Users may choose to only read the pages concerning DOCA libraries required for their application.

This section contains the following pages:

- DOCA Common
- DOCA Flow
- DPA Subsystem
- DOCA DMA
- DOCA Comch
- DOCA UROM
- DOCA RDMA
- DOCA Ethernet
- DOCA GPUNetIO
- DOCA App Shield
- DOCA Compress
- DOCA SHA
- DOCA Erasure Coding
- DOCA AES-GCM
- DOCA Rivermax
- DOCA Telemetry Exporter
- DOCA Device Emulation
Info

For questions, comments, and feedback, please contact us at DOCA-Feedback@exchange.nvidia.com.
DOCA Common

DOCA Common is comprised of the following libraries:

- **DOCA Core**
- **DOCA Log**

DOCA Core

This document provides guidelines on using DOCA Core objects as part of DOCA SDK programming.

Introduction

**Note**

The DOCA Core library is supported at beta level.

DOCA Core objects provide a unified and holistic interface for application developers to interact with various DOCA libraries. The DOCA Core API and objects bring a standardized flow and building blocks for applications to build upon while hiding the internal details of dealing with hardware and other software components. DOCA Core is designed to give the right level of abstraction while maintaining performance.

DOCA Core has the same API (header files) for both NVIDIA® BlueField® and CPU installations, but specific API calls may return **DOCA_ERROR_NOT_SUPPORTED** if the API is not implemented for that processor. However, this is not the case for Windows and Linux as DOCA Core does have API differences between Windows and Linux installations.

DOCA Core exposes C-language API to application writers and users must include the right header file to use according to the DOCA Core facilities needed for their application.
DOCA Core can be divided into the following software modules:

<table>
<thead>
<tr>
<th>DOCA Core Module</th>
<th>Description</th>
</tr>
</thead>
</table>
| General          | - DOCA Core enumerations and basic structures  
                  - Header files – doca_error.h, doca_types.h |
| Device handling  | - Queries device information (host-side and BlueField) and device capabilities (e.g., device's PCIe BDF address)  
                  - On BlueField  
                  - Gets local BlueField devices  
                  - Gets representors list (representing host local devices)  
                  - On the host  
                  - Gets local devices  
                  - Queries device capabilities and library capabilities  
                  - Opens and uses the selected device representor  
                  - Relevant entities – doca_devinfo, doca_devinfo_rep, doca_dev, doca_dev_rep  
                  - Header files – doca_dev.h |
| Memory management| - Handles optimized memory pools to be used by applications and enables sharing resources between DOCA libraries (while hiding hardware-related technicalities)  
                  - Data buffer services (e.g., linked list of buffers to support scatter-gather list)  
                  - Maps host memory to BlueField for direct access  
                  - Relevant entities – doca_buf, doca_mmap, doca_buf_inventory, doca_buf_array, doca_bufpool  
                  - Header files – doca_buf.h, doca_buf_inventory.h, doca_mmap.h, doca_buf_array.h, doca_bufpool |

ℹ️ **Info**

There is a symmetry between device entities on host and its representor (on BlueField). The convention of adding `rep` to the API or the object hints that it is representor-specific.
<table>
<thead>
<tr>
<th>DOCA Core Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Progress engine and task execution</td>
<td>- Enables submitting tasks to DOCA libraries and track task progress (supports both polling mode and event-driven mode)</td>
</tr>
<tr>
<td></td>
<td>- Relevant entities – doca_ctx, doca_task, doca_event, doca_event_handle_t, doca_pe</td>
</tr>
<tr>
<td></td>
<td>- Header files – doca_ctx.h</td>
</tr>
<tr>
<td>Sync events</td>
<td>- Sync events are used to synchronize different processors (e.g., synchronize BlueField and host)</td>
</tr>
<tr>
<td></td>
<td>- header files – doca_dpa_sync_event.h, doca_sync_event.h</td>
</tr>
</tbody>
</table>

The following sections describe DOCA Core's architecture and subsystems along with some basic flows that help users get started using DOCA Core.

**Prerequisites**

DOCA Core objects are supported on NVIDIA® BlueField® networking platforms (DPU or SuperNIC) and the host machine. Both must meet the following prerequisites:

- DOCA version 2.0.2 or greater
- NVIDIA® BlueField® software 4.0.2 or greater
- NVIDIA® BlueField®-3 firmware version 32.37.1000 and higher
- NVIDIA® BlueField®-2 firmware version 24.37.1000 and higher
- Please refer to the [DOCA Backward Compatibility Policy](#)

**Changes From Previous Releases**

**Changes in 2.8.0**

**Added**

- doca_bitfield.h
- `doca_error_t doca_buf_inventory_expand(struct doca_buf_inventory *inventory, uint32_t num_elements)`

- `void doca_ctx_flush_tasks(struct doca_ctx *ctx)`

- `doca_error_t doca_devinfo_cap_is_notification_moderation_supported(const struct doca_devinfo *devinfo, uint8_t *is_notification_moderation_supported)`

- **New DOCA errors:** `DOCA_ERROR_AUTHENTICATION`, `DOCA_ERROR_BAD_CONFIG`, `DOCA_ERROR_SKIPPED`

- `doca_error_t doca_task_submit_ex(struct doca_task *task, uint32_t flags)`

- `doca_error_t doca_pe_set_notification_affinity(struct doca_pe *pe, uint32_t core_id)`

- `doca_error_t doca_pe_is_set_notification_affinity_supported(const struct doca_devinfo *devinfo, uint8_t *is_set_notification_affinity_supported)`

---

**Changed**

- `doca_error_t doca_devinfo_get_active_rate(const struct doca_devinfo *devinfo, double uint64_t *active_rate); // Gb/s -> bits/s`

- `doca_buf_set_data_len` is **STABLE API**

- **Imported mmap can be exported to RDMA**

---

**Architecture**

The following sections describe the architecture for the various DOCA Core software modules. Please refer to the NVIDIA DOCA Library APIs for DOCA header documentation.

**General**

All core objects adhere to same flow that later helps in doing no allocations in the fast path.

The flow is as follows:
1. Create the object instance (e.g., `doca_mmap_create`).

2. Configure the instance (e.g., `doca_mmap_set_memory_range`).

3. Start the instance (e.g., `doca_mmap_start`).

After the instance is started, it adheres to zero allocations and can be used safely in the data path. After the instance is complete, it must be stopped and destroyed (`doca_mmap_stop`, `doca_mmap_destroy`).

There are core objects that can be reconfigured and restarted again (i.e., create configure start stop configure start). Please read the header file to see if specific objects support this option.

**doca_error_t**

All DOCA APIs return the status in the form of `doca_error_t`.

```c
typedef enum doca_error {
    DOCA_SUCCESS,
    DOCA_ERROR_UNKNOWN,
    DOCA_ERROR_NOT_PERMITTED, /**< Operation not permitted */
    DOCA_ERROR_IN_USE, /**< Resource already in use */
    DOCA_ERROR_NOT_SUPPORTED, /**< Operation not supported */
    DOCA_ERROR_AGAIN, /**< Resource temporarily unavailable, try again */
    DOCA_ERROR_INVALID_VALUE, /**< Invalid input */
    DOCA_ERROR_NO_MEMORY, /**< Memory allocation failure */
    DOCA_ERROR_INITIALIZATION, /**< Resource initialization failure */
    DOCA_ERROR_TIME_OUT, /**< Timer expired waiting for resource */
    DOCA_ERROR_SHUTDOWN, /**< Shut down in process or completed */
    DOCA_ERROR_CONNECTION_RESET, /**< Connection reset by peer */
    DOCA_ERROR_CONNECTION_ABORTED, /**< Connection aborted */
    DOCA_ERROR_CONNECTION_INPROGRESS, /**< Connection in progress */
    DOCA_ERROR_NOT_CONNECTED, /**< Not Connected */
    DOCA_ERROR_NO_LOCK, /**< Unable to acquire required lock */
    DOCA_ERROR_NOT_FOUND, /**< Resource Not Found */
    DOCA_ERROR_IO_FAILED, /**< Input/Output Operation Failed */
    DOCA_ERROR_BAD_STATE, /**< Bad State */
    DOCA_ERROR_UNSUPPORTED_VERSION, /**< Unsupported version */
    DOCA_ERROR_OPERATING_SYSTEM, /**< Operating system call failure */
    DOCA_ERROR_DRIVER, /**< DOCA Driver call failure */
    DOCA_ERROR_UNEXPECTED, /**< An unexpected scenario was detected */
} doca_error;
```
DOCA_ERROR_ALREADY_EXIST, /**< Resource already exist */
DOCA_ERROR_FULL,    /**< No more space in resource */
DOCA_ERROR_EMPTY,   /**< No entry is available in resource */
DOCA_ERROR_IN_PROGRESS, /**< Operation is in progress */
DOCA_ERROR_TOO_BIG, /**< Requested operation too big to be contained */
} doca_error_t;

See doca_error.h for more.

**Generic Structures/Enum**

The following types are common across all DOCA APIs.

```c
union doca_data {
    void *ptr;
    uint64_t u64;
};

enum doca_access_flags {
    DOCA_ACCESS_LOCAL_READ_ONLY     = 0,
    DOCA_ACCESS_LOCAL_READ_WRITE    = (1 << 0),
    DOCA_ACCESS_RDMA_READ           = (1 << 1),
    DOCA_ACCESS_RDMA_WRITE          = (1 << 2),
    DOCA_ACCESS_RDMA_ATOMIC         = (1 << 3),
    DOCA_ACCESS_DPU_READ_ONLY       = (1 << 4),
    DOCA_ACCESS_DPU_READ_WRITE      = (1 << 5),
};

enum doca_pci_func_type {
    DOCA_PCI_FUNC_PF = 0, /* physical function */
    DOCA_PCI_FUNC_VF, /* virtual function */
    DOCA_PCI_FUNC_SF, /* sub function */
};
```

For more see doca_types.h.
DOCA Device

Local Device and Representor

Prerequisites

For the representors model, BlueField must be operated in DPU mode. See NVIDIA BlueField Modes of Operation.

Topology

The DOCA device represents an available processing unit backed by hardware or software implementation. The DOCA device exposes its properties to help an application in choosing the right device(s). DOCA Core supports two device types:

- Local device – this is an actual device exposed in the local system (BlueField or host) and can perform DOCA library processing tasks.

- Representor device – this is a representation of a local device. The local device is usually on the host (except for SFs) and the representor is always on BlueField side (a proxy on BlueField for the host-side device).

The following figure provides an example topology:
The diagram shows a BlueField device (on the right side of the figure) connected to a host (on the left side of the figure). The host topology consists of two physical functions (PF0 and PF1). Furthermore, PF0 has two child virtual functions, VF0 and VF1. PF1 has only one VF associated with it, VF0. Using the DOCA SDK API, the user gets these five devices as local devices on the host.

The BlueField side has a representor-device per each host function in a 1-to-1 relation (e.g., hpf0 is the representor device for the host's PF0 device and so on) as well as a representor for each SF function, such that both the SF and its representor reside in BlueField.

If the user queries local devices on the BlueField (not representor devices), they get the two (in this example) BlueField DPU PFs, p0 and p1. These two BlueField local devices are the parent devices for:

- 7 representor devices –
5 representor devices shown as arrows to/from the host (devices with the prefix hpf*) in the diagram

2 representor devices for the SF devices, pf0sf0 and pf1sf0

- 2 local SF devices (not the SF representors), p0s0 and p1s0

In the diagram, the topology is split into two parts (note the dotted line), each part is represented by a BlueField physical device, p0 and p1, each of which is responsible for creating all other local devices (host PFs, host VFs, and BlueField SFs). As such, the BlueField physical device can be referred to as the parent device of the other devices and would have access to the representor of every other function (via doca_devinfo_rep_list_create).

**Local Device and Representor Matching**

Based on the topology diagram, the mmap export APIs can be used as follows:

<table>
<thead>
<tr>
<th>Device to Select on Host When Using doca_mmap_export_dpu()</th>
<th>BlueField Matching Representor</th>
<th>Device to Select on BlueField When Using doca_mmap_create_from_export()</th>
</tr>
</thead>
<tbody>
<tr>
<td>pf0 – 0b:00.0</td>
<td>hpf0 – 0b:00.0</td>
<td>p0 – 03:00.0</td>
</tr>
<tr>
<td>pf0vf0 – 0b:00.2</td>
<td>hpf0vf0 – 0b:00.2</td>
<td></td>
</tr>
<tr>
<td>pf0vf1 – 0b:00.3</td>
<td>hpf0vf1 – 0b:00.3</td>
<td></td>
</tr>
<tr>
<td>pf1 – 0b:00.1</td>
<td>hpf1 – 0b:00.1</td>
<td></td>
</tr>
<tr>
<td>pf1vf0 – 0b:00.4</td>
<td>hpf1vf0 – 0b:00.4</td>
<td></td>
</tr>
</tbody>
</table>

**Expected Flow**

**Device Discovery**
To work with DOCA libraries or DOCA Core objects, application must open and use a device on BlueField or host.

There are usually multiple devices available depending on the setup. See section "Topology" for more information.

An application can decide which device to select based on capabilities, the DOCA Core API, and every other library which provides a wide range of device capabilities. The flow is as follows:

1. The application gets a list of available devices.
2. Select a specific doca_devinfo to work with according to one of its properties and capabilities. This example looks for a specific PCIe address.
3. Once the doca_devinfo that suits the user's needs is found, open doca_dev.
4. After the user opens the right device, they can close the doca_devinfo list and continue working with doca_dev. The application eventually must close the doca_dev.

**Representor Device Discovery**
To work with DOCA libraries or DOCA Core objects, some applications must open and use a representor device on BlueField. Before they can open the representor device and use it, applications need tools to allow them to select the appropriate representor device with the necessary capabilities. The DOCA Core API provides a wide range of device capabilities to help the application select the right device pair (device and its BlueField representor). The flow is as follows:

1. The application "knows" which device it wants to use (e.g., by its PCIe BDF address). On the host, it can be done using DOCA Core API or OS services.

2. On the BlueField side, the application gets a list of device representors for a specific BlueField local device.

3. Select a specific doca_devinfo_rep to work with according to one of its properties. This example looks for a specific PCIe address.

4. Once the doca_devinfo_rep that suits the user's needs is found, open doca_dev_rep.

5. After the user opens the right device representor, they can close the doca_devinfo_rep list and continue working with doca_dev_rep. The application eventually must close doca_dev_rep too.

As mentioned previously, the DOCA Core API can identify devices and their representors that have a unique property (e.g., the BDF address, the same BDF for the device, and its BlueField representor).
DOCA Memory Subsystem

DOCA memory subsystem is designed to optimize performance while keeping a minimal memory footprint (to facilitate scalability) as main design goal.

DOCA memory has the following main components:

- **doca_buf** – this is the data buffer descriptor. This is not the actual data buffer, rather, it is a descriptor that holds metadata on the "pointed" data buffer.

- **doca_mmap** – this is the data buffers pool which **doca_buf** points at. The application provides the memory as a single memory region, as well as permissions for certain devices to access it.

As the **doca_mmap** serves as the memory pool for data buffers, there is also an entity called **doca_buf_inventory** which serves as a pool of **doca_buf** with same characteristics (see more in sections "DOCA Core Buffers" and "DOCA Core Inventories"). As all DOCA entities, memory subsystem objects are opaque and can be instantiated by DOCA SDK only.

The following diagram shows the various modules within the DOCA memory subsystem.

---

**Note**

Regarding representor device property caching, the function **doca_devinfo_rep_create_list** provides a snapshot of the DOCA representor device properties when it is called. If any representor's properties are changed dynamically (e.g., BDF address changes after bus reset), the device properties that the function returns would not reflect this change. One should create the list again to get the updated properties of the representors.
In the diagram, you may see two `doca_buf_inventory`s. Each `doca_buf` points to a portion of the memory buffer which is part of a `doca_mmap`. The mmap is populated with one continuous memory buffer `memrange` and is mapped to two devices, `dev1` and `dev2`.

**Requirements and Considerations**

- The DOCA memory subsystem mandates the usage of pools as opposed to dynamic allocation
  - Pool for `doca_buf` `doca_buf_inventory`
  - Pool for data memory `doca_mmap`
- The memory buffer in the mmap can be mapped to one device or more
- Devices in the mmap are restricted by access permissions defining how they can access the memory buffer
- `doca_buf` points to a specific memory buffer (or part of it) and holds the metadata for that buffer
- The internals of mapping and working with the device (e.g., memory registrations) is hidden from the application
As best practice, the application should start the `doca_mmap` in the initialization phase as the start operation is time consuming. `doca_mmap` should not be started as part of the data path unless necessary.

The host-mapped memory buffer can be accessed by BlueField

**doca_mmap**

`doca_mmap` is more than just a data buffer as it hides a lot of details (e.g., RDMA technicalities, device handling, etc.) from the application developer while giving the right level of abstraction to the software using it. `doca_mmap` is the best way to share memory between the host and BlueField so BlueField can have direct access to the host-side memory or vice versa.

DOCA SDK supports several types of mmap that help with different use cases: local mmap and mmap from export.

**Local mmap**

This is the basic type of mmap which maps local buffers to the local device(s).

1. The application creates the `doca_mmap`.

2. The application sets the memory range of the mmap using `doca_mmap_set_memrange`. The memory range is memory that the application allocates and manages (usually holding the pool of data sent to the device's processing units).

3. The application adds devices, granting the devices access to the memory region.

4. The application can specify the access permission for the devices to that memory range using `doca_mmap_set_permissions`.

   - If the mmap is used only locally, then `DOCA_ACCESS_LOCAL_*` must be specified
   - If the mmap is created on the host but shared with BlueField (see step 6), then `DOCA_ACCESS_PCI_*` must be specified
   - If the mmap is created on BlueField but shared with the host (see step 6), then `DOCA_ACCESS_PCI_*` must be specified
If the mmap is shared with a remote RDMA target, then DOCA_ACCESS_RDMA_* must be specified.

5. The application starts the mmap.

![Note]

From this point no more changes can be made to the mmap.

6. To share the mmap with BlueField/host or the RDMA remote target, call doca_mmap_export_pci or doca_mmap_export_rdma respectively. If appropriate access has not been provided, the export fails.

![Warning]

The exported data contains sensitive information. Make sure to pass this data through a secure channel!

7. The generated blob from the previous step can be shared out of band using a socket. If shared with a BlueField, it is recommended to use the DOCA Comm Channel instead. See the DMA Copy application for the exact flow.

**mmap from Export**

This mmap is used to access the host memory (from BlueField) or the remote RDMA target's memory.

1. The application receives a blob from the other side. The blob contains data returned from step 6 in the former bullet.

2. The application calls doca_mmap_create_from_export and receives a new mmap that represents memory defined by the other side.
Now the application can create `doca_buf` to point to this imported `mmap` and have direct access to the other machine's memory.

**Note**

BlueField can access memory exported to BlueField if the exporter is a host on the same machine. Or it can access memory exported through RDMA which can be on the same machine, a remote host, or on a remote BlueField.

**Note**

The host can only access memory exported through RDMA. This can be memory on a remote host, remote BlueField, or BlueField on same machine.
Buffers

The DOCA buffer object is used to reference memory that is accessible by BlueField hardware. The buffer can be utilized across different BlueField accelerators. The buffer may reference CPU, GPU, host, or even RDMA memory. However, this is abstracted so once a buffer is created, it can be handled in a similar way regardless of how it got created. This section covers usage of the DOCA buffer after it is allocated.

The DOCA buffer has an address and length describing a memory region. Each buffer can also point to data within the region using the data address and data length. This distinguishes three sections of the buffer: The headroom, the dataroom, and the tailroom.

- Headroom – memory region starting from the buffer's address up to the buffer's data address
- Dataroom – memory region starting from the buffer's data address with a length indicated by the buffer's data length
- Tailroom – memory region starting from the end of the dataroom to the end of the buffer
- Buffer length – the total length of the headroom, the dataroom, and the tailroom

Buffer Considerations
There are multiple ways to create the buffer but, once created, it behaves in the same way (see section "Inventories").

- The buffer may reference memory that is not accessible by the CPU (e.g., RDMA memory)
- The buffer is a thread-unsafe object
- The buffer can be used to represent non-continuous memory regions (scatter/gather list)
- The buffer does not own nor manage the data it references. Freeing a buffer does not affect the underlying memory.

**Headroom**

The headroom is considered user space. For example, this can be used by the user to hold relevant information regarding the buffer or data coupled with the data in the buffer's dataroom.

This section is ignored and remains untouched by DOCA libraries in all operations.

**Dataroom**

The dataroom is the content of the buffer, holding either data on which the user may want to perform different operations using DOCA libraries or the result of such operations.

**Tailroom**

The tailroom is considered as free writing space in the buffer by DOCA libraries (i.e., a memory region that may be written over in different operations where the buffer is used as output).
Buffer as Source

When using `doca_buf` as a source buffer, the source data is considered as the data section only (the dataroom).

Buffer as Destination

When using `doca_buf` as a destination buffer, data is written to the tailroom (i.e., appended after existing data, if any).

When DOCA libraries append data to the buffer, the data length is increased accordingly.

Scatter/Gather List

To execute operations on non-continuous memory regions, it is possible to create a buffer list. The list would be represented by a single `doca_buf` which represents the head of the list.

To create a list of buffers, the user must first allocate each buffer individually and then chain them. Once they are chained, they can be unchained as well:

- The chaining operation, `doca_buf_chain_list()`, receives two lists (heads) and appends the second list to the end of the first list
- The unchaining operation, `doca_buf_unchain_list()`, receives the list (head) and an element in the list, and separates them
- Once the list is created, it can be traversed using `doca_buf_get_next_in_list()`. NULL is returned once the last element is reached.

Passing the list to another library is same as passing a single buffer; the application sends the head of the list. DOCA libraries that support this feature can then treat the memory regions that comprise the list as one contiguous.

When using the buffer list as a source, the data of each buffer (in the dataroom) is gathered and used as continuous data for the given operation.
When using the buffer list as destination, data is scattered in the tailroom of the buffers in the list until it is all written (some buffers may not be written to).

**Buffer Use Cases**

The DOCA buffer is widely used by the DOCA acceleration libraries (e.g., DMA, compress, SHA). In these instances, the buffer can be provided as a source or as a destination.

Buffer use-case considerations:

- If the application wishes to use a linked list buffer and concatenate several `doca_bufs` to a scatter/gather list, the application is expected to ensure the library indeed supports a linked list buffer. For example, to check linked-list support for DMA memcpy task, the application may call `doca_dma_cap_task_memcpy_get_max_buf_list_len()`.

- Operations made on the buffer’s data are not atomic unless stated otherwise.

- Once a buffer has been passed to the library as part of the task, ownership of the buffer moves to the library until that task is complete.

**Note**

When using `doca_buf` as an input to some processing library (e.g., `doca_dma`), `doca_buf` must remain valid and unmodified until processing is complete.

- Writing to an in-flight buffer may result in anomalous behavior. Similarly, there are no guarantees for data validity when reading from an in-flight buffer.

**Inventories**

The inventory is the object responsible for allocating DOCA buffers. The most basic inventory allows allocations to be done without having to allocate any system memory.
Other inventories involve enforcing that buffer addresses do not overlap.

**Inventory Considerations**

- All inventories adhere to zero allocation after start.

- Allocation of a DOCA buffer requires a data source and an inventory.
  - The data source defines where the data resides, what can access it, and with what permissions.
  - The data source must be created by the application. For creation of mmaps, see `doca_mmap`.

- The inventory describes the allocation pattern of the buffers, such as, random access or pool, variable-size or fixed-size buffers, and continuous or non-continuous memory.

- Some inventories require providing the data source, `doca_mmap`, when allocating the buffers, others require it on creation of the inventory.

- All inventory types are thread-unsafe.

**Inventory Types**

<table>
<thead>
<tr>
<th>Inventory Type</th>
<th>Characteristics</th>
<th>When to Use</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>doca_buf_inventory</code></td>
<td>Multiple mmaps, flexible address, flexible buffer size.</td>
<td>When multiple sizes or mmaps are used.</td>
<td>Most common use case.</td>
</tr>
<tr>
<td><code>doca_buf_array</code></td>
<td>Single mmap, fixed buffer size. User receives an array of pointers to DOCA buffers. In case of DPA, mmap and buffer size can be</td>
<td>Use for creating DOCA buffers on GPU or DPA.</td>
<td><code>doca_buf_arr</code> can be configured on the CPU and created on the GPU or DPA</td>
</tr>
</tbody>
</table>
| Inven-
<table>
<thead>
<tr>
<th>tory</th>
<th>Characteristics</th>
<th>When to Use</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|      | unconfigured and later can be set from the DPA. |             |       |

|      | Single mmap, fixed buffer size, address not controlled by the user. | Use as a pool of buffers of the same characteristics when buffer address is not important. | Slightly faster than doca_buf_inventory. |

## Example Flow

The following is a simplified example of the steps expected for exporting the host mmap to BlueField to be used by DOCA for direct access to the host memory (e.g., for DMA):

1. Create mmap on the host (see section "Expected Flow" for information on how to choose the doca_dev to add to mmap if exporting to BlueField). This example adds a single doca_dev to the mmap and exports it so the BlueField/RDMA endpoint can use it.

```c
1. doca_mmap_create(mmap_out: struct doca_mmap **): doca_error_t
2. doca_mmap_set_memrange(mmap: struct doca_mmap *, address: void *, length: size_t): doca_error_t
3. doca_mmap_set_permissions(mmap: struct doca_mmap *, access_mask: uint32_t): doca_error_t
4. doca_mmap_add_dev(mmap: struct doca_mmap *, dev: struct doca_dev *): doca_error_t
5. doca_mmap_start(mmap: struct doca_mmap *): doca_error_t
```

2. Import to the BlueField/RDMA endpoint (e.g., use the mmap descriptor output parameter as input to doca_mmap_create_from_export).
The execution model is based on hardware processing on data and application threads. DOCA does not create an internal thread for processing data.

Example: Chain the Two buffers

doca_buf_dec_refcount(buf: struct doca_buf *, refcount: uint16_t *): doca_error_t

doca_buf_inventory_stop(inventory: struct doca_buf_inventory *): doca_error_t

doca_buf_inventory_destroy(inventory: struct doca_buf_inventory *): doca_error_t

doca_mmap_destroy(mmap: struct doca_mmap *): doca_error_t

DOCA Execution Model
The workload is made up of tasks and events. Some tasks transform source data to destination data. The basic transformation is a DMA operation on the data which simply copies data from one memory location to another. Other operations allow users to receive packets from the network or involve calculating the SHA value of the source data and writing it to the destination.

For instance, a transform workload can be broken into three steps:

1. Read source data (doca_buf see memory subsystem).
2. Apply an operation on the read data (handled by a dedicated hardware accelerator).
3. Write the result of the operation to the destination (doca_buf see memory subsystem).

Each such operation is referred to as a task (doca_task).

Tasks describe operations that an application would like to submit to DOCA (hardware or BlueField). To do so, the application requires a means of communicating with the hardware/BlueField. This is where the doca_pe comes into play. The progress engine (PE) is a per-thread object used to queue tasks to offload to DOCA and eventually receive their completion status.

doca_pe introduces three main operations:

1. Submission of tasks.
2. Checking progress/status of submitted tasks.
3. Receiving a notification on task completion (in the form of a callback).

A workload can be split into many different tasks that can be executed on different threads; each thread represented by a different PE. Each task must be associated to some context, where the context defines the type of task to be done.

A context can be obtained from some libraries within the DOCA SDK. For example, to submit DMA tasks, a DMA context can be acquired from doca_dma.h, whereas SHA context can be obtained using doca_sha.h. Each such context may allow submission of several task types.

A task is considered asynchronous in that once an application submits a task, the DOCA execution engine (hardware or BlueField) would start processing it, and the application
can continue to do some other processing until the hardware finishes. To keep track of which task has finished, there are two modes of operation: **polling mode** and **event-driven mode**.

### Requirements and Considerations

- The task submission/execution flow/API is optimized for performance (latency)

- DOCA does not manage internal (operating system) threads. Rather, progress is managed by application resources (calling DOCA API in polling mode or waiting on DOCA notification in event-driven mode).

- The basic object for executing the task is a **doca_task**. Each task is allocated from a specific DOCA library context.

- **doca_pe** represents a logical thread of execution for the application and tasks submitted to the progress engine (PE)

#### Note

PE is not thread safe and it is expected that each PE is managed by a single application thread (to submit a task and manage the PE).

- Execution-related elements (e.g., **doca_pe**, **doca_ctx**, **doca_task**) are opaque and the application performs minimal initialization/configuration before using these elements

- A task submitted to PE can fail (even after the submission succeeds). In some cases, it is possible to recover from the error. In other cases, the only option is to reinitialize the relevant objects.

- PE does not guarantee order (i.e., tasks submitted in certain order might finish out-of-order). If the application requires order, it must impose it (e.g., submit a dependent task once the previous task is done).

- A PE can either work in polling mode or event-driven mode, but not in both at same time
All DOCA contexts support polling mode (i.e., can be added to a PE that supports polling mode)

DOCA Context

DOCA Context (struct `doca_ctx`) defines and provides (implements) task/event handling. A context is an instance of a specific DOCA library (i.e., when the library provides a DOCA Context, its functionality is defined by the list of tasks/events it can handle). When more than one type of task is supported by the context, it means that the supported task types have a certain degree of similarity to implement and utilize common functionality.

The following list defines the relationship between task contexts:

- Each context utilizes at least one DOCA Device functionality/accelerated processing capabilities
- For each task type there is one and only context type supporting it
- A context virtually contains an inventory per supported task type
- A context virtually defines all parameters of processing/execution per task type (e.g., size of inventory, device to accelerate processing)

Each context needs an instance of progress engine (PE) as a runtime for its tasks (i.e., a context must be associated with a PE to execute tasks).

The following diagram shows the high-level (domain model) relations between various DOCA Core entities.

```
«Entity» 1 connected contexts «Control»
  «Entity»
  «Doca PE» doca_pe 1..* doca_ctx
  «Doca Task»
  1..* submit contains
  «Doca Device»

1. `doca_task` is associated to a relevant `doca_ctx` that executes the task (with the help of the relevant `doca_dev`).
```
2. `doca_task`, after it is initialized, is submitted to `doca_pe` for execution.

3. `doca_ctxs` are connected to the `doca_pe`. Once a `doca_task` is queued to `doca_pe`, it is executed by the `doca_ctx` that is associated with that task in this PE.

The following diagram describes the initialization sequence of a context:

![Diagram of context initialization sequence]

After the context is started, it can be used to enable the submission of tasks to a PE based on the types of tasks that the context supports. See section "DOCA Progress Engine" for more information.

**Note**

Context is a thread-unsafe object which can be connected to a single PE only.

**Configuration Phase**

A DOCA context must be configured before attempting to start it using `doca_ctx_start()`. Some configurations are mandatory (e.g., providing `doca_dev`) while others are not.

- Configurations can be useful to allow certain tasks/events, to enable features which are disabled by default, and to optimize performance depending on a specific workload.

- Configurations are provided using setter functions. Refer to context documentation for a list of mandatory and optional configurations and their corresponding APIs.

- Configurations are provided after creating the context and before starting it. Once the context is started, it can no longer be configured unless it is stopped again.
Examples of common configurations:

- Providing a device – usually done as part of the create API
- Enabling tasks or registering to events – all tasks are disabled by default

**Execution Phase**

Once context configuration is complete, the context can be used to execute tasks. The context executes the tasks by offloading the workload to hardware, while software polls the tasks (i.e., waits) until they are complete.

In this phase, an application uses the context to allocate and submit asynchronous tasks, and then polls tasks (waits) until completion.

The application must build an event loop to poll the tasks (wait), utilizing one of the following modes:

- **Polling Mode**
- **Notification-driven Mode**

In this phase, the context and all core objects perform zero allocations by utilizing memory pools. It is recommended that the application utilizes same approach for its own logic.

**State Machine**

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
</table>
| Idle  | - 0 in-flight tasks  
       | - On init (right after `doca_<T>_create(ctx)`): All configuration APIs enabled  
<pre><code>   | - On reconf (on transition from stopping state): Some configuration APIs enabled |
</code></pre>
<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting</td>
<td>This state is mandatory for CTXs where transition to running state is conditioned by one or more async op completions/external events. For example, when a client connects to comm channel, it enters running state. Waiting for state change can be terminated by a voluntary (user) <code>doctx_ctx_stop()</code> call or involuntary context state change due to internal error.</td>
</tr>
</tbody>
</table>
| Running  | - Task allocation/submission enabled (disabled in all other states)  
- All configuration APIs are disabled                                                                                                                                                     |
| Stopping | - Preparation before stopped state  
- Clean all in-flight tasks that may not complete in near future  
- Procedures relying on external entity actions should be terminated by CTX logic                                                                                                         |

The following diagram describes DOCA Context state transitions:
**Internal Error**

DOCA Context states can encounter internal errors at any time. If the state is starting or running, an internal error can cause an involuntary transition to stopping state.

For instance, an involuntary transition from running to stopping can happen when a task execution fails. This results in a completion with error for the failed task and all subsequent task completions.

After stopping, the state may become idle. However, `doca_ctx_start()` may fail if there is a configuration issue or if an error event prevented proper transition to starting or running state.
DOCA Task

A task is a unit of (functional/processing) workload offload-able to hardware. The majority of tasks utilize NVIDIA® BlueField® and NVIDIA® ConnectX® hardware to provide accelerated processing of the workload defined by the task. Tasks are asynchronous operations (e.g., tasks submitted for processing via non-blocking `doca_task_submit()` API).

Upon task completion, the preset completion callback is executed in context of `doca_pe_progress()` call. The completion callback is a basic/generic property of the task, similar to user data. Most tasks are IO operations executed/accelerated by NVIDIA device hardware.

**Task Properties**

Task properties share generic properties which are common to all task types and type-specific properties. Since task structure is opaque (i.e., its content not exposed to the user), the access to task properties provided by set/get APIs.

The following are generic task properties:

- Setting completion callback – it has separate callbacks for successful completion and completion with failure.
- Getting/setting user data – used in completion callback as some structure associated with specific task object.
- Getting task status – intended to retrieve error code on completion with failure.

For each task there is only one owner: a context object. There is a `doca_task_get_ctx()` API to get generic context object.

The following are generic task APIs:

- Allocating and freeing from CTX (internal/virtual) inventory
- Configuring via setters (or init API)
- Submit-able (i.e., implements `doca_task_submit(task)`)

DOCA Libraries
Upon completion, there is a set of getters to access the results of the task execution.

**Task Lifecycle**

This section describes the lifecycle of DOCA Task. Each DOCA Task object lifecycle:

- starts on the event of entering *Running* state by the DOCA Context owning the task i.e., once *Running* state entered application can obtain the task from CTX by calling `doca_<CTX name>_task_<Task name>_alloc_init(ctx, ... &task)`.

- ends on the event of entering *Stopped* state by the DOCA Context owning the task i.e., application can no longer allocate tasks once the related DOCA Context left the *Running* state.

From application perspective DOCA Context provides a virtual task inventory The diagram below shows the how ownership if the DOCA Task passed from DOCA Context virtual inventory to application and than from application back to CTX, pay attention to the colors used in activation bars for application (APP) participant & DOCA Context (CTX) participant and DOCA Context Task virtual inventory (Task).

The diagram below shows the lifecycle of DOCA Task staring from its allocation to its submission.
The diagram above displays following ownership transitions during DOCA Task object lifecycle:

- starting from allocation task ownership passed from context to application

- application may modify task attributes via API templated as `doca_<CTX name>_task_<Task name>_set_<Parameter name>(task, param);` on return from the task modification call the ownership of the task object returns to application.

- submit the task for processing in the PE, once all required modifications/settings of the task object completed. On task submission the ownership of the object passed to the related context.

The next two diagrams below shows the lifecycle of DOCA Task on its completion.
The diagram above displays following ownership transitions during DOCA Task object lifecycle:

- on **DOCA Task** completion the appropriate handler provided by application invoked; on handler invocation the **DOCA Task** ownership passed to application.

- after **DOCA Task** completion application may access task attributes & result fields utilizing appropriate APIs; application remains owner of the task object.

- application may call `doca_task_free()` when task is no longer needed; on return from the call task ownership passed to **DOCA Context** while task became uninitialized & pre-allocated till the context enters Idle state.
The diagram above displays similar to the previous diagram ownership transitions during DOCA Task object lifecycle with the only difference that instead of `doca_task_free(task)` `doca_task_submit(task)` was called:

- **DOCA Task** result (related attributes) can be accessed right after enter successful task completion callback, similar to the previous case

- lifecycle of the **DOCA Task** results ends on exit from the task completion callback scope.

- On `doca_task_free()` or `doca_<CTX name>_task_<Task name>_set_<Parameter name>(task, param)` call all task results should be considered invalidated regardless of scope.

The diagram below shows the lifecycle of **DOCA Task** set-able parameters while API to set such a parameter templated as `doca_<CTX name>_task_<Task name>_set_<Parameter name>(task, param)`.
**Green activation** of **param** participant describes the time slice when all *DOCA Task* parameters owned by DOCA library. On `doca_task_submit()` call the ownership on all task arguments passed from application to the DOCA Context the related Task object belongs to. The ownership of task arguments passed back to application on task completion. The application should not modify and/or destroy/free Task argument related objects if it doesn’t own the argument.

**DOCA Progress Engine**

The progress engine (PE) enables asynchronous processing and handling of multiple tasks and events of different types in a single-threaded execution environment. It is an event loop for all context-based DOCA libraries, with I/O completion being the most common event type.

PE is designed to be thread unsafe (i.e., it can only be used in one thread at a time) but a single OS thread can use multiple PEs. The user can assign different priorities to different contexts by adding them to different PEs and adjusting the polling frequency for each PE.
accordingly. Another way to view the PE is as a queue of workload units that are scheduled for execution.

There are no explicit APIs to add and/or schedule a workload to/on a PE but a workload can be added by:

- Adding a DOCA context to PE
- Registering a DOCA event to probe (by the PE) and executing the associated handler if the probe is positive

PE is responsible for scheduling workloads (i.e., picking the next workload to execute). The order of workload execution is independent of task submission order, event registration order, or order of context associations with a given PE object. Multiple task completion callbacks may be executed in an order different from the order of related task submissions.

The following diagram describes the initialization flow of the PE:

After a PE is created and connected to contexts, it can start progressing tasks which are submitted to the contexts. Refer to context documentation to find details such as what tasks can be submitted using the context.

Note that the PE can be connected to multiple contexts. Such contexts can be of the same type or of different types. This allows submitting different task types to the same PE and waiting for any of them to finish from the same place/thread.

After initializing the PE, an application can define an event loop using one of these modes:

- **Polling Mode**
- **Notification-driven Mode**
PE as Event Loop Mode of Operation

All completion handlers for both tasks and events are executed in the context of `doca_pe_progress()`. `doca_pe_progress()` loops for every workload (i.e., for each workload unit) scheduled for execution:

Run the selected workload unit. For the following cases:

- Task completion, execute associated handler and break the loop and return status made some progress
- Positive probe of event, execute associated handler and break the loop and return status made some progress
- Considerable progress is made to contribute to future task completion or positive event probe, break the loop and return status made some progress

Otherwise, reach the end of the loop and return status no progress.

Polling Mode

In this mode, the application submits a task and then does busy-wait to find out when the task has completed.

The following diagram demonstrates this sequence:
1. The application submits all tasks (one or more) and tracks the number of task completions to know if all tasks are done.

2. The application waits for a task to complete by consecutive polls on `doca_pe_progress()`.

   1. If `doca_pe_progress()` returns 1, it means progress is being made (i.e., some task completed or some event handled).

   2. Each time a task is completed or an event is handled, its preset completion or event handling callback is executed accordingly.

   3. If a task is completed with an error, preset task completion with error callback is executed (see section "Error Handling").
3. The application may add code to completion callbacks or event handlers for tracking the amount of completed and pending workloads.

**Note**

In this mode, the application is always using the CPU even when it is doing nothing (busy-wait).

**Blocking Mode - Notification Driven**

In this mode, the application submits a task and then waits for a notification to be received before querying the status.

The following diagram demonstrates this sequence:
1. The application gets a notification handle from the `doa_pe` representing a Linux file descriptor which is used to signal the application that some work has finished.

2. The application then arms the PE with `doa_pe_request_notification()`.

**Note**
This must be done every time an application is interested in receiving a notification from the PE.

**Note**

After `doca_pe_request_notification()`, no calls to `doca_pe_progress()` are allowed. In other words, `doca_pe_request_notification()` should be followed by `doca_pe_clear_notification` before any calls to `doca_pe_progress()`.

3. The application submits a task.

4. The application waits (e.g., Linux epoll/select) for a signal to be received on the `pe-fd`.

5. The application clears the notifications received, notifying the PE that a signal has been received and allowing it to perform notification handling.

6. The application attempts to handle received notifications via (multiple) calls to `doca_pe_progress()`.

**Note**

There is no guarantee that the call to `doca_pe_progress()` would execute any task completion/event handler, but the PE can continue the operation.

7. The application handles its internal state changes caused by task completions and event handlers called in the previous step.

8. Repeat steps 2-7 until all tasks are completed and all expected events are handled.
Progress Engine versus Epoll

The epoll mechanism in Linux and the DOCA PE handles high concurrency in event-driven architectures. Epoll, like a post office, tracks "mailboxes" (file descriptors) and notifies the "postman" (the `epoll_wait` function) when a "letter" (event) arrives. DOCA PE, like a restaurant, uses a single "waiter" to handle "orders" (workload units) from "customers" (DOCA contexts). When an order is ready, it is placed on a "tray" (task completion handler/event handler execution) and delivered in the order received. Both systems efficiently manage resources while waiting for events or tasks to complete.

DOCA Event

An event is a type of occurrence that can be detected or verified by the DOCA software, which can then trigger a handler (a callback function) to perform an action. Events are associated with a specific source object, which is the entity whose state or attribute change defines the event's occurrence. For example, a context state change event is caused by the change of state of a context object.

To register an event, the user must call the `doca_<event_type>_reg(pe, ...)` function, passing a pointer to the user handler function and an opaque argument for the handler. The user must also associate the event handler with a PE, which is responsible for running the workloads that involve event detection and handler execution.

Once an event is registered, it is periodically checked by the `doca_pe_progress()` function, which runs in the same execution context as the PE to which the event is bound. If the event condition is met, the handler function is invoked. Events are not thread-safe objects and should only be accessed by the PE to which they are bound.
Error Handling

After a task is submitted successfully, consequent calls to `doca_pe_progress()` may fail (i.e., task failure completion callback is called).

Once a task fails, the context may transition to stopping state, in this state, the application has to progress all in-flight tasks until completion before destroying or restarting the context.

The following diagram shows how an application may handle an error from `doca_pe_progress()`:
1. Application runs event loop.

2. Any of the following may happen:
   - [Optional] Task fails, and the task failed completion handler is called
     - This may be caused by bad task parameters or another fatal error
     - Handler releases the task and all associated resources
   - [Optional] Context transitions to stopping state, and the context state changed handler is called
This may be caused by failure of a task or another fatal error

- In this state, all in-flight tasks are guaranteed to fail
- Handler releases tasks that are not in-flight if such tasks exist
  - [Optional] Context transitions to idle state, and the context state changed handler is called
    - This may happen due to encountering an error and the context does not have any resources that must be freed by the application
    - In this case, the application may decide to recover the context by calling start again or it may decide to destroy the context and possibly exit the application

**Task and Event Batching**

DOCA Batching is an approach for grouping multiple tasks or events of the same type and handling them as a single unit. DOCA offers two options of achieving this as described in the following subsections.

**Batch Task/Event**

In this batching option, a library (e.g., `doca_eth_txq`) offers a task that represents a batched operation (e.g., sending multiple packets), the task is considered a batch task and has a task type that is separate from the non-batched operation (e.g., sending a single packet).

To submit the batch task, the user is required to build the batch and then submit it at once, similar to submitting a regular task.

The completion of the batch is based on the completion of all items in the batch and is handled as the completion of a single unit. This allows for multiple DOCA Task initialization/submission and multiple DOCA Task/Event completion handling in a single API call (see DOCA Ethernet for example).

**Iterative Batch**
In this batching option, it is possible to utilize existing task types to build a batch operation, where each task within the batch is submitted individually and each task receives its own completion.

Furthermore, the batch is built iteratively, where the user is not required to have information for the entire batch ahead of time.

To utilize this option, the user can submit each task in the batch using an extended submit API `doca_task_submit_ex` while providing additional submit flags.

The extended submit API is similar to a regular submit API (`doca_task_submit`) but with the ability to receive submit flags. These flags are used as hints to the library that executes the tasks. They can have implications on the current task but may also have implications on previously submitted flags, as described in the following table:

<table>
<thead>
<tr>
<th>Submit Flag</th>
<th>Effect on Current Task</th>
<th>Effect on Previous Tasks</th>
<th>Default Behavior of <code>doca_task_submit</code></th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag Provided</td>
<td>Flag not Provided</td>
<td>Flag Provided</td>
<td>Flag not Provided</td>
<td>As long as the task is unflushed, it never completes. The flag allows batching such that multiple tasks are flushed at once, instead of individually.</td>
</tr>
<tr>
<td>Submit Flag</td>
<td>Effect on Current Task</td>
<td>Effect on Previous Tasks</td>
<td>Default Behavior of doca_task_submit</td>
<td>Comments</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------</td>
<td>--------------------------</td>
<td>--------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>DOCA_TASK_SUBMIT_FLAG_OPTIMIZE_REPORTS</td>
<td>The user does not receive task completion after hardware has completed execution of the task, and the completion is considered &quot;unreported&quot;.</td>
<td>The user receives task completion after hardware has completed execution of the task, and the completion is considered &quot;reported&quot;.</td>
<td>None</td>
<td>Once the hardware completes execution of this task, all previous unreported completions become reported.</td>
</tr>
</tbody>
</table>

As long as the task is unreported, the user would never know that it has been completed. The completion of a task is reported through a completion callback using the progress engine. The library does not guarantee any order of execution/completion of tasks. The flag allows batching, such that multiple task completions are reported using a single hardware completion, instead of receiving a
<table>
<thead>
<tr>
<th>Submit Flag</th>
<th>Effect on Current Task</th>
<th>Effect on Previous Tasks</th>
<th>Default Behavior of <code>doca_task_submit</code></th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>completion for every task.</td>
</tr>
</tbody>
</table>

1. Note that these flags are hints which may allow internal optimizations. However, on a task by task basis, the library may decide to ignore user flags and revert to default submit behavior.

2. "Previous tasks" only refers to tasks submitted to the same library instance (doca_ctx). The flags do not allow optimizations across different library instances.

3. "previous" refers to tasks that have been submitted before this one.

**DOCA Graph Execution**

DOCA Graph facilitates running a set of actions (tasks, user callbacks, graphs) in a specific order and dependencies. DOCA Graph runs on a DOCA progress engine.

DOCA Graph creates graph instances that are submitted to the progress engine (doca_graph_instance_submit).

**Nodes**

DOCA Graph is comprised of context, user, and sub-graph nodes. Each of these types can be in any of the following positions in the network:

- Root nodes – a root node does not have a parent. The graph can have one or more root nodes. All roots begin running when the graph instance is submitted.
• Edge nodes – an edge node is a node that does not have child nodes connected to it. The graph instance is completed when all edge nodes are completed.

• Intermediate node – a node connected to parent and child nodes

**Context Node**

A context node runs a specific DOCA task and uses a specific DOCA context (doca_ctx). The context must be connected to the progress engine before the graph is started.

The task lifespan must be longer or equal to the life span of the graph instance.

**User Node**

A user node runs a user callback to facilitate performing actions during the run time of the graph instance (e.g., adjust next node task data, compare results).

**Sub-graph Node**

A sub-graph node runs an instance of another graph.

**Using DOCA Graph**

1. Create the graph using doca_graph_create.

2. Create the graph nodes (e.g., doca_graph_node_create_from_ctx).

3. Define dependencies using doca_graph_add_dependency.
4. Start the graph using `doca_graph_start`.

5. Create the graph instance using `doca_graph_instance_create`.

6. Set the nodes data (e.g., `doca_graph_instance_set_ctx_node_data`).

7. Submit the graph instance to the pe using `doca_graph_instance_submit`.

8. Call `doca_pe_progress` until the graph callback is invoked.
   - Progress engine can run graph instances and standalone tasks simultaneously.

**DOCA Graph Limitations**

- DOCA Graph does not support circle dependencies
- DOCA Graph must contain at least one context node. A graph containing a sub-graph with at least one context node is a valid configuration.

**DOCA Graph Sample**

The graph sample is based on the DOCA DMA library. The sample copies 2 buffers using DMA.

The graph ends with a user callback node that compares source and destinations.

**Running DOCA Graph Sample**

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_common/graph/
   meson build
   ninja -C build
   ```

3. Sample (e.g., doca_graph) usage:

   ```
   ./build/doca_graph
   ```

   No parameters required.

**Alternative Data Path**

DOCA Progress Engine utilizes the CPU to offload data path operations to hardware. However, some libraries support utilization of DPA and/or GPU.

Considerations:

- Not all contexts support alternative datapath
- Configuration phase is always done on CPU
- Datapath operations are always offloaded to hardware. The unit that offloads the operation itself can be either CPU/DPA/GPU.
- The default mode of operation is CPU
- Each mode of operation introduces a different set of APIs to be used in execution path. The used APIs are mutually exclusive for specific context instance.
DPA

Users must first refer to the programming guide of the relevant context (e.g., DOCA RDMA) to check if datapath on DPA is supported. Additionally, the guide provides what operations can be used.

To set the datapath mode to DPA, acquire a DOCA DPA instance, then use the `doca_ctx_set_datapath_on_dpa()` API.

After the context has been started with this mode, it becomes possible to get a DPA handle, using an API defined by the relevant context (e.g., `doca_rdma_get_dpa_handle()`). This handle can then be used to access DPA data path APIs within DPA code.

GPU

Users must first refer to the programming guide of the relevant context (E.g., DOCA Ethernet) to check if datapath on GPU is supported. Additionally, the guide provides what operations can be used.

To set the data path mode to GPU, acquire a DOCA GPU instance, then use the `doca_ctx_set_datapath_on_gpu()` API.

After the context has been started with this mode, it becomes possible to get a GPU handle, using an API defined by the relevant context (e.g., `doca_eth_rxq_get_gpu_handle()`). This handle can then be used to access GPU data path APIs within GPU code.

Object Life Cycle

Most DOCA Core objects share the same handling model in which:

1. The object is allocated by DOCA so it is opaque for the application (e.g., `doca_buf_inventory_create`, `doca_mmap_create`).

2. The application initializes the object and sets the desired properties (e.g., `doca_mmap_set_memrange`).
3. The object is started, and no configuration or attribute change is allowed (e.g.,
   doca_buf_inventory_start, doca_mmap_start).

4. The object is used.

5. The object is stopped and deleted (e.g., doca_buf_inventory_stop
   doca_buf_inventory_destroy, doca_mmap_stop doca_mmap_destroy).

The following procedure describes the mmap export mechanism between two machines
(remote machines or host-BlueField):

1. Memory is allocated on Machine1.

2. Mmap is created and is provided memory from step 1.

3. Mmap is exported to the Machine2 pinning the memory.

4. On the Machine2, an imported mmap is created and holds a reference to actual
   memory residing on Machine1.

5. Imported mmap can be used by Machine2 to allocate buffers.

6. Imported mmap is destroyed.

7. Exported mmap is destroyed.

8. Original memory is destroyed.

**RDMA Bridge**

The DOCA Core library provides building blocks for applications to use while abstracting
many details relying on the RDMA driver. While this takes away complexity, it adds
flexibility especially for applications already based on rdma-core. The RDMA bridge allows
interoperability between DOCA SDK and rdma-core such that existing applications can
convert DOCA-based objects to rdma-core-based objects.

**Requirements and Considerations**

- This library enables applications already using rdma-core to port their existing
  application or extend it using DOCA SDK.
• Bridge allows converting DOCA objects to equivalent rdma-core objects.

**DOCA Core Objects to RDMA Core Objects Mapping**

The RDMA bridge allows translating a DOCA Core object to a matching RDMA Core object. The following table shows how the one object maps to the other.

<table>
<thead>
<tr>
<th>RDMA Core Object</th>
<th>DOCA Equivalent</th>
<th>RDMA Object to DOCA Object</th>
<th>DOCA Object to RDMA Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibv_pd</td>
<td>doca_dev</td>
<td>doca_rdma_bridge_open_dev_from_pd</td>
<td>doca_rdma_bridge_get_dev_pd</td>
</tr>
<tr>
<td>ibv_mr</td>
<td>doca_buf</td>
<td>-</td>
<td>doca_rdma_bridge_get_buf_mk</td>
</tr>
</tbody>
</table>

**DOCA Core Samples**

ℹ️ **Info**

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

**Progress Engine Samples**

All progress engine (PE) samples use DOCA DMA because of its simplicity. PE samples should be used to understand the PE not DOCA DMA.

**pe_common**

`pe_common.c` and `pe_common.h` contain code that is used in most or all PE samples.
Users can find core code (e.g., create MMAP) and common code that uses PE (e.g., poll_for_completion).

Struct pe_sample_state_base (defined in pe_common.h) is the base state for all PE samples, containing common members that are used by most or all PE samples.

**pe_polling**

The polling sample is the most basic sample for using PE. Start with this sample to learn how to use DOCA PE.

![Info]

You can diff between `pe_polling_sample.c` and any other `pe_x_sample.c` to see the unique features that the other sample demonstrates.

The sample demonstrates the following functions:

- How to create a PE
- How to connect a context to the PE
- How to allocate tasks
- How to submit tasks
- How to run the PE
- How to cleanup (e.g., destroy context, destroy PE)
The sample performs the following:

1. Uses one DMA context.
2. Allocates and submits 16 DMA tasks.
3. Polls until all tasks are completed.

**Info**

Task completion callback checks that the copied content is valid.

3. Polls until all tasks are completed.

**pe_async_stop**

A context can be stopped while it still processes tasks. This stop is asynchronous because the context must complete/abort all tasks.

The sample demonstrates the following functions:

- How to asynchronously stop a context
- How to implement a context state changed callback (with regards to context moving from stopping to idle)
- How to implement task error callback (check if this is a real error or if the task is flushed)

The sample performs the following:

1. Submits 16 tasks and stops the context after half of the tasks are completed.

Pay attention to the order of destruction (e.g., all contexts must be destroyed before the PE).
2. Polls until all tasks are complete (half are completed successfully, half are flushed).

The difference between `pe_polling_sample.c` and `pe_async_stop_sample.c` is to learn how to use PE APIs for event-driven mode.

**pe_event**

Event-driven mode reduces CPU utilization (wait for event until a task is complete) but may increase latency or reduce performance.

The sample demonstrates the following functions:

- How to run the PE in event-driven mode

The sample performs the following:

1. Runs 16 DMA tasks.
2. Waits for event.

The difference between `pe_polling_sample.c` and `pe_event_sample.c` is to learn how to use PE APIs for event-driven mode.

**pe_multi_context**

A PE can host more than one instance of a specific context. This facilitates running a single PE with multiple BlueField devices.

The sample demonstrates the following functions:

- How to run a single PE with multiple instances of a specific context

The sample performs the following:

1. Connects 4 instances of DOCA DMA context to the PE.
2. Allocates and submits 4 tasks to every context instance.
3. Polls until all tasks are complete.

The difference between `pe_polling_sample.c` and `pe_multi_context_sample.c` is to learn how to use PE with multiple instances of a context.

**pe_reactive**

PE and contexts can be maintained in callbacks (task completion and state changed).

The sample demonstrates the following functions:

- How to maintain the context and PE in the callbacks instead of the program’s main function

The user must make sure to:

- Review the task completion callback and the state changed callbacks
- Review the difference between `poll_to_completion` and the polling loop in main

The sample performs the following:

1. Runs 16 DMA tasks.
2. Stops the DMA context in the completion callback after all tasks are complete.

The difference between `pe_polling_sample.c` and `pe_reactive_sample.c` is to learn how to use PE in reactive model.

**pe_single_task_cb**

A DOCA task can invoke a success or error callback. Both callbacks share the same structure (same input parameters).

DOCA recommends using 2 callbacks:

- Success callback – does not need to check the task status, thereby improving performance
• Error callback – may need to run a different flow than success callback

The sample demonstrates the following functions:

• How to use a single callback instead of two callbacks

The sample performs the following:

1. Runs 16 DMA tasks.

2. Handles completion with a single callback.

The difference between `pe_polling_sample.c` and `pe_single_task_comp_cb_sample.c` is to learn how to use PE with a single completion callback.

**pe_task_error**

Task execution may fail causing the associated context (e.g., DMA) to move to stopping state due to this fatal error.

The sample demonstrates the following functions:

• How to mitigate a task error during runtime

The user must make sure to:

• Review the state changed callback and the error callback to see how the sample mitigates context error

The sample performs the following:

1. Submits 255 tasks.

2. Allocates the second task with invalid parameters that cause hardware to fail.

3. Mitigates the failure and polls until all submitted tasks are flushed.

The difference between `pe_polling_sample.c` and `pe_task_error_sample.c` is to learn how to mitigate context error.
pe_task_resubmit

A task can be freed or reused after it is completed:

- Task resubmit can improve performance because the program does not free and allocate the task.
- Task resubmit can reduce memory usage (using a smaller task pool).
- Task members (e.g., source or destination buffer) can be set, so resubmission can be used if the source or destination are changed every iteration.

The sample demonstrates the following functions:

- How to re-submit a task in the completion callback
- How to replace buffers in a DMA task (similar to other task types)

The sample performs the following:

1. Allocates a set of 4 tasks and 16 buffer pairs.
2. Uses the tasks to copy all sources to destinations by resubmitting the tasks.

The difference between pe_polling_sample.c and pe_task_resubmit_sample.c is to learn how to use task resubmission.

pe_task_try_submit

doca_task_submit does not validate task inputs (to increase performance). Developers can use doca_task_try_submit to validate the tasks during development.

⚠️ Note
The sample demonstrates the following functions:

- How to use `doca_task_try_submit` instead of `doca_task_submit`

The sample performs the following:

1. Allocates and tries to submit tasks using `doca_task_try_submit`.

The difference between `pe_polling_sample.c` and `pe_task_try_submit_sample.c` is to learn how to use `doca_task_try_submit`.

### Graph Sample

The graph sample demonstrates how to use DOCA graph with PE. The sample can be used to learn how to build and use DOCA graph.

The sample uses two nodes of DOCA DMA and one user node.

The graph runs both DMA nodes (copying a source buffer to two destinations). Once both nodes are complete, the graph runs the user node that compares the buffers.

The sample runs 10 instances of the graph in parallel.

### Backward Compatibility of DOCA Core `doca_buf`

This section lists changes to the DOCA SDK which impacts backward compatibility.

**DOCA Core `doca_buf`**
Up to DOCA 2.0.2, the data length of the buffer is ignored when using the buffer as an output parameter, and the new data was written over the data that was there beforehand. From now on, new data is appended after existing data (if any) while updating the data length accordingly.

Because of this change, it is recommended that a destination buffer is allocated without a data section (data length 0), for ease of use.

In cases where the data length is 0 in a destination buffer, this change would go unnoticed (as appending the data and writing to the data section has the same result).

Reusing buffers requires resetting the data length when wishing to write to the same data address (instead of appending the data), overwriting the existing data. A new function, `doca_buf_reset_data_len()`, has been added specifically for this need.

## Sync Event

**Note**

DOCA Sync Event API is considered thread-unsafe

**Note**

DOCA Sync Event does not currently support GPU related features.

### Introduction

DOCA Sync Event (SE) is a software synchronization mechanism for parallel execution across the CPU, DPU, DPA and remote nodes. The SE holds a 64-bit counter which can be
updated, read, and waited upon from any of these units to achieve synchronization between executions on them.

To achieve the best performance, DOCA SE defines a subscriber and publisher locality, where:

- Publisher – the entity which updates (sets or increments) the event value
- Subscriber – the entity which gets and waits upon the SE

Both publisher and subscriber can read (get) the actual counter's value.

Based on hints, DOCA selects memory locality of the SE counter, closer to the subscriber side. Each DOCA SE is configured with a single publisher location and a single subscriber location which can be the CPU or DPU.

The SE control path happens on the CPU (either host CPU or DPU CPU) through the DOCA SE CPU handle. It is possible to retrieve different execution-unit-specific handles (DPU/DPA/GPU/remote handles) by exporting the SE instance through the CPU handle. Each SE handle refers to the DOCA SE instance from which it is retrieved. By using the execution-unit-specific handle, the associated SE instance can be operated from that execution unit.

In a basic scenario, synchronization is achieved by updating the SE from one execution and waiting upon the SE from another execution unit.

**Prerequisites**

DOCA SE can be used as a context which follows the architecture of a DOCA Core Context, it is recommended to read the following sections of the DOCA Core page before proceeding:

- DOCA Execution Model
Environment

DOCA SE based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target and can involve DPA, GPU and other remote nodes.

Using DOCA SE with DPU requires BlueField to be configured to work in DPU mode as described in NVIDIA BlueField Modes of Operation.

Info

Asynchronous wait on a DOCA SE requires NVIDIA® BlueField-3® or newer.

Architecture

DOCA SE can be converted to a DOCA Context as defined by DOCA Core. See DOCA Context for more information.

As a context, DOCA SE leverages DOCA Core architecture to expose asynchronous tasks/events offloaded to hardware.

The figure that follows demonstrates components used by DOCA SE. DOCA Device provides information on the capabilities of the configured HW used by SE to control system resources.

DOCA DPA, GPUNetIO, and RDMA modules are required for cross-device synchronization (could be DPA, GPU, or remote peer respectively).

DOCA SE allows flexible memory management by its ability to specify an external buffer, where a DOCA mmap module handles memory registration for advanced synchronization scenarios.

For asynchronous operation scheduling, SE uses the DOCA Progress Engine (PE) module.
DOCA Sync Event Components Diagram

The following diagram represents DOCA SE synchronization abilities on various devices.

DOCA Sync Event Interaction Diagram

DOCA Sync Event Objects
DOCA SE exposes different types of handles per execution unit as detailed in the following table.

<table>
<thead>
<tr>
<th>Execution Unit</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (host/DPU)</td>
<td>struct doca_sync_event</td>
<td>Handle for interacting with the SE from the CPU</td>
</tr>
<tr>
<td>DPU</td>
<td>struct doca_sync_event</td>
<td>Handle for interacting with the SE from the DPU</td>
</tr>
<tr>
<td>DPA</td>
<td>doca_dpa_dev_sync_event_t</td>
<td>Handle for interacting with the SE from the DPA</td>
</tr>
<tr>
<td>GPU</td>
<td>doca_gpu_dev_sync_event_t</td>
<td>Handle for interacting with the SE from the GPU</td>
</tr>
<tr>
<td>Remote net CPU</td>
<td>doca_sync_event_remote_net</td>
<td>Handle for interacting with the SE from a remote CPU</td>
</tr>
<tr>
<td>Remote net DPA</td>
<td>doca_dpa_dev_sync_event_remote_net_t</td>
<td>Handle for interacting with the SE from a remote DPA</td>
</tr>
<tr>
<td>Remote net GPU</td>
<td>doca_gpu_dev_sync_event_remote_net_t</td>
<td>Handle for interacting with the SE from a remote GPU</td>
</tr>
</tbody>
</table>

Each one of these handle types has its own dedicated API for creating the handle and interacting with it.

**Configuration Phase**

Any DOCA SE creation starts with creating CPU handle by calling `doca_sync_event_create` API.

After creation, the SE entity could be shared with local PCIe, remote CPU, DPA, and GPU by a dedicated handle creation via the DOCA SE export flow, as illustrated in the following diagram:
Operation Modes

DOCA SE exposes two different APIs for starting it depending on the desired operation mode, synchronous or asynchronous.

Note
Once started, SE operation mode cannot be changed.

Synchronous Mode

Start the SE to operate in synchronous mode by calling `doca_sync_event_start`.

In synchronous operation mode, each data path operation (get, update, wait) blocks the calling thread from continuing until the operation is done.

Note
An operation is considered done if the requested change fails and the exact error can be reported or if the requested change has taken effect.
Asynchronous Mode

To start the SE to operate in asynchronous mode, convert the SE instance to `doca_ctx` by calling `doca_sync_event_as_ctx`. Then use DOCA CTX API to start the SE and DOCA PE API to submit tasks on the SE (see section "DOCA Progress Engine" for more).

Configurations

Mandatory Configurations

These configurations must be set by the application before attempting to start the SE:

- DOCA SE CPU handle must be configured by providing the runtime hints on the publisher and subscriber locations. Both the subscriber and publisher locations must be configured using the following APIs:
  
  - `doca_sync_event_add_publisher_location_<cpu|dpa|gpu|remote_pci|remote_net>`
  
  - `doca_sync_event_add_subscriber_location_<cpu|dpa|gpu|remote_pci>`

- For the asynchronous use case, at least one task/event type must be configured. See configuration of tasks.

Optional Configurations

- If these configurations are not set, a default value is used.

- These configurations provide an 8-byte buffer to be used as the backing memory of the SE. If set, it is user responsibility to handle the memory (i.e., preserve the
memory allocated during DOCA SE lifecycle and free it after DOCA SE destruction). If not provided, the SE backing memory is allocated by the SE.

- doca_sync_event_set_addr
- doca_sync_event_set_doca_buf

**Export DOCA Sync Event to Another Execution Unit**

To use an SE from an execution unit other than the CPU, it must be exported to get a handle for the specific execution unit:

- **DPA** – `doca_sync_event_get_dpa_handle` returns a DOCA SE DPA handle (`doca_dpa_dev_sync_event_t`) which can be passed to the DPA SE data path APIs from the DPA kernel

- **GPU** – `doca_sync_event_get_gpu_handle` returns a DOCA SE GPU handle (`doca_gpu_dev_sync_event_t`) which can be passed to the GPU SE data path APIs for the CUDA kernel

- **DPU** – `doca_sync_event_export_to_remote_pci` returns a blob which can be used from the DPU CPU to instantiate a DOCA SE DPU handle (`struct doca_sync_event`) using the `doca_sync_event_create_from_export` function

DOCA SE allows notifications from remote peers (remote net) utilizing capabilities of the DOCA RDMA library. The following figure illustrates the remote net export flow:
- Remote net CPU – `doca_sync_event_export_to_remote_net` returns a blob which can be used from the remote net CPU to instantiate a DOCA SE remote net CPU handle (`struct doca_sync_event_remote_net`) using the `doca_sync_event_remote_net_create_from_export` function. The handle can be used directly for submitting asynchronous tasks through the `doca_rdma` library or exported to the remote DPA/GPU.

- Remote net DPA – `doca_sync_event_remote_net_get_dpa_handle` returns a DOCA SE remote net DPA handle (`doca_dpa_dev_sync_event_remote_net_t`) which can be passed to the DPA RDMA data path APIs from a DPA kernel.

- Remote net GPU – `doca_sync_event_remote_net_get_gpu_handle` returns a DOCA SE remote net GPU handle (`doca_gpu_dev_sync_event_remote_net_t`) which can be passed to the GPU RDMA data path APIs from a CUDA kernel.

⚠️ **Note**

The CPU handle (`struct doca_sync_event`) can be exported only to the location where the SE is configured.

⚠️ **Note**

Prior to calling any export function, users must first verify it is supported by calling the corresponding export capability getter:

- `doca_sync_event_cap_is_export_to_dpa_supported`
- `doca_sync_event_cap_is_export_to_gpu_supported`
- `doca_sync_event_cap_is_export_to_remote_pci_supported`, or
- `doca_sync_event_cap_is_export_to_remote_net_supported`.

⚠️ **Note**

Prior to calling any `_create_from_export` function, users must first verify it is supported by calling the corresponding create from the export.
**capability getter**: 
doca_sync_event_cap_is_create_from_export_supported OR
doca_sync_event_cap_remote_net_is_create_from_export_supported.

---

**Note**

Once created from an export, both the SE DPU handle `struct
doca_sync_event` and the SE remote net CPU handle `struct
doca_sync_event_remote_net` cannot be configured, but only the SE DPU
handle must be started before it is used.

---

**Warning**

Data exported in `doca_sync_event_export_to_*` functions contains sensitive
information. Make sure to pass this data through a secure channel!

---

**Device Support**

DOCA SE needs a device to operate. For instructions on picking a device, see DOCA Core
device discovery.

---

**Info**

Both NVIDIA® BlueField ® -2 and BlueField ® -3 devices are
supported as well as any `doca_dev` is supported.
As device capabilities may change in the future (see DOCA Capability Check), it is recommended to choose your device using any relevant capability method (starting with the prefix `doca_sync_event_cap_*`).

Capability APIs to query whether sync event can be constructed from export blob:

- `doca_sync_event_cap_is_create_from_export_supported`
- `doca_sync_event_cap_remote_net_is_create_from_export_supported`

Capability APIs to query whether sync event can be exported to other execution units:

- `doca_sync_event_cap_is_export_to_remote_pci_supported`
- `doca_sync_event_cap_is_export_to_dpa_supported`
- `doca_sync_event_cap_is_export_to_gpu_supported`
- `doca_sync_event_cap_is_export_to_remote_net_supported`
- `doca_sync_event_cap_remote_net_is_export_to_dpa_supported`
- `doca_sync_event_cap_remote_net_is_export_to_gpu_supported`

Capability APIs to query whether an asynchronous task is supported:

- `doca_sync_event_cap_task_get_is_supported`
- `doca_sync_event_cap_task_notify_set_is_supported`
- `doca_sync_event_cap_task_notify_add_is_supported`
- `doca_sync_event_cap_task_wait_eq_is_supported`

Info

Asynchronous wait (blocking/polling) is supported on NVIDIA® BlueField® -3 and NVIDIA® ConnectX®-7 and later.
DOCA Sync Event Data Path Operations

The DOCA SE synchronization mechanism is achieved using exposed datapath operations. The API exposes a function for "writing" to the SE and for "reading" the SE.

The synchronous API is a set of functions which can be called directly by the user, while the asynchronous API is exposed by defining a corresponding `doca_task` for each synchronous function to be submitted on a DOCA PE (see DOCA Progress Engine and DOCA Context for additional information).

**Info**

Remote net CPU handle (`struct doca_sync_event_remote_net`) can be used for submitting asynchronous tasks using the DOCA RDMA library.

**Note**

Prior to asynchronous task submission, users must check if the job is supported using `doca_error_t doca_sync_event_cap_task_<task_type>_is_supported`.

The following subsections describe the DOCA SE datapath operation with respect to synchronous and asynchronous operation modes.

- `doca_sync_event_cap_task_wait_neq_is_supported`
Publishing on DOCA Sync Event

Setting DOCA Sync Event Value

Users can set DOCA SE to a 64-bit value:

- Synchronously by calling `doca_sync_event_update_set`
- Asynchronously by submitting a `doca_sync_event_task_notify_set` task

Adding to DOCA Sync Event Value

Users can atomically increment the value of a DOCA SE:

- Synchronously by calling `doca_sync_event_update_add`
- Asynchronously by submitting a `doca_sync_event_task_notify_add` task

Subscribe on DOCA Sync Event

Getting DOCA Sync Event Value

Users can get the value of a DOCA SE:

- Synchronously by calling `doca_sync_event_get`
- Asynchronously by submitting a `doca_sync_event_task_get` task

Waiting on DOCA Sync Event

Waiting for an event is the main operation for achieving synchronization between different execution units.

Users can wait until an SE reaches a specific value in a variety of ways.
Synchronously

doca_sync_event_wait_gt waits for the value of a DOCA SE to be greater than a specified value in a "polling busy wait" manner (100% processor utilization). This API enables users to wait for an SE in real time.

doca_sync_event_wait_gt_yield waits for the value of a DOCA SE to be greater than a specified value in a "periodically busy wait" manner. After each polling iteration, the calling thread relinquishes the CPU, so a new thread gets to run. This API allows a tradeoff between real-time polling to CPU starvation.

doca_sync_event_wait_eq waits for the value of a DOCA SE to be equal to a specified value in a "polling busy wait" manner (100% processor utilization). This API enables users to wait for an SE in real time.

doca_sync_event_wait_eq_yield waits for the value of a DOCA SE to be equal to a specified value in a "periodically busy wait" manner. After each polling iteration, the calling thread relinquishes the CPU so a new thread gets to run. This API allows a tradeoff between real-time polling to CPU starvation.

doca_sync_event_wait_neq waits for the value of a DOCA SE to not be equal to a specified value in a "polling busy wait" manner (100% processor utilization). This API enables users to wait for an SE in real time.

doca_sync_event_wait_neq_yield waits for the value of a DOCA SE to not be equal to a specified value in a "periodically busy wait" manner. After each polling iteration, the calling thread relinquishes the CPU so a new thread gets to run. This API allows a tradeoff between real-time polling to CPU starvation.

Note

This wait method is supported only from the CPU.

Asynchronously
DOCA SE exposes an asynchronous wait method by defining a `doca_sync_event_task_wait_eq` and `doca_sync_event_task_wait_neq` tasks.

Users can wait for wait-job completion in the following methods:

- **Blocking** – get a `doca_event_handle_t` from the `doca_pe` to blocking-wait on
- **Polling** – poll the wait task by calling `doca_pe_progress`

**Info**

Asynchronous wait (blocking/polling) is supported on BlueField-3 and ConnectX-7 and later.

**Note**

Users may leverage the `doca_sync_event_task_get` job to implement asynchronous wait by asynchronously submitting the task on a DOCA PE and comparing the result to some threshold.

**Tasks**

DOCA SE context exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See [DOCA Core Task](#).

**Get Task**

The get task retrieves the value of a DOCA SE.
Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_sync_event_task_get_set_conf</td>
<td>doca_sync_event_cap_task_get_is_supported</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_sync_event_task_get_set_conf</td>
<td>-</td>
</tr>
</tbody>
</table>

Task Input

Common input described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return value</td>
<td>8-bytes memory pointer to hold the DOCA SE value</td>
</tr>
</tbody>
</table>

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the return value memory holds the DOCA SE value.

Task Completion Failure

If the task fails midway:

- The context may enter a stopping state if a fatal error occurs
- The return value memory may be modified
Task Limitations

All limitations are described in DOCA Core Task.

Notify Set Task

The notify set task allows setting the value of a DOCA SE.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_sync_event_task_notify_set_set_conf</td>
<td>doca_sync_event_cap_task_notify_set_is_supported</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_sync_event_task_notify_set_set_conf</td>
<td>-</td>
</tr>
</tbody>
</table>

Task Input

Common input described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set value</td>
<td>64-bit value to set the DOCA SE value to</td>
</tr>
</tbody>
</table>

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the DOCA SE value is set to the given set value.
Task Completion Failure

If the task fails midway, the context may enter a stopping state if a fatal error occurs.

Task Limitations

This operation is not atomic. Other limitations are described in DOCA Core Task.

Notify Add Task

The notify add task allows atomically setting the value of a DOCA SE.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_sync_event_task_notify_add_set_conf</td>
<td>doca_sync_event_cap_task_notify_add_is_supported</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_sync_event_task_notify_add_set_conf</td>
<td>-</td>
</tr>
</tbody>
</table>

Task Input

Common input described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment value</td>
<td>64-bit value to atomically increment the DOCA SE value by</td>
</tr>
<tr>
<td>Fetched value</td>
<td>8-bytes memory pointer to hold the DOCA SE value before the increment</td>
</tr>
</tbody>
</table>
**Task Output**

Common output described in [DOCA Core Task](#).

**Task Completion Success**

After the task is completed successfully, the following occurs:

- The DOCA SE value is incremented according to the given increment value
- The fetched value memory holds the DOCA SE value before the increment

**Task Completion Failure**

If the task fails midway:

- The context may enter a stopping state if a fatal error occurs
- The fetched value memory may be modified.

**Task Limitations**

All limitations are described in [DOCA Core Task](#).

**Wait Equal-to Task**

The wait-equal task allows atomically waiting for a DOCA SE value to be equal to some threshold.

**Task Configuration**
<table>
<thead>
<tr>
<th>Description</th>
<th>API to set the configuration</th>
<th>API to query support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_sync_event_task_wait_eq_set_conf</code></td>
<td><code>doca_sync_event_cap_task_wait_eq_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_sync_event_task_wait_eq_set_conf</code></td>
<td>-</td>
</tr>
</tbody>
</table>

**Task Input**

Common input described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait threshold</td>
<td>64-bit value to wait for the DOCA SE value to be equal to</td>
</tr>
<tr>
<td>Mask</td>
<td>64-bit mask to apply on the DOCA SE value before comparing with the wait threshold</td>
</tr>
</tbody>
</table>

**Task Output**

Common output described in DOCA Core Task.

**Task Completion Success**

After the task is completed successfully, the following occurs:

- The DOCA SE value is equal to the given wait threshold.

**Task Completion Failure**

If the task fails midway, the context may enter a stopping state if a fatal error occurs.
Task Limitations

Other limitations are described in DOCA Core Task.

Wait Not-equal-to Task

The wait-not-equal task allows atomically waiting for a DOCA SE value to not be equal to some threshold.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to set the configuration</th>
<th>API to query support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_sync_event_task_wait_neq_set_conf</td>
<td>doca_sync_event_cap_task_wait_neq_is_supported</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_sync_event_task_wait_neq_set_conf</td>
<td>-</td>
</tr>
</tbody>
</table>

Task Input

Common input described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait threshold</td>
<td>64-bit value to wait for the DOCA SE value to be not equal to</td>
</tr>
<tr>
<td>Mask</td>
<td>64-bit mask to apply on the DOCA SE value before comparing with the wait threshold</td>
</tr>
</tbody>
</table>

Task Output

Common output described in DOCA Core Task.
Task Completion Success

After the task is completed successfully, the following occurs:

- The DOCA SE value is not equal to the given wait threshold.

Task Completion Failure

If the task fails midway, the context may enter a stopping state if a fatal error occurs.

Task Limitations

Limitations are described in DOCA Core Task.

Events

DOCA SE context exposes asynchronous events to notify about changes that happen unexpectedly, according to the DOCA Core architecture.

The only event DOCA SE context exposes is common events as described in DOCA Core Event.

State Machine

The DOCA SE context follows the Context state machine as described in DOCA Core Context State Machine.

The following subsection describe how to move to specific states and what is allowed in each state.
Idle

In this state, it is expected that the application will:

- Destroy the context; or
- Start the context

Allowed operations in this state:

- Configure the context according to section "Configurations"
- Start the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all tasks have been freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all tasks are completed and then freed</td>
</tr>
</tbody>
</table>

Starting

This state cannot be reached.

Running

In this state, it is expected that the application will:

- Allocate and submit tasks
- Call progress to complete tasks and/or receive events

Allowed operations in this state:
- Allocate previously configured task
- Submit an allocated task
- Call stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

**Stopping**

In this state, it is expected that the application will:

- Call progress to complete all inflight tasks (tasks will complete with failure)
- Free any completed tasks

Allowed operations in this state:

- Call progress

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Call progress and fatal error occurs</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop without freeing all tasks</td>
</tr>
</tbody>
</table>

**DOCA Sync Event Tear Down**

Multiple SE handles (for different execution units) associated with the same DOCA SE instance can live simultaneously, though the teardown flow is performed only from the CPU on the CPU handle.
Stopping DOCA Sync Event

To stop a DOCA SE:

- Synchronous – call `doca_sync_event_stop` on the CPU handle
- Asynchronous – stop the DOCA context associated with the DOCA SE instance

Note

Users must validate active handles associated with the CPU handle during the teardown flow because DOCA SE does not do that.

Note

Stopping a DOCA SE must be followed by destruction. Refer to section "Destroying DOCA Sync Event" for details.

Destroying DOCA Sync Event

Once stopped, a DOCA SE instance can be destroyed by calling `doca_sync_event_destroy` on the CPU handle.

Remote net CPU handle instance terminates and frees by calling `doca_sync_event_remote_net_destroy` on the remote net CPU handle.

Upon destruction, all the internal resources are released, allocated memory is freed, associated `doca_ctx` (if it exists) is destroyed, and any associated exported handles (other than CPU handles) and their resources are destroyed.
Alternative Datapath Options

DOCA SE supports datapath on CPU (see section "Execution Phase") and also on DPA and GPU.

GPU Datapath

DOCA SE does not currently support GPU related features.

DPA Datapath

Info

An SE with DPA-subscriber configuration currently supports synchronous APIs only.

Once a DOCA SE DPA handle (doca_dpa_dev_sync_event_t) has been retrieved it can be used within a DOCA DPA kernel as described in DOCA DPA Sync Event.

DOCA Sync Event Sample

This section provides DOCA SE sample implementation on top of the BlueField DPU.

The sample demonstrates how to share an SE between the host and the DPU while simultaneously interacting with the event from both the host and DPU sides using different handles.

Running DOCA Sync Event Sample

1. Refer to the following documents:
- NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

- NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_common/sync_event_<local|remote>_pci
   meson /tmp/build
   ninja -C /tmp/build
   ```

   **Note**

   The binary `doca_sync_event_<local|remote>_pci` is created under `/tmp/build/`.

3. Sample usage:

   **Usage:** `doca_sync_event_remote_pci [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=CRTICAL, 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=CRTICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>`
   - `-j`, `--json <path>` Parse all command flags from an input json file

   **Program Flags:**
   - `-d`, `--pci-addr` Device PCI address
   - `-r`, `--rep-pci-addr` DPU representor PCI address
   - `--async` Start DOCA Sync Event in asynchronous mode (synchronous mode by default)
4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_sync_event_<local|remote>_pci -h
```

**Samples**

**Sync Event Remote PCIe**

**Note**

This sample should be run (on the DPU or on the host) before Sync Event Local PCIe.

This sample demonstrates creating an SE from an export which is associated with an SE on a local PCIe (host or the DPU) and interacting with the SE to achieve synchronization between the host and DPU.

The sample logic includes:

1. Reading configuration files and saving their content into local buffers.
2. Locating and opening DOCA devices and DOCA representors (if running on the DPU) matching the given PCIe addresses.

3. Initializing DOCA Comm Channel.

4. Receiving SE blob through Comm Channel.

5. Creating SE from export.

6. Starting the above SE in the requested operation mode (synchronous or asynchronous).

7. Interacting with the SE:
   1. Waiting for signal from the host – synchronously or asynchronously (with busy wait polling) according to user input.
   2. Signaling the SE for the host – synchronously or asynchronously, using set or atomic add, according to user input.

8. Cleaning all resources.

Reference:

- /opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/sync_event_remote_pci_sample.c
- /opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/sync_event_remote_pci_main.c
- /opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/meson.build

**Sync Event Local PCIe**

ℹ️ **Note**

This sample should run (on the DPU or on the Host) only after **Sync Event Remote PCIe** has been started.
This sample demonstrates how to initialize a SE to be shared with a remote PCIe (host or the DPU) how to export it to a remote PCIe, and how to interact with the SE to achieve synchronization between the host and DPU.

The sample logic includes:

1. Reading configuration files and saving their content into local buffers.
2. Locating and opening DOCA devices and DOCA representors (if running on the DPU) matching the given PCIe addresses.
3. Creating and configuring the SE to be shared with a remote PCIe.
4. Starting the above SE in the requested operation mode (synchronous or asynchronous).
5. Initializing DOCA Comm Channel.
6. Exporting the SE and sending it through the Comm Channel.
7. Interacting with the SE:
   1. Signaling the SE for the remote PCIe – synchronously or asynchronously, using set or atomic add, according to user input.
   2. Waiting for a signal – synchronously or asynchronously, with busy wait polling, according to user input.
8. Cleaning all resources.

Reference:
- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/sync_event_local_pci_sample.c`
- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/sync_event_local_pci_main.c`
- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/meson.build`
Mmap Advise

Introduction

DOCA Mmap Advise is used to give advanced memory-related instructions to NVIDIA® BlueField® DPUs in order to improve system or application performance.

Note

To use DOCA Mmap Advise with BlueField, the device must be configured to work in DPU mode as described in NVIDIA BlueField Modes of Operation.

The operations in the instructions are meant to influence the performance of the application, but not its semantics. The operations allow an application to inform the NIC how it expects it to use some mapped memory areas, so the BlueField's hardware can choose appropriate optimization techniques.

Prerequisites

DOCA Mmap Advise is a context and follows the architecture of a DOCA Core Context, it is recommended to read the following sections of the DOCA Core page before proceeding:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

Architecture

DOCA Mmap Advise is a DOCA Context as defined by DOCA Core. See DOCA Core Context for more information.

DOCA Mmap Advise currently supports the following list of advised operations:
• Cache Invalidate Operation

**Cache Invalidate Operation**

When data is processed by BlueField's cores it may be temporarily stored in the cores’ system-level cache (i.e., L3 cache). When a cache line is occupied and new data must be written to it, the cache management sub-system evicts the existing data, usually based on LRU policy, by performing a write-back operation to store this data in the main (DDR) memory. When this data is not required to be stored in the BlueField's memory (e.g., it is host data and is no longer needed after it is copied to the host's memory), the cache’s write-back operation wastes memory bandwidth that reduces overall system performance, which is undesirable. The simplest to avoid this write-back operation is to mark the appropriate cache lines as "invalid". This enables their immediate reuse, without additional operations.

The cache invalidate operation facilitates invalidating a set of cache lines.

**Environment**

Applications based on DOCA Mmap Advise can run on the BlueField target.

**Objects**

**Device and Device Representor**

The MMAP Advise context requires a DOCA Device to operate. The device is used to access memory and perform the copy operation. See [DOCA Core Device Discovery](#).

**Info**

For the same DPU, it does not matter which device is used (i.e., PF, VF, SF) as all these devices utilize the same hardware components.
Memory Buffers

The cache invalidate task requires one DOCA Buffer containing the address space to invalidate depending on the allocation pattern of the buffers (refer to the table in section "Inventory Types"). To find what kind of memory is supported, refer to the table in section "Buffer Support".

Buffers must not be modified or read during the cache invalidate operation.

Configuration Phase

To start using the context, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

Configurations

The context can be configured to match the application's use case.

To find if a configuration is supported, or what the min/max value for it is, refer to section "Device Support".

Mandatory Configurations

Note

The device must stay valid for as long as the MMAP Advise instance is not destroyed.
These configurations are mandatory and must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of tasks and/or events in sections "Tasks" and "Events" respectively for information.
- A device with appropriate support must be provided upon creation

**Device Support**

DOCA Mmap Advise requires a device to operate. To pick a device, refer to DOCA Core Device Discovery.

As device capabilities may change (see DOCA Core Device Support), it is recommended to select your device using the following method:

- `doca_mmap_advise_cap_task_cache_invalidate_is_supported`

Some devices expose different capabilities as follows:

- Maximum cache invalidate buffer size may differ.

**Buffer Support**

Tasks support buffers with the following features:

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local mmap buffer</td>
<td>Yes</td>
</tr>
<tr>
<td>MMAP from PCIe export buffer</td>
<td>No</td>
</tr>
<tr>
<td>MMAP from RDMA export buffer</td>
<td>No</td>
</tr>
<tr>
<td>Linked list buffer</td>
<td>No</td>
</tr>
</tbody>
</table>
Execution Phase

This section describes execution on the CPU using DOCA Core Progress Engine.

Tasks

DOCA Mmap Advise exposes asynchronous tasks that leverage DPU hardware according to the DOCA Core architecture. See DOCA Core Task for information.

Cache Invalidate Task

The cache invalidate task facilitates invalidating a set of cache lines, preventing them from being written back to the RAM (thus increasing performance).

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_mmap_advise_task_invalidate_cache_set_conf</td>
<td>doca_mmap_advise_cap_task_cache_invalidate_is_supported</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_mmap_advise_task_invalidate_cache_set_conf</td>
<td>–</td>
</tr>
<tr>
<td>Maximal buffer</td>
<td>–</td>
<td>doca_mmap_advise_task_cache_invalidate_get_max_buf_size</td>
</tr>
<tr>
<td>size</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Maximal buffer</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>list size</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>Buffer that points to the memory to be invalidated</td>
</tr>
</tbody>
</table>

Task Output
Common output as described in DOCA Core Task.

**Task Completion Success**

After the task is completed successfully:

- The cache is invalidated

**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state, if a fatal error occurs
- The cache is not invalidated

**Task Limitations**

- The operation is not atomic
- Once the task has been submitted, the buffer should not be read/written to
- Other limitations are described in DOCA Core Task

**Events**

DOCA Mmap Advise exposes asynchronous events to notify on changes that happen unexpectedly, according to DOCA Core architecture.

The only events DOCA Mmap Advise exposes are common events as described in DOCA Core Event.
State Machine

DOCA Mmap Advise context follows the context state machine as described in DOCA Core Context State Machine.

The following section describes how to move states and what is allowed in each state.

Idle

In this state it is expected that the application:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all tasks have been freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all tasks are completed and freed</td>
</tr>
</tbody>
</table>

Starting

This state cannot be reached.
Running

In this state, it is expected that the application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating a previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

Stopping

In this state it is expected that the application:

- Calls progress to complete all in-flight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Call progress

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Call progress and fatal error occurs</td>
</tr>
<tr>
<td>Previous State</td>
<td>Transition Action</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop without freeing all tasks</td>
</tr>
</tbody>
</table>

## Alternative Datapath Options

DOCA Mmap Advise only supports datapath on the CPU. See section "Execution Phase".

## Samples

### Cache Invalidate Sample

The sample illustrates how to invalidate the cache for a memory range after copying it using DOCA DMA.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Populating DOCA memory map with two relevant buffers.
4. Allocating element in DOCA buffer inventory for each buffer.
5. Initializing DOCA DMA memory copy task object.
6. Initializing DOCA Mmap Advise cache invalidate task object
7. Submitting DMA task.
8. Polling for completion:
   1. Handling DMA task completion – submitting the cache invalidate task in the DMA task completion callback body.
   2. Handling cache invalidate task completion.
10. Destroying DMA, DOCA MMAP Advise, and DOCA Core objects.

Reference:

- /opt/mellanox/doca/samples/doca_common/cache_invalidate/cache_invalidate_sample.c
- /opt/mellanox/doca/samples/doca_common/cache_invalidate/cache_invalidate_main.c
- /opt/mellanox/doca/samples/doca_common/cache_invalidate/meson.build

DOCA Log

DOCA logging infrastructure allows printing DOCA SDK library error messages, and printing debug and error messages from applications.

To work with the DOCA logging mechanism, the header file `doca_log.h` must be included in every source code using it.

Log Verbosity Level Enumerations

The following verbosity levels are supported by the DOCA logging:

```c
enum doca_log_level {
    DOCA_LOG_LEVEL_DISABLE = 10, /**< Disable log messages */
    DOCA_LOG_LEVEL_CRIT = 20, /**< Critical log level */
    DOCA_LOG_LEVEL_ERROR = 30, /**< Error log level */
    DOCA_LOG_LEVEL_WARNING = 40, /**< Warning log level */
    DOCA_LOG_LEVEL_INFO = 50, /**< Info log level */
    DOCA_LOG_LEVEL_DEBUG = 60, /**< Debug log level */
    DOCA_LOG_LEVEL_TRACE = 70, /**< Trace log level */
};
```

ℹ️ Note
The DOCA_LOG_LEVEL_TRACE verbosity level is available only if the macro
DOCA_LOGGING_ALLOW_TRACE is set before the compilation.

See doca_log.h for more information.

Logging Backends

DOCA's logging backend is the target to which log messages are directed.

The following backend types are supported:

- FILE * – file stream which can be any open file or stdout/stderr
- file descriptor – any file descriptor that the system supports, including (but not limited to) raw files, sockets, and pipes
- buf – memory buffer (address and size) that can hold a single message and a callback to be called for every logged message
- syslog – system standard logging

Every logger is created with the following default lower and upper verbosity levels:

- Lower level – DOCA_LOG_LEVEL_INFO
- Upper level – DOCA_LOG_LEVEL_CRIT

SDK and application logging have different default configuration values and can be controlled separately using the appropriate API.

Every message is printed to every created backend if its verbosity level allows it.

Enabling DOCA SDK Libraries Logging

The DOCA SDK libraries print debug and error messages to all the backends created using the following functions:

- doca_log_backend_create_with_file_sdk()
- `doca_log_backend_create_with_fd_sdk()`
- `doca_log_backend_create_with_buf_sdk()`
- `doca_log_backend_create_with_syslog_sdk()`

A newly created SDK backend verbosity level is set to the SDK global verbosity level value. This value can be changed using `doca_log_level_set_global_sdk_limit()`.

`doca_log_level_set_global_sdk_limit()` sets the verbosity level for all existing SDK backends and sets the SDK global verbosity level.

`doca_log_backend_set_sdk_level()` sets the verbosity level of a specific SDK backend.

`doca_log_level_get_global_sdk_limit()` gets the SDK global verbosity level.

---

Note

Messages may change between different versions of DOCA. Users cannot rely on message permanence or formatting.

---

Enabling DOCA Application Logging

Any source code that uses DOCA can use DOCA logging infrastructure.

Every debug and error messages is printed to all backends created using the following functions:

- `doca_log_backend_create_with_file()`
- `doca_log_backend_create_with_fd()`
- `doca_log_backend_create_with_buf()`
- `doca_log_backend_create_with_syslog()`

The lower and upper levels of a newly created backend are set to the default values. Those values can be changed using `doca_log_backend_set_level_lower_limit()` and
DOCA log backend set level upper limit().

DOCA log backend create standard() creates a default non-configurable set of two backends:

- stdout prints the range from a global minimum level up to DOCA_LOG_LEVEL_INFO
- stderr prints the range from DOCA_LOG_LEVEL_WARNING level up to DOCA_LOG_LEVEL_CRIT

DOCA log backend set level lower limit strict() marks the lower log level limit of a backend as strict, preventing it from being lowered by future log level changes. It is both global and direct.

DOCA log backend set level upper limit strict() marks the upper log level limit of a backend as strict, preventing it from being raised by future log level changes. It is both global and direct.

DOCA log level set global lower limit() sets the lower limit for all existing backends not marked as strict and sets the global application lower limit.

DOCA log level set global upper limit() sets the upper limit for all existing backends not marked as strict and sets the global application upper limit.

**Logging DOCA Application Messages**

To use the DOCA logging infrastructure with your source code to log its messages, users must call, at the beginning of the file, the macro DOCA_LOG_REGISTER(source) just before using the DOCA logging functionality. This macro handles the registration and the teardown from the DOCA logging.

Printing a message can be done by calling one of the following macros (with the same usage as printf()):

- DOCA_LOG_CRIT(format, ...)
- DOCA_LOG_ERR(format, ...)
- DOCA_LOG_WARN(format, ...)
- DOCA_LOG_INFO(format, ...)
- DOCA_LOG_DBG(format, ...)
• DOCA_LOG_TRC(format, ...)

The message is printed to all the application's backends with configured lower and upper logging limits.
DOCA Flow

This guide describes how to deploy the DOCA Flow library, the philosophy of the DOCA Flow API, and how to use it. The guide is intended for developers writing network function applications that focus on packet processing (such as gateways). It assumes familiarity with the network stack and DPDK.

Introduction

DOCA Flow is the most fundamental API for building generic packet processing pipes in hardware. The DOCA Flow library provides an API for building a set of pipes, where each pipe consists of match criteria, monitoring, and a set of actions. Pipes can be chained so that after a pipe-defined action is executed, the packet may proceed to another pipe.

Using DOCA Flow API, it is easy to develop hardware-accelerated applications that have a match on up to two layers of packets (tunneled).

- MAC/VLAN/ETHERTYPE
- IPv4/IPv6
- TCP/UDP/ICMP
- GRE/VXLAN/GTP-U/ESP/PSP
- Metadata

The execution pipe can include packet modification actions such as the following:

- Modify MAC address
- Modify IP address
- Modify L4 (ports)
- Strip tunnel
- Add tunnel
- Set metadata
- Encrypt/Decrypt

The execution pipe can also have monitoring actions such as the following:

- Count
- Policers

The pipe also has a forwarding target which can be any of the following:

- Software (RSS to subset of queues)
- Port
- Another pipe
- Drop packets

**Prerequisites**

A DOCA Flow-based application can run either on the host machine or on an NVIDIA® BlueField® DPU target. Flow-based programs require an allocation of huge pages, hence the following commands are required:

```bash
echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
sudo mkdir /mnt/huge
sudo mount -t hugetlbfs nodev /mnt/huge
```

**Note**

On some operating systems (RockyLinux, OpenEuler, CentOS 8.2) the default huge page size on the DPU (and Arm hosts) is larger than 2MB, often 512MB. Users can check the size of the huge pages on their OS using the following command:
Architecture

The following diagram shows how the DOCA Flow library defines a pipe template, receives a packet for processing, creates the pipe entry, and offloads the flow rule in NIC hardware.
Features of DOCA Flow:

- User-defined set of matches parser and actions
- DOCA Flow pipes can be created or destroyed dynamically
- Packet processing is fully accelerated by hardware with a specific entry in a flow pipe
- Packets that do not match any of the pipe entries in hardware can be sent to Arm cores for exception handling and then reinjected back to hardware

The DOCA Flow pipe consists of the following components:

- Monitor (MON in the diagram) - counts, meters, or mirrors
- Modify (MDF in the diagram) - modifies a field
- Forward (FWD in the diagram) - forwards to the next stage in packet processing

Steering Domains
DOCA Flow organizes pipes into high-level containers named domains to address the specific needs of the underlying architecture.

A key element in defining a domain is the packet direction and a set of allowed actions.

- A domain is a pipe attribute (also relates to shared objects)
- A domain restricts the set of allowed actions
- Transition between domains is well-defined (packets cannot cross domains arbitrarily)
- A domain may restrict the sharing of objects between packet directions
- Packet direction can restrict the move between domains

**List of Steering Domains**

DOCA Flow provides the following set of predefined steering domains:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
</table>
| DOCA_FLOW_PIPE_DOMAIN_DEFAULT | • Default domain for actions on ingress traffic  
• Encapsulated and secure actions are not allowed here  
• The next milestone is queue or pipe in the EGRESS domain  
• Miss action is: Drop |
| DOCA_FLOW_PIPE_DOMAIN_SECURE_INGRESS | • For secure actions on ingress traffic  
• Encapsulation and encrypting actions not allowed here  
• The only allowed domain for decrypting secure actions  
• The next milestone is queue or pipe in the DEFAULT or EGRESS domain  
• Only meta register is preserved  
• Miss action is: Drop  
• Memory may be optimized if set with DOCA_FLOW_DIRECTION_NETWORK_TO_HOST direction information |
| DOCA_FLOW_PIPE_DOMAIN_EGRESS | • Domain for actions on egress traffic  
• Decapsulation and secure actions are not allowed here |
<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
</table>
| ____                                                                 | - The next milestone is wire/representor or pipe in **SECURE_EGRESS** domain  
| ____                                                                 | - Miss action is: Send to wire/representor                                                                                                     |
| DOCA_FLOW_PIPE_DOMAIN_SECURE_EGRESS                           | - Domain for secure actions on egress traffic  
| ____                                                                 | - Decapsulation actions are not allowed here  
| ____                                                                 | - The only allowed domain for encrypting secure action  
| ____                                                                 | - The next milestone is wire/representor  
| ____                                                                 | - Miss action is: Send to wire/representor  
| ____                                                                 | - Memory may be optimized if set with **DOCA_FLOW_DIRECTION_HOST_TO_NETWORK** direction information                                      |

**Domains in VNF Mode**
Domains in Switch Mode
API

DOCA API is available through the NVIDIA DOCA Library APIs page.

Info

The pkg-config (*.pc file) for the DOCA Flow library is doca-flow.

Flow Life Cycle
Initialization Flow

Before using any DOCA Flow function, it is mandatory to call DOCA Flow initialization, `doca_flow_init()`, which initializes all resources required by DOCA Flow.

Pipe Mode

This mode (`mode_args`) defines the basic traffic in DOCA. It creates some miss rules when a DOCA port initializes. Currently, DOCA supports 3 modes:

- **vfn**

  A packet arriving from one of the device's ports is processed, and can be sent to another port. By default, missed packets go to RSS.

  The following diagram shows the basic traffic flow in vfn mode. Packet1 firstly misses and is forwarded to host RSS. The app captures this packet and decides how to process it and then creates a pipe entry. Packet2 will hit this pipe entry and do the action, for example, for VXLAN, will do decap, modify, and encap, then is sent out from P1.

- **switch**

  Used for internal switching, only representor ports are allowed, for example, uplink representors and SF/VF representors. Packet is forwarded from one port to another. If a packet arrives from an uplink and does not hit the rules defined by the user's pipe, then the packet is received on all RSS queues of the representor of the uplink.
The following diagram shows the basic flow of traffic in switch mode. Packet1 firstly misses to host RSS queues. The app captures this packet and decides to which representor the packet goes, and then sets the rule. Packets hit this rule and go to representor0.

If the SWITCH is in ARM, VFs are in host

```
HOST/ARM
  DOCA_FLOW
    APP
      rep0
      rep1

switch

packet1 - miss flow

packet2
```

doca_dev field is mandatory in doca_flow_port_cfg (using doca_flow_port_cfg_set_dev()) and isolated mode should be specified.

**Note**

The application must avoid initialization of the VF/SF representor ports in DPDK API (i.e., the following functions `rte_eth_dev_configure()`, `rte_eth_rx_queue_setup()`, `rte_eth_dev_start()`) must not be called for VF/SF representor ports).

DOCA Flow switch mode unifies all the ports to the switch manager port for traffic management. This means that all the traffic is handled by switch manager port. Users only have to create an RSS pipe on the switch manager port to get the missed traffic, and they should only manage the pipes on the switch manager port. Switch mode can work with two different mode_args configurations: With or without expert. The way to retrieve the miss traffic source's port_id depends on this configuration:
If `expert` is not set, the traffic misses to software would be tagged with `port_id` information in the mbuf CQE field to allow users to deduce the source `port_id`. Meanwhile, users can set the destination `port_id` to mbuf meta and the packet is sent out directly to the destination port based on the meta information.

**Note**

Only one RSS pipe is supported in switch mode, users can add multiple RSS pipe entries to that RSS pipe. Traffic missed from the user’s pipe without a specified `fwd_miss` target is sent to the kernel if it is isolated mode, or sent to DOCA application (bypassing the kernel) if it is non-isolated (default) mode.

**Info**

Please refer to the "Flow Switch to Wire" sample to get more information regarding the `port_id` management with missed traffic mbuf.
- If expert is set, the `port_id` is not added to the packet. Users can configure the pipes freely to implement their own solution.

### Note
Traffic cloned from the VF to the RSS pipe misses its `port_id` information due to firmware limitation.

- **remote-vnf**

Remote mode is a BlueField mode only, with two physical ports (uplinks). Users must use `doca_flow_port_pair` to pair one physical port and one of its representors. A packet from this uplink, if it does not hit any rules from the users, is firstly received on this representor. Users must also use `doca_flow_port_pair` to pair two physical uplinks. If a packet is received from one uplink and hits the rule whose FWD action is to another uplink, then the packets are sent out from it.

The following diagram shows the basic traffic flow in remote-vnf mode. Packet1, from BlueField uplink P0, firstly misses to host VF0. The app captures this packet and decides whether to drop it or forward it to another uplink (P1). Then, using gRPC to set rules on P0, packet2 hits the rule, then is either dropped or is sent out from P1.
Start Point

DOCA Flow API serves as an abstraction layer API for network acceleration. The packet processing in-network function is described from ingress to egress and, therefore, a pipe must be attached to the origin port. Once a packet arrives to the ingress port, it starts the hardware execution as defined by the DOCA API.

doca_flow_port is an opaque object since the DOCA Flow API is not bound to a specific packet delivery API, such as DPDK. The first step is to start the DOCA Flow port by calling doca_flow_port_start(). The purpose of this step is to attach user application ports to the DOCA Flow ports.

When DPDK is used, the following configuration must be provided:

```c
enum doca_flow_port_type type = DOCA_FLOW_PORT_DPDK_BY_ID;

const char *devargs = "1";
```
The `devargs` parameter points to a string that has the numeric value of the DPDK `port_id` in decimal format. The port must be configured and started before calling this API. Mapping the DPDK port to the DOCA port is required to synchronize application ports with hardware ports.

**Port Operation State**

DOCA Flow ports can be initialized multiple times from different instances. Each instance prepares its pipeline, but only one actively receives port traffic at a time. The instance actively handling the port traffic depends on the operation state set by the `doca_flow_port_cfg_set_operation_state()` function:

- **DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE** – The instance actively handles incoming and outgoing traffic
- **DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP** – The instance handles traffic actively when no other active instance is available
- **DOCA_FLOW_PORT_OPERATION_STATE_STANDBY** – The instance handles traffic only when no active or active_ready_to_swap instance is available
- **DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED** – The instance does not handle traffic, regardless of the state of other instances

If the `doca_flow_port_cfg_set_operation_state()` function is not called, the default state `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` is applied.

**Note**

When a port is configured with a state that expects to handle traffic, it takes effect only after root pipes are created for this port.

When the active port is closed, either gracefully or due to a crash, the standby instance automatically becomes active without any action required.
The port operation state can be modified after the port is started using the 
doca_flow_port_operation_state_modify() function.

**Use Case Examples**

**Hot Upgrade**

This operation state mechanism allows upgrading the DOCA Flow program without losing any traffic.

To upgrade an existing DOCA Flow program with ports started in DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE state (Instance A):

1. Open a new Instance B and start its ports in DOCA_FLOW_PORT_OPERATION_STATE_STANDBY state.

2. Modify Instance A's ports from DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE to DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED state. At this point, Instance B starts receiving traffic.

3. Close Instance A.

4. Open a new Instance C with DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED state. Instance C is the upgraded version of Instance A.

5. Create the entire pipeline for Instance C.

6. Change Instance C's state from DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED to DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE. At this point, Instance B stops receiving traffic and Instance C starts.

7. Instance B can either be closed or kept as a backup should Instance C crash.

**Swap Existing Instances**

This mechanism also facilitates swapping two different DOCA Flow programs without losing any traffic.

To swap between two existing DOCA Flow programs with ports started in DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE and DOCA_FLOW_PORT_OPERATION_STATE_STANDBY
states (Instance A and Instance B, respectively):

1. Modify Instance A's ports from `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP`.

2. Modify Instance B's ports from `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`. At this point, Instance B starts receiving traffic.

3. Modify Instance A's ports from `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP` to `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY`.

**Limitations**

- Supported only in switch mode – the `mode_args` string must include "switch".

- Only the switch port supports states; its representors are affected by its state. Starting a representor port or calling the modify function with a non-active operation state should fail.

- Two instances cannot be in the same operation state simultaneously, except for `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED`.

**Create Pipe and Pipe Entry**

Pipe is a template that defines packet processing without adding any specific hardware rule. A pipe consists of a template that includes the following elements:

- Match
- Monitor
- Actions
- Forward
The following diagram illustrates a pipe structure.

The creation phase allows the hardware to efficiently build the execution pipe. After the pipe is created, specific entries can be added. A subset of the pipe may be used (e.g., skipping the monitor completely, just using the counter, etc).

**Pipe Matching or Action Applying**

DOCA Flow allows defining criteria for matching on a packet or for taking actions on a matched packet by modifying it. The information defining these criteria is provided through the following pointers:

- Match or action pointer – given at pipe or entry creation
- Mask pointer – optionally given at pipe creation

Defining criteria for matching or actions on a packet can be done at the pipe level, where it applies to all packets of a pipe, or specified on a per entry basis, where each entry defines the operation on either the match, actions, or both.

In DOCA Flow terminology, when a field is identified as CHANGEABLE at pipe creation, this means that the actual criterion of the field is deferred to entry creation. Different entries can provide different criteria for a CHANGEABLE field.

A match or action field can be categorized, during pipe creation, as one of the following:

- IGNORED – Ignored in either the match or action taking process
- CHANGEABLE – When the actual behavior is deferred to the entry creation stage
• SPECIFIC – Value is used as is in either match or action process

A mask field can either be provided, in which case it is called it explicit matching, or action applying. If the mask pointer is NULL, we call it implicit matching or action applying. The following subsections provide the logic governing matching and action applying.

When a field value is specified as 0xffff it means that all the field's bits are set (e.g., for TTL it means 0xff and for IPv4 address it means 0xffffffff).

**Matching**

Matching is the process of selecting packets based on their fields' values and steering them for further processing. Processing can either be further matching or actions applying.

![Diagram of packet processing](image)

The packet enters the green filter which modifies it by masking it with the value A. The output value, P&A, is then compared to the value B, and if they are equal, then that is a match.

The values of A and B are evaluated according to the values of the pipe configuration and entry configuration fields, according to the tables in sections "Implicit matching" and "Explicit matching".

**Implicit Matching**

<table>
<thead>
<tr>
<th>Match Type</th>
<th>Pipe Match Value (V)</th>
<th>Pipe Match Mask (M)</th>
<th>Entry Match Value (E)</th>
<th>Filter (A)</th>
<th>Rule (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignore</td>
<td>0</td>
<td>NULL</td>
<td>N/A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Constant</td>
<td>0&lt;V&lt;0xffff</td>
<td>NULL</td>
<td>N/A</td>
<td>0xffff</td>
<td>V</td>
</tr>
<tr>
<td>Changeable (per entry)</td>
<td>0xffff</td>
<td>NULL</td>
<td>0≤E≤0xffff</td>
<td>0xffff</td>
<td>E</td>
</tr>
</tbody>
</table>
Explicit Matching

<table>
<thead>
<tr>
<th>Match Type</th>
<th>Pipe Match Value (V)</th>
<th>Pipe Match Mask (M)</th>
<th>Entry Match Value (E)</th>
<th>Filter (A)</th>
<th>Rule (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>V!=0xffff</td>
<td>0&lt;M≤0xffffffff</td>
<td>0≤E≤0xffffffff</td>
<td>M</td>
<td>M&amp;V</td>
</tr>
<tr>
<td>Changeable</td>
<td>V==0xffffffff</td>
<td>0&lt;M≤0xffffffff</td>
<td>0≤E≤0xffffffff</td>
<td>M</td>
<td>M&amp;E</td>
</tr>
<tr>
<td>Ignored</td>
<td>0≤V&lt;0xffffffff</td>
<td>M==0</td>
<td>0≤E≤0xffffffff</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Action Applying

Implicit Action Applying

<table>
<thead>
<tr>
<th>Action Type</th>
<th>Pipe Action value (V)</th>
<th>Pipe Action Mask (M)</th>
<th>Entry Action value (E)</th>
<th>Action on the field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignore</td>
<td>0</td>
<td>NULL</td>
<td>N/A</td>
<td>none</td>
</tr>
<tr>
<td>Constant</td>
<td>0 &lt; V &lt; 0xffffffff</td>
<td>NULL</td>
<td>N/A</td>
<td>set to V</td>
</tr>
<tr>
<td>Changeable</td>
<td>0xffffffff</td>
<td>NULL</td>
<td>E</td>
<td>set to E</td>
</tr>
</tbody>
</table>

Implicit action applying example:

- Destination IPv4 address is 255.255.255.255
- No mask provided
- Entry value is 192.168.0.1
- Result – The action field is changeable. Therefore, the value is provided by the entry. If a match on the packet occurs, the packet destination IPv4 address is changed to 192.168.0.1.
Explicit Action Applying

Info

Assume P is packet's field value.

<table>
<thead>
<tr>
<th>Action Type</th>
<th>Pipe Action value (V)</th>
<th>Pipe Action Mask (M)</th>
<th>Entry Action value (E)</th>
<th>Action on the field</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>V!=0xffff</td>
<td>0≤M≤0xffff</td>
<td>0≤E≤0xffff</td>
<td>set to (~M &amp; P)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>In words: modify only bits that are set on the mask to the values in V</td>
</tr>
<tr>
<td>Changeable</td>
<td>V==0xffff</td>
<td>0&lt;M≤0xffff</td>
<td>0≤E≤0xffff</td>
<td>set to (~M &amp; P)</td>
</tr>
<tr>
<td>Ignored</td>
<td>0≤V&lt;0xffff</td>
<td>M==0</td>
<td>0≤E≤0xffff</td>
<td>none</td>
</tr>
</tbody>
</table>

Explicit action applying example:

- Destination IPv4 address is 192.168.10.1
- Mask is provided and equals 255.255.0.0
- Entry value is ignored
- Result – If a match on the packet occurs, the packet destination IPv4 value changes to 192.168.0.0.
Match is a mandatory parameter when creating a pipe. Using the `doca_flow_match` struct, users must define the packet fields to be matched by the pipe.

For each `doca_flow_match` field, users select whether the field type is:

- **Ignore (match any)** – the value of the field is ignored in a packet. In other words, match on any value of the field.

- **Constant (specific)** – all entries in the pipe have the same value for this field. Users should not put a value for each entry.

- **Changeable** – the value of the field is defined per entry. Users must provide it upon adding an entry.

**Note**

L4 type, L3 type, and tunnel type cannot be changeable.

The match field type can be defined either implicitly or explicitly using the `doca_flow_pipe_cfg_set_match(struct doca_flow_pipe_cfg *cfg, const doca_flow_match *match, const doca_flow_match *match_mask)` function. If `match_mask == NULL`, then it is done implicitly. Otherwise, it is explicit.

In the tables in the following subsections, an example is used of a 16-bit field (such as layer-4 destination port) where:

**Note**

The same concept would apply to any other field (such as an IP address occupying 32 bits).

- P stands for the packet field value
- V stands for the pipe match field value
• M stands for the pipe mask field value

• E stands for the match entry field value

**Implicit Match**

<table>
<thead>
<tr>
<th>Match Type</th>
<th>Pipe Match Value (V)</th>
<th>Pipe Match Mask (M)</th>
<th>Entry Match Value (E)</th>
<th>Filter (A)</th>
<th>Rule (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignore</td>
<td>0</td>
<td>NULL</td>
<td>N/A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Constant</td>
<td>0&lt;V&lt;0xffff</td>
<td>NULL</td>
<td>N/A</td>
<td>0xffff</td>
<td>V</td>
</tr>
<tr>
<td>Changeable (per entry)</td>
<td>0xffffffff</td>
<td>NULL</td>
<td>0≤E≤0xffff</td>
<td>0xffffffff</td>
<td>E</td>
</tr>
</tbody>
</table>

To match implicitly, the following considerations should be taken into account.

• **Ignored fields:**
  - Field is zeroed
  - Pipeline has no comparison on the field

• **Constant fields** – These are fields that have a constant value among all entries. For example, as shown in the following, the tunnel type is VXLAN:

  ```
  match.tun.type = DOCA_FLOW_TUN_VXLAN;
  ```

  These fields must only be configured once at pipe build stage, not once per new pipeline entry.

• **Changeable fields** – These are fields whose value may change per entry. For example, the following shows match on a destination IPv4 address of variable per-entry value (outer 5-tuple):

  ```
  match.outer.ip4.dst_ip = 0xffffffff;
  ```

• The following is an example of a match, where:
● Outer 5-tuple
  ■ L3 type is IPv4 – constant among entries by design
  ■ L4 type is UDP – constant among entries by design
  ■ Tunnel type is DOCA_Flow_Tun_VxLAN – constant among entries by design
  ■ IPv4 destination address varies per entry
  ■ UDP destination port is always DOCA_VXLAN_DEFAULT_PORT
  ■ VXLAN tunnel ID varies per entry
  ■ The rest of the packet fields are ignored

● Inner 5-tuple
  ■ L3 type is IPv4 – constant among entries by design
  ■ L4 type is TCP – constant among entries by design
  ■ IPv4 source and destination addresses vary per entry
  ■ TCP source and destination ports vary per entry
  ■ The rest of the packet fields are ignored

// filter creation
static void build_underlay_overlay_match(struct doca_flow_match *match) {

   //outer
   match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
   match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
   match->tun.type = DOCA_FLOW_TUN_VXLAN;
   match->outer.ip4.dst_ip = 0xffffffff;
   match->outer.udp.l4_port.dst_port = DOCA_VXLAN_DEFAULT_PORT;
   match->tun.vxlan_tun_id = 0xffffffff;

   //inner
   match->inner.l3_type = DOCA_FLOW_L3_TYPE_IP4;
   match->inner.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP;
match->inner.ip4.dst_ip = 0xffffffff;
match->inner.ip4.src_ip = 0xffffffff;
match->inner.tcp.l4_port.src_port = 0xffff;
match->inner.tcp.l4_port.dst_port = 0xffff;
}

// create entry specifying specific values to match upon
doca_error_t add_entry(struct doca_flow_pipe *pipe, struct doca_flow_port *port,
    struct doca_flow_pipe_entry **entry)
{
    struct doca_flow_match match = {};
    struct entries_status status = {};
    doca_error_t result;

    match.outer.ip4.dst_ip = BE_IPV4_ADDR(7, 7, 7, 1);
    match.tun.vxlan_tun_id = RTE_BE32(9876);
    match.inner.ip4.src_ip = BE_IPV4_ADDR(8, 8, 8, 1);
    match.inner.ip4.dst_ip = BE_IPV4_ADDR(9, 9, 9, 1);
    match.inner.tcp.l4_port.src_port = rte_cpu_to_be_16(5678);
    match.inner.tcp.l4_port.dst_port = rte_cpu_to_be_16(1234);
    result = doca_flow_pipe_add_entry(0, pipe, &match, &actions, NULL, NULL, 0, &status, entry);
}

\[\text{Note}\]

The fields of the `doca_flow_meta` struct inside the match are not subject to implicit match rules and must be paired with explicit mask values.

**Explicit Match**

<table>
<thead>
<tr>
<th>Match Type</th>
<th>Pipe Match Value (V)</th>
<th>Pipe Match Mask (M)</th>
<th>Entry Match Value (E)</th>
<th>Filter (A)</th>
<th>Rule (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>V!=0xffff</td>
<td>0&lt;M≤0xffff</td>
<td>0≤E≤0xffff</td>
<td>M</td>
<td>M&amp;V</td>
</tr>
<tr>
<td>Match Type</td>
<td>Pipe Match Value (V)</td>
<td>Pipe Match Mask (M)</td>
<td>Entry Match Value (E)</td>
<td>Filter (A)</td>
<td>Rule (B)</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>---------------------</td>
<td>-----------------------</td>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>Changeable</td>
<td>V==0xffff</td>
<td>0&lt;M≤0xffff</td>
<td>0≤E≤0xffff</td>
<td>M</td>
<td>M&amp;E</td>
</tr>
<tr>
<td>Ignored</td>
<td>0≤V&lt;0xffff</td>
<td>M==0</td>
<td>0≤E≤0xffff</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

In this case, there are two `doca_flow_match` items, the following considerations should be considered:

- **Ignored fields**
  - M equals zero. This can be seen from the table where the rule equals 0. Since mask is also 0, the resulting packet after the filter is 0. Thus, the comparison always succeeds.

  ```
  match_mask.inner.ip4.dst_ip = 0;
  ```

- **Constant fields**

  These are fields that have a constant value. For example, as shown in the following, the inner 5-tuple match on IPv4 destination addresses belonging to the 0.0.0.0/24 subnet, and this match is constant among all entries:

  ```
  // BE_IPV4_ADDR converts 4 numbers A,B,C,D to a big endian representation of IP address A.B.C.D
  match.inner.ip4.dst_ip = 0;
  match_mask.inner.ip4.dst_ip = BE_IPV4_ADDR(255, 255, 255, 0);
  ```

  For example, as shown in the following, the inner 5-tuple match on IPv4 destination addresses belonging to the 1.2.0.0/16 subnet, and this match is constant among all entries. The last two octets of the `match.inner.ip4.dst_ip` are ignored because the `match_mask` of 255.255.0.0 is applied:

  ```
  // BE_IPV4_ADDR converts 4 numbers A,B,C,D to a big endian representation of IP address A.B.C.D
  ```
Once a field is defined as constant, the field's value cannot be changed per entry.

**Tip**

Users should set constant fields to zero when adding entries for better code readability.

A more complex example of constant matches may be achieved as follows:

```c
match.mask.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0xf0f0);
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0x5020)
```

The following ports would be matched:

- 0x5020 - 0x502f
- 0x5120 - 0x512f
- ...
- 0x5f20 - 0x5f2f

**Changeable fields**

The following example matches on either FTP or TELNET well known port numbers and forwards packets to a server after modifying the destination IP address and destination port numbers. In the example, either FTP or TELNET are forwarded to the same server. FTP is forwarded to port 8000 and TELNET is forwarded to port 9000.

```c
// at Pipe creation
pipe_cfg.attr.name = "PORT_MAPPER";
```
Relaxed Match

Relaxed matching is the default working mode in DOCA flow. However, it can be disabled per pipe using the `enable_strict_matching` pipe attribute. This mode grants the user more control on matching fields such that only explicitly set match fields by the user (either specific or changeable) are matched by the pipe.

Consider the following strict matching mode example. There are three pipes:

- **Basic pipe A** with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP;` and `match.outer.tcp.flags = 1;`
- **Basic pipe B** with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;` and `match.outer.udp.l4_port.src_port = 8080;`
- **Control pipe X** with two entries to direct TCP traffic to pipe A and UDP to pipe B. The first entry has `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP;` while the second has

```c
pipe_cfg.attr.type = DOCA_FLOW_PIPE_BASIC;
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0xffff); // v
match_mask.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0xffff); // M
pipe_cfg.match_mask = &match_mask;
pipe_cfg.match = &match;
actions_arr[0] = &actions;
pipe_cfg.actions = actions_arr;
pipe_cfg.attr.is_root = true;
pipe_cfg.attr.nb_actions = 1;

// Adding entries
// FTP
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(20); // E
actions.outer.ip4.src_ip = server_addr;
actions.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(8000);
result = doca_flow.pipe_add_entry(0, pipe, &match, &actions, NULL, NULL, 0, &status, entry);

// TELNET
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(23); // E
actions.outer.ip4.src_ip = server_addr;
actions.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(9000);
result = doca_flow.pipe_add_entry(0, pipe, &match, &actions, NULL, NULL, 0, &status, entry);
```
As a result, the hardware performs match on the L4 header type twice:

- First, when the packet enters the filter in control pipe X to decide the next pipe
- Second, when the packet enters the filter of pipe A or pipe B to do the match on L4 header fields

With particularly large pipelines, such double matches decrease performance and increase the memory footprint in hardware. Relaxed matching mode gives the user greater control of the match to solve the performance problems.

In relaxed mode, type selectors in the outer, inner, and tun parts of the doca_flow_match are used only for the type cast (or selectors) of the underlying unions. Header-type matches are available using the parser_meta API.

Thus, the aforementioned scenario may be overwritten in the following manner. There are three pipes:

- **Basic pipe A** with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP;` and `match.outer.tcp.flags = 1;`

- **Basic pipe B** with `match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;` and `match.outer.udp.l4_port.src_port = 8080;`

- **Control pipe X** with two entries to direct TCP traffic to pipe A and UDP to pipe B. The first entry has `match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_TCP;` while the second has `match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_UDP;`.

As a result, the hardware performs the L4 header-type match only once, when the packet enters the filter of control pipe. Basic pipes' `match.outer.l4_type_ext` are used only for the selection of the `match.outer.tcp` or `match.outer.udp` structures.

**Example**

The following code snippet is used to demonstrate relaxed matching mode:

```c
// filter creation
static void build_underlay_overlay_match(struct doca_flow_match *match) {
```
This match code above is an example of a match where:

- With relaxed matching disabled (i.e., `enable_strict_matching` attribute set to `true`), the following hardware matches are performed:
  - L3 type is IPv4 – constant among entries by design
  - L4 type is UDP – constant among entries by design
  - Tunnel type is `DOCA_FLOW_TUN_VXLAN` – constant among entries by design
  - IPv4 destination address varies per entry
  - UDP source port is constant among entries
  - VXLAN tunnel ID varies per entry
  - The rest of the packet fields are ignored

- With relaxed matching enabled (default mode), the following hardware matches are performed:
  - IPv4 destination address varies per entry
  - UDP source port is constant among entries
  - VXLAN tunnel ID varies per entry

In summary, with relaxed matching L3, L4, tunnel protocol types, and similar no longer indicate a match on the specific protocol. They are used solely as a selector for the relevant header fields. For example, to match on `outer.ip4.dst_ip`, users must set `outer.l3_type = DOCA_FLOW_L3_TYPE_IP4`. That is, the L3 header is checked for the IPv4 destination.
address. There is no check that it is of IPv4 type. It is user responsibility to make sure that packets arriving to such a filter indeed have an L3 header of type IPv4 (same goes for L4 UDP header/VXLAN tunnel).

**Protocols/Tunnels Type Match**

The following section explains how to match on a protocol's and a tunnel's type with relaxed matching.

To match on a specific protocol/tunnel type, consider the following:

- To match on an inner/outer L3/L4 protocol type, one can use relevant `doca_flow_parser_meta` fields (e.g., for outer protocols, `parser_meta.outer_l[3,4]_type` fields can be used).

- To match on a specific tunnel type (e.g., VXLAN/GRE and so on), users should match on a tunnel according to its specification (e.g., for VXLAN, a match on UDP destination port 4789 can be used). Another option is to use the L3 next protocol field (e.g., for IPv4 with next header GRE, one can match on the IPv4 header's next protocol field value to match GRE IP protocol number 47).

**Example**

Using the aforementioned example, to add the match on the same L3,L4 protocol type and on a VXLAN tunnel with relaxed matching enabled, the following function implementation should be considered:

```c
// filter creation
static void build_underlay_overlay_match(struct doca_flow_match *match)
{
    //outer
    match->parser_meta.outer_l3_type = DOCA_FLOW_L3_META_IPV4;
    match->parser_meta.outer_l4_type = DOCA_FLOW_L4_META_UDP;
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
    match->tun.type = DOCA_FLOW_TUN_VXLAN;
    match->outer.ip4.dst_ip = 0xffffffff;
    match->outer.udp.l4_port.src_port = 22;
    match->outer.udp.l4_port.dst_port = DOCA_VXLAN_DEFAULT_PORT;
```
match->tun.vxlan_tun_id = 0xffffffff;
}

The match code above is an example of a match, where:

- With relaxed matching disabled (i.e., `enable_strict_matching` attribute set to `true`), the following hardware matches are performed:
  - L3 type is IPv4 – constant among entries by design
  - L4 type is UDP – constant among entries by design
  - Tunnel type is `DOCA_FLOW_TUN_VXLAN` – constant among entries by design
  - IPv4 destination address varies per entry
  - UDP source port is always 22
  - UDP destination port is always `DOCA_VXLAN_DEFAULT_PORT`
  - VXLAN tunnel ID varies per entry
  - The rest of the packet fields are ignored

- With relaxed matching enabled (default mode), the following hardware matches are performed:
  - L3 type is IPv4 – constant among entries by design
  - L4 type is UDP – constant among entries by design
  - IPv4 destination address varies per entry
  - UDP source port is always 22
  - UDP destination port is always `DOCA_VXLAN_DEFAULT_PORT`
  - VXLAN tunnel ID varies per entry

Note
With relaxed matching, if any of the selectors is used without setting a relevant field, the pipe/entry creation would fail with the following error message:

```
failed building active opcode - active opcode <opcode number> is protocol only
```

**Setting Pipe Actions**

**Pipe Execution Order**

When setting actions, they are executed in the following order:

1. Crypto (decryption)
2. Decapsulation
3. Pop
4. Meta
5. Outer
6. Tun
7. Push
8. Encapsulation
9. Crypto (encryption)

**Note**
Auto-modification

Similarly to setting pipe match, actions also have a template definition.

Similarly to `doca_flow_match` in the creation phase, only the subset of actions that should be executed per packet are defined. This is done in a similar way to match, namely by classifying a field of `doca_flow_match` to one of the following:

- **Ignored field** – field is zeroed, modify is not used.
- **Constant fields** – when a field must be modified per packet, but the value is the same for all packets, a one-time value on action definitions can be used.
- **Changeable fields** – fields that may have more than one possible value, and the exact values are set by the user per entry.

```
actions.outer.ip4.dst_ip = 0xffffffff
```

**Note**

The `action_mask` should be set as `0xffffffff` and `action` as `0` if the user wants to configure `0` to this field.

Explicit Modification Type
It is possible to force constant modification or per-entry modification with action mask. For example:

```c
static void
create_constant_modify_actions(struct doca_flow_actions *actions
    struct doca_flow_actions *actions_mask,
    struct doca_flow_action_descs *descs)
{
    actions->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
    actions->outer.udp.src_port = 0x1234;
    actions_mask->outer.udp.src_port = 0xffff;
}
```

**Copy Field**

The action descriptor can be used to copy between the packet field and metadata. For example:

```c
#define META_U32_BIT_OFFSET(idx) (offsetof(struct doca_flow_meta, u32[(idx)]) << 3)

static void
create_copy_packet_to_meta_actions(struct doca_flow_match *match
    struct doca_flow_action_desc *desc)
{
    desc->type = DOCA_FLOW_ACTION_COPY;
    desc->field_op.src.field_string = "outer.ipv4.src_ip";
    desc->field_op.src.bit_offset = 0;
    desc->field_op.dst.field_string = "meta.data";
    desc->field_op.dst.bit_offset = META_U32_BIT_OFFSET(1); /* Bit offset of meta.u32[1] */;
}
```

**Multiple Actions List**

Creating a pipe is possible using a list of multiple actions. For example:
Summary of Action Types

<table>
<thead>
<tr>
<th>Pipe Creation</th>
<th>Entry Creation</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>action_desc</td>
<td>Pipe Actions</td>
<td>Entry Actions</td>
</tr>
<tr>
<td>DOCA_FLOW_ACTION_ADD/ action_desc = NULL</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>mask != 0</td>
</tr>
<tr>
<td></td>
<td>val != 0 &amp;&amp; val != 0xFF</td>
<td>mask != 0</td>
</tr>
<tr>
<td></td>
<td>val = 0xFF</td>
<td>mask = 0</td>
</tr>
<tr>
<td></td>
<td>val = 0xFF</td>
<td>mask != 0</td>
</tr>
</tbody>
</table>

```c
static void create_multi_actions_for_pipe_cfg()
{
    struct doca_flow_actions *actions_arr[2];
    struct doca_flow_actions actions_0 = {0}, actions_1 = {0};
    struct doca_flow_pipe_cfg *pipe_cfg;
    /* input configurations for actions_0 and actions_1 */
    actions_arr[0] = &actions_0;
    actions_arr[1] = &actions_1;
    doca_flow_pipe_cfg_set_actions(pipe_cfg, actions_arr, NULL, NULL, 2);
}
```
<table>
<thead>
<tr>
<th>Pipe Creation</th>
<th>Entry Creation</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add field value or from src</td>
<td>width</td>
<td>val == 0</td>
</tr>
<tr>
<td>Define the src and dst fields and width</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOCA_FLOW_ACTION_COPY</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Copy field to another field</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Setting Pipe Monitoring**

If a meter policer should be used, then it is possible to have the same configuration for all policers on the pipe or to have a specific configuration per entry. The meter policer is determined by the FWD action. If an entry has NULL FWD action, the policer FWD action is taken from the pipe.

If a mirror should be used, mirror can be shared on the pipe or configured to have a specific value per entry.
The monitor also includes the aging configuration, if the aging time is set, this entry ages out if timeout passes without any matching on the entry.

For example:

```c
static void build_entry_monitor(struct doca_flow_monitor *monitor, void *user_ctx)
{
    monitor->aging_sec = 10;
}
```

Refer to [Pipe Entry Aged Query](#) for more information.

### Setting Pipe Forwarding

The FWD (forwarding) action is the last action in a pipe, and it directs where the packet goes next. Users may configure one of the following destinations:

- Send to software (representor)
- Send to wire
- Jump to next pipe
- Drop packets

The FORWARDING action may be set for pipe create, but it can also be unique per entry.

A pipe can be defined with constant forwarding (e.g., always send packets on a specific port). In this case, all entries will have the exact same forwarding. If forwarding is not defined when a pipe is created, users must define forwarding per entry. In this instance, pipes may have different forwarding actions.

When a pipe includes meter monitor `<cir, cbs>`, it must have `fwd` defined as well as the policer.

If a pipe is created with a dedicate constant mirror with FWD, the pipe FWD can be from a mirror FWD or a pipe FWD and the two FWDS are exclusive. It is not allowed to specify a mirror with a FWD to a pipe with FWD also.
If a mirror FWD is not configured, the FWD is from the pipe configuration. The FWD of the pipe with a mirror cannot be direct RSS, only shared RSS from NULL FWD is allowed.

The following is an RSS forwarding example:

```c
fwd->type = DOCA_FLOW_FWD_RSS;
fwd->rss_queues = queues;
fwd->rss_flags = DOCA_FLOW_RSS_IP | DOCA_FLOW_RSS_UDP;
fwd->num_of_queues = 4;
```

Queues point to the `uint16_t` array that contains the queue numbers. When a port is started, the number of queues is defined, starting from zero up to the number of queues minus 1. RSS queue numbers may contain any subset of those predefined queue numbers. For a specific match, a packet may be directed to a single queue by having RSS forwarding with a single queue.

Changeable RSS forwarding is supported. When creating the pipe, the `num_of_queues` must be set to `0xffffffff`, then different forwarding RSS information can be set when adding each entry.

```c
fwd->num_of_queues = 0xffffffff;
```

The packet is directed to the port. In many instances the complete pipe is executed in the hardware, including the forwarding of the packet back to the wire. The packet never arrives to the software.

Example code for forwarding to port:

```c
struct doca_flow_fwd *fwd = malloc(sizeof(struct doca_flow_fwd));
memset(fwd, 0, sizeof(struct doca_flow_fwd));
fwd->type = DOCA_FLOW_FWD_PORT;
fwd->port_id = port_id; // this should the same port_id that was set in doca_flow_port_cfg_set_devargs()
```

The type of forwarding is `DOCA_FLOW_FWD_PORT` and the only data required is the `port_id` as defined in `DOCA_FLOW_PORT`. 
Changeable port forwarding is also supported. When creating the pipe, the `port_id` must be set to 0xffff, then different forwarding `port_id` values can be set when adding each entry.

```c
fwd->port_id = 0xffff;
```

**Shared Resources**

DOCA Flow supports several types of resources that can be shared. The supported types of resources can be:

- Meters
- Counters
- RSS queues
- Mirrors
- PSPs
- Encap
- Decap
- IPsec SA

Shared resources can be used by several pipes and can save device and memory resources while promoting better performance.

To create and configure shared resource, the user should go through the steps detailed in the following subsections.

**Creating Shared Resource Configuration Object**

Call `doca_flow_cfg_create(&flow_cfg)`, passing a pointer to `struct doca_flow_cfg` to be used to fill the required parameters for the shared resource.
Setting Number of Shared Resources per Shared Resource Type

This can be done by calling `doca_flow_cfg_set_nr_shared_resource()`. Refer to the API documentation for details on the configuration process.

Conclude the configuration by calling `doca_flow_init()`.

Configuring Shared Resource

When shared resources are allocated, they are assigned identifiers ranging from 0 and increasing incrementally. For example, if the user configures two shared counters, they would bear the identifiers 0 and 1.

Note

- Note: The `struct doca_flow_cfg` object is used for configuring other resources besides the aforementioned shared resources, but this section only refers to the configuration of shared resources.

- Note: Note that each resource has its own identifier space. So, if users have two shared counters and three meters, they would bear identifiers 0..1 and 0..2 respectively.

Configuring the shared resources requires the user to call `doca_flow_shared_resource_set_cfg()`.

Binding Shared Resource
A shared resource must be bound by calling `doa_flow_shared_resources_bind()` which binds the resource to a pointer. The object to which the resource is bound is usually a `struct doca_flow_port` pointer.

**Using Shared Resources**

After a resource has been configured, it can be used by referring to its ID.

In the case of meters, counters, and mirrors, they are referenced through `struct doca_flow_monitor` during pipe creation or entry addition.

**Querying Shared Resource**

Querying shared resources can be done by calling `doa_flow_shared_resources_query()`. The function accepts the resource type and an array of resource numbers, and returns an array of `struct doca_flow_shared_resource_result` with the results.

**Shared Meter Resource**

A shared meter can be used in multiple pipe entries (hardware steering mode support only).

The shared meter action marks a packet with one of three colors: Green, Yellow, and Red. The packet color can then be matched in the next pipe, and an appropriate action may be taken. For example, packets marked in red color are usually dropped. So, the next pipe to meter action may have an entry which matches on red and has fwd type `DOCA_FLOW_FWD_DROP`.

DOCA Flow supports three marking algorithms based on RFCs: 2697, 2698, and 4115.

*RFC 2697 – Single-rate Three Color Marker (srTCM)*
CBS (committed burst size) is the bucket size which is granted credentials at a CIR (committed information rate). If CBS overflow occurs, credentials are passed to the EBS (excess burst size) bucket. Packets passing through the meter consume credentials. A packet is marked green if it does not exceed the CBS, yellow if it exceeds the CBS but not the EBS, and red otherwise. A packet can have an initial color upon entering the meter. A pre-colored yellow packet will start consuming credentials from the EBS.

RFC 2698 – Two-rate Three Color Marker (trTCM)
CBS and CIR are defined as in RFC 2697. PBS (peak burst size) is a second bucket which is granted credentials at a PIR (peak information rate). There is no overflow of credentials from the CBS bucket to the PBS bucket. The PIR must be equal to or greater than the CIR. Packets consuming CBS credentials consume PBS credentials as well. A packet is marked red if it exceeds the PIR. Otherwise, it is marked either yellow or green depending on whether it exceeds the CIR or not. A packet can have an initial color upon entering the meter. A pre-colored yellow packet starts consuming credentials from the PBS.

**RFC 4115 – trTCM without Peak-rate Dependency**

EBS is a second bucket which is granted credentials at a EIR (excess information rate) and gets overflowed credentials from the CBS. For the packet marking algorithm, refer to RFC 4115.

The following sections present the steps for configuring and using shared meters to mark packets.

**Shared IPsec SA Resource**

The IPsec Security Association (SA) shared resource is used for IPsec ESP encryption protocol. The resource should be pointed from the `doca_flow_crypto_actions` struct that inside
By default, the resource manages the state of the sequence number (SN), incrementing each packet on the encryption side, and performing anti-replay protection on the decryption side.

To control the SN in software, `sn_offload` should be disabled per port in the configuration for `doca_flow_port_start` (see DOCA API documentation for details). Once `sn_offload` is disabled, the following fields are ignored: `sn_offload_type`, `win_size`, `sn_initial`, and `lifetime_threshold`.

When shared resource query is called for an IPsec SA resource, the current SN is retrieved for the encryption resource and the lower bound of anti-replay window is retrieved for the decryption resource. Querying IPsec SA can only be called when `sn_offload` is enabled.

To maintain a valid state of the resource during its usage, `doca_flow_crypto_ipsec_resource_handle` should be called periodically.

**Shared Mirror Resource**

The mirror shared resource is used to clone packets to other pipes, vports (switch mode only), RSS queues (VNF mode only), or drop.

- **Info**
  
  The maximum supported mirror number is 4K.

- **Info**
  
  The maximum supported mirror clone destination is 254.
Mirror clone destination as `next_pipe` cannot be intermixed with `port` or `rss` types. Only clone destination and origin destination both as `next_pipe` is supported.

The register copy for packet after mirroring is not saved.

**Note**

For switch mode, there are several mirror limitations which should be noted:

- Mirror should be cloned to `DOCA_FLOW_DIRECTION_BIDIRECTIONAL` pipe
- The register copy for pkt after mirroring is not saved
- Mirror should not be cloned to RSS pipe directly
- Encap is supported while cloning a packet to a wire port only
- Mirror must not be configured on a resizable pipe

If mirror creation fails, users should check the resulting syndrome for failure details.

**Mirroring and Packet Order**

To maintain the order of the mirrored packets in relation to the non-mirrored ones, set a first mirror target forward destination equivalent to the non-mirrored packets as illustrated in the following diagram:

In NVIDIA® BlueField®-3, NVIDIA® ConnectX®-7, and lower, when using the mirror action in the egress domain, mirrored packets cannot preserve the order with the non-mirrored packets due to the high latency of the mirror operation. To maintain the order, use `DOCA_FLOW_FWD_DROP` as the target forward as illustrated in the following diagram:
Shared Encap Resource

The encap shared resource is used for encapsulation. A shared encap ID represents one kind of encap configuration and can be used in multiple pipes and entries (hardware steering mode support only).

The shared encap action encapsulates the packet with the configured tunnel information.

Shared Decap Resource

The decap shared resource is used for decapsulation. A shared decap ID represents one kind of decap configuration and can be used in multiple pipes and entries (hardware steering mode support only).

The shared decap action decapsulates the packet. Ethernet information should be provided when is_l2 is false.
The PSP shared resource is used for PSP encryption. The resource should be pointed to from the `doca_flow_crypto_actions struct` in `doca_flow_actions`.

The resource should be configured with a key to encrypt the packets. See NVIDIA DOCA Library API documentation for PSP key generation for a reference about key handling on decrypt side.

**Basic Pipe Create**

Once all parameters are defined, the user should call `doca_flow_pipe_create` to create a pipe.

The return value of the function is a handle to the pipe. This handle should be given when adding entries to pipe. If a failure occurs, the function returns `NULL`, and the error reason and message are put in the error argument if provided by the user.

Refer to the NVIDIA DOCA Library APIs to see which fields are optional and may be skipped. It is typically recommended to set optional fields to 0 when not in use. See Miss Pipe and Control Pipe for more information.

Once a pipe is created, a new entry can be added to it. These entries are bound to a pipe, so when a pipe is destroyed, all the entries in the pipe are removed. Please refer to section Pipe Entry for more information.

There is no priority between pipes or entries. The way that priority can be implemented is to match the highest priority first, and if a miss occurs, to jump to the next PIPE. There can be more than one PIPE on a root as long the pipes are not overlapping. If entries overlap, the priority is set according to the order of entries added. So, if two pipes have overlapping matching and PIPE1 has higher priority than PIPE2, users should add an entry to PIPE1 after all entries are added to PIPE2.

**Pipe Entry (doca_flow_pipe_add_entry)**

An entry is a specific instance inside of a pipe. When defining a pipe, users define match criteria (subset of fields to be matched), the type of actions to be done on matched packets, monitor, and, optionally, the FWD action.
When a user calls `doca_flow.pipe_add_entry()` to add an entry, they should define the values that are not constant among all entries in the pipe. And if FWD is not defined then that is also mandatory.

DOCA Flow is designed to support concurrency in an efficient way. Since the expected rate is going to be in millions of new entries per second, it is mandatory to use a similar architecture as the data path. Having a unique queue ID per core saves the DOCA engine from having to lock the data structure and enables the usage of multiple queues when interacting with hardware.

Each core is expected to use its own dedicated `pipe_queue` number when calling `doca_flow.pipe_entry`. Using the same `pipe_queue` from different cores causes a race condition and has unexpected results.

![Diagram of DOCA Flow](image)

**Note**

Applications are expected to avoid adding, removing, or updating pipe entries from within a `doca_flow.entry_process_cb`.

**Failure Path**
Entry insertion can fail in two places, add_entry and add_entry_cb.

- When add_entry fails, no cleanup is required.
- When add_entry succeeds, a handle is returned to the user. If the subsequent add_entry_cb fails, the user is responsible for releasing the handle through a rm_entry call. This rm_entry call is expected to return DOCA_SUCCESS and is expected to invoke doca_rm_entry_cb with a successful return code.

**Pipe Entry Counting**

By default, no counter is added. If defined in monitor, a unique counter is added per entry.

**Note**

Having a counter per entry affects performance and should be avoided if it is not required by the application.

The retrieved statistics are stored in struct doca_flow_query.

**Pipe Entry Aged Query**

When a user calls doca_flow_aging_handle(), this query is used to get the aged-out entries by the time quota in microseconds. The user callback is invoked by this API with the aged entries.

Since the number of flows can be very large, the query of aged flows is limited by a quota in microseconds. This means that it may return without all flows and requires the user to call it again. When the query has gone over all flows, a full cycle is done.
Pipe Entry With Multiple Actions

Users can define multiple actions per pipe. This gives the user the option to define different actions per entry in the same pipe by providing the `action_idx` in `struct doca_flow_actions`.

For example, to create two flows with the same match but with different actions, users can provide two actions upon pipe creation, `Action_0` and `Action_1`, which have indices 0 and 1 respectively in the actions array in the pipe configuration. `Action_0` has `modify_mac`, and `Action_1` has `modify_ip`.

Users can also add two kinds of entries to the pipe, the first one with `Action_0` and the second with `Action_1`. This is done by assigning 0 in the `action_idx` field in `struct doca_flow_actions` when creating the first entry and 1 when creating the second one.

Miss Pipe and Control Pipe

ℹ️ Note

Only one root pipe is allowed. If more than one is needed, create a control pipe as root and forward the packets to relevant non-root pipes.

To set priority between pipes, users must use miss-pipes. Miss pipes allow to look up entries associated with pipe X, and if there are no matches, to jump to pipe X+1 and perform a lookup on entries associated with pipe X+1.

The following figure illustrates the hardware table structure:
The first lookup is performed on the table with priority 0. If no hits are found, then it jumps to the next table and performs another lookup.

The way to implement a miss pipe in DOCA Flow is to use a miss pipe in FWD. In struct `doca_flow_fwd`, the field `next_pipe` signifies that when creating a pipe, if a `fwd_miss` is configured then if a packet does not match the specific pipe, steering should jump to `next_pipe` in `fwd_miss`.

Note

`fwd_miss` is of type `struct doca_flow_fwd` but it only implements two forward types of this struct:

- `DOCA_FLOW_FWD_PIPE` – forwards the packet to another pipe
- `DOCA_FLOW_FWD_DROP` – drops the packet

Other forwarding types (e.g., forwarding to port or sending to RSS queue) are not supported.

`next_pipe` is defined as `doca_flow_pipe` and created by `doca_flow_pipe_create`. To separate `miss_pipe` and a general one, `is_root` is introduced in struct `doca_flow_pipe_cfg`. If `is_root` is true, it means the pipe is a root pipe executed on packet arrival. Otherwise, the pipe is `next_pipe`.

When `fwd_miss` is not null, the packet that does not match the criteria is handled by `next_pipe` which is defined in `fwd_miss`. 
In internal implementations of `doca_flow_pipe_create`, if `fwd_miss` is not null and the forwarding action type of `miss_pipe` is `DOCA_FLOW_FWD_PIPE`, a flow with the lowest priority is created that always jumps to the group for the `next_pipe` of the `fwd_miss`. Then the flow of `next_pipe` can handle the packets, or drop the packets if the forwarding action type of `miss_pipe` is `DOCAFLOW_FWD_DROP`.

For example, VXLAN packets are forwarded as RSS and hairpin for other packets. The `miss_pipe` is for the other packets (non-VXLAN packets) and the match is for general Ethernet packets. The `fwd_miss` is defined by `miss_pipe` and the type is `DOCAFLOW_FWD_PIPE`. For the VXLAN pipe, it is created by `doca_flow_create()` and `fwd_miss` is introduced.

Since, in the example, the jump flow is for general Ethernet packets, it is possible that some VXLAN packets match it and cause conflicts. For example, VXLAN flow entry for ipA is created. A VXLAN packet with ipB comes in, no flow entry is added for ipB, so it hits `miss_pipe` and is hairpinned.

A control pipe is introduced to handle the conflict. After creating a control pipe, the user can add control entries with different matches, forwarding, and priorities when there are conflicts.

The user can add a control entry by calling `doca_flow_control_pipe_add_entry()`.

priority must be defined as higher than the lowest priority (3) and lower than the highest one (0).

The other parameters represent the same meaning of the parameters in `doca_flow_pipe_create`. In the example above, a control entry for VXLAN is created. The VLXAN packets with ipB hit the control entry.

**doca_flow_pipe_lpm**

`doca_flow_pipe_lpm` uses longest prefix match (LPM) matching. LPM matching is limited to a single field of the match provided by the user at pipe creation (e.g., the outer destination IP). Each entry is consisted of a value and a mask (e.g., 10.0.0.0/8, 10.10.0.0/16, etc). The LPM match is defined as the entry that has the maximum matching bits. For example, using the two entries 10.7.0.0/16 and 10.0.0.0/8, the IP 10.1.9.2 matches on 10.0.0.0/8 and IP 10.7.9.2 matches on 10.7.0.0/16 because 16 bits are the longest prefix matched.

In addition to the longest prefix match logic, LPM supports exact match (EM) logic on the `meta.u32`, inner destination MAC and VNI. Only index 1 is supported for `meta.u32`. Any
combination of these three fields can be chosen for EM. However, if inner destination MAC is chosen for LPM, then it should not be chosen for EM as well. If more than one field is chosen for EM, a logical AND is applied. Support for EM on meta allows working with any single field by copying its value to the meta.u32[1] on pipes before LPM. EM is performed at the same time as LPM matching (i.e., a logical AND is applied for both logics). For example, if there is a match on LPM logic, but the value in the fields chosen for EM is not exactly matched, this constitutes an LPM pipe miss.

To enable EM logic in an LPM pipe, two steps are required:

1. Provide match_mask to the LPM pipe creation with meta.u32[1] being fully masked and/or inner.eth.dst_mac and/or tun.vxlan_tun_id, while setting match_mask.tun.type to DOCA_FLOW_TUN_VXLAN. Thus, the match parameter is responsible for the choice of field for LPM logic, while the match_mask parameter is responsible for the enablement of EM logic. Separation into two parameters is done to distinguish which field is for LPM logic and which is for EM logic, when both fields can be used for LPM (e.g., destination IP address and source MAC address).

2. Per entry, provide values to do exact match using the match structure. match_mask is used only for LPM-related masks and is not involved into EM logic.

EM logic allows inserting many entries with different meta values for the same pair of LPM-related data. Regarding IPv4-based LPM logic with exact match enabled: LPM pipe can have 1.1.1.1/32 with meta 42, 555, and 1020. If a packet with 1.1.1.1/32 goes through such an LPM pipe, its meta value is compared against 42, 555, and 1020.

The actions and FWD of the DOCA Flow LPM pipe work the same as the basic DOCA Flow pipe.

**Note**

The monitor only supports non-shared counters in the LPM pipe.

doca_flow_pipe_lpm insertion max latency can be measured in milliseconds in some cases and, therefore, it is better to insert it from the control path. To get the best insertion performance, entries should be added in large batches.
**Note**

An LPM pipe cannot be a root pipe. You must create a pipe as root and forward the packets to the LPM pipe.

**Note**

An LPM pipe can only do LPM matching on inner and outer IP and MAC addresses.

**Note**

For monitoring, an LPM pipe only supports non-shared counters and does not support other capabilities of **doca_flow_monitor**.

**doca_flow_pipe_acl**

doca_flow_pipe_acl uses an access-control list (ACL) matching. ACL matching is five-tuple of the doca_flow_match. Each entry consists of a value and a mask (e.g., 10.0.0.0/8, 10.10.0.0/16, etc.) for IP address fields, port range, or specific port in the port fields, protocol, and priority of the entry.

**ACL entry port configuration:**

- Mask port is 0 ==> Any port
- Mask port is equal to match port ==> Exact port. Port with mask 0xffff.
- Mask port > match port ==> Match port is used as port from and mask port is used as port to

Monitor actions are not supported in ACL. FWD of the DOCA Flow ACL pipe works the same as the basic DOCA Flow pipe.

ACL supports the following types of FWD:

- **DOCA_FLOW_FWD_PORT**
- **DOCA_FLOW_FWD_PIPE**
- **DOCA_FLOW_FWD_DROP**

doca_flow_pipe_lpm insertion max latency can be measured in milliseconds in some cases and, therefore, it is better to insert it from the control path. To get the best insertion performance, entries should be added in large batches.

**Note**

An ACL pipe can be a root pipe.

**Note**

An ACL pipe can be in ingress and egress domain.

**Note**

An ACL pipe must be accessed on a single queue. Different ACL pipes may be accessed on different queues.
**Note**

Adding an entry to the ACL pipe after sending an entry with flag `DOCA_FLOW_NO_WAIT` is not supported.

**Note**

Removing an entry from an ACL pipe is not supported.

doca_flow_pipe_ordered_list

doca_flow_pipe_ordered_list allows the user to define a specific order of actions and multiply the same type of actions (i.e., specific ordering between counter/meter and encap/decap).

An ordered list pipe is defined by an array of actions (i.e., sequences of actions). Each entry can be an instance of these sequences. An ordered list pipe may consist of up to an array of 8 different actions. The maximum size of each action array is 4 elements. Resource allocation may be optimized when combining multiple action arrays in one ordered list pipe.

doca_flow_pipe_hash

doca_flow_pipe_hash allows the user to insert entries by index. The index represents the packet hash calculation.

An hash pipe gets doca_flow_match only on pipe creation and only mask. The mask provides all fields to be used for hash calculation.
The monitor, actions, actions_descs, and FWD of the DOCA Flow hash pipe works the same as the basic DOCA Flow pipe.

**Note**

The nb_flows in doca_flow_pipe_attr should be a power of 2.

## Hardware Steering Mode

Users can enable hardware steering mode by setting devarg `dv_flow_en` to 2.

The following is an example of running DOCA with hardware steering mode:

```
... –a 03:00.0, dv_flow_en=2 –a 03:00.1, dv_flow_en=2....
```

The following is an example of running DOCA with software steering mode:

```
... –a 03:00.0 –a 03:00.1 ....
```

The `dv_flow_en=2` means that hardware steering mode is enabled.

In the struct `doca_flow_cfg`, setting `mode_args` using `(doca_flow_cfg_set_mode_args())` represents DOCA applications. If it is set with `hws` (e.g., "vnf,hws", "switch,hws", "remmote_vnf,hws") then hardware steering mode is enabled.

In switch mode, `fdb_def_rule_en=0,vport_match=1,repr_matching_en=0,dv_xmeta_en=4` should be added to DPDK PMD devargs, which makes DOCA Flow switch module take over all the traffic.

To create an entry by calling `doca_flow_pipe_add_entry`, the parameter flags can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT`:
• **DOCA_FLOW_WAIT_FOR_BATCH** means that this flow entry waits to be pushed to hardware. Batch flows then can be pushed only at once. This reduces the push times and enhances the insertion rate.

• **DOCA_FLOW_NO_WAIT** means that the flow entry is pushed to hardware immediately.

The parameter *usr_ctx* is handled in the callback set in struct *doca_flow_cfg*.

doca_flow_entries_process processes all the flows in this queue. After the flow is handled and the status is returned, the callback is executed with the status and *usr_ctx*.

If the user does not set the callback in *doca_flow_cfg*, the user can get the status using *doca_flow_entry_get_status* to check if the flow has completed offloading or not.

**Isolated Mode**

In non-isolated mode (default) any received packets (following an RSS forward, for example) can be processed by the DOCA application, bypassing the kernel. In the same way, the DOCA application can send packets to the NIC without kernel knowledge. This is why, by default, no replies are received when pinging a host with a running DOCA application. If only specific packet types (e.g., DNS packets) should be processed by the DOCA application, while other packets (e.g., ICMP ping) should be handled directly the kernel, then isolated mode becomes relevant.

In isolated mode, packets that match root pipe entries are steered to the DOCA application (as usual) while other packets are received/sent directly by the kernel.

If you plan to create a pipe with matches followed by action/monitor/forward operations, due to functional/performance considerations, it is advised that root pipes entries include the matches followed by a next pipe forward operation. In the next pipe, all the planned matches actions/monitor/forward operations could be specified. Unmatched packets are received and sent by the kernel.
To activate isolated mode, two configurations are required:

1. DOCA configuration: Update the string member `mode_args` (struct doca_flow_cfg) using `doca_flow_cfg_set_mode_args()` which represents the DOCA application mode and add "isolated" (separated by comma) to the other mode arguments. For example:
   ```
   doca_flow_cfg_set_mode_args(cfg, "vnf,hws,isolated")
   doca_flow_cfg_set_mode_args(cfg, "switch,isolated")
   ```

2. DPDK configuration: Set `isolated_mode` to 1 (struct application_port_config). For example, if DPDK is initialized by the API: `dpdk_queues_and_ports_init(struct application_dpdk_config *app_dpdk_config)`.

```
struct application_dpdk_config app_dpdk_config = {
    .port_config = {
        .isolated_mode = 1,
        .nb_ports = ...
        ...
    },
    ...
};
```

### Pipe Resize

The move to HWS improves performance because rule insertion is implemented in hardware rather than software. However, this move imposes additional limitations, such as the need to commit in advance on the size of the pipes (the number of rule entries). For applications that require pipe sizes to grow over time, a static size can be challenging: Committing to a pipe size too small can cause the application to fail once the number of rule entries exceeds the committed number, and pre-committing to an excessively high number of rules can result in memory over-allocation.

This is where pipe resizing comes in handy. This feature allows the pipe size to increase during runtime with support for all entries in a new resized pipe.
Increasing Pipe Size

It is possible to set a congestion level by percentage (CONGESTION_PERCENTAGE). Once the number of entries in the pipe exceeds this value, a callback is invoked. For example, for a pipe with 1000 entries and a CONGESTION_PERCENTAGE of 80%, the CONGESTION_REACHED callback is invoked after the 800th entry is added.

Following the CONGESTION_REACHED callback, the application should call the pipe resize API (resize()). The following are optional callbacks during the resize callback:

- A callback on the new number of entries allocated to the pipe
- A callback on each entry that existed in the smaller pipe and is now allocated to the resized pipe

Upon completion of the internal transfer of all entries from the small pipe to the resized pipe, a RESIZED callback is invoked.

A CONGESTION_REACHED callback is received exactly once before the RESIZED callback. Receiving another CONGESTION_REACHED only happens after calling resize() and receiving its completion with a RESIZED callback.

List of Callbacks
• CONGESTION_REACHED – on the updated number of entries in the pipe (if pipe is resizable)

.getInfo

Receiving a CONGESTION_REACHED callback can occur after adding a small number of entries and for moving entries from a small to resized pipe. The application must always call pipe resize after receiving the CONGESTION_REACHED callback to handle such cases.

Note

Calling pipe resize returns immediately. It starts an internal process that ends later with the RESIZED callback.

• RESIZED – upon completion of the resize operation

• NR_ENTRIES_CHANGED (optional) – on the new max number of entries in the pipe

• ENTRY_RELOCATE (optional) – on each entry moved from the small pipe to the resized pipe

Order of Operations for Pipe Resizing

1. Set a process callback on flow configuration:

```c
struct doca_flow_cfg *flow_cfg;
doca_flow_cfg_create(&flow_cfg);
doca_flow_cfg_set_cb_pipe_process(flow_cfg, <pipe-process-callback>);
```
2. Set the following pipe attribute configurations:

```c
struct doca_flow_pipe_cfg *pipe_cfg;
doca_flow_pipe_cfg_create(&pipe_cfg, port);
doca_flow_pipe_cfg_set_nr_entries(pipe_cfg, <initial-number-of-entries>);
doca_flow_pipe_cfg_set_is_resizable(pipe_cfg, true);
doca_flow_pipe_cfg_set_congestion_level_threshold(pipe_cfg, <CONGESTION_PERCENTAGE>);
doca_flow_pipe_cfg_set_user_ctx(pipe_cfg, <pipe-user-context>);
```

3. Start adding entries:

```c
/* Basic pipe */
doca_flow_pipe_add_entry()
/* Control pipe */
doca_flow_pipe_control_add_entry()
```

4. Once the number of entries in the pipe crosses the congestion threshold, an OP_CONGESTION_REACHED operation callback is received.

5. Mark the pipe's congestion threshold event and, upon return, call doca_flow_pipe_resize(). For this call, add the following parameters:

- The new threshold percentage for calculating the new size.
- A callback on the new pipe size (optional):

```c
doca_flow_pipe_resize_nr_entries_changed_cb nr_entries_changed_cb
```
A callback on the entries to be transferred to the resized pipe:

```
doca_flow_pipe_resize_entry_relocate_cb entry_relocation_cb
```

6. Call `doca_flow_entries_process()` to trigger the transfer of entries. It is relevant for both a basic pipe and a control pipe.

7. At this phase, adding new entries to the pipe is permitted. The entries are added directly to the resized pipe and therefore do not need to be transferred.

8. Once all entries are transferred, an `OP_RESIZED` operation callback is received. Also, at this point a new `OP_CONGESTION_REACHED` operation callback can be received again.

9. At this point calling `doca_flow_entries_process()` can be stopped for a control pipe. For a basic pipe an additional call is required to complete the call to `doca_flow_pipe_add_entry()`.

**Info**

`doca_flow_entries_process()` has the following roles:

- Triggering entry transfer from the smaller to the bigger pipe (until an `OP_RESIZED` callback is received)
- Follow up API on previous `add_entries` API (basic pipe relevance only)

**Hairpin Configuration**

In switch mode, if `dev` is set in struct `doca_flow_port_cfg` (using `doca_flow_port_cfg_set_dev()`), then an internal hairpin is created for direct wire-to-wire fwd. Users may specify the hairpin configuration using `mode_args`. The supported options as follows:
• hairpinq_num=[n] – the hairpin queue number
• use_huge_mem – determines whether the Tx buffer uses hugepage memory
• lock_rx_mem – locks Rx queue memory

Teardown

Pipe Entry Teardown

When an entry is terminated by the user application or ages-out, the user should call the entry destroy function, doca_flow_pipe_rm_entry(). This frees the pipe entry and cancels hardware offload.

Pipe Teardown

When a pipe is terminated by the user application, the user should call the pipe destroy function, doca_flow_pipe_destroy(). This destroys the pipe and the pipe entries that match it.

When all pipes of a port are terminated by the user application, the user should call the pipe flush function, doca_flow_port_pipes_flush(). This destroys all pipes and all pipe entries belonging to this port.

⚠️ Warning

During doca_flow_pipe_destroy() execution, the application must avoid adding/removing entries or checking for aged entries of any other pipes.

Port Teardown
When the port is not used anymore, the user should call the port stop function, `doca_flow_port_stop()`. This stops the DOCA port, disables the traffic, destroys the port and frees all resources of the port.

**Flow Teardown**

When the DOCA Flow is not used anymore, the user should call the flow destroy function, `doca_flow_destroy()`. This releases all the resources used by DOCA Flow.

**Metadata**

- **Info**

  A scratch area exists throughout the pipeline whose maximum size is `DOCA_FLOW_META_MAX` bytes.

The user can set a value to metadata, copy from a packet field, then match in later pipes. Mask is supported in both match and modification actions.

The user can modify the metadata in different ways based on its actions' masks or descriptors:

- **ADD** – set metadata scratch value from a pipe action or an action of a specific entry. Width is specified by the descriptor.

- **COPY** – copy metadata scratch value from a packet field (including the metadata scratch itself). Width is specified by the descriptor.
Some DOCA pipe types (or actions) use several bytes in the scratch area for internal usage. So, if the user has set these bytes in PIPE-1 and read them in PIPE-2, and between PIPE-1 and PIPE-2 there is PIPE-A which also uses these bytes for internal purpose, then these bytes are overwritten by the PIPE-A. This must be considered when designing the pipe tree.

The bytes used in the scratch area are presented by pipe type in the following table:

<table>
<thead>
<tr>
<th>Pipe Type/Action</th>
<th>Bytes Used in Scratch</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered_list</td>
<td>[0, 1, 2, 3]</td>
</tr>
<tr>
<td>LPM</td>
<td>[0, 1, 2, 3]</td>
</tr>
<tr>
<td>LPM EM</td>
<td>[0, 1, 2, 3, 4, 5, 6, 7]</td>
</tr>
<tr>
<td>Mirror</td>
<td>[0, 1, 2, 3]</td>
</tr>
<tr>
<td>ACL</td>
<td>[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]</td>
</tr>
<tr>
<td>Fwd from ingress to egress</td>
<td>[0, 1, 2, 3]</td>
</tr>
</tbody>
</table>

**Packet Processing**

In situations where there is a port without a pipe defined, or with a pipe defined but without any entry, the default behavior is that all packets arrive to a port in the software.

Once entries are added to the pipe, if a packet has no match then it continues to the port in the software. If it is matched, then the rules defined in the pipe are executed.
If the packet is forwarded in RSS, the packet is forwarded to software according to the RSS definition. If the packet is forwarded to a port, the packet is redirected back to the wire. If the packet is forwarded to the next pipe, then the software attempts to match it with the next pipe.

Note that the number of pipes impacts performance. The longer the number of matches and actions that the packet goes through, the longer it takes the hardware to process it. When there is a very large number of entries, the hardware must access the main memory to retrieve the entry context which increases latency.

**Debug and Trace Features**

DOCA Flow supports trace and debugging of DOCA Flow applications which enable collecting predefined internal key performance indicators (KPIs) and pipeline visualization.

**Installation**

The set of DOCA's SDK development packages include also a developer-oriented package that includes additional trace and debug features which are not included in the production libraries:

- .deb based systems – libdoca-sdk-flow-trace
- .rpm based systems – doca-sdk-flow-trace
These packages install the trace-version of the libraries under the following directories:

- .deb based systems – /opt/mellanox/doca/lib/<arch>/trace
- .rpm based systems – /opt/mellanox/doca/lib64/trace

**Using Trace Libraries**

The trace libraries are designed to allow a user to link their existing (production) program to the trace library without needing to recompile the program. To do so, one should simply update the matching environment variable so that the OS will prioritize loading libraries from the above trace directory:

```
LD_LIBRARY_PATH=/opt/mellanox/doca/lib/aarch64-linux-gnu/trace:${LD_LIBRARY_PATH}
doca_ipsec_security_gw <program parameters>
```

**Trace Features**

**DOCA Log – Trace Level**

DOCA’s trace logging level (`DOCA_LOG_LEVEL_TRACE`) is compiled as part of this trace version of the library. That is, any program compiled against the library can activate this additional logging level through DOCA’s API or even through DOCA’s built-in argument parsing (ARGP) library:

```
LD_LIBRARY_PATH=/opt/mellanox/doca/lib/aarch64-linux-gnu/trace:${LD_LIBRARY_PATH}
doca_ipsec_security_gw <program parameters> --sdk-log-level 70
```

**DOCA Flow Samples**

This section provides DOCA Flow sample implementation on top of the BlueField.
Sample Prerequisites

A DOCA Flow-based program can either run on the host machine or on the BlueField.

Flow-based programs require an allocation of huge pages, hence the following commands are required:

```bash
echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
sudo mkdir /mnt/huge
sudo mount -t hugetlbfs nodev /mnt/huge
```

Note

On some OSs (RockyLinux, OpenEuler, CentOS 8.2), the default huge page size on the BlueField (and Arm hosts) is larger than 2MB, often 512MB. Users can check the size of the huge pages on their OS using the following command:

```bash
$ grep -i huge /proc/meminfo
```

```
AnonHugePages:   0 kB
ShmemHugePages:  0 kB
FileHugePages:   0 kB
HugePages_Total:        4
HugePages_Free:        4
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:          524288 kB
```
Running the Sample

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

   ```
cd /opt/mellanox/doca/samples/doca_flow/<sample_name>
meson /tmp/build
ninja -C /tmp/build
   ```

   **Note**

   The binary `doca_<sample_name>` will be created under `/tmp/build/`.

3. Sample (e.g., `flow_aging`) usage:

   ```
   echo '4' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-524288kB/nr_hugepages
   ```
Usage: doca_flow_aging [DPDK Flags] -- [DOCA Flags]

DOCA Flags:
- `--help`   Print a help synopsis
- `--version` Print program version information
- `--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
- `--json <path>` Parse all command flags from an input json file

4. For additional information per sample, use the `--help` option after the `--` separator:

```
/tmp/build/doca_<sample_name> -- --help
```

5. DOCA Flow samples are based on DPDK libraries. Therefore, the user is required to
provide DPDK flags. The following is an example from an execution on the DPU:

- **CLI example for running the samples with "vnf" mode:**

```
/tmp/build/doca_<sample_name> -a auxiliary:mlx5_core.sf.2 -a auxiliary:mlx5_core.sf.3 -- --l 60
```

- **CLI example for running the VNF samples with vnf,hws mode:**

```
/tmp/build/doca_<sample_name> -a auxiliary:mlx5_core.sf.2,dv_flow_en=2 -a auxiliary:mlx5_core.sf.3,dv_flow_en=2 -- --l 60
```

- **CLI example for running the switch samples with switch,hws mode:**

```
/tmp/build/doca_<sample_name> -- -p 03:00.0 -r sf[2-3] --l 60
```
**Note**

When running on the BlueField with `switch,hws mode`, it is not necessary to configure the OVS.

DOCA switch sample hides the extra `fdb_def_rule_en=0,vport_match=1,repr_matching_en=0,dv_xmeta_en=4` DPDK devargs with a simple `-p` and `-r` to specify the PCIe ID and representor information.

**Note**

When running on the DPU using the command above, sub-functions must be enabled according to the NVIDIA BlueField DPU Scalable Function User Guide.

**Note**

When running on the host, virtual functions must be used according to the instructions in the NVIDIA DOCA Virtual Functions User Guide.

## Samples

### Flow ACL

This sample illustrates how to use the access-control list (ACL) pipe.

The sample logic includes:
1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   
   1. Building an ACL pipe that matches changeable:
      
      1. Source IPv4 address
      2. Destination IPv4 address
      3. Source port
      4. Destination port
   
   2. Adding four example 5-tuple entries:
      
      1. The first entry with:
         
         - Full mask on source IPv4 address
         - Full mask on destination IPv4 address
         - Null mask on source port (any source port)
         - Null mask on destination port (any destination port)
         - TCP protocol
         - Priority 10
         - Action "deny" (drop action)
      
      2. The second entry with:
         
         - Full mask on source IPv4 address
         - Full mask on destination IPv4 address
         - Null mask on source port (any source port)
         - Value set in mask on destination port is used as part of port range:
- Destination port in match is used as port from
- Destination port in mask is used as port to
  - UDP protocol
  - Priority 50
  - Action "allow" (forward port action)

3. The third entry with:
  - Full mask on source IPv4 address
  - Full mask on destination IPv4 address
  - Value set in mask on source port is equal to the source port in match. It is the exact port. ACL uses the port with full mask.
  - Null mask on destination port (any destination port)
  - TCP protocol
  - Priority 40
  - Action "allow" (forward port action)

4. The fourth entry with:
  - 24-bit mask on source IPv4 address
  - 24-bit mask on destination IPv4 address
  - Value set in mask on source port is used as part of port range: source port in match is used as port from, source port in mask is used as port to.
  - Value set in mask on destination port is equal to the destination port in match. It is the exact port. ACL uses the port with full mask.
  - TCP protocol
3. The sample shows how to run the ACL pipe on ingress and egress domains. To change the domain, use the global parameter `flow_acl_sample.c`.

1. Ingress domain: ACL is created as root pipe

2. Egress domain:
   - Building a control pipe with one entry that forwards the IPv4 traffic hairpin port.
   - ACL is created as a root pipe on the hairpin port.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_acl/flow_acl_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_acl/flow_acl_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_acl/meson.build`

**Flow Aging**

This sample illustrates the use of DOCA Flow's aging functionality. It demonstrates how to build a pipe and add different entries with different aging times and user data.

The sample logic includes:

1. Initializing DOCA Flow with `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow port.

3. On each port:
   1. Building a pipe with changeable 5-tuple match and forward port action.
   2. Adding 10 entries with different 5-tuple match, a monitor with different aging time (5-60 seconds), and setting user data in the monitor. The user data will
contain the port ID, entry number, and entry pointer.

4. Handling aging every 5 seconds and removing each entry after age-out.

5. Running these commands until all entries age out.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_aging/flow_aging_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_aging/flow_aging_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_aging/meson.build

**Flow Control Pipe**

This sample shows how to use the DOCA Flow control pipe and decap action.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:

   1. Building VXLAN pipe with match on VNI field, decap action, action descriptor for decap, and forwarding the matched packets to the second port.

   2. Building VXLAN-GPE pipe with match on VNI plus next protocol fields, and forwarding the matched packets to the second port.

   3. Building GRE pipe with match on GRE key field, decap and build eth header actions, action descriptor for decap, and forwarding the matched packets to the second port.

   4. Building NVGRE pipe with match on protocol is 0x6558, `vs_id`, `flow_id`, and inner UDP source port fields, and forwarding the matched packets to the second port. This pipe has a higher priority than the GRE pipe. The NVGRE packets are matched first.
5. Building MPLS pipe with match on third MPLS label field, decap and build eth header actions, action descriptor for decap, and forwarding the matched packets to the second port.

6. Building a control pipe with the following entries:

- If L4 type is UDP and destination port is 4789, forward to VXLAN pipe
- If L4 type is UDP and destination port is 4790, forward to VXLAN-GPE pipe
- If L4 type is UDP and destination port is 6635, forward to MPLS pipe
- If tunnel type and L4 type is GRE, forward to GRE pipe

**Note**

When any tunnel is decapped, it is user responsibility to identify if it is an L2 or L3 tunnel within the action. If the tunnel is L3, the complete outer layer, tunnel, and inner L2 are removed and the inner L3 layer is exposed. To keep the packet valid, the user should provide the ETH header to encap the inner packet. For example:

```c
actions.decap_type = DOCA_FLOW_RESOURCE_TYPE_NON_SHARED;
actions.decap_cfg.is_l2 = false;
/* append eth header after decap GRE tunnel */
SET_MAC_ADDR(actions.decap_cfg.eth.src_mac, src_mac[0], src_mac[1], src_mac[2], src_mac[3], src_mac[4], src_mac[5]);
SET_MAC_ADDR(actions.decap_cfg.eth.dst_mac, dst_mac[0], dst_mac[1], dst_mac[2], dst_mac[3], dst_mac[4], dst_mac[5]);
actions.decap_cfg.eth.type = DOCA_FLOW_L3_TYPE_IP4;
```

For a VXLAN tunnel, since VXLAN is a L2 tunnel, the user must indicate it within the action:

```c
actions.decap_type = DOCA_FLOW_RESOURCE_TYPE_NON_SHARED;
```
Flow Copy to Meta

This sample shows how to use the DOCA Flow copy-to-metadata action to copy the source MAC address and then match on it.

The sample logic includes:

1. Initializing DOCA Flow by indicating `ode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Building a pipe with changeable match on `meta_data` and forwarding the matched packets to the second port.
   2. Adding an entry that matches an example source MAC that has been copied to metadata.
   3. Building a pipe with changeable 5-tuple match, copying source MAC action, and `fwd` to the first pipe.
   4. Adding example 5-tuple entry to the pipe.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_control_pipe/flow_copy_to_meta_sample.c`
Flow Add to Metadata

This sample shows how to use the DOCA Flow add-to-metadata action to accumulate the source IPv4 address for double to meta and then match on the meta.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Building a pipe with changeable match on `meta_data` and forwarding the matched packets to the second port.
   2. Adding an entry that matches an example double of source IPv4 address that has been added to metadata.
   3. Building a pipe with changeable 5-tuple match, copying the source IPv4, and adding the value again to the meta action, and forwarding to the first pipe.
   4. Adding an example 5-tuple entry to the pipe.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_add_to_meta/flow_add_to_meta_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_add_to_meta/flow_add_to_meta_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_add_to_meta/meson.build

Flow Drop
This sample illustrates how to build a pipe with 5-tuple match, forward action drop, and forward miss action to the hairpin pipe. The sample also demonstrates how to dump pipe information to a file and query entry.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Building a hairpin pipe with an entry that matches all traffic and forwarding traffic to the second port.
   2. Building a pipe with a changeable 5-tuple match, forwarding action drop, and miss forward to the hairpin pipe. This pipe serves as a root pipe.
   3. Adding an example 5-tuple entry to the drop pipe with a counter as monitor to query the entry later.

4. Waiting 5 seconds and querying the drop entry (total bytes and total packets).

5. Dumping the pipe information to a file.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_drop/flow_drop_sample_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_drop/flow_drop_sample_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_drop/meson.build`

**Flow ECMP**

This sample illustrates ECMP feature using a hash pipe.

The sample enables users to determine how many port are included in ECMP distribution:
• The number of ports, $n$, is determined by DPDK device argument `representor=sf[0-m]` where $m=n-1$.

• CLI example for running this samples with $n=4$ ports:

```
/tmp/build/doca_flow_ecmp -- -p 03:00.0 -r sf[0-3] -l 60 --sdk-log-level 60
```

• $n$ should be power of 2. Max supported value is $n=8$.

The sample logic includes:

1. Calculate the number of SF representors ($n$) created by DPDK according to user input.

2. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` structure.


4. On switch port:

   1. Constructing a hash pipe that signifies the `match_mask` structure to compute the hash based on the outer IPv6 flow label field.

   2. Adding $n$ entries to the created pipe, each of which forwards packets to a different port representor.

5. Waiting 15 seconds and querying the entries.

6. Print the ECMP results per port (number packets in each port related to total packets).

Reference:

• `/opt/mellanox/doca/samples/doca_flow/flow_ecmp/flow_ecmp_sample.c`

• `/opt/mellanox/doca/samples/doca_flow/flow_ecmp/flow_ecmp_main.c`

• `/opt/mellanox/doca/samples/doca_flow/flow_ecmp/meson.build`
**Flow ESP**

This sample illustrates how to match ESP fields in two ways:

- Exact match for both `esp_spi` and `esp_en` fields using the `doca_flow_match` structure.
- Comparison match for `esp_en` field using the `doca_flow_match_condition` structure.

![Note]

This sample is supported for ConnectX-7, BlueField-3, and above.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
   1. Building a control pipe with entry that match `esp_en > 3` (GT pipe).
   2. Building a control pipe with entry that match `esp_en < 3` (LT pipe).
   3. Building a root pipe with changeable `next_pipe` FWD and `esp_spi` match along with specific `esp_sn` match + IPv4 and ESP exitance (matching `parser_meta`).
   4. Adding example `esp_spi = 8` entry to the root pipe which forwards to GT pipe (and miss condition).
   5. Adding example `esp_spi = 5` entry to the root pipe which forwards to LT pipe (and hit condition).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_esp/flow_esp_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_esp/flow_esp_main.c`
Flow Forward Miss

The sample illustrates how to use FWD miss query and update with or without miss counter.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Building a copy pipe with a changeable outer L3 type match and forwarding traffic to the second port.
   2. Add entries doing different copy action depending on the outer L3 type:
      1. IPv4 – copy IHL field into Type Of Service field.
      2. IPv6 – copy Payload Length field into Traffic Class field.
   3. Building a pipe with a IPv4 addresses match, forwarding traffic to the second port, and miss forward to the copy pipe.
   4. Building an IP selector pipe with outer L3 type match, forwarding IPv4 traffic to IPv4 pipe, and miss forward to the copy pipe with miss counter.
   5. Building a root pipe with outer L3 type match, forwarding IPv4 and IPv6 traffic to IP selector pipe, and dropping all other traffic by miss forward with miss counter.

4. Waiting 5 seconds for first batch of traffic.

5. On each port:
   1. Querying the miss counters using `doca_flow_query_pipe_miss` API.
   2. Printing the miss results.
6. On each port:
   
   1. Building a push pipe that pushes VLAN header and forwarding traffic to the second port.
   
   2. Updating both IP selector and IPv4 pipes miss FWD pipe target to push pipe using doca_flow_pipe_update_miss API.

7. Waiting 5 seconds for second batch of traffic, same flow as before.

8. On each port:
   
   1. Querying again the miss counters using doca_flow_query_pipe_miss API.
   
   2. Printing the miss results again, the results should include miss packets coming either before or after miss action updating.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_fwd_miss/flow_fwd_miss_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_fwd_miss/flow_fwd_miss_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_fwd_miss/meson.build

Flow Forward Target (DOCA_FLOW_TARGET_KERNEL)

The sample illustrates how to use DOCA_FLOW_FWD_TARGET type of forward, as well as the doca_flow_get_target API to obtain an instance of struct doca_flow_target.

The sample logic includes:

1. Initializing DOCA Flow with "vnf,isolated,hws".

2. Initializing two ports.

3. Obtaining an instance of doca_flow_target by calling doca_flow_get_target(DOCA_FLOW_TARGET_KERNEL, &kernel_target);

4. On each port, creating:
1. Non-root basic pipe with 5 tuple match.
   
   1. If hit – forward the packet to another port.
   
   2. If miss – forward the packet to the kernel for processing by using the instance of `doca_flow_target` obtained in previous steps.
   
   3. Then add a single entry with a specific 5-tuple which is hit, and the rest is forwarded to the kernel.

2. Root control pipe with a match on outer L3 type being IPv4.
   
   1. If hit – forward the packet to the non-root pipe.
   
   2. If miss – drop the packet.
   
   3. Add a single entry that implements the logic described.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_target/flow_fwd_target_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_target/flow_fwd_target_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_fwd_target/meson.build`

**Flow GENEVE Encap**

This sample illustrates how to use DOCA Flow actions to create a GENEVE tunnel.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   
   1. Building ingress pipe with changeable 5-tuple match, copying to `pkt_meta` action, and forwarding port action.
2. Building egress pipe with `pkt_meta` match and 4 different encapsulation actions:
   - L2 encap without options
   - L2 encap with options
   - L3 encap without options
   - L3 encap with options

3. Adding example 5-tuple and encapsulation values entries to the pipes.

Reference:
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/flow_geneve_encap_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/flow_geneve_encap_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_geneve_encap/meson.build`

**Flow GENEVE Options**

This sample illustrates how to prepare a GENEVE options parser, match on configured options, and decap GENEVE tunnel.

Note

This sample works only with PF. VFs and SFs are not supported.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
1. Building GENEVE options parser, same input for all ports.

2. Building match pipe with GENEVE VNI and options match and forwards decap pipe.

3. Building decap pipe with more GENEVE options match, and 2 different decapsulation actions:
   - L2 decap
   - L3 decap with changeable mac addresses

4. Adding example GENEVE options and MAC address values entries to the pipes.

Reference:
- /opt/mellanox/doca/samples/doca_flow/flow_geneve_opt/flow_geneve_opt_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_geneve_opt/flow_geneve_opt_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_geneve_opt/meson.build

**Flow Hairpin VNF**

This sample illustrates how to build a pipe with 5-tuple match and to forward packets to the other port.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Building a pipe with changeable 5-tuple match and forwarding port action.
   2. Adding example 5-tuple entry to the pipe.

Reference:
Flow Switch to Wire

This sample illustrates how to build a pipe with 5-tuple match and forward packets from the wire back to the wire.

The sample shows how to build a basic pipe in a switch and hardware steering (HWS) mode. Each pipe contains two entries, each of which forwards matched packets to two different representors.

The sample also demonstrates how to obtain the switch port of a given port using `doca_flow_port_switch_get()`.

Note

The test requires one PF with three representors (either VFs or SFs).

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.

2. Starting DOCA Flow ports with `doca_dev` in struct `doca_flow_port_cfg`.

3. On the switch’s PF port:
   1. Building ingress, egress, vport, and RSS pipes with changeable 5-tuple match and forwarding port action.

   2. Adding example 5-tuple entry to the pipe.
3. The matched traffic goes to its destination port, the missed traffic is handled by the rx\_tx function and is sent to a dedicate port based on the protocol.

- **Ingress pipe:**

  Entry 0: IP src 1.2.3.4 / TCP src 1234 dst 80 -> egress pipe
  Entry 1: IP src 1.2.3.5 / TCP src 1234 dst 80 -> vport pipe

- **Egress pipe (test ingress to egress cross domain):**

  Entry 0: IP dst 8.8.8.8 / TCP src 1234 dst 80 -> port 0
  Entry 1: IP dst 8.8.8.9 / TCP src 1234 dst 80 -> port 1
  Entry 2: IP dst 8.8.8.10 / TCP src 1234 dst 80 -> port 2
  Entry 3: IP dst 8.8.8.11 / TCP src 1234 dst 80 -> port 3

- **Vport pipe (test ingress direct to vport):**

  Entry 0: IP dst 8.8.8.8 / TCP src 1234 -> port 0
  Entry 1: IP dst 8.8.8.9 / TCP src 1234 -> port 1
  Entry 2: IP dst 8.8.8.10 / TCP src 1234 -> port 2
  Entry 3: IP dst 8.8.8.11 / TCP src 1234 -> port 3

- **RSS pipe (test miss traffic port\_id get and destination port\_id set):**

  Entry 0: IPv4 / TCP -> port 0
  Entry 0: IPv4 / UDP -> port 1
  Entry 0: IPv4 / ICMP -> port 2

**Reference:**

- /opt/mellanox/doca/samples/doca_flow/flow_switch_to_wire/flow_switch_to_wire_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_switch_to_wire/flow_switch_to_wire_main.c
Flow Hash Pipe

This sample illustrates how to build a hash pipe in hardware steering (HWS) mode.

The hash pipe contains two entries, each of which forwards "matched" packets to two different SF representors. For each received packet, the hash pipe calculates the entry index to use based on the IPv4 destination address.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.

2. Starting DOCA Flow ports: Physical port and two SF representors.

3. On switch port:
   
   1. Building a hash pipe while indicating which fields to use to calculate the hash in the `struct match_mask`.

   2. Adding two entries to the created pipe, each of which forwards packets to a different port representor.

4. Printing the hash result calculated by the software with the following message: "hash value for" for dest ip = 192.168.1.1.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_hash_pipe/flow_hash_pipe_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_hash_pipe/flow_hash_pipe_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_hash_pipe/meson.build`

Flow IPv6 Flow Label

This sample shows how to use DOCA Flow actions to update IPv6 flow label field after encapsulation.
As a side effect, it shows also example for IPv6 + MPLS encapsulation.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Building an ingress pipe with changeable L4 type and ports matching, which updates metadata and goes to the peer port.
   2. Adding example UDP/TCP type and ports and metadata values entries to the pipe. This pipe is L3 type agnostic.
   3. Building an egress pipe on the peer port with changeable metadata matching, which encapsulates packets with IPv6 + MPLS headers, and goes to the next pipe.
   4. Adding entries to the pipe, with different encapsulation values for different metadata values.
   5. Building another egress pipe on the peer port with changeable L3 inner type matching, which copies value into outer IPv6 flow label field.
   6. Adding two entries to the pipe:
      1. L3 inner type is IPv6 - copy IPv6 flow label from inner to outer.
      2. L3 inner type is IPv6 - copy outer IPv6 flow label from metadata.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ipv6_flow_label/flow_ipv6_flow_label_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ipv6_flow_label/flow_ipv6_flow_label_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ipv6_flow_label/meson.build`
Flow Loopback

This sample illustrates how to implement packet re-injection, or loopback, in VNF mode.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   
   1. Building a UDP pipe that matches a changeable source and destination IPv4 address, while the forwarding component is RSS to queues. Upon match, setting the packet meta on this UDP pipe which is referred to as an `RSS_UDP_IP` pipe.

   2. Adding one entry to the `RSS_UDP_IP` pipe that matches a packet with a specific source and destination IPv4 address and setting the meta to 10.

   3. Building a TCP pipe that matches changeable 4-tuple source and destination IPv4 and port addresses, while the forwarding component is RSS to queues (this pipe is called `RSS_TCP_IP` and it is the root pipe on ingress domain).

   4. Adding one entry to the `RSS_TCP_IP` pipe, that matches a packet with a specific source and destination port and IPv4 addresses.

   5. On the egress domain, creating the loopback pipe, which is root, and matching TCP over IPv4 with changeable 4-tuple source and destination port and IPv4 addresses, while encapsulating the matched packets with VXLAN tunneling and setting the destination and source MAC addresses to be changeable per entry.

   6. Adding one entry to the loopback pipe with specific values for the match and actions part while setting the destination MAC address to the port to which to inject the packet (in this case, it is the ingress port where the packet arrived).

   7. Starting to receive packets loop and printing the metadata

   - For packets that were re-injected, metadata equaling 10 is printed
Otherwise, 0 is be printed as metadata (indicating that it is the first time the packet has been encountered)

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_loopback/flow_loopback_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_loopback/flow_loopback_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_loopback/meson.build

Flow LPM

This sample illustrates how to use LPM (Longest Prefix Match) pipe

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
   1. Building an LPM pipe that matches changeable source IPv4 address.
   2. Adding two example 5-tuple entries:
      1. The first entry with full mask and forward port action
      2. The second entry with 16-bit mask and drop action
   3. Building a control pipe with one entry that forwards IPv4 traffic to the LPM pipe.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_lpm/flow_lpm_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_lpm/flow_lpm_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_lpm/meson.build
Flow LPM with exact match (EM)

This sample illustrates how to use LPM (Longest Prefix Match) pipe with exact match logic (EM) enabled.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Building LPM pipe that matches changeable source IPv4 address (using `match`) with exact-match logic on `meta.u32[1]` and the inner destination MAC and VNI (using `match_mask`).

2. Adding five entries to the LPM:
   1. Default entry with IPv4 subnet 0 to drop the packets which are unmatched in LPM with EM

   2. Fully masked 1.2.3.4 IPv4 address with meta value 1, inner destination mac 1:1:1:1:1, VNI 0xabcde1 to forward to the next port

   3. Fully masked 1.2.3.4 IPv4 address with meta value 2, inner destination mac 2:2:2:2:2, VNI 0xabcde2 to forward to the next port

   4. Fully masked 1.2.3.4 IPv4 address with meta value 3, inner destination mac 3:3:3:3:3, VNI 0xabcde3 to drop

   5. First 16 bit masked 1.2.0.0 IPv4 address with meta value 3, inner destination mac 3:3:3:3:3, VNI 0xabcde3 to forward to the next port

3. Building basic root pipe which matches everything, copies the `outer.eth_vlan0.tci` value to the `meta.u32[1]` and forwards the packet to the LPM pipe.

4. Adding single entry to the main pipe.
The sample uses the counters to show the packets per entry. Here are the packets that can be used for the test and the expected response of the sample to them:

- Ether()/Dot1Q(vlan=1)/IP(src="1.2.3.4")/UDP(dport=4789)/VXLAN(vni=0xabced30)/Ether(dst="1:1:1:1:1") – to be forwarded to next port by entry number 1

- Ether()/Dot1Q(vlan=2)/IP(src="1.2.3.4")/UDP(dport=4789)/VXLAN(vni=0xabced30)/Ether(dst="2:2:2:2:2") – to be forwarded to next port by entry number 2

- Ether()/Dot1Q(vlan=3)/IP(src="1.2.3.4")/UDP(dport=4789)/VXLAN(vni=0xabced30)/Ether(dst="3:3:3:3:3") – to be dropped by entry number 3

- Ether()/Dot1Q(vlan=3)/IP(src="1.2.125.125")/UDP(dport=4789)/VXLAN(vni=0xabced30)/Ether(dst="3:3:3:3:3") – to be forwarded to next port by entry number 4

- Ether()/Dot1Q(vlan=5)/IP(src="1.2.125.125")/UDP(dport=4789)/VXLAN(vni=0xabced30)/Ether(dst="42:42:42:42:42") – to be dropped by entry number 0 (default)

- Ether()/Dot1Q(vlan=1)/IP(src="1.2.3.4")/UDP(dport=4789)/VXLAN(vni=0xabced30)/Ether(dst="1:1:1:1:2") – to be dropped by entry number 0 (default)

- Ether()/Dot1Q(vlan=1)/IP(src="1.2.3.4")/UDP(dport=4789)/VXLAN(vni=0xabced30)/Ether(dst="1:1:1:1:1") – to be dropped by entry number 0 (default)

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_lpm_em/flow_lpm_em_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_lpm_em/flow_lpm_em_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_lpm_em/meson.build

**Flow Modify Header**

This sample illustrates how to use DOCA Flow actions to modify the specific packet fields.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.
3. On each port, creating serial pipes and jumping to the next pipe if traffic is unmatched:

1. Building a pipe with action `dec_ttl=true` and changeable `mod_dst_mac`. The pipe matches IPv4 traffic with a changeable destination IP and forwards the matched packets to the second port.
   - Adding an entry with an example destination IP (8.8.8.8) and `mod_dst_mac` value.

2. Building a pipe with action-changeable `mod_vxlan_tun_rsvd1`. The pipe matches IPv4 traffic with a changeable UDP destination port and VXLAN-GPE tunnel ID then forwards the matched packets to the second port.
   - Adding an entry with an example VXLAN-GPE tunnel ID (100) and UDP destination port (4790), then `mod_vxlan_tun_rsvd1` value.

3. Building a pipe with action-changeable `mod_vxlan_tun_rsvd1`. The pipe matches IPv4 traffic with a changeable UDP destination port and VXLAN tunnel ID then forwards the matched packets to the second port.
   - Adding an entry with an example VXLAN tunnel ID (100) and UDP destination port (4789), then `mod_vxlan_tun_rsvd1` value.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_modify_header/flow_modify_header_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_modify_header/flow_modify_header_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_modify_header/meson.build

**Flow Monitor Meter**

This sample illustrates how to use DOCA Flow monitor meter.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.

3. On each port:

   1. Building a pipe with monitor meter flag and changeable 5-tuple match. The pipe forwards the matched packets to the second port.

   2. Adding an entry with an example CIR and CBS values.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/flow_monitor_meter_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/flow_monitor_meter_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_monitor_meter/meson.build`

**Flow Multi-actions**

This sample shows how to use a DOCA Flow array of actions in a pipe.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:

   1. Building a pipe with changeable source IP match which forwards the matched packets to the second port and sets different actions in the actions array:

      - Changeable modify source MAC address
      - Changeable modify source IP address

   2. Adding two entries to the pipe with different source IP match:

      1. The first entry with an example modify source MAC address.
      2. The second with a modify source IP address.
Flow Multi-fwd

This sample shows how to use a different forward in pipe entries.

The sample logic includes:

1. Initializing DOCA Flow by indicating \texttt{mode_args=\textasciitilde\texttt{vnf,hws}} in the \texttt{doca_flow_cfg} struct.

2. Starting two DOCA Flow ports.

3. On each port:

   1. Building a pipe with changeable source IP match and sending NULL in the forward.

   2. Adding two entries to the pipe with different source IP match, and different forward:

      - The first entry with forward to the second port
      - The second with drop

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/flow_multi_fwd_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/flow_multi_fwd_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_multi_fwd/meson.build

Flow Ordered List
This sample shows how to use a DOCA Flow ordered list pipe.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:

   1. Building a root pipe with changeable 5-tuple match and forwarding to an ordered list pipe with a changeable index.

   2. Adding two entries to the pipe with an example value sent to a different index in the ordered list pipe.

   3. Building ordered list pipe with two lists, one for each entry:

      - First list uses meter and then shared counter
      - Second list uses shared counter and then meter

4. Waiting 5 seconds and querying the entries (total bytes and total packets).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ordered_list/flow_ordered_list_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ordered_list/flow_ordered_list_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ordered_list/meson.build`

**Flow Parser Meta**

This sample shows how to use some of `match.parser_meta` fields from 3 families:

- IP fragmentation – matching on whether a packet is IP fragmented
- Integrity bits – matching on whether a specific protocol is OK (length, checksum etc.)
- Packet types – matching on a specific layer packet type
The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Building a root pipe with outer IP fragmentation match:
      - If a packet is IP fragmented – forward it to the second port regardless of next pipes in the pipeline
      - If a packet is not IP fragmented – proceed with the the pipeline by forwarding it to integrity pipe
   
2. Building an "integrity" pipe with a single entry which continues to the next pipe when:
   - The outer IPv4 checksum is OK
   - The inner L3 is OK (incorrect length should be dropped)

3. Building a "packet type" pipe which forwards packets to the second port when:
   - The outer L3 type is IPv4
   - The inner L4 type is either TCP or UDP

4. Waiting 5 seconds for traffic to arrive.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_parser_meta/flow_parser_meta_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_parser_meta/flow_parser_meta_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_parser_meta/meson.build`

**Flow Random**
This sample shows how to use `match.parser_meta.random` field for 2 different use-cases:

- Sampling – sampling certain percentage of traffic regardless of flow content
- Distribution – distributing traffic in 8 different queues

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
   1. Building a root pipe with changeable 5-tuple match and forwarding to specific use-case pipe according to changeable source IP address.
   2. Adding two entries to the pipe with different source IP match, and different forward:
      - The first entry with forward to the sampling pipe.
      - The second entry with forward to the distribution pipe.
   3. Building a "sampling" pipe with a single entry and preparing the entry to sample 12.5% of traffic.
   4. Building a "distribution" hash pipe with 8 entries and preparing the entries to get 12.5% of traffic for each queue.
4. Waiting 15 seconds and querying the entries (total packets after sampling/distribution related to total packets before).

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_random/flow_random_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_random/flow_random_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_random/meson.build`
Flow RSS ESP

This sample shows how to use DOCA Flow forward RSS according to ESP SPI field, and distribute the traffic between queues.

The sample logic includes:

1. Initializing DOCA Flow by indicating mode_args="vnf,hws" in the doca_flow_cfg struct.
2. Starting two DOCA Flow ports.
3. On each port:
   1. Building a pipe with both L3 and L4 types match, copy the SPI field into packet meta data, and forwarding to RSS with 7 queues.
   2. Adding an entry with both IPv4 and ESP existence matching.
4. Waiting 15 seconds for traffic to arrived.
5. On each port:
   1. Calculates the traffic percentage distributed into each port and prints the result.
   2. Printing for each packet its SPI value. (only in debug mode, -l ≥ 60)

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_rss_esp/flow_rss_esp_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_rss_esp/flow_rss_esp_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_rss_esp/meson.build

Flow RSS Meta

This sample shows how to use DOCA Flow forward RSS, set meta action, and then retrieve the matched packets in the sample.
The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Building a pipe with a changeable 5-tuple match, forwarding to RSS queue with index 0, and setting changeable packet meta data.
   2. Adding an entry with an example 5-tuple and metadata value to the pipe.

4. Retrieving the packets on both ports from a receive queue, and printing the packet metadata value.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/flow_rss_meta_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/flow_rss_meta_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_rss_meta/meson.build`

**Flow Sampling**

This sample shows how to sample certain percentage of traffic regardless of flow content using `doca_flow_match_condition` structure with `parser_meta.random.value` field string.

*Note*

This sample is supported for ConnectX-7/BlueField-3 and above.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Starting DOCA Flow ports: Physical port and two SF representors.

3. On switch port:
   1. Building a root pipe with changeable 5-tuple match and forwarding to sampling pipe.
   2. Adding entry with an example 5-tuple to the pipe.
   3. Building a "sampling" control pipe with a single entry.
   4. calculating the requested random value for getting 35% of traffic.
   5. Adding entry with an example condition random value to the pipe.

4. Waiting 15 seconds and querying the entries (total packets after sampling related to total packets before).

Reference:
- /opt/mellanox/doca/samples/doca_flow/flow_sampling/flow_sampling_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_sampling/flow_sampling_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_sampling/meson.build

**Flow Set Meta**

This sample shows how to use the DOCA Flow set metadata action and then match on it.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Building a pipe with a changeable match on metadata and forwarding the matched packets to the second port.
2. Adding an entry that matches an example metadata value.

3. Building a pipe with changeable 5-tuple match, changeable metadata action, and fwd to the first pipe.

4. Adding entry with an example 5-tuple and metadata value to the pipe.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_set_meta/flow_set_meta_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_set_meta/flow_set_meta_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_set_meta/meson.build

**Flow Shared Counter**

This sample shows how to use the DOCA Flow shared counter and query it to get the counter statistics.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. On each port:

   1. Binding the shared counter to the port.

   2. Building a pipe with changeable 5-tuple match with UDP protocol, changeable shared counter ID and forwarding the matched packets to the second port.

   3. Adding an entry with an example 5-tuple match and shared counter with ID=port_id.

   4. Building a pipe with changeable 5-tuple match with TCP protocol, changeable shared counter ID and forwarding the matched packets to the second port.
5. Adding an entry with an example 5-tuple match and shared counter with ID=port_id.

6. Building a control pipe with the following entries:
   - If L4 type is UDP, forwards the packets to the UDP pipe
   - If L4 type is TCP, forwards the packets to the TCP pipe

4. Waiting 5 seconds and querying the shared counters (total bytes and total packets).

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_shared_counter/flow_shared_counter_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_shared_counter/flow_shared_counter_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_shared_counter/meson.build

Flow Shared Meter

This sample shows how to use the DOCA Flow shared meter.

The sample logic includes:

1. Initializing DOCA Flow by indicating mode_args="vnf,hws" in the doca_flow_cfg struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Config a shared meter with specific cir and cbs values.
   2. Binding the shared meter to the port.
   3. Building a pipe with a changeable 5-tuple match with UDP protocol, changeable shared meter ID and forwarding the matched packets to the second port.
   4. Adding an entry with an example 5-tuple match and shared meter with ID=port_id.
5. Building a pipe with a changeable 5-tuple match with TCP protocol, changeable shared meter ID and forwarding the matched packets to the second port.

6. Adding an entry with an example 5-tuple match and shared meter with ID=port_id.

7. Building a control pipe with the following entries:
   - If L4 type is UDP, forwards the packets to the UDP pipe
   - If L4 type is TCP, forwards the packets to the TCP pipe

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_shared_meter/flow_shared_meter_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_shared_meter/flow_shared_meter_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_shared_meter/meson.build

**Flow Switch Control Pipe**

This sample shows how to use the DOCA Flow control pipe in switch mode.

The sample logic includes:

1. Initializing DOCA Flow by indicating mode_args="switch,hws" in the doca_flow_cfg struct.

2. Starting two DOCA Flow ports.

3. On each port:
   1. Building control pipe with match on VNI field.

   2. Adding two entries to the control pipe, both matching TRANSPORT (UDP or TCP proto) over IPv4 with source port 80 and forwarding to the other port, where the first entry matches destination port 1234 and the second 12345.

   3. Both entries have counters, so that after the successful insertions of both entries, the sample queries those counters to check the number of matched packets.
Flow Switch – Multiple Switches

This sample illustrates how to use two switches working concurrently on two different physical functions.

It shows how to build a basic pipe in a switch and hardware steering (HWS) mode. Each pipe contains two entries, each of which forwards matched packets to two different representors.

The sample also demonstrates how to obtain the switch port of a given port using `doca_flow_port_switch_get()`.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.

2. Starting DOCA Flow ports: Two physical ports and two representors each (totaling six ports).

3. On the switch port:

   The test requires two PFs with two (either VF or SF) representors on each.
1. Building a basic pipe while indicating which fields to match on using `struct
doca_flow_match match`.

2. Adding two entries to the created pipe, each of which forwards packets to a
different port representor.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_switch/flow_switch_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch/flow_switch_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_switch/meson.build`

**Flow Switch – Single Switch**

This sample is identical to the previous sample, before the flow switch sample was extended to take advantage of the capabilities of DOCA to support multiple switches concurrently, each based on a different physical device.

The reason we add this original version is that it removes the constraints imposed by the modified flow switch version, allowing to use arbitrary number of representors in the switch configuration.

The logic of this sample is identical to that of the previous sample with 2 new pipes.

- A user RSS pipe which receives the packets which missed TC rules (in the kernel domain in this case)
- A simple pipe forwarding packets to kernel domain by using `DOCA_FLOW_FWD_TARGET`

In the `to_kernel_pipe`, all the IPv4 packets are forwarded to the kernel (i.e., entry 0 in `to Kernel pipe`). In the kernel domain, all the IPv4 packets are missed to the NIC domain if there is no TC rule. In the NIC domain, the IPv4 packets missed from the NIC domain are forwarded to slow path (i.e., the representor of the PF/VF).

- Root pipe:

  Entry 0: IP src 1.2.3.4 / dst 8.8.8.8 / TCP src 1234 dst 80 -> port 0
Flow Switch (Direction Info)

This sample illustrates how to give a hint to the driver for potential optimizations based on the direction information.

Info

This sample requires a single PF with two representors (either VF or SF).
The sample also demonstrates usage of the `match.parser_meta.port_meta` to detect by the switch pipe the source from where the packet has arrived.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.

2. Starting 3 DOCA Flow ports, 1 physical port and 2 representors.

3. On the switch port:

   1. Network-to-host pipe:

   1. Building basic pipe with a changeable `ipv4.next_proto` field and configuring the pipe with the hint of direction by setting `attr.dir_info = DOCA_FLOW_DIRECTION_NETWORK_TO_HOST`.

   2. Adding two entries:

      - If `ipv4.next_proto` is TCP, the packet is forwarded to the first representor, to the host.
      - If `ipv4.next_proto` is UDP, the packet is forwarded to the second representor, to the host.

   2. Host-to-network pipe:

   1. Building a basic pipe with a match on `aa:aa:aa:aa:aa:aa` as a source MAC address and configuring a pipe with the hint of direction by setting `attr.dir_info = DOCA_FLOW_DIRECTION_HOST_TO_NETWORK`.

   2. Adding an entry. If the source MAC is matched, forward the packet to the physical port (i.e., to the network).

3. Switch pipe:

   1. Building a basic pipe with a changeable `parser_meta.port_meta` to detect where the packet has arrived from.

   2. Adding 3 entries:
- If the packet arrived from port 0 (i.e., the network), forward it to the network-to-host pipe to decide for further logic.

- If the packet arrived from port 1 (i.e., the host's first representor), forward it to the host-to-network pipe to decide for further logic.

- If the packet arrived from port 2, (i.e., the host's second representor), forward it to the host-to-network pipe to decide for further logic.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_switch_direction_info/flow_switch_direction_info_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_switch_direction_info/flow_switch_direction_info_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_switch_direction_info/meson.build

**Flow Switch Hot Upgrade**

This sample demonstrates how to use the port operation state mechanism for a hot upgrade use case. It shows how to configure the state of a port during initialization and how to modify the state after the port has already been started.

**Prerequisites**

The test requires two physical functions (PFs) with two (either VFs or SFs) representors on each.

**Command-line Arguments**

The sample allows users to specify the operation state of the instance using the `--state <value>` argument. The relevant values are:

- 0 for `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`
- 1 for `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP`
Sample Logic

1. Initialize DOCA Flow:
   - Indicate mode_args="switch" in the doca_flow_cfg structure.

2. Start DOCA Flow ports:
   - Two physical ports and two representors each (totaling six ports) are started.
   - Both switch ports are configured with DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED state.

3. Configure each switch port:
   1. Build a basic pipe with a miss counter matching on outer L3 type (specific IPv4) and outer L4 type (changeable).
   2. Add two entries to the created pipe with counters, each forwarding packets to a different port representor.
   3. Modify the port operation state from DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED to the required state.

4. Traffic handling:
   - Wait for traffic until a SIGQUIT signal (Ctrl+) is received.
   - While traffic is being received, traffic statistics are printed to stdout.

Hot Upgrade Use Case

To illustrate the hot upgrade use case, follow these steps:

1. Create two different instances in separate windows with different states.
2. Close the active process by typing Ctrl+\ while traffic is being received. The traffic statistics will start printing in the standby instance.

3. Restart the first instance. The traffic statistics will stop printing in the standby instance and start printing in the active instance again.

Swap Use Case

When both instances are running, the swap use case can be demonstrated by typing Ctrl+C:

- Typing Ctrl+C in the active instance changes its state to DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP
- Typing Ctrl+C in the standby instance changes its state to DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE
- Typing Ctrl+C in the active instance again changes its state to DOCA_FLOW_PORT_OPERATION_STATE_STANDBY

Note

DPDK prevents users from creating two primary instances. To avoid this limitation, use the --file-prefix EAL argument.

- Example for the "active" instance:
  
  ```
  /tmp/build/samples/doca_flow_switch_hot_upgrade -- -p 08:00.0 -p 08:00.1 -r vf[0-1] -r vf[0-1] -l 70
  ```

- Example for the "stand-by" instance:
  
  ```
  /tmp/build/samples/doca_flow_switch_hot_upgrade --file-prefix standby -- -p 08:00.0 -p 08:00.1 -r vf[0-1] -r vf[0-1] -l 70 --state 2
  ```
References

- /opt/mellanox/doca/samples/doca_flow/flow_switch_hot_upgrade/flow_switch_hot_upgrade_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_switch_hot_upgrade/flow_switch_hot_upgrade_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_switch_hot_upgrade/meson.build

Flow VXLAN Encap

This sample shows how to use DOCA Flow actions to create a VXLAN/VXLANGPE/VXLANGBP tunnel as well as illustrating the usage of matching TCP and UDP packets in the same pipe.

The sample logic includes:

1. Initializing DOCA Flow by indicating mode_args="vnf,hws" in the doca_flow_cfg struct.

2. Starting two DOCA Flow ports.

3. On each port:

   1. Building a pipe with changeable 5-tuple match, encap action, and forward port action.

   2. Adding example 5-tuple and encapsulation values entry to the pipe. Every TCP or UDP over IPv4 packet with the same 5-tuple is matched and encapsulated.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/flow_vxlan_encap_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/flow_vxlan_encap_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_vxlan_encap/meson.build
Flow Shared Mirror

This sample shows how to use the DOCA Flow shared mirror.

Note

A current limitation does not allow using shared mirror IDs bearing the value zero.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. On each port:
   1. Configuring a shared mirror with a clone destination hairpin to the second port.
   2. Binding the shared mirror to the port.
   3. Building a pipe with a changeable 5-tuple match with UDP protocol, changeable shared mirror ID, and forwarding the matched packets to the second port.
   4. Adding an entry with an example 5-tuple match and shared mirror with ID=port_id+1.
   5. Building a pipe with a changeable 5-tuple match with TCP protocol, changeable shared mirror ID, and forwarding the matched packets to the second port.
   6. Adding an entry with an example 5-tuple match and shared mirror with ID=port_id+1.
   7. Building a control pipe with the following entries:
      - If L4 type is UDP, forwards the packets to the UDP pipe
If L4 type is TCP, forwards the packets to the TCP pipe

8. Waiting 15 seconds to clone any incoming traffic. Should see the same two packets received on the second port (one from the clone and another from the original).

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/flow_shared_mirror_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/flow_shared_mirror_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_shared_mirror/meson.build

Flow Match Comparison

This sample shows how to use the DOCA Flow match with a comparison result.

The sample logic includes:

1. Initializing DOCA Flow by indicating mode_args="vnf,hws" in the doca_flow_cfg struct.

2. Starting two DOCA Flow ports.

3. On each port:

   1. Building a pipe with a changeable match on meta_data[0] and forwarding the matched packets to the second port.

   2. Adding an entry that matches on meta_data[0] equal with TCP header length.

   3. Building a control pipe for comparison purpose.

   4. Adding an entry to the control pipe match with comparison result the meta_data[0] value greater than meta_data[1] and forwarding the matched packets to match with the meta pipe.

   5. Building a pipe with a changeable 5-tuple match, copying ipv4.total_len to meta_data[1], and accumulating ipv4.version_ihl << 2 tcp.data_offset << 2 to meta_data[1], then forwarding to the second pipe.
6. Adding an example 5-tuple entry to the pipe.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_match_comparison/flow_match_comparison_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_match_comparison/flow_match_comparison_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_match_comparison/meson.build

Flow Pipe Resize

This sample shows how the DOCA Flow pipe resize feature behaves as pipe size increases. The pipe type under resize (basic or control) can be specified in the command line.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws,cpds"` in the `doca_flow_cfg` struct.

2. Starting a PF with two representors of SFs or VFs and selecting the pipe type under resize. For example:

```
./doca_flow_pipe_resize -- --pipe-type <basic|control> -p 08:00.0 -r sf[0-1] -l 60 --sdk-log-level 50
```

Info

The CPDS (control pipe dynamic size) argument is relevant for a control pipe only. By default, a control pipe's internal tables have a default size of 64 entries. Using the CPDS mode, each table's initial size matches the control pipe size.
3. Starting with a pipe of a max size of 10 entries then adding 80 entries. Instead of failing on adding the 11th entry, the pipe continues increasing in the following manner:

1. Receiving a `CONGESTION_REACHED` callback whenever the number of current entries exceeds a threshold level of 80%.

2. Calling `doca_flow_pipe_resize()` with threshold percentage of 50%. Roughly, the new size is calculated as: \((\text{current entries}) / (50\%)\) rounded up to the nearest power of 2. A callback can indicate the exact number of entries.

3. Receiving a callback on the exact new calculated size of the pipe:

   ```c
   typedef doca_error_t (*doca_flow_pipe_resize_nr_entries_changed_cb)(void *pipe_user_ctx, uint32_t nr_entries);
   ```

4. Start calling `doca_flow_entries_process()` in a loop on each thread ID to trigger the entry relocations.

   **Info**
   
   The loop should continue as long as the resize process was not ended.

5. Receiving a callback on each entry relocated to the new resized pipe:

   ```c
   typedef doca_error_t (*doca_flow_pipe_resize_entry_relocate_cb)(void *pipe_user_ctx, uint16_t pipe_queue, void *entry_user_ctx, void **new_entry_user_ctx)
   ```

6. Receiving a `PIPE_RESIZED` callback upon completion of the resize process. At this point, in case of a control pipe, calling `doca_flow_entries_process()` should stop. In case of a basic pipe, continue calling `doca_flow_entries_process()` to process the last entries being added to the pipe.
7. Waiting 5 seconds to send any traffic that matches the flows and seeing them on the other port.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_pipe_resize/flow_pipe_resize_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_pipe_resize/flow_pipe_resize_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_pipe_resize/meson.build

**Flow Entropy**

This sample shows how to use the DOCA Flow entropy calculation.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.

2. Starting one DOCA Flow port.
3. Configuring the doca_flow_entropy_format structure with 5-tuple values.

4. Calling to doca_flow_port_calc_entropy to get the calculated entropy.

5. Logging the calculated entropy.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_entropy/flow_entropy_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_entropy/flow_entropy_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_entropy/meson.build

Flow VXLAN Shared Encap

This sample shows how to use DOCA Flow actions to create a VXLAN tunnel as well as illustrating the usage of matching TCP and UDP packets in the same pipe.

The VXLAN tunnel is created by shared_resource_encap.

The sample logic includes:

1. Initializing DOCA Flow by indicating mode_args="vnf,hws" in the doca_flow_cfg struct.

2. Starting two DOCA Flow ports.

3. On each port:

   1. Configure and bind shared encap resources. The encap resources are for VXLAN encap.

   2. Building a pipe with changeable 5-tuple match, shared_encap_id, and forward port action.

   3. Adding example 5-tuple and encapsulation values entry to the pipe. Every TCP or UDP over IPv4 packet with the same 5-tuple is matched and encapsulated.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_vxlan_shared_encap/flow_vxlan_shared_encap_sample.c
# Field String Support Appendix

## Supported Field String

The following is a list of all the API fields available for matching criteria and action execution.

<table>
<thead>
<tr>
<th>String Field</th>
<th>Path in The Structure</th>
<th>Set Actions</th>
<th>Add Actions</th>
<th>Copy Actions</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta.data (bit_offset &lt; 32)</td>
<td>meta.pkt_meta</td>
<td>meta.pkt_meta</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>meta.data (bit_offset ≥ 32)</td>
<td>meta.u32[i]</td>
<td>meta.u32[i]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>meta.mark</td>
<td>meta.mark</td>
<td>meta.mark</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser_meta.hash.result</td>
<td>None</td>
<td>N / A</td>
<td>N / A</td>
<td>N / A</td>
<td></td>
</tr>
<tr>
<td>parser_meta.port.id</td>
<td>parser_meta.port_meta</td>
<td>N / A</td>
<td>N / A</td>
<td>N / A</td>
<td></td>
</tr>
<tr>
<td>parser_meta.ipsec.syndrome</td>
<td>parser_meta.ipsec_syndrome</td>
<td>N / A</td>
<td>N / A</td>
<td>N / A</td>
<td></td>
</tr>
<tr>
<td>parser_meta.psp.syndrome</td>
<td>parser_meta.psp_syndrome</td>
<td>N / A</td>
<td>N / A</td>
<td>N / A</td>
<td></td>
</tr>
<tr>
<td>String Field</td>
<td>Path in The Structure</td>
<td>Set</td>
<td>Add</td>
<td>Copy</td>
<td>Condition</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>---------------------------</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td>Match</td>
<td>Dst</td>
<td>Src</td>
<td>Dst</td>
<td>Src</td>
</tr>
<tr>
<td>parser_meta.random.value</td>
<td>parser_meta.random</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>parser_meta.meter.color</td>
<td>parser_meta.meter_color</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>parser_meta.packet_type.l2_outter</td>
<td>parser_meta.outer_l2_type</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>parser_meta.packet_type.l3_outter</td>
<td>parser_meta.outer_l3_type</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>parser_meta.packet_type.l4_outter</td>
<td>parser_meta.outer_l4_type</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>parser_meta.packet_type.l2_inner</td>
<td>parser_meta.inner_l2_type</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>parser_meta.packet_type.l3_inner</td>
<td>parser_meta.inner_l3_type</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>parser_meta.packet_type.l4_inner</td>
<td>parser_meta.inner_l4_type</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>parser_meta.outer_ip_fragmented</td>
<td>parser_meta.outer_ip_fragmented</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>parser_meta.inner_ip_fragmented</td>
<td>parser_meta.inner_ip_fragmented</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>String Field</td>
<td>Path in The Structure</td>
<td>Set</td>
<td>Add</td>
<td>Copy</td>
<td>Condition</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------------</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td>Match</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser_meta.outer_integrity.l3_</td>
<td>parser_meta.outer_l3_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser_meta.outer_integrity.ipv</td>
<td>parser_meta.outer_ip4_checksum_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4_checksum_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser_meta.outer_integrity.l4_</td>
<td>parser_meta.outer_l4_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser_meta.outer_integrity.l4_</td>
<td>parser_meta.outer_l4_checksum_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>checksum_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser_meta.inner_integrity.l3_</td>
<td>parser_meta.inner_l3_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser_meta.inner_integrity.ipv</td>
<td>parser_meta.inner_ip4_checksum_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4_checksum_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser_meta.inner_integrity.l4_</td>
<td>parser_meta.inner_l4_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parser_meta.inner_integrity.l4_</td>
<td>parser_meta.inner_l4_checksum_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>checksum_ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.eth.dst_mac</td>
<td>outer.eth.dst_mac</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.eth.src_mac</td>
<td>outer.eth.src_mac</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.eth.type</td>
<td>outer.eth.type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Field</td>
<td>Path in The Structure</td>
<td>Match</td>
<td>Actions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------</td>
<td>-------------</td>
<td>---------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.eth_vlan0.tci</td>
<td>outer.eth_vlan[0].tci</td>
<td>outer.eth_vlan[0].tci</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.eth_vlan1.tci</td>
<td>outer.eth_vlan[1].tci</td>
<td>outer.eth_vlan[1].tci</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv4.src_ip</td>
<td>outer.ip4.src_ip</td>
<td>outer.ip4.src_ip</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv4.dst_ip</td>
<td>outer.ip4.dst_ip</td>
<td>outer.ip4.dst_ip</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv4.dscp_ecn</td>
<td>outer.ip4.dscp_ecn</td>
<td>outer.ip4.dscp_ecn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv4.next_proto</td>
<td>outer.ip4.next_proto</td>
<td>outer.ip4.next_proto</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv4.ttl</td>
<td>outer.ip4.ttl</td>
<td>outer.ip4.ttl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv4.version_ihl</td>
<td>outer.ip4.version_ihl</td>
<td>outer.ip4.version_ihl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv4.total_len</td>
<td>outer.ip4.total_len</td>
<td>outer.ip4.total_len</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv6.src_ip</td>
<td>outer.ip6.src_ip</td>
<td>outer.ip6.src_ip</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv6.dst_ip</td>
<td>outer.ip6.dst_ip</td>
<td>outer.ip6.dst_ip</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv6.traffic_class</td>
<td>outer.ip6.traffic_class</td>
<td>outer.ip6.traffic_class</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv6.flow_label</td>
<td>outer.ip6.flow_label</td>
<td>outer.ip6.flow_label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv6.next_proto</td>
<td>outer.ip6.next_proto</td>
<td>outer.ip6.next_proto</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.ipv6.hop_limit</td>
<td>outer.ip6.hop_limit</td>
<td>outer.ip6.hop_limit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.udp.src_port</td>
<td>outer.udp.l4_port.src_port</td>
<td>outer.udp.l4_port.src_port</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.udp.dst_port</td>
<td>outer.udp.l4_port.dst_port</td>
<td>outer.udp.l4_port.dst_port</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Field</td>
<td>Path in The Structure</td>
<td>Match</td>
<td>Actions</td>
<td>Condition</td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------</td>
<td>--------------------------------</td>
<td>------------------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>outer.transport.src_port</td>
<td>outer.transport.src_port</td>
<td>outer.transport.src_port</td>
<td>outer.transport.src_port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.transport.dst_port</td>
<td>outer.transport.dst_port</td>
<td>outer.transport.dst_port</td>
<td>outer.transport.dst_port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.tcp.src_port</td>
<td>outer.tcp.l4_port.src_port</td>
<td>outer.tcp.l4_port.src_port</td>
<td>outer.tcp.l4_port.src_port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.tcp.dst_port</td>
<td>outer.tcp.l4_port.dst_port</td>
<td>outer.tcp.l4_port.dst_port</td>
<td>outer.tcp.l4_port.dst_port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.tcp.flags</td>
<td>outer.tcp.flags</td>
<td>outer.tcp.flags</td>
<td>outer.tcp.flags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.tcp.data_offset</td>
<td>outer.tcp.data_offset</td>
<td>outer.tcp.data_offset</td>
<td>outer.tcp.data_offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.icmp4.type</td>
<td>outer.icmp.type</td>
<td>outer.icmp.type</td>
<td>outer.icmp.type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.icmp4.code</td>
<td>outer.icmp.code</td>
<td>outer.icmp.code</td>
<td>outer.icmp.code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.icmp4.ident</td>
<td>outer.icmp.ident</td>
<td>outer.icmp.ident</td>
<td>outer.icmp.ident</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.icmp6.type</td>
<td>outer.icmp.type</td>
<td>outer.icmp.type</td>
<td>outer.icmp.type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outer.icmp6.code</td>
<td>outer.icmp.code</td>
<td>outer.icmp.code</td>
<td>outer.icmp.code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.gre.protocol</td>
<td>tun.protocol</td>
<td>tun.protocol</td>
<td>tun.protocol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.gre_key.value</td>
<td>tun.gre_key</td>
<td>tun.gre_key</td>
<td>tun.gre_key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.nvgre.protocol</td>
<td>tun.protocol</td>
<td>tun.protocol</td>
<td>tun.protocol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.nvgre.nvgre_vs_id</td>
<td>tun.nvgre_vs_id</td>
<td>tun.nvgre_vs_id</td>
<td>tun.nvgre_vs_id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.nvgre.nvgre_flow_id</td>
<td>tun.nvgre_flow_id</td>
<td>tun.nvgre_flow_id</td>
<td>tun.nvgre_flow_id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.vxlan.vni</td>
<td>tun.vxlan_tun_id</td>
<td>tun.vxlan_tun_id</td>
<td>tun.vxlan_tun_id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.vxlan_gpe.vni</td>
<td>tun.vxlan_tun_id</td>
<td>tun.vxlan_tun_id</td>
<td>tun.vxlan_tun_id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.vxlan_gbp.vni</td>
<td>tun.vxlan_tun_id</td>
<td>tun.vxlan_tun_id</td>
<td>tun.vxlan_tun_id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.vxlan_gpe.next_proto</td>
<td>tun.vxlan_next_protocol</td>
<td>tun.vxlan_next_protocol</td>
<td>tun.vxlan_next_protocol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Field</td>
<td>Path in The Structure</td>
<td>Set</td>
<td>Add</td>
<td>Copy</td>
<td>Condition</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td>Match</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.vxlan_gbp.policy_id</td>
<td>tun.vxlan_group_policy_id</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.vxlan.rsvd1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.vxlan_gpe.rsvd1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.vxlan_gbp.rsvd1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.gtp.teid</td>
<td>tun.gtp_teid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.esp.spi</td>
<td>tun.esp_spi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.esp.sn</td>
<td>tun.esp_sn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.psp.nexthdr</td>
<td>tun.psp.nexthdr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.psp.hdrextlen</td>
<td>tun.psp.hdrextlen</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.psp.res_cryptofst</td>
<td>tun.psp.res_cryptofst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.psp.s_d_ver_v</td>
<td>tun.psp.s_d_ver_v</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.psp.spi</td>
<td>tun.psp.spi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.psp.iv</td>
<td>tun.psp.iv</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.psp.vc</td>
<td>tun.psp.vc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.mpls[0].label</td>
<td>tun.mpls[0].label</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.mpls[1].label</td>
<td>tun.mpls[1].label</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.mpls[2].label</td>
<td>tun.mpls[2].label</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.mpls[3].label</td>
<td>tun.mpls[3].label</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.mpls[4].label</td>
<td>tun.mpls[4].label</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Field</td>
<td>Path in The Structure</td>
<td>Match</td>
<td>Actions</td>
<td>Set</td>
<td>Add</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------</td>
<td>-------</td>
<td>---------------------------------------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>tunnel.geneve.ver_opt_len</td>
<td>tun.geneve.ver_opt_len</td>
<td></td>
<td>tun.geneve.ver_opt_len</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.geneve.o_c</td>
<td>tun.geneve.o_c</td>
<td></td>
<td>tun.geneve.o_c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.geneve.next_proto</td>
<td>tun.geneve.next_proto</td>
<td></td>
<td>tun.geneve.next_proto</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.geneve.vni</td>
<td>tun.geneve.vni</td>
<td></td>
<td>tun.geneve.vni</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.geneve_opt[i].type</td>
<td></td>
<td>None</td>
<td>See section &quot;Copy Geneve Options&quot; for details.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.geneve_opt[i].class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tunnel.geneve_opt[i].data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.eth.dst_mac</td>
<td>inner.eth.dst_mac</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.eth.src_mac</td>
<td>inner.eth.src_mac</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.eth.type</td>
<td>inner.eth.type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.eth_vlan0.tci</td>
<td>inner.eth_vlan[0].tci</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.eth_vlan1.tci</td>
<td>inner.eth_vlan[1].tci</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv4.src_ip</td>
<td>inner.ip4.src_ip</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv4.dst_ip</td>
<td>inner.ip4.dst_ip</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv4.dscp_ecn</td>
<td>inner.ip4.dscp_ecn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv4.next_proto</td>
<td>inner.ip4.next_proto</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv4.ttl</td>
<td>inner.ip4.ttl</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv4.version_ihl</td>
<td>inner.ip4.version_ihl</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv4.total_len</td>
<td>inner.ip4.total_len</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv6.src_ip</td>
<td>inner.ip6.src_ip</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv6.dst_ip</td>
<td>inner.ip6.dst_ip</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Field</td>
<td>Path in The Structure</td>
<td>Set</td>
<td>Add</td>
<td>Copy</td>
<td>Condition</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>inner.ipv6.traffic_class</td>
<td>inner.ipv6.traffic_class</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv6.flow_label</td>
<td>inner.ipv6.flow_label</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv6.next_proto</td>
<td>inner.ipv6.next_proto</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv6.hop_limit</td>
<td>inner.ipv6.hop_limit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.ipv6.payload_len</td>
<td>inner.ipv6.payload_len</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.udp.src_port</td>
<td>inner.udp.l4_port.src_port</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.udp.dst_port</td>
<td>inner.udp.l4_port.dst_port</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.transport.src_port</td>
<td>inner.transport.src_port</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.transport.dst_port</td>
<td>inner.transport.dst_port</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.tcp.src_port</td>
<td>inner.tcp.l4_port.src_port</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.tcp.dst_port</td>
<td>inner.tcp.l4_port.dst_port</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.tcp.flags</td>
<td>inner.tcp.flags</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.tcp.data_offset</td>
<td>inner.tcp.data_offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.icmp4.type</td>
<td>inner.icmp.type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.icmp4.code</td>
<td>inner.icmp.code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.icmp4.ident</td>
<td>inner.icmp.ident</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.icmp6.type</td>
<td>inner.icmp.type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inner.icmp6.code</td>
<td>inner.icmp.code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. tun.vxlan_tun_rsvd1 modifications only work for traffic with the default UDP destination port (i.e., 4789 for VXLAN and VXLAN-GBP and 4790 for VXLAN-GPE)   ___  ___
Supported Non-field String

Users can modify fields which are not included in doca_flow_match structure.

Copy Hash Result

Users can copy the the matcher hash calculation into other fields using the "parser_meta.hash" string.

Copy GENEVE Options

User can copy GENEVE option type/class/data using the following strings:

- "tunnel.geneve_opt[i].type" – Copy from/to option type (only for option configured with DOCA_FLOW_PARSER_GENEVE_OPT_MODE_MATCHABLE).
- "tunnel.geneve_opt[i].class" – Copy from/to option class (only for option configured with DOCA_FLOW_PARSER_GENEVE_OPT_MODE_MATCHABLE).
- "tunnel.geneve_opt[i].data" – Copy from/to option data, the bit offset is from the start of the data.

i is the index of the option in tlv_list array provided in doca_flow_parser_geneve_opt_create.

DOCA Flow Connection Tracking

This guide provides an overview and configuration instructions for DOCA Flow CT API.

Introduction

DOCA Flow Connection Tracking (CT) is a 5-tuple table which supports the following:
• Track 5-tuple sessions (or 6-tuple when a zone is available)

• Zone based – virtual tables

• Aging (i.e., removes idle connections)

• Sets metadata for a connection

• Bidirectional packet handling

• High rate of connections per second (CPS)

The CT module makes it simple and efficient to track connections by leveraging hardware resources. The module supports both autonomous and managed mode.

**Architecture**

DOCA Flow CT pipe handles non-encapsulated TCP and UDP packets. The CT pipe only supports forward to next pipe or miss to next pipe actions:

• All packets matching known connection 6-tuples are forwarded to the CT’s forward pipe

• Non-matching packets are forwarded to the miss pipe

The user application must handle packets accordingly.

The DOCA Flow CT API is built around four major parts:

• CT module manipulation – configuring CT module resources

• CT connection entry manipulation – adding, removing, or updating connection entries

• Callbacks – handling asynchronous entry processing result

• Pipe and entry statistics
Aging

Aging time is a time in seconds that sets the maximum allowed time for a session to be maintained without a packet seen. If that time elapses with no packet being detected, the session is terminated.

To support aging, a dedicated aging thread is started to poll and check counters for all connections.

Autonomous Mode

In this mode, DOCA runs multiple CT workers internally, to handle connections in parallel.

A connection's lifecycle is controlled by the connection state encapsulated in the packet and time-based aging.

CT workers establish and close connections automatically based on the connection's state stored in packet meta.

Packet meta is defined as follows:

```c
uint32_t src : 1; /**< Source port in multi-port E-Switch mode */
uint32_t hairpin : 1; /**< Subject to forward using hairpin. */
uint32_t type : 2; /**< CT packet type: New, End or Update */
```
- data – CT table matches on packet meta (zone) and 5-tuples
- type – can have the following values:
  - NONE – (known) if packet hit any connection rule
  - NEW – if new TCP or UDP connection
  - END – if TCP connection closed
- src and hairpin – used for forwarding pipe and worker to deliver packet

Managed Mode

The application is responsible for managing the worker threads in this mode, parsing and handling the connection’s lifecycle.

Managed mode uses DOCA Flow CT management APIs to create or destroy the connections.

The CT aging module notifies on aged out connections by calling callbacks.
Users can create connection rules with a different pattern, meta, or counter, for each packet direction.

**Info**

Users are responsible for defining meta and mask to match and modify.

Users can create one rule of a connection first, then create another rule using API `doca_flow_ct_entry_add_dir()`.

**Info**

Other DOCA Flow APIs like CT entry status query and pipe miss query are not supported.

DOCA Flow API can be used to process CT entries with a CT-dedicated queue.

- `doca_flow_entries_process` – process pipe entries in queue
- `doca_flow_aging_handle` – handle pipe entries aging
Prerequisites

DPU

To enable DOCA Flow CT on the DPU, perform the following on the Arm:

1. Enable iommu.passthrough in Linux boot commands (or disable SMMU from the DPU BIOS):
   1. Run:
      
      ```
      sudo vim /etc/default/grub
      ```
   2. Set GRUB_CMDLINE_LINUX="iommu.passthrough=1".
   3. Run:
      
      ```
      sudo update-grub
      sudo reboot
      ```

2. Configure DPU firmware with LAG_RESOURCE_ALLOCATION=1:
   
   ```
   sudo mlxconfig -d <device-id> s LAG_RESOURCE_ALLOCATION=1
   ```

⚠️ Info

Retrieve device-id from the output of the mst status -v command. If, under the MST tab, the value is N/A, run the mst start command.
3. Update `/etc/mellanox/mlnx-bf.conf` as follows:

```
ALLOW_SHARED_RQ="no"
```

4. Perform power cycle on the host and Arm sides.

5. If working with a single port, set the DPU into e-switch mode:

```
sudo devlink dev eswitch set pci/<pcie-address> mode switchdev
sudo devlink dev param set pci/<pcie-address> name esw_multiport value false cmode runtime
```

**Info**

Retrieve `pcie-address` from the output of the `mst status -v` command.

6. If working with two PF ports, set the DPU into multi-port e-switch mode (for the 2 PCIe devices):

```
sudo devlink dev param set pci/<pcie-address> name esw_multiport value true cmode runtime
```

**Info**

Retrieve `pcie-address` from the output of the `mst status -v` command.

7. Define huge pages (see DOCA Flow prerequisites).
## ConnectX

To enable DOCA Flow CT on the NVIDIA® ConnectX®, perform the following:

1. Configure firmware with `LAGRESOURCE_ALLOCATION=1`:

   ```
   sudo mlxconfig -d <device-id> s LAGRESOURCE_ALLOCATION=1
   ```

   **Info**

   Retrieve `device-id` from the output of the `mst status -v` command. If, under the MST tab, the value is N/A, run the `mst start` command.

2. Perform power cycle.

3. If working with a single port:

   ```
   sudo devlink dev eswitch set pci/<pcie-address> mode switchdev
   sudo devlink dev param set pci/<pcie-address> name esw_multiport value false cmode runtime
   ```

   **Info**

   Retrieve `pcie-address` from the output of the `mst status -v` command.

4. If working with two PF ports:
5. Define huge pages (see DOCA Flow prerequisites).

**Actions**

DOCA Flow CT supports actions based on meta and NAT operations. Each action can be defined as either shared or non-shared.

**Shared Actions**

Actions that can be shared between entries. Shared actions are predefined and reused in multiple entries.

The user gets a handle per shared action created and uses this handle as a reference to the action where required.

**Info**

It is user responsibility to track shared actions and to remove them when they become irrelevant.
Shared actions are defined using a control queue (see `struct doca_flow_ct_cfg`).

## Non-shared Actions

Actions provided with their data during entry create/update.

These actions are completely managed by DOCA Flow CT and cannot be reused in multiple flows (i.e., NAT operations).

### Action Sets in Pipe Creation

Users must define action sets during DOCA Flow CT pipe creation (as with any other pipe).

#### Info

Only actions for meta and NAT are accepted (according to `struct doca_flow_ct_actions`).

During entry create/update, different actions can be provided per direction (different action content and/or different type).

## Feature Enable

To enable user actions, configure the following parameters:

- User action templates during DOCA Flow CT pipe creation
- Maximum number of user actions (`nb_user_actions` on DOCA Flow CT init)
Using Actions in Autonomous Mode

**Init**

Configure the following parameters on `doca_flow_ct_init()`:

- `nb_ctrl_queues` – number of control queues for defining shared actions
- `nb_user_actions` – maximum number of actions (shared and non-shared)
- `worker_cb` – callbacks required to communicate with the user

**Create DOCA Flow CT Pipe**

Configure actions sets on `doca_flow_pipe_create()`.

**Create Shared Actions**

Use `doca_flow_ct_actions_add_shared()` with one of the control queues.

Shared actions can be added at any time before use.

**Implement Worker Callbacks**

Callbacks are called from each worker thread to acquire synchronization with the user code and on the first packet of a flow.

On `doca_flow_ct_rule_pkt_cb`:

- Determine how the packet should be treated
- If rules are required, return the actions handles to use
Using Actions in Managed Mode

Init

Configure the following parameters on `doca_flow_ct_init()`:

- `nb_ctrl_queues` – number of control queues for defining shared actions
- `nb_user_actions` – maximum number of user actions. Both shared control queues and non-shared control queues cache actions IDs to speed up ID allocation, each queue cache max 1024 IDs. The user must configure expected number of actions + total queues * 1024. The number can't exceed the number of actions hardware supported.

Create DOCA Flow CT Pipe

Configure actions sets on `doca_flow_pipe_create()`.

Create Shared Actions

Use `doca_flow_ct_actions_add_shared()` with one of the control queues.

Shared actions can be added at any time before use.

Add Entry

Entry can be created in one of the following ways:

- Using an action handle of a predefined shared action
- Using action data, which is specific to the flow, not sharable (e.g., for NAT operations)

The entry can have different actions and/or different action types per direction.
**Remove Entry**

Non-shared actions associated with an entry are implicitly destroyed by DOCA Flow CT.

Shared actions are not destroyed. They can be used by the user until they decide to remove them.

**Update Entry**

Entry actions can be updated per direction. All combinations of shared/non-shared actions are applicable (e.g., update from shared to non-shared).

**Changeable Forward**

DOCA Flow CT allows using a different forward pipe per flow direction.

DOCA Flow CT supports the forward pipe in two levels:

- Pipe level – a single forward pipe defined during DOCA Flow CT pipe creation and used for all entries
- Entry level – forward pipe defined during entry create
- DOCA Flow CT operates in one of the two levels

DOCA CT forward in entry level has the following characteristics:

- Supports only `DOCA_FLOW_FWD_PIPE` (up to 4 different forward pipes)
- Supports forward pipe per flow direction (both directions can have same/different forward pipe)
- Must set forward pipes on each entry create (no default forward pipe)

Turn on the feature:
1. Create DOCA Flow CT pipe with forward type = DOCA_FLOW_FWD_PIPE and next_pipe = NULL.

2. Call to doca_flow_ct_fwd_register to register forward pipes and get fwd_handles in return.

**Using Changeable Forward in Managed Mode**

1. Initialize DOCA Flow CT (doca_flow_ct_init).

2. Register forward pipes (doca_flow_ct_fwd_register).
   
   ○ Define pipes that can be used for forward

3. Create DOCA Flow CT pipe (doca_flow_pipe_create) with definition of possible forward pipes.

4. Add entry (doca_flow_ct_add_entry).
   
   ○ Set origin and/or reply fwd_handles returned from doca_flow_ct_fwd_register.

5. Update forward for entry direction (doca_flow_ct_update_entry).

**Note**

Updating forward handle requires setting all other parameters with their previous values.

**Using Changeable Forward in Autonomous Mode**

1. Initialize DOCA Flow CT (doca_flow_ct_init).

2. Register forward pipes (doca_flow_ct_fwd_register).
   
   ○ Define pipes that can be used for forward.
3. Create DOCA Flow CT pipe (doca_flow_pipe_create) with definition of possible forward pipes.

4. CT workers start to handle traffic.

5. On the first flow packet, doca_flow_ct_rule_pkt callback is called.
   - In this callback, determine if the entry should be created, and which actions and/or forward handles should be used for this entry.

Info

Update forward for entry direction is not supported.

API

For the library API reference, refer to DOCA Flow and CT API documentation in the NVIDIA DOCA Library APIs.

Note

The pkg-config (*.pc file) for the Flow CT library is included in DOCA's regular definitions :doca.

The following sections provide additional details about the library API.

enum doca_flow_ct_flags

DOCA Flow CT configuration optional flags.
<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOCA_FLOW_CT_FLAG_STATS = 1u &lt;&lt; 0</td>
<td>Enable internal pipe counters for packet tracking purposes. Call doca_flow_pipe_dump(&lt;ct_pipe&gt;) to dump counter values. Each call dumps values changed.</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_FLAG_WORKER_STATS = 1u &lt;&lt; 1</td>
<td>Enable worker thread internal debug counter periodical dump. Autonomous mode only.</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_FLAG_NO_AGING = 1u &lt;&lt; 2</td>
<td>Disable aging</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_FLAG_SW_PKT_PARSING = 1u &lt;&lt; 3</td>
<td>Enable CT worker software packet parsing to support VLAN, IPv6 options, or special tunnel types</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_FLAG_MANAGED = 1u &lt;&lt; 4</td>
<td>Enable managed mode in which user application is responsible for managing packet handling, and calling the CT API to manipulate CT connection entries</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_FLAGASYMMETRIC = 1u &lt;&lt; 5</td>
<td>Allows different 6-tuple table definitions for the origin and reply directions. Default to symmetric mode, uses same meta and reverse 5-tuples for reply direction. Managed mode only.</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_FLAGASYMMETRIC_COUNTER = 1u &lt;&lt; 6</td>
<td>Enable different counters for the origin and reply directions. Managed mode only.</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_FLAG_NO_COUNTER = 1u &lt;&lt; 7</td>
<td>Disable counter and aging to save aging thread CPU cycles</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_FLAG_DEFAULT_MIS = 1u &lt;&lt; 8</td>
<td>Check TCP SYN flags and UDP in CT miss flow to identify ADD type packets.</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_FLAG_WIRE_TO_WIRE = 1u &lt;&lt; 9</td>
<td>Hint traffic comes from uplink wire and forwards to uplink wire.</td>
</tr>
</tbody>
</table>

**Note**

If this flag is set, the direction info must be **DOCA_FLOW_DIRECTION_NETWORK_TO_HOST**.
<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOCA_FLOW_CT_FLAG_LAG_CALC_TUN_IP_CHKSUM</td>
<td>Enable hardware to calculate and set the checksum on L3 header (IPv4)</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_FLAG_DUP_FILTER_UDP_ONLY</td>
<td>Apply the connection duplication filter for UDP connections only</td>
</tr>
</tbody>
</table>

**enum doca_flow_ct doca_flow_ct_entry_flags**

DOCA Flow CT Entry optional flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_NO_WAIT</td>
<td>Entry is not buffered; send to hardware immediately</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_DIR_ORIGIN</td>
<td>Apply flags to origin direction</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_DIR_REPLY</td>
<td>Apply flags to reply direction</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_IPV6_ORIGIN</td>
<td>Origin direction is IPv6; origin match union in struct doca_flow_ct_match is IPv6</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_IPV6_REPLY</td>
<td>Reply direction is IPv6; reply match union in struct doca_flow_ct_match is IPv6</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_ORIGIN</td>
<td>Apply counter to origin direction</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_REPLY</td>
<td>Apply counter to reply direction</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_SHARED</td>
<td>Counter is shared for both direction (origin and reply)</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_FLOW_LOG</td>
<td>Enable flow log on entry removed</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_ALLOC_ON_MISS</td>
<td>Allocate on entry not found when calling doca_flow_ct_entry_prepare() API</td>
</tr>
<tr>
<td>Flag</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_DUP_FILTER_ORIGIN = (1 &lt;&lt; 10)</td>
<td>Enable duplication filter on origin direction</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_ENTRY_FLAGS_DUP_FILTER_REPLY = (1 &lt;&lt; 11)</td>
<td>Enable duplication filter on reply direction</td>
</tr>
</tbody>
</table>

**enum doca_flow_ct_rule_opr**

Options for handling flows in autonomous mode with shared actions. The decision is taken on the first flow packet.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOCA_FLOW_CT_RULE_OK</td>
<td>Flow should be defined in the CT pipe using the required shared actions handles</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_RULE_DROP</td>
<td>Flow should not be defined in the CT pipe. The packet should be dropped.</td>
</tr>
<tr>
<td>DOCA_FLOW_CT_RULE_TX_ONLY</td>
<td>Flow should not be defined in the CT pipe. The packet should be transmitted.</td>
</tr>
</tbody>
</table>

**struct direction_cfg**

Managed mode configuration for origin or reply direction.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool match_inner</td>
<td>5-tuple match pattern applies to packet inner layer</td>
</tr>
<tr>
<td>struct doca_flow_meta *zone_match_mask</td>
<td>Mask to indicate meta field and bits to match</td>
</tr>
<tr>
<td>struct doca_flow_meta *meta_modify_mask</td>
<td>Mask to indicate meta field and bits to modify on connection packet match</td>
</tr>
</tbody>
</table>

**struct doca_flow_ct_worker_callbacks**
Set of callbacks for using shared actions in autonomous mode.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>doca_flow_ct_sync_acquire_cb worker_init</td>
<td>Called at the start of a worker thread to sync with the user context</td>
</tr>
<tr>
<td>doca_flow_ct_sync_release_cb worker_release</td>
<td>Called at the end of a worker thread</td>
</tr>
<tr>
<td>doca_flow_ct_rule_pkt_cb rule_pkt</td>
<td>Called on the first packet of a flow</td>
</tr>
</tbody>
</table>

**struct doca_flow_ct_cfg**

DOCA Flow CT configuration.

```c
uint32_t nb_arm_queues;
uint32_t nb_ctrl_queues;
uint32_t nb_user_actions;
uint32_t nb_arm_sessions[DOCA_FLOW_CT_SESSION_MAX];
uint32_t flags;
uint16_t aging_core;
uint16_t aging_query_delay_s;
doca_flow_ct_flow_log_cb flow_log_cb;
struct doca_flow_ct_aging_ops *aging_ops;
uint32_t base_core_id;
uint32_t dup_filter_sz;
union {
    /* Managed mode configuration for origin and reply direction. */
    struct direction_cfg direction[2];

    /* Below fields are dedicate for autonomous mode */
    struct {
        uint16_t tcp_timeout_s;
        uint16_t tcp_session_del_s;
        uint16_t udp_timeout_s;
        enum doca_flow_tun_type tunnel_type;
        uint16_t vxlan_dst_port;
        enum doca_flow_ct_hash_type hash_type;
        uint32_t meta_user_bits;
        uint32_t meta_action_bits;
    }
};
```
Where:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint32_t nb_arm_queues</code></td>
<td>Number of CT queues. In autonomous mode, also the number of worker threads.</td>
</tr>
<tr>
<td><code>uint32_t nb_ctrl_queues</code></td>
<td>Number of CT control queues used for defining shared actions</td>
</tr>
<tr>
<td><code>uint32_t nb_user_actions</code></td>
<td>Maximum number of user actions supported (shared and non-shared)</td>
</tr>
<tr>
<td></td>
<td>Minimum value is $1K \times \text{nb_ctrl_queues}$</td>
</tr>
<tr>
<td><code>uint32_t nb_arm_sessions[DOCA_FLOW_CT_SESSION_MAX]</code></td>
<td>Maximum number of IPv4 and IPv6 CT connections</td>
</tr>
<tr>
<td><code>uint32_t flags</code></td>
<td>CT configuration flags</td>
</tr>
<tr>
<td><code>uint16_t aging_core</code></td>
<td>CPU core ID for CT aging thread to bind.</td>
</tr>
<tr>
<td><code>uint16_t aging_core_delay</code></td>
<td>CT aging code delay.</td>
</tr>
<tr>
<td><code>doxa_flow_ct_flow_log_cb flow_log_cb</code></td>
<td>Flow log callback function, when set</td>
</tr>
<tr>
<td><code>struct doxa_flow_ct_aging_ops *aging_ops</code></td>
<td>User-defined aging logic callback functions. Fallback to default aging logic</td>
</tr>
<tr>
<td><code>uint32_t base_core_id</code></td>
<td>Base core ID for the workers</td>
</tr>
<tr>
<td><code>uint32_t dup_filter_sz</code></td>
<td>Number of connections to cache in the duplication filter</td>
</tr>
<tr>
<td><code>struct direction_cfg direction</code></td>
<td>Managed mode configuration for origin or reply direction</td>
</tr>
<tr>
<td><code>uint16_t tcp_timeout_s</code></td>
<td>TCP timeout in seconds</td>
</tr>
<tr>
<td><code>uint16_t tcp_session_del_s</code></td>
<td>Time to delay or kill TCP session after RST/FIN</td>
</tr>
<tr>
<td><code>enum doxa_flow_tun_type tunnel_type</code></td>
<td>Encapsulation tunnel type</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>uint16_t vxlan_dst_port</td>
<td>VXLAN outer UDP destination port in big endian</td>
</tr>
<tr>
<td>enum doca_flow_ct_hash_type hash_type</td>
<td>Type of connection hash table type: NONE or SYMMETRIC_HASH</td>
</tr>
<tr>
<td>uint32_t meta_user_bits</td>
<td>User packet meta bits to be owned by the user</td>
</tr>
<tr>
<td>uint32_t meta_action_bits</td>
<td>User packet meta bits to be carried by identified connection packet</td>
</tr>
<tr>
<td>struct doca_flow_meta *meta_zone_mask</td>
<td>Mask to indicate meta field and bits saving zone information</td>
</tr>
<tr>
<td>struct doca_flow_meta *connection_id_mask</td>
<td>Mask to indicate meta field and bits for CT internal connection ID</td>
</tr>
<tr>
<td>struct doca_flowct_worker_callbacks worker_cb</td>
<td>Worker callbacks to use shared actions</td>
</tr>
</tbody>
</table>

**struct doca_flow_ct_actions**

This structure is used in the following cases:

- For defining shared actions. In this case, action data is provided by the user. The action handle is returned by DOCA Flow CT.

- For defining an entry with actions. The structure can be filled with two options:
  - With action handle of a previously created shared action
  - With non-shared action data

DOCA Flow CT action structure.

```c
enum doca_flow_resource_type resource_type;
union {
    /* Used when creating an entry with a shared action. */
    uint32_t action_handle;

    /* Used when creating an entry with non-shared action or when creating a shared action. */
    struct {
```
uint32_t action_idx;
struct doca_flow_meta meta;
struct doca_flow_header_l4_port l4_port;
union {
    struct doca_flow_ct_ip4 ip4;
    struct doca_flow_ct_ip6 ip6;
} ip;
} data;
};

Where:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum doca_flow_resource_type resource_type</td>
<td>Shared/non-shared action</td>
</tr>
<tr>
<td>uint32_t action_handle</td>
<td>Shared action handle</td>
</tr>
<tr>
<td>uint32_t action_idx</td>
<td>Actions template index</td>
</tr>
<tr>
<td>struct doca_flow_meta meta</td>
<td>Modify meta values</td>
</tr>
<tr>
<td>struct doca_flow_header_l4_port l4_port</td>
<td>UDP or TCP source and destination port</td>
</tr>
<tr>
<td>struct doca_flow_ct_ip4 ip4</td>
<td>Source and destination IPv4 addresses</td>
</tr>
<tr>
<td>struct doca_flow_ct_ip6 ip6</td>
<td>Source and destination IPv6 addresses</td>
</tr>
</tbody>
</table>

**DOCA Flow Connection Tracking Samples**

This section describes DOCA Flow CT samples based on the DOCA Flow CT pipe.

The samples illustrate how to use the library API to manage TCP/UDP connections.

ℹ️ **Info**

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.
Running the Samples

1. Refer to the following documents:
   
   o NVIDIA DOCA Installation Guide for Linux for details on how to install
     BlueField-related software.
   
   o NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with
     the installation, compilation, or execution of DOCA samples.

2. To build a given sample:
   
   ```
   cd /opt/mellanox/doca/samples/doca_flow/flow_ct_udp
   meson /tmp/build
   ninja -C /tmp/build
   ```

   **Info**
   
   The binary `doca_flow_ct_udp` is created under `/tmp/build/samples/`.

3. Sample (e.g., `doca_flow_ct_udp`) usage:
   
   Usage: `doca_<sample_name> [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:**
4. For additional information per sample, use the `-h` option:

```
/tmp/build/samples/<sample_name> -h
```

5. The following is a CLI example for running the samples when port 03:00.0 is configured (multi-port e-switch) as manager port:

```
/tmp/build/samples/doca_<sample_name> -- -p 03:00.0 -l 60
```

**Info**

To avoid the test being impacted by unexpected packets, it only accepts packets like the following examples:

- IPv4 destination address is "1.1.1.1"
- IPv6 destination address is "0101:0101:0101:0101:0101:0101:0101:0101"

**Samples**

**Note**

All CT UDP samples demonstrate the usage of the connection’s duplication filter. Duplication filter is used if the user is interested in
Flow CT 2 Ports

This sample illustrates how to create a simple pipeline on two standalone e-switches. Multi-port e-switch must be disabled.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Initializing DOCA Flow CT.
3. Starting two DOCA Flow uplink ports where port 0 and 1 each has a special role of being a switch manager port.

4. Creating a pipeline on each port:
   1. Building an UDP pipe to filter non-UDP packets.
   2. Building a CT pipe to hold UDP session entries.
   3. Building a counter pipe with an example 5-tuple entry to which non-unidentified UDP sessions should be sent.

---

**Info**

Ports are configured according to the parameters provided to `doca_dpdk_port_probe()` in the main function.

```bash
sudo devlink dev eswitch set pci/<pcie-address0> mode switchdev
sudo devlink dev eswitch set pci/<pcie-address1> mode switchdev
sudo devlink dev param set pci/<pcie-address0> name esw_multiport value false cmode runtime
```
4. Building a hairpin pipe to send back packets.

5. Building an RSS pipe from which all packets are directed to the sample main thread for parsing and processing.

5. Packet processing on each port:

1. The first UDP packet triggers the miss flow as the CT pipe is empty.

2. Performing 5-tuple packet parsing.

3. Calling `doca_flow_ct_add_entry()` to create a hardware rule according to the parsed 5-tuple info.

4. The second UDP packet based on the the same 5-tuple should be sent again. Packet hits the hardware rule inserted before and sent back to egress.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/flow_ct_2_ports_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/flow_ct_2_ports_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/meson.build`

**Flow CT UDP**

This sample illustrates how to create a simple UDP pipeline with a CT pipe in it.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.

2. Initializing DOCA Flow CT.

3. Starting two DOCA Flow uplink representor ports where port 0 has a special role of being a switch manager port.
4. Creating a pipeline on the main port:

1. Building an UDP pipe to filter non-UDP packets.
2. Building a CT pipe to hold UDP session entries.
3. Building a counter pipe with an example 5-tuple entry to which non-unidentified UDP sessions should be sent.
4. Building a VXLAN encapsulation pipe to encapsulate all identified UDP sessions.
5. Building an RSS pipe from which all packets are directed to the sample main thread for parsing and processing.

5. Packet processing:

1. The first UDP packet triggers the miss flow as the CT pipe is empty.
2. 5-tuple packet parsing is performed.
3. `doca_flow_ct_add_entry()` is called to create a hardware rule according to the parsed 5-tuple info.
4. The second UDP packet based on the the same 5-tuple should be sent again. Packet hits the HW rule inserted before and directed to port 0 after VXLAN encapsulation.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/flow_ct_udp_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/flow_ct_udp_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp/meson.build`
**Flow CT UDP Query**

This sample illustrates how to query a Flow CT UDP session entry. The query can be done according to session direction (origin or reply). The pipeline is identical to that of the Flow CT UDP sample.

This sample adds the following logic:

1. Dumping port 0 information into a file at `/port_0_info.txt`.

2. Querying UDP session hardware entry created after receiving the first UDP packet:
   - Origin total bytes received
   - Origin total packets received
   - Reply total bytes received
   - Reply total packets received

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp_query/flow_ct_udp_query_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp_query/flow_ct_udp_query_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_udp_query/meson.build`

**Flow CT UDP Update**

This sample illustrates how a CT entry can be updated after creation.

The pipeline is identical to that of the Flow CT UDP sample. In case of non-active UDP sessions, a relevant entry shall be updated with an aging timeout.

This sample adds the following logic:

1. Querying all UDP sessions for the total number of packets received in both the origin and reply directions.
2. Updating entry aging timeout to 2 seconds once a session is not active (i.e., no packets received on either side).

3. Waiting until all non-active session are aged and deleted.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_ct_udp_update/flow_ct_udp_update_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_ct_udp_update/flow_ct_udp_update_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_ct_udp_update/meson.build

**Flow CT UDP Single Match**

This sample is based on the Flow CT UDP sample. The sample illustrates that a hardware entry can be created with a single match (matching performed in one direction only) in the API call `doca_flow_ct_add_entry()`.

**Flow CT Aging**

This sample illustrates the use of the DOCA Flow CT aging functionality. It demonstrates how to build a pipe and add different entries with different aging times and user data.

No packets need to be sent for this sample.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.

2. Initializing DOCA Flow CT.

3. Starting two DOCA Flow uplink representor ports where port 0 has a special role of being a switch manager port.
4. Building a UDP pipe to serve as the root pipe.

5. Building a counter pipe with an example 5-tuple entry to which CT forwards packets.

6. Adding 32 entries with a different 5-tuple match, different aging time (3-12 seconds), and setting user data. User data will contain the port ID, entry number, and status.

7. Handling aging in small intervals and removing each entry after age-out.

8. Running these commands until all 32 entries age out.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_ct_aging/flow_ct_aging_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_ct_aging/flow_ct_aging_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_ct_aging/meson.build

**Flow CT TCP**

This sample illustrates how to manage TCP flags with CT to achieve better control over TCP sessions.

1. Initializing DOCA Flow by indicating `mode_args="switch,hws"` in the `doca_flow_cfg` struct.
2. Initializing DOCA Flow CT.

3. Starting two DOCA Flow uplink representor ports where port 0 has a special role of being a switch manager port.

![Info]

- Ports are configured according to the parameters provided to `doca_dpdk_port_probe()` in the main function.

4. Creating a pipeline on the main port:
   1. Building a TCP pipe to filter non-TCP packets.
   2. Building a CT pipe to hold TCP session entries.
   3. Building a CT miss pipe which forwards all packets to RSS pipe.
   4. Building an RSS pipe from which all packets are directed to the sample main thread for parsing and processing.
   5. Building a TCP flags filter pipe which identifies the TCP flag inside the packets. `SYN`, `FIN`, and `RST` packets are forwarded to the RSS pipe while all others are forwarded to the EGRESS pipe.
   6. Building an EGRESS pipe to forward packets to uplink representor port 1.

5. Packet processing:
   1. The first TCP packet triggers the miss flow as the CT pipe is empty.
   2. 5-tuple packet parsing is performed.
   3. TCP flag is examined.
      - In case of a `SYN` flag, a hardware entry is created.
For **FIN** or **RST** flags, the HW entry is removed and all packets are transferred to uplink representor port 1 using `rte_eth_tx_burst()` on port 0 (proxy port) by `rte_flow_dynf_metadata_set()` to 1.

4. From this point on, all TCP packets belonging to the above session are offloaded directly to uplink port representor 1.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp/flow_ct_tcp_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp/flow_ct_tcp_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp/meson.build`

### Flow CT TCP Actions

This sample illustrates how to add shared and non-shared actions to CT TCP sessions. The pipeline is identical to that of the Flow CT TCP sample.

![Info](https://via.placeholder.com/150)

**Info**

The sample expects to receive at least **SYN** and **FIN** packets.

This sample adds a shared action on one side of the session that placed the value 1 in the packet’s metadata, while on the other side of the session a non-shared action is placed. The non-shared action simply flips the order of the source-destination IP addresses and port numbers.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_actions/flow_ct_tcp_actions_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_actions/flow_ct_tcp_actions_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_actions/meson.build`
**Flow CT TCP Flow Log**

This sample illustrate how to use the flow log callback to alert when a session is aged/removed.

*Info*

The sample expects to receive at least SYN and FIN packets.

This sample is based on the Flow CT TCP sample. Once a session is removed (after receiving FIN packet), the callback is triggered and session counters are queried.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_flow_log/flow_ct_tcp_flow_log_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_flow_log/flow_ct_tcp_flow_log_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_ct_tcp_flow_log/meson.build`

**Flow CT TCP IPv4/IPv6**

This sample illustrates how to manage a flow with a different IP type per direction.

In case of a SYN flag:

1. A single HW entry of IPv4 is created as origin direction
2. An additional HW entry of IPv6 is created as reply direction
3. From this point on, all IP v4 TCP packets (belonging to the origin direction) and all IPv6 TCP packets (belonging to the reply direction) are offloaded.

Reference:
DOCA Flow Tune Server

This guide provides an overview and configuration instructions for DOCA Flow Tune Server API.

Introduction

DOCA Flow Tune Server (TS), DOCA Flow subcomponent, exposes an API to collect predefined internal key performance indicators (KPIs) and pipeline visualization of a running DOCA Flow application.

Supported port KPIs:

- Total add operations across all queues
- Total update operations across all queues
- Total remove operations across all queues
- Pending operations number across all queues
- Number of NO_WAIT flag operations across all queues
- Number of shared resources and counters
- Number of pipes

Supported application KPIs:

- Number of ports
- Number of queues
- Queues depth

Pipeline information is saved to a JSON file to simplify its structure. Visualization is supported for the following DOCA Flow pipes:

- Basic
- Control

Each pipe contains the following fields:

- Type
- Name
- Domain
- Is root
- Match
- Match mask
- FWD
- FWD miss

Supported entry information:

- Basic
  - FWD
- Control
  - FWD
  - Match
  - Match mask
Prerequisites

DOCA Flow Tune Server API is available only by using the DOCA Flow and DOCA Flow Tune Server trace libraries.

Info

For more detailed information, refer to section "DOCA Flow Debug and Trace" under DOCA Flow.

API

Info

For more detailed information on DOCA Flow API, refer to NVIDIA DOCA Library APIs.

The following subsections provide additional details about the library API.

`enum doca_flow_tune_server_kpi_type`

DOCA Flow TS KPI flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TUNE_SERVER_KPI_TYPE_NR_PORTS,</td>
<td>Retrieve port number</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_NR_QUEUES,</td>
<td>Retrieve queue number</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_QUEUE_DEPTH,</td>
<td>Retrieve queue depth</td>
</tr>
<tr>
<td>Flag</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_NR_SHARED_RESOURCES</td>
<td>Retrieve shared resource and counter numbers</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_NR_PIPES,</td>
<td>Retrieve number of pipes per port</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_ENTRIES_OPS_ADD,</td>
<td>Retrieve entry add operations per port</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_ENTRIES_OPS_UPDATE,</td>
<td>Retrieve entry update operations per port</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_ENTRIES_OPS_REMOVE,</td>
<td>Retrieve entry remove operations per port</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_PENDING_OPS,</td>
<td>Retrieve entry pending operations per port</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_NO_WAIT_OPS,</td>
<td>Retrieve entry NO_WAIT flag operations per port</td>
</tr>
</tbody>
</table>

**struct doca_flow_tune_server_shared_resources_kpi_res**

Holds the number of each shared resources and counters per port.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t nr_meter</td>
<td>Number of meters</td>
</tr>
<tr>
<td>uint64_t nr_counter</td>
<td>Number of counters</td>
</tr>
<tr>
<td>uint64_t nr_rss</td>
<td>Number of RSS</td>
</tr>
<tr>
<td>uint64_t nr_mirror</td>
<td>Number of mirrors</td>
</tr>
<tr>
<td>uint64_t nr_psp</td>
<td>Number of PSP</td>
</tr>
<tr>
<td>uint64_t nr_encap</td>
<td>Number of encap</td>
</tr>
<tr>
<td>uint64_t nr_decap</td>
<td>Number of decap</td>
</tr>
</tbody>
</table>

**struct doca_flow_tune_server_kpi_res**

Holds the KPI result.
**Note**

This structure is required when calling `doca_flow_tune_server_get_kpi` or `doca_flow_tune_server_get_port_kpi`.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enum doca_flow_tune_server_kpi_type</code> type</td>
<td>KPI result type</td>
</tr>
<tr>
<td><code>struct doca_flow_tune_server_shared_resources_kpi_res shared_resources_kpi</code></td>
<td>Shared resource result values</td>
</tr>
<tr>
<td><code>uint64_t val</code></td>
<td>Result value</td>
</tr>
</tbody>
</table>

**doca_flow_tune_server_cfg_create**

Creates DOCA Flow Tune Server configuration structure.

```c
doca_error_t doca_flow_tune_server_cfg_create(struct doca_flow_tune_server **cfg);
```

**doca_flow_tune_server_cfg_set_bind_path**

Adds local path to the configuration struct on which the DOCA Flow Tune Server AF_UNIX socket binds.

```c
doca_error_t doca_flow_tune_server_cfg_set_bind_path(struct doca_flow_tune_server *cfg, const char *path, size_t path_len);
```

**doca_flow_tune_server_cfg_destroy**
Destroys DOCA Flow Tune Server configuration structure.

```c
doca_error_t doca_flow_tune_server_cfg_destroy(struct doca_flow_tune_server *cfg);
```

doca_flow_tune_server_init

Initializes DOCA Flow Tune Server internal structures.

```c
doca_error_t doca_flow_tune_server_init(void);
```

doca_flow_tune_server_destroy

Destroys DOCA Flow Tune Server internal structures.

```c
void doca_flow_tune_server_destroy(void);
```

doca_flow_tune_server_query_pipe_line

Queries and dumps pipeline info for all ports to a JSON file pointed by fp.

```c
doca_error_t doca_flow_tune_server_query_pipe_line(FILE *fp);
```

doca_flow_tune_server_get_port_ids

Retrieves ports identification numbers.
doca_flow_tune_server_get_kpi
Retrieves application scope KPI.

```c
doca_error_t doca_flow_tune_server_get_kpi(enum doca_flow_tune_server_kpi_type kpi_type,
                                          struct doca_flow_tune_server_kpi_res *res)
```

doca_flow_tune_server_get_port_kpi
Retrieves port scope KPI.

```c
doca_error_t doca_flow_tune_server_get_port_kpi(uint16_t port_id,
                                               enum doca_flow_tune_server_kpi_type kpi_type,
                                               struct doca_flow_tune_server_kpi_res *res);
```

DOCA Flow Tune Server Samples
This section describes DOCA Flow Tune Server samples.

The samples illustrate how to use the library API to retrieve KPIs or save pipeline information into a JSON file.

️ Info
All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Running the Samples

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

   ```bash
   cd /opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline
   meson /tmp/build
   ninja -C /tmp/build
   
   Info
   The binary `doca_flow_tune_server_dump_pipeline` is created under `/tmp/build/samples/`.
   
3. Sample (e.g., `doca_flow_tune_server_dump_pipeline`) usage:

   Usage: `doca_<sample_name> [DOCA Flags] [Program Flags]`

   DOCA Flags:
   - `-h, --help`  Print a help synopsis
   - `-v, --version`  Print program version information
4. For additional information per sample, use the -h option:

```
/tmp/build/samples/<sample_name> -h
```

5. The following is a CLI example for running the samples:

```
/tmp/build/doca_<sample_name> -a auxiliary:mlx5_core.sf.2,dv_flow_en=2 -a auxiliary:mlx5_core.sf.3,dv_flow_en=2 -- -l 60
```

## Samples

### Flow Tune Server KPI

This sample illustrates how to use DOCA Flow Tune Server API to retrieve KPIs.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting a single DOCA Flow port.
3. Creating a server configuration struct using the `doca_flow_tune_server_cfg_create` function.
4. Initializing DOCA Flow server using the `doca_flow_tune_server_init` function. This must be done after calling the `doca_flow_port_start` function (or the `init_doca_flow_ports` helper function).
5.Querying existing port IDs using the `doca_flow_tune_server_get_port_ids` function.
6. Querying application level KPIs using `doca_flow_tune_server_get_kpi` function. The following KPI are read:
   - Number of queues
   - Queue depth

7. KPIs per port on which the basic pipe is created:
   1. Add operation entries.

8. Adding 20 entries followed by a second call to query entries add operations.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_kpi/flow_tune_server_kpi_sample.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_kpi/flow_tune_server_kpi_main.c`
- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_kpi/meson.build`

**Flow Tune Server Dump Pipeline**

This sample illustrates how to use DOCA Flow Tune Server API to dump pipeline information into a JSON file.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting two DOCA Flow ports.

3. Creating server configuration struct using the `doca_flow_tune_server_cfg_create` function.


>Note

| DocA Libraries | 282 |
5. Opening a file called `sample_pipeline.json` for writing.

6. For each port:
   1. Creating a pipe to drop all traffic.
   2. Creating a pipe to hairpin traffic from port 0 to port 1
   3. Creating FWD pipe to forward traffic based on 5-tuple.
   4. Adding two entries to FWD pipe, each entry with different 5-tuple.
   5. Creating a control pipe and adding the FWD pipe as an entry.

7. Dumping the pipeline information into a file.

Reference:

- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline/flow_tune_server_dump_pipeline`
- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline/flow_tune_server_dump_pipeline`
- `/opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline/meson.build`

**Flow Visualization**

Once a DOCA Flow application pipeline has been exported to a JSON file, it is easy to visualize it using tools such as Mermaid.

1. Save the following Python script locally to a file named `doca-flow-viz.py` (or similar). This script converts a given JSON file produced by DOCA Flow TS to a Mermaid diagram embedded in a markdown document.

```bash
#!/usr/bin/python3
```
# Copyright (c) 2024 NVIDIA CORPORATION & AFFILIATES, ALL RIGHTS RESERVED.
# This software product is a proprietary product of NVIDIA CORPORATION &
# AFFILIATES (the "Company") and all right, title, and interest in and to the
# software product, including all associated intellectual property rights, are
# and shall remain exclusively with the Company.
#
# This software product is governed by the End User License Agreement
# provided with the software product.
#
import glob
import json
import sys
import os.path

class MermaidConfig:
    def __init__(self):
        self.prefix_pipe_name_with_port_id = False
        self.show_match_criteria = False
        self.show_actions = False

class MermaidFormatter:
    def __init__(self, cfg):
        self.cfg = cfg
        self.syntax = ""
        self.prefix_pipe_name_with_port_id = cfg.prefix_pipe_name_with_port_id

    def format(self, data):
        self.prefix_pipe_name_with_port_id = self.cfg.prefix_pipe_name_with_port_id
        if not 'ports' in data:
            port_id = data.get('port_id', 0)
            data = {
                'ports': [
                    {
                        'port_id': port_id,
                        'pipes': data['pipes']
                    }
                ]
            }
        
if not 'ports' in data:
    port_id = data.get('port_id', 0)
    data = {
        'ports': [
            {
                'port_id': port_id,
                'pipes': data['pipes']
            }
        ]
    }
self.syntax = "
self.append("```mermaid\ngraph LR\nself.declare_terminal_states(data)\n
for port in data['ports']:\n    self.process_port(port)

self.append("\nreturn self.syntax

def append(self, text, endline = "\n"): self.syntax += text + endline
def declare_terminal_states(self, data):
    all_fwd_types = self.get_all_fwd_types(data)
    if 'drop' in all_fwd_types:
        self.append('    drop[[drop]]')
    if 'rss' in all_fwd_types:
        self.append('    RSS[[RSS]]')

def get_all_fwd_types(self, data):
    # Gather all 'fwd' and 'fwd_miss' types from pipes and 'fwd' types from entries
    all_fwd_types = {\n        fwd_type
        for port in data.get('ports', [])
        for pipe in port.get('pipes', [])
        for tag in ['fwd', 'fwd_miss'] # Process both 'fwd' and 'fwd_miss' for each pipe
        for fwd_type in [pipe.get(tag, {}).get('type', None)] # Extract the 'type'
        if fwd_type
    } | {\n        fwd_type
        for port in data.get('ports', [])
        for pipe in port.get('pipes', [])
        for tag in ['fwd']
        for entry in pipe.get('entries', []) # Process all entries in each pipe
        for fwd_type in [entry.get(tag, {}).get('type', None)]
        if fwd_type
    }
    return all_fwd_types

def process_port(self, port):
    port_id = port['port_id']
    pipe_names = self.resolve_pipe_names(port)
    self.declare_pipes(port, pipe_names)
for pipe in port.get('pipes', []):
    self.process_pipe(pipe, port_id)

def resolve_pipe_names(self, port):
    pipe_names = {}
    port_id = port['port_id']
    for pipe in port.get('pipes', []):
        id = pipe['pipe_id']
        name = pipe['attributes'].get('name', f'pipe_{id}"
        if self.prefix_pipe_name_with_port_id:
            name = f'p(port_id).{name}"
        pipe_names[id] = name
    return pipe_names

def declare_pipes(self, port, pipe_names):
    port_id = port['port_id']
    for pipe in port.get('pipes', []):
        id = pipe['pipe_id']
        name = pipe_names[id]
        self.declare_pipe(port_id, pipe, name)

def declare_pipe(self, port_id, pipe, pipe_name):
    id = pipe['pipe_id']
    attr = '\n(root)" if self.pipe_is_root(pipe) else ""'
    if self.cfg.show_match_criteria and not self.pipe_is_ctrl(pipe):
        fields_matched = self.pipe_match_criteria(pipe, 'match')
        attr += f'\nmmatch: {fields_matched}"
    self.append(f'     p(port_id).pipe_{id}{{{{"{pipe_name}{attr}"}}}}}')

def pipe_match_criteria(self, pipe, key: ['match', 'match_mask']):
    return '\n'.join(self.extract_match_criteria_paths(None, pipe.get(key, {}))) or 'None'

def extract_match_criteria_paths(self, prefix, match):
    for k,v in match.items():
        if isinstance(v, dict):
            new_prefix = f"{prefix}.{k}" if prefix else k
            for x in self.extract_match_criteria_paths(new_prefix, v):
                yield x
        else:
            # ignore v, the match value
            yield f'\"{prefix}.{k}" if prefix else k

def pipe_is_ctrl(self, pipe):
    return pipe['attributes']['type'] == 'control'

def pipe_is_root(self, pipe):
    return pipe['attributes'].get('is_root', False)
def process_pipe(self, pipe, port_id):
    pipe_id = f"pipe_{pipe['pipe_id']}"
    is_ctrl = self.pipe_is_ctrl(pipe)
    self.declare_fwd(port_id, pipe_id, '-->', self.get_fwd_target(pipe.get('fwd', {}), port_id))
    self.declare_fwd(port_id, pipe_id, '-->', self.get_fwd_target(pipe.get('fwd_miss', {}), port_id))
    for entry in pipe.get('entries', []):
        fields_matched = self.pipe_match_criteria(entry, 'match') if is_ctrl else None
        fields_matched = f"{fields_matched}" if fields_matched else ""
        self.declare_fwd(port_id, pipe_id, f"{fields_matched}", self.get_fwd_target(entry.get('fwd', {}), port_id))
    if self.pipe_is_root(pipe):
        self.declare_fwd(port_id, None, '-->', f"p{port_id}.pipe_{pipe_id}")

def get_fwd_target(self, fwd, port_id):
    fwd_type = fwd.get('type', None)
    if not fwd_type:
        return None
    if fwd_type == 'changeable':
        return None
    elif fwd_type == 'pipe':
        pipe_id = fwd.get('pipe_id', fwd.get('value', None))
        target = f"p{port_id}.pipe_{pipe_id}"
    elif fwd_type == 'port':
        port_id = fwd.get('port_id', fwd.get('value', None))
        target = f"p{port_id}.egress"
    else:
        target = f"{fwd_type}"  
    return target

def declare_fwd(self, port_id, pipe_id, arrow, target):
    if target:
        src = f"p{port_id}.pipe_{pipe_id}" if pipe_id else f"p{port_id}.ingress"
        self.append(f"    {src} {arrow} {target}"

def json_to_md(infile, outfile, cfg):
    formatter = MermaidFormatter(cfg)
    data = json.load(infile)
    mermaid_syntax = formatter.format(data)
    outfile.write(mermaid_syntax)

def json_dir_to_md_inplace(dir, cfg):
    for infile in glob.glob(dir + '/**/*.json', recursive=True):
        outfile = os.path.splitext(infile)[0] + '.md'
        print(f"{infile} --> {outfile}")
2. The resulting Markdown can be viewed in several ways, including:

- Microsoft Visual Studio Code (using an available Mermaid plugin, such as this one)
- In the GitHub and GitLab built-in Markdown renderer (after committing the output to a Git repo)
- By pasting only the Flowchart content into the Online FlowChart and Diagram Editor

3. The Python script can be invoked as follows:

```python
python3 doca-flow-viz.py sample_pipeline.json sample_pipeline.md
```

In the case of the `flow_tune_server_dump_pipeline` sample, the script produces the following diagram:
The NVIDIA® BlueField®-3 data-path accelerator (DPA) is an embedded subsystem designed to accelerate workloads that require high-performance access to the NIC engines in certain packet and I/O processing workloads. Applications leveraging DPA capabilities run faster on the DPA than on host. Unlike other programmable embedded technologies, such as FPGAs, the DPA enables a high degree of programmability using the C programming model, multi-process support, tools chains like compilers and debuggers, SDKs, dynamic application loading, and management.

The DPA architecture is optimized for executing packet and I/O processing workloads. As such, the DPA subsystem is characterized by having many execution units that can work in parallel to overcome latency issues (such as access to host memory) and provide an overall higher throughput.

The following diagram illustrates the DPA subsystem. The application accesses the DPA through the DOCA library (DOCA DPA) or the DOCA driver layer (FlexIO SDK). On the host or DPU side, the application loads its code into the DPA (shown as "Running DPA Process") as well as allocates memory, NIC queues, and more resources for the DPA process to access. The DPA process can use device side libraries to access the resources. The provided APIs support signaling of the DPA process from the host or DPU to explicitly pass control or to obtain results from the DPA.
The threads on the DPA can react independently to incoming messages via interrupts from the hardware, thereby providing full bypass of DPU or Arm CPU for datapath operations.

The following sections provide an overview of the DPA platform design.

**Multiple Processes on Multiple Execution Units**

The DPA platform supports multiple processes with each process having multiple threads. Each thread can be mapped to a different execution unit to achieve parallel execution. The processes operate within their own address spaces and their execution contexts are isolated. Processes are loaded and unloaded dynamically per the user's request. This is achieved by the platform's hardware design (i.e., privilege layers, memory translation units, DMA engines) and a light-weight real-time operating system (RTOS). The RTOS enforces the privileges and isolation among the different processes.
**DPA RTOS**

The RTOS is designed to rely on hardware-based scheduling to enable low activation latency for the execution handlers. The RTOS works in a cooperative run-to-completion scheduling model.

Under cooperative scheduling, an execution handler can use the execution unit without interrupts until it relinquishes it. Once relinquished, the execution unit is handed back to the RTOS to schedule the next handler. The RTOS sets a watchdog for the handlers to prevent any handler from unduly monopolizing the execution units.

**DPA Memory and Caches**

The following diagram illustrates the DPA memory hierarchy. Memory accessed by the DPA can be cached at three levels (L1, L2, and L3). Each execution unit has a private L1 data cache. The L1 code cache is shared among all the execution units in a DPA core. The L2 cache is shared among all the DPA cores. The DPA execution units can access external memory via load/store operations through the Memory Apertures.

The external memory that is fetched can be cached directly in L1. The DPA caches are backed by NIC private memory, which is located in the DPU's DDR memory banks. Therefore, the address spaces are scalable and bound only by the size of the NIC’s private memory, which in turn is limited only by the DPU's DDR capacity.

See "Memory Model" for more details.
DPA Access to NIC Accelerators

The DPA can send and receive any kind of packet toward the NIC and utilize all the accelerators that reside on the BlueField DPU (e.g., encryption/decryption, hash computation, compression/decompression).

The DPA platform has efficient DMA accelerators that enable the different execution units to access any memory location accessible by the NIC in parallel and without contention. This includes both synchronous and asynchronous DMA operations triggered by the execution units. In addition, the NIC can DMA data to the DPA caches to enable low-latency access and fast processing. For example, a packet received from the wire may be "DMA-gathered" directly to the DPA's last level caches.

DPA Development

Overview

DOCA Libs and Drivers

The NVIDIA DOCA framework is the key for unlocking the potential of NVIDIA® BlueField®-3 platforms.

DOCA's software environment allows developers to program the DPA to accelerate workloads. Specifically, DOCA includes:

- DOCA DPA SDK – a high-level SDK for application-level protocol acceleration
- DOCA FlexIO SDK – a low-level SDK to load DPA programs into the DPA, manage the DPA memory, create the execution handlers and the needed hardware rings and contexts
- DPACC – DPA toolchain for compiling and ELF file manipulation of the DPA code

Programming Model
The DPA is intended to accelerate datapath operations for the DPU and host CPU. The accelerated portion of the application using DPA is presented as a library for the host application. The code within the library is invoked in an event-driven manner in the context of a process that is running on the DPA. One or many DPA execution units may work to handle the work associated with network events. The programmer specifies different conditions when each function should be called using the appropriate SDK APIs on the host or DPU.

The DPA cannot be used as a standalone CPU.

Management of the DPA, such as loading processes and allocating memory, is performed from a host or DPU process. The host process discovers the DPA capabilities on the device and drives the control plane to set up the different DPA objects. The DPA objects exist as long as the host process exists. When the host process is destroyed, the DPA objects are freed. The host process decides which functions it wants to accelerate using the DPA: Either its entire data plane or only a part of it.

The following diagram illustrates the different processes that exist in the system:
Compiler

DPACC is a compiler for the DPA processor. It compiles code targeted for the DPA processor into an executable and generates a DPA program. A DPA program is a host library with interfaces encapsulating the DPA executable.

This DPA program is linked with the host application to generate a host executable. The host executable can invoke the DPA code through the DPA SDK's runtime.

Compiler Keywords

DPACC implements the following keywords:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Application Usage</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dpa_global</strong></td>
<td>Annotate all event handlers that execute on the DPA and all common</td>
<td>Used by the compiler to generate entry points in the DPA</td>
</tr>
<tr>
<td>Keyword</td>
<td>Application Usage</td>
<td>Comment</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>user-defined datatypes (including user-defined structures) which are passed from</td>
<td>executable and automatically replicate user-defined datatypes between the host and DPA.</td>
</tr>
<tr>
<td></td>
<td>the host to the DPA as arguments.</td>
<td></td>
</tr>
<tr>
<td><strong>dpa_rpc</strong></td>
<td>Annotate all RPC calls which are invoked by the host and execute on the DPA.</td>
<td>Used by the compiler to generate RPC specific entry points.</td>
</tr>
<tr>
<td></td>
<td>RPC calls return a value of uint64_t.</td>
<td></td>
</tr>
</tbody>
</table>

Please refer to NVIDIA DOCA DPACC Compiler for more details.

**FlexIO**

Supported at beta level.

FlexIO is a low-level event-driven library to program and accelerate functions on the DPA.

**FlexIO Execution Model**

To load an application onto the DPA, the user must create a process on the DPA, called a FlexIO process. FlexIO processes are isolated from each other like standard host OS processes.

FlexIO supports the following options for executing a user-defined function on the DPA:

1. FlexIO event handler – the event handler executes its function each time an event occurs. An event on this context is a completion event (CQE) received on the NIC completion queue (CQ) when the CQ was in the armed state. The event triggers an internal DPA interrupt that activates the event handler. When the event handler is activated, it is provided with a user-defined argument. The argument in most cases is a pointer to the software execution context of the event handler.

The following pseudo-code example describes how to create an event handler and attach it to a CQ:
2. RPC – remote, synchronous, one-time call of a specific function. RPC is mainly used for the control path to update DPA memory contexts of a process. The RPC's return value is reported back to the host application.

The following pseudo-code example describes how to use the RPC:

```c
// Device code
__dpa_global__ void myFunc(flexiouintptr_t myArg) {
    struct my_db *db = (struct my_db *)myArg;
    get_completion(db->myCq)
    work();
    arm_cq(myCq);
    // reschedule the thread
    flexio_dev_thread_reschedule();
}

// Host code
main() {

    /* Load the application code into the DPA */
    flexio_process_create(device, application, &myProcess);

    /* Create event handler to run my_func with my_arg */
    flexio_event_handler_create(myProcess, myFunc, myArg, &myEventHandler);

    /* Associate the event handler with a specific CQ */
    create_cq(&myCQ,..., myEventHandler)

    /* Start the event handler */
    flexio_event_handler_run(myEventHandler)
    ...
}

// Device code
__dpa_rpc__ uint64_t myFunc(myArg) {
    struct my_db *db = (struct my_db *)myArg;
    if (db->flag) return 1;
    db->flag = 1;
    return 0;
}
```
FlexIO Memory Management

The DPA process can access several memory locations:

- Global variables defined in the DPA process.
- Stack memory – local to the DPA execution unit. Stack memory is not guaranteed to be preserved between different execution of the same handler.
- Heap memory – this is the process' main memory. The heap memory contents are preserved as long as the DPA process is active.
- External registered memory – remote to the DPA but local to the server. The DPA can access any memory location that can be registered to the local NIC using the provided API. This includes BlueField DRAM, external host DRAM, GPU memory, and more.

The heap and external registered memory locations are managed from the host process. The DPA execution units can load/store from stack/heap and external memory locations. Note that for external memory locations, the window should be configured appropriately using FlexIO Window APIs.

FlexIO allows the user to allocate and populate heap memory on the DPA. The memory can later be used by in the DPA application as an argument to the execution context (RPC and event handler):

```c
// Host code
main() {
    ...

    /* Load the application code into the DPA */
    flexio_process_create(device, application, &myProcess);

    /* run the function */
    flexio_process_call(myProcess, myFunc, myArg, &returnValue);
    ...
}
```
FlexIO allows accessing external registered memory from the DPA execution units using FlexIO Window. FlexIO Window maps a memory region from the DPA process address space to an external registered memory. A memory key for the external memory region is required to be associated with the window. The memory key is used for address translation and protection. FlexIO window is created by the host process and is configured and used by the DPA handler during execution. Once configured, LD/ST from the DPA execution units access the external memory directly.

The access for external memory is not coherent. As such, an explicit memory fencing is required to flush the cached data to maintain consistency. See section "Memory Fences" for more.

The following example code demonstrates the window management:

```c
// Device code
__dpa_rpc__ uint64_t myFunc(arg1, arg2, arg3)
{
    struct flexio_dev_thread_ctx *dtctx;
    flexio_dev_get_thread_ctx(&dtctx);
    uint32_t windowId = arg1;
    uint32_t mkey = arg2;
    uint64_t *dev_ptr;
    flexio_dev_window_config(dtctx, windowId, mkey);
    /* get ptr to the external memory (arg3) from the DPA process address space */
    flexio_dev_status status = flexio_dev_window_ptr_acquire (dtctx, arg3, dev_ptr);
    /* will set the external memory */
    *dev_ptr = 0xff;
    /* flush the data out */
}```
Send and Receive Operation

A DPA process can initiate send and receive operations using the FlexIO outbox object. The FlexIO outbox contains memory-mapped IO registers that enable the DPA application to issue device doorbells to manage the send and receive planes. The DPA outbox can be configured during run time to perform send and receive from a specific NIC function exposed by the DPU. This capability is not available for Host CPUs that can only access their assigned NIC function.

Each DPA execution engine has its own outbox. As such, each handler can efficiently use the outbox without needing to lock to protect against accesses from other handlers. To enforce the required security and isolation, the DPA outbox enables the DPA application to send and receive only for queues created by the DPA host process and only for NIC functions the process is allowed to access.

Like the FlexIO window, the FlexIO outbox is created by the host process and configured and used at run time by the DPA process.
struct flexio_dev_thread_ctx *dtctx;

flexio_dev_get_thread_ctx(&dtctx);

uint32_t outbox = arg1;
flexio_dev_outbox_config(dtctx, outbox);

/* Create some wqe and post it on sq */
/* Send DB on sq*/
flexio_dev_qp_sq_ring_db(dtctx, sq_pi,arg3);

/* Poll CQ (cq number is in arg2) */
return 0;
}

// Host code
main() {

    /* Load the application code into the DPA */
    flexio_process_create(device, application, &myProcess);

    /* Allocate uar */
    uar = ibv_alloc_uar(ibv_ctx);

    /* Create queues*/
    flexio_cq_create(myProcess, ibv_ctx, uar, cq_attr, &myCQ);
    my_hwcq = flexio_cq_get_hw_cq (myCQ);

    flexio_sq_create(myProcess, ibv_ctx, myCQ, uar, sq_attr, &mySQ);
    my_hwsq = flexio_sq_get_hw_sq(mySQ);

    /* Outbox will allow access only for queues created with the same UAR*/
    flexio_outbox_create(process, ibv_ctx, uar, &myOutbox);

    /* Run the function */
    flexio_process_call(myProcess, myFunc, myOutbox, my_hwcq->cq_num, my_hwsq->sq_num,
    &return_value);
}
Synchronization Primitives

The DPA execution units support atomic instructions to protect from concurrent access to the DPA process heap memory. Using those instructions, multiple synchronization primitives can be designed.

FlexIO currently supports basic spin lock primitives. More advanced thread pipelining can be achieved using DOCA DPA events.

DOCA DPA

Supported at beta level.

The DOCA DPA SDK eases DPA code management by providing high-level primitives for DPA work offloading, synchronization, and communication. This leads to simpler code but lacks the low-level control that FlexIO SDK provides.

User-level applications and libraries wishing to utilize the DPA to offload their code may choose DOCA DPA. Use-cases closer to the driver level and requiring access to low-level NIC features would be better served using FlexIO.

The implementation of DOCA DPA is based on the FlexIO API. The higher level of abstraction enables the user to focus on their program logic and not the low-level mechanics.

Info

Refer to DOCA DPA documentation for more details.

Memory Model

The DPA offers a coherent but weakly ordered memory model. The application is required to use fences to impose the desired memory ordering. Additionally, where
applicable, the application is required to write back data for the data to be visible to NIC engines (see the coherency table).

The memory model offers "same address ordering" within a thread. This means that, if a thread writes to a memory location and subsequently reads that memory location, the read returns the contents that have previously been written.

The memory model offers 8-byte atomicity for aligned accesses to atomic datatypes. This means that all eight bytes of read and write are performed in one indivisible transaction.

The DPA does not support unaligned accesses, such as accessing \( N \) bytes of data from an address not evenly divisible by \( N \).

The DPA processes memory can be divided into the following memory spaces:

<table>
<thead>
<tr>
<th>Memory Space</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>Memory locations within the DPA process heap. Referenced as _DPA_HEAP in the code.</td>
</tr>
<tr>
<td>Memory</td>
<td>Memory locations belonging to the DPA process (including stack, heap, BSS and data segment) except the memory-mapped IO. Referenced as _DPA_MEMORY in the code.</td>
</tr>
<tr>
<td>MMIO (memory-mapped I/O)</td>
<td>External memory outside the DPA process accessed via memory-mapped IO. Window and Outbox accesses are considered MMIO. Referenced as _DPA_MMIO in the code.</td>
</tr>
<tr>
<td>System</td>
<td>All memory locations accessible to the thread within Memory and MMIO spaces as described above. Referenced as _DPA_SYSTEM in the code.</td>
</tr>
</tbody>
</table>

The coherency between the DPA threads and NIC engines is described in the following table:

<table>
<thead>
<tr>
<th>Producer</th>
<th>Observer</th>
<th>Cohereency</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPA thread</td>
<td>NIC engine</td>
<td>Not coherent</td>
<td>Data to be read by the NIC must be written back using the appropriate intrinsic (see section &quot;Memory Fence and Cache Control Usage Examples&quot;).</td>
</tr>
<tr>
<td>Producer</td>
<td>Observer</td>
<td>Coherence</td>
<td>Comments</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>NIC engine</td>
<td>DPA Thread</td>
<td>Coherent</td>
<td>Data written by the NIC is eventually visible to the DPA threads. The order in which the writes are visible to the DPA threads is influenced by the ordering configuration of the memory region (see <code>IBV_ACCESS_RELAXED_ORDERING</code>). In a typical example of the NIC writing data and generating a completion entry (CQE), it is guaranteed that when the write to the CQE is visible, the DPA thread can read the data without additional fences.</td>
</tr>
<tr>
<td>DPA thread</td>
<td>DPA thread</td>
<td>Coherent</td>
<td>Data written by a DPA thread is eventually visible to the other DPA threads without additional fences. The order in which writes made by a thread are visible to other threads is undefined when fences are not used. Programmers can enforce ordering of updates using fences (see section &quot;Memory Fences&quot;).</td>
</tr>
</tbody>
</table>

**Memory Fences**

Fence APIs are intended to impose memory access ordering. The fence operations are defined on the different memory spaces. See information on memory spaces under section "Memory Model".

The fence APIs apply ordering between the operations issued by the calling thread. As a performance note, the fence APIs also have a side effect of writing back data to the memory space used in the fence operation. However, programmers should not rely on this side effect. See section "Cache Control" for explicit cache control operations. The fence APIs have an effect of a compiler-barrier which means that memory accesses are not reordered around the fence API invocation by the compiler.

A fence applies between the "predecessor" and the "successor" operations. The predecessor and successor ops can be refenced using `__DPA_R`, `__DPA_W`, and `__DPA_RW` in the code.

The generic memory fence operation can operate on any memory space and any set of predecessor and successor operations. The other fence operations are provided as convenient shortcuts that are specific to the use case. It is preferable for programmers to use the shortcuts when possible.

Fence operations can be included using the `dpaintrin.h` header file.
Generic Fence

```c
void __dpa_thread_fence(memory_space, pred_op, succ_op);
```

This fence can apply to any DPA thread memory space. Memory spaces are defined under section "Memory Model". The fence ensures that all operations (`pred_op`) performed by the calling thread, before the call to `__dpa_thread_fence()`, are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before all operations (`succ_op`) to the memory space after the call to `__dpa_thread_fence()`.

System Fence

```c
void __dpa_thread_system_fence();
```

This is equivalent to calling `__dpa_thread_fence(__DPA_SYSTEM, __DPA_RW, __DPA_RW)`.

Outbox Fence

```c
void __dpa_thread_outbox_fence(pred_op, succ_op);
```

This is equivalent to calling `__dpa_thread_fence(__DPA_MMIO, pred_op, succ_op)`.

Window Fence

```c
void __dpa_thread_window_fence(pred_op, succ_op);
```

This is equivalent to calling `__dpa_thread_fence(__DPA_MMIO, pred_op, succ_op)`.
Memory Fence

```c
void __dpa_thread_memory_fence(pred_op, succ_op);
```

This is equivalent to calling `__dpa_thread_fence(__DPA_MEMORY, pred_op, succ_op)`.

Cache Control

Cache control operations allow the programmer to exercise fine-grained control over data resident in the DPA’s caches. They have an effect of a compiler-barrier. The operations can be included using the `dpaintrin.h` header file.

Window Read Contents Invalidation

```c
void __dpa_thread_window_read_inv();
```

The DPA can cache data that was fetched from external memory using a window. Subsequent memory accesses to the window memory location may return the data that is already cached. In some cases, it is required by the programmer to force a read of external memory (see example under "Polling Externally Set Flag"). In such a situation, the window read contents cached must be dropped.

This function ensures that contents in the window memory space of the thread before the call to `__dpa_thread_window_read_inv()` are invalidated before read operations made by the calling thread after the call to `__dpa_thread_window_read_inv()`.

Window Writeback
Writes to external memory must be explicitly written back to be visible to external entities.

This function ensures that contents in the window space of the thread before the call to `__dpa_thread_window_writeback()` are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before any write operation after the call to `__dpa_thread_window_writeback()`.

---

**Memory Writeback**

Writes to DPA memory space may need to be written back. For example, the data must be written back before the NIC engines can read it. Refer to the coherency table for more.

This function ensures that the contents in the memory space of the thread before the call to `__dpa_thread_writeback_memory()` are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before any write operation after the call to `__dpa_thread_writeback_memory()`.

---

**Memory Fence and Cache Control Usage Examples**

These examples illustrate situations in which programmers must use fences and cache control operations.

In most situations, such direct usage of fences is not required by the application using FlexIO or DOCA DPA SDKs as fences are used within the APIs.

---

**Issuing Send Operation**
In this example, a thread on the DPA prepares a work queue element (WQE) that is read by the NIC to perform the desired operation.

The ordering requirement is to ensure the WQE data contents are visible to the NIC engines read it. The NIC only reads the WQE after the doorbell (MMIO operation) is performed. Refer to coherency table.

<table>
<thead>
<tr>
<th>#</th>
<th>User Code – WQE Present in DPA Memory</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write WQE</td>
<td>Write to memory locations in the DPA (memory space = __DPA_MEMORY)</td>
</tr>
<tr>
<td>2</td>
<td>__dpa_thread_memory_writeback();</td>
<td>Cache control operation</td>
</tr>
<tr>
<td>3</td>
<td>Write doorbell</td>
<td>MMIO operation via Outbox</td>
</tr>
</tbody>
</table>

In some cases, the WQE may be present in external memory. See the description of flexio_qmem below. The table of operations in such a case is below.

<table>
<thead>
<tr>
<th>#</th>
<th>User Code – WQE Present in External Memory</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write WQE</td>
<td>Write to memory locations in the DPA (memory space = __DPA_MMIO)</td>
</tr>
<tr>
<td>2</td>
<td>__dpa_thread_window_writeback();</td>
<td>Cache control operation</td>
</tr>
<tr>
<td>3</td>
<td>Write doorbell</td>
<td>MMIO operation via Outbox</td>
</tr>
</tbody>
</table>

**Posting Receive Operation**

In this example, a thread on the DPA is writing a WQE for a receive queue and advancing the queue's producer index. The DPA thread will have to order its writes and writeback the doorbell record contents so that the NIC engine can read the contents.

<table>
<thead>
<tr>
<th>#</th>
<th>User Code – WQE Present in DPA Memory</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write WQE</td>
<td>Write to memory locations in the DPA (memory space = __DPA_MEMORY)</td>
</tr>
<tr>
<td>#</td>
<td>User Code – WQE Present in DPA Memory</td>
<td>Comment</td>
</tr>
<tr>
<td>----</td>
<td>--------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>2</td>
<td>__dpa_thread_memory_fence(__DPA_W, __DPA_W);</td>
<td>Order the write to the doorbell record with respect to WQE</td>
</tr>
<tr>
<td>3</td>
<td>Write doorbell record</td>
<td>Write to memory locations in the DPA (memory space = __DPA_MEMORY)</td>
</tr>
<tr>
<td>4</td>
<td>__dpa_thread_memory_writeback();</td>
<td>Ensure that contents of doorbell record are visible to the NIC engine</td>
</tr>
</tbody>
</table>

**Polling Externally Set Flag**

In this example, a thread on the DPA is polling on a flag that will be updated by the host or other peer device. The memory is accessed by the DPA thread via a window. The DPA thread must invalidate the contents so that the underlying hardware performs a read.

<table>
<thead>
<tr>
<th>User Code – Flag Present in External Memory</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (f)</td>
<td>flag is a memory location read using a window</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>__dpa_thread_window_read_inv();</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

**Thread-to-thread Communication**

In this example, a thread on the DPA is writing a data value and communicating that the data is written to another thread via a flag write. The data and flag are both in DPA memory.

<table>
<thead>
<tr>
<th>User Code – Thread 1</th>
<th>User Code – Thread 2</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1 = x;</td>
<td>while(* (volatile int</td>
<td>Initial condition, flag = 0</td>
</tr>
<tr>
<td></td>
<td>Thread 1 - write to var1</td>
<td></td>
</tr>
<tr>
<td>User Code – Thread 1</td>
<td>User Code – Thread 2</td>
<td>Comment</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>*)&amp;flag) !=1);</td>
<td><strong>Thread 2 - flag is accessed as a volatile variable, so the compiler preserves the intended program order of reads</strong></td>
<td></td>
</tr>
<tr>
<td>__dpa_thread_memory_fence(__DPA_W, __DPA_W);</td>
<td><strong>Thread 1 – write to flag cannot bypass write to var1</strong></td>
<td></td>
</tr>
<tr>
<td>var_t2 = var1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>flag = 1;</td>
<td><strong>var_t2 must be equal to x</strong></td>
<td></td>
</tr>
<tr>
<td>assert(var_t2 == x);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Setting Flag to be Read Externally

In this example, a thread on the DPA sets a flag that is observed by a peer device. The flag is written using a window.

<table>
<thead>
<tr>
<th>User Code – Flag Present in External Memory</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag = data;</td>
<td>flag is updated in local DPA memory</td>
</tr>
<tr>
<td>__dpa_thread_window_writeback();</td>
<td>Contents from DPA memory for the window are written to external memory</td>
</tr>
</tbody>
</table>

### Polling Completion Queue

In this example, a thread on the DPA reads a NIC completion queue and updates its consumer index.

First, the DPA thread polls the memory location for the next expected CQE. When the CQE is visible, the DPA thread processes it. After processing is complete, the DPA thread updates the CQ's consumer index. The consumer index is read by the NIC to determine whether a completion queue entry has been read by the DPA thread. The consumer index is used by the NIC to monitor a potential completion queue overflow situation.
<table>
<thead>
<tr>
<th>User Code – CQE in DPA Memory</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(*((volatile uint8_t *)cq_op_open) &amp; 0x1 == hw_owner);</td>
<td>Poll CQ owner bit in DPA memory until the value indicates the CQE is in software ownership. Coherency model ensures update to the CQ is visible to the DPA execution unit without additional fences or cache control operations. Coherency model ensures that data in the CQE or referenced by it are visible when the CQE changes ownership to software.</td>
</tr>
<tr>
<td>process_cqe();</td>
<td>User processes the CQE according to the application’s logic.</td>
</tr>
<tr>
<td>cq_cq_index++; // next CQ index. Handle wraparound if necessary</td>
<td>Calculate the next CQ index taking into account any wraparound of the CQ depth.</td>
</tr>
<tr>
<td>update_cq_dbr(cq, cq_index); // writes cq_index to DPA memory</td>
<td>Memory operation to write the new consumer index.</td>
</tr>
<tr>
<td>__dpa_thread_memory_writeback();</td>
<td>Ensures that write to CQ’s consumer index is visible to the NIC. Depending on the application’s logic, the __dpa_thread_memory_writeback() may be coalesced or eliminated if the CQ is configured in overrun ignore mode.</td>
</tr>
<tr>
<td>arm_cq();</td>
<td>Arm the CQ to generate an event if this handler is going to call flexio_dev_thread_reschedule(). Arming the CQ is not required if the handler calls flexio_dev_thread_finish().</td>
</tr>
</tbody>
</table>

**DPA-specific Operations**

The DPA supports some platform-specific operations. These can be accessed using the functions described in the following subsections. The operations can be included using the dpaintrin.h header file.

**Clock Cycles**
Returns a counter containing the number of cycles from an arbitrary start point in the past on the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This function also acts as a compiler barrier, preventing the compiler from moving instructions around the location where it is used.

**Timer Ticks**

Returns the number of timer ticks from an arbitrary start point in the past on the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This intrinsic also acts as a compiler barrier, preventing the compiler from moving instructions around the location where the intrinsic is used.

**Instructions Retired**

Returns a counter containing the number of instructions retired from an arbitrary start point in the past by the execution unit the thread is currently scheduled on.
Note that the value returned by this function in the software thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This intrinsic also acts as a compiler barrier, preventing the compiler from moving instructions around the location where the intrinsic is used.

**Fixed Point Log2**

```c
int __dpa_fxp_log2(unsigned int);
```

This function evaluates the fixed point Q16.16 base 2 logarithm. The input is an unsigned integer.

**Fixed Point Reciprocal**

```c
int __dpa_fxp_rcp(int);
```

This function evaluates the fixed point Q16.16 reciprocal (1/x) of the value provided.

**Fixed Point Pow2**

```c
int __dpa_fxp_pow2(int);
```

This function evaluates the fixed point Q16.16 power of 2 of the provided value.
FlexIO

This chapter provides an overview and configuration instructions for DOCA FlexIO SDK API.

The DPA processor is an auxiliary processor designed to accelerate packet processing and other data-path operations. The FlexIO SDK exposes an API for managing the DPA device and executing native code over it.

The DPA processor is supported on NVIDIA® BlueField®-3 DPUs and later generations.

After DOCA installation, FlexIO SDK headers may be found under `/opt/mellanox/flexio/include` and libraries may be found under `/opt/mellanox/flexio/lib`.

Prerequisites

DOCA FlexIO applications can run either on the host machine or on the target DPU.

Developing programs over FlexIO SDK requires knowledge of DPU networking queue usage and management.

Architecture

FlexIO SDK library exposes a few layers of functionality:

- `libflexio` – library for Host-side operations. It is used for resource management.
- `libflexio_dev` – library for DPA-side operations. It is used for data path implementation.
- `libflexio_libc` – a lightweight C library for DPA device code. `libflexio_libc` may expose very partial functionality compared to a standard `libc`.

A typical application is composed of two parts: One running on the host machine or the DPU target and another running directly over the DPA.

API
Please refer to the NVIDIA DOCA Driver APIs.

**Resource Management**

DPA programs cannot create resources. The responsibility of creating resources, such as FlexIO process, thread, outbox and window, as well as queues for packet processing (completion, receive and send), lies on the DPU program. The relevant information should be communicated (copied) to the DPA side and the address of the copied information should be passed as an argument to the running thread.

**Example**

**Host side:**

1. Declare a variable to hold the DPA buffer address.

   ```c
   flexio_uintptr_t app_data_dpa_daddr;
   ```

2. Allocate a buffer on the DPA side.

   ```c
   flexio_buf_dev_alloc(flexio_process, sizeof(struct my_app_data), &app_data_dpa_daddr);
   ```

3. Copy application data to the DPA buffer.

   ```c
   flexio_host2dev_memcpy(flexio_process, (uintptr_t)app_data, sizeof(struct my_app_data),
                          app_data_dpa_daddr);
   ```

   **struct my_app_data** should be common between the DPU and DPA applications so the DPA application can access the struct fields.

   The event handler should get the address to the DPA buffer with the copied data:
As mentioned previously, the DPU program is responsible for allocating buffers on the DPA side (same as resources). The DPU program should allocate device memory in advance for the DPA program needs (e.g., queues data buffer and rings, buffers for the program functionality, etc).

The DPU program is also responsible for releasing the allocated memory. For this purpose, the FlexIO SDK API exposes the following memory management functions:

- `flexio_event_handler_create(flexio_process, net_entry_point, app_data_dpa_daddr, NULL, flexio_outbox, &app_ctx.net_event_handler)`

```c
__dpa_rpc__ uint64_t event_handler_init(uint64_t thread_arg)
{
    struct my_app_data *app_data;
    app_data = (my_app_data *)thread_arg;
    ...
}
```

- `flexio_status flexio_buf_dev_alloc(struct flexio_process *process, size_t buff_bsize, flexio_ptr_t *dest_daddr_p);`
- `flexio_status flexio_buf_dev_free(flexio_ptr_t daddr_p);`
- `flexio_status flexio_host2dev_memcpy(struct flexio_process *process, void *src_haddr, size_t buff_bsize, flexio_ptr_t dest_daddr);`
- `flexio_status flexio_buf_dev_memset(struct flexio_process *process, int value, size_t buff_bsize, flexio_ptr_t dest_daddr);`

### DPA Memory Management

**Allocating NIC Queues for Use by DPA**
The FlexIO SDK exposes an API for allocating work queues and completion queues for the DPA. This means that the DPA may have direct access and control over these queues, allowing it to create doorbells and access their memory.

When creating a FlexIO SDK queue, the user must pre-allocate and provide memory buffers for the queue's work queue elements (WQEs). This buffer may be allocated on the DPU or the DPA memory.

To this end, the FlexIO SDK exposes the `flexio_qmem` struct, which allows the user to provide the buffer address and type (DPA or DPU).

### Memory Allocation Best Practices

To optimize process device memory allocation, it is recommended to use the following allocation sizes (or closest to it):

- Up to 1 page (4KB)
- $2^6$ pages (256KB)
- $2^{11}$ pages (8MB)
- $2^{16}$ pages (256MB)

Using these sizes minimizes memory fragmentation over the process device memory heap. If other buffer sizes are required, it is recommended to round the allocation up to one of the listed sizes and use it for multiple buffers.

### DPA Window

DPA windows are used to access external memory, such as on the DPU's DDR or host's memory. DPA windows are the software mechanism to use the Memory Apertures mentioned in section "DPA Memory and Caches". To use the window functionality, DPU or host memory must be registered for the device using the `ibv_reg_mr()` call.
Both the address and size provided to this call must be 64 bytes aligned for the window to operate. This alignment may be obtained using the `posix_memalign()` allocation call.

**DPA Event Handler**

**Default Window/Outbox**

The DPA event handler expects a DPA window and DPA outbox structs upon creation. These are used as the default for the event handler thread. Users may choose to set one or both to NULL, in which case there would be no valid default value for one/both of them.

Upon thread invocation on the DPA side, the thread context is set for the provided default IDs. If, at any point, the outbox/window IDs are changed, then the thread context on the next invocation is restored to the default IDs. This means that the DPA Window MKey must be configured each time the thread is invoked, as it has no default value.

**Execution Unit Management**

DPA execution units (EUs) are the equivalent to logical cores. For a DPA program to execute, it must be assigned an EU.

It is possible to set EU affinity for an event handler upon creation. This causes the event handler to execute its DPA program over specific EUs (or a group of EUs).

DPA supports three types of affinity: none, strict, group.

The affinity type and ID, if applicable, are passed to the event handler upon creation using the `affinity` field of the `flexio_event_handler_attr` struct.

For more information, please refer to [NVIDIA DOCA DPA Execution Unit Management Tool](#).

**Execution Unit Partitions**

To work over DPA, an EU partition must be created for the used device. A partition is a selection of EUs marked as available for a device. For the DPU ECPF, a default partition is created upon boot with all EUs available in it. For any other device (i.e., function), the user
must create a partition. This means that running an application on a non-ECPF function without creating a partition would result in failure.

FlexIO SDK uses strict and none affinity for internal threads, which require a partition with at least one EU for the participating devices. Failing to comply with this assumption may cause failures.

**Virtual Execution Units**

Users should be aware that beside the default EU partition, which is exposed to the real EU numbers, all other partitions created use virtual EUs.

For example, if a user creates a partition with the range of EUs 20-40, querying the partition info from one of its virtual HCAs (vHCAs) it would display EUs from 0-20. So, the real EU number, 39 in this example, would correspond to the virtual EU number 19.

**Version API and Backward Compatibility**

FlexIO SDK supports partial backward compatibility. The may follow one of the following options:

1. Work only with the latest version. The user must align their entire code according to the changes in the FlexIO SDK API listed in the document accompanying each version.

2. Ensure partial backward compatibility for the working code. The user must inform the SDK which version they intend to work with. The SDK provides a set of tools that ensure backward compatibility. The set consists of compile-time and runtime tools.

**Version API Toolkit**

To support backward compatibility, the FlexIO SDK uses the macros `FLEXIO_VER` for the host and `FLEXIO_DEV_VER` for the DPA device. The macros have 3 parameters, where the first is the major version (year), the second is the minor version (month), and the third is the sub-minor version (not used, always 0).
**Compile-time**

This toolkit is available for both the host and DPA device. The header files `flexio_ver.h` and `flexio_dev_ver.h` contain the macros `FLEXIO_VER` and `FLEXIO_VER_LATEST` for the host and `FLEXIO_DEV_VER` and `FLEXIO_DEV_VER_LATEST` for the DPA device. For example, to set backward compatibility for version 24.04, the user must declare the following construct for the host:

```c
#include <libflexio/flexio_ver.h>
#define FLEXIO_VER_USED FLEXIO_VER(24, 4, 0)
#include <libflexio/flexio.h>
```

And the user must declare the following construct for the DPA device:

```c
#include <libflexio-dev/flexio_dev_ver.h>
#define FLEXIO_DEV_VER_USED FLEXIO_DEV_VER(24, 4, 0)
#include <libflexio-dev/flexio_dev.h>
```

Where 24 is the major version, and 4 is the minor version.

⚠️ **Warning**

The files `flexio.h` and `flexio_dev.h` have the macros `FLEXIO_CURRENT_VERSION` and `FLEXIO_LAST_SUPPORTED_VERSION` for the host, `FLEXIO_DEV_CURRENT_VERSION` and `FLEXIO_DEV_LAST_SUPPORTED_VERSION` for the DPA device. These versions are provided for internal use and user information. The user should not use these macros.

**Runtime**
This toolkit is only present for the host. For backward compatibility in runtime, the user can call the function `flexio_status flexio_version_set(uint64_t version);` in `flexio.h` once before calling any other function from the API, with the version parameter they wish to work with. The function returns an error in the following cases:

- If the specified version is less than `FLEXIO_LAST_SUPPORTED_VERSION`
- If it exceeds `FLEXIO_CURRENT_VERSION`
- If the function is called again with a version value different from the previous one

```c
status = flexio_version_set(FLEXIO_VER(24, 4, 0));
if (status == FLEXIO_STATUS_FAILED)
{
    return ERROR;
}
```

It is recommended to use the `FLEXIO_VER_USED` macro as a parameter:

```c
flexio_version_set(FLEXIO_VER_USED);
```

**End of Backward Compatibility**

The backward compatibility tools are designed to have an endpoint. With each new version, it is possible to gradually raise the value of `FLEXIO_LAST_SUPPORTED_VERSION` for the host and `FLEXIO_DEV_LAST_SUPPORTED_VERSION` for the DPA device. If `FLEXIO_VER_USED` equals `FLEXIO_LAST_SUPPORTED_VERSION`, then the compiler will issue a warning. This is a sign for the user to start transitioning to a newer version. This way the user has time at least until the next version to modify their code to comply with the older version. If `FLEXIO_VER_USED` is lower than `FLEXIO_LAST_SUPPORTED_VERSION`, then the compiler will issue errors. This is a sign for the user to immediately transition to a newer version. The same behavior for the DPA device.
Application Debugging

Because application execution is divided between the host side and the DPA processor services, debugging may be somewhat challenging, especially since the DPA side does not have a terminal allowing the use of the C stdio library printf services.

Using Device Messaging Stream API

Another logging (messaging) option is to use FlexIO SDK infrastructure to send strings or formatted text in general, from the DPA side to the host side console or file. The host side's flexio.h file provides the flexio_msg_stream_create API function for initializing the required infrastructures to support this. Once initialized, the DPA side must have the thread context, which can be obtained by calling flexio_dev_get_thread_ctx. flexio_dev_msg can then be called to write a string generated on the DPA side to the stream created (using its ID) on the host side, where it is directed to the console or a file, according to user configuration in the creation stage.

It is important to call flexio_msg_stream_destroy when exiting the DPU application to ensure proper clean-up of the print mechanism resources.

Device messages use an internal QP for communication between the DPA and the DPU. When running over an InfiniBand fabric, the user must ensure that the subnet is well-configured, and that the relevant device's port is in active state.

Message Stream Functionality

The user can create as many streams as they see fit, up to a maximum of FLEXIO_MSG_DEV_MAX_STREAMS_AMOUNT as defined in flexio.h.

Every stream has its own messaging level which serves as a filter where messages with a level below that of the stream are filtered out.

The first stream created is the default_stream gets stream ID 0, and it is created with messaging level FLEXIO_MSG_DEV_INFO by default.

The stream ID defined by FLEXIO_MSG_DEV_BROADCAST_STREAM serves as a broadcast stream which means it messages all open streams (with the proper messaging level).

A stream can be configured with a synchronization mode attribute according to the following options:
• **sync** – displays the messages as soon as they are sent from the device to the host side using the verb **SEND**.

• **async** – uses the verb **RDMA write**. When the programmer calls the stream's flush functionality, all the messages in the buffer are displayed (unless there was a wraparound due to the size of messages being bigger than the size allocated for them). In this synchronization mode, the flush should be called at the end of the run.

• **batch** – uses **RDMA write** and **RDMA write with immediate**. It works similarly to the async mode, except the fact each batch size of messages is being flushed and therefore displayed automatically in every batch. The purpose is to allow the host to use fewer resources for device messaging.

**Device Messaging Assumptions**

Device messaging uses RPC calls to create, modify, and destroy streams. By default, these RPC calls run with affinity **none**, which requires at least one available EU on the default group. If the user wants to set the management affinity of a stream to a different option (any affinity option is supported, including forcing **none**, which is the default behavior) they should specify this in the stream attributes using the **mgmt_affinity** field.

**Printf Support**

Only limited functionality is implemented for printf. Not all libc printf is supported.

Please consult the following list for supported modifiers:

- **Formats** – `%c, %s, %d, %ld, %u, %lu, %i, %li, %x, %hx, %hx, %lx, %lX, %lo, %p, %%

- **Flags** – -, *, - , + , #

- **General supported modifiers**:
  - "0" padding
  - Min/max characters in string
• General unsupported modifiers:
  o Floating point modifiers – %e, %E, %f, %lf, %LF
  o Octal modifier %o is partially supported
  o Precision modifiers

**Core Dump**

If the DPA process encounters a fatal error, the user can create a core dump file to review the application's status at that point using a GDB app.

Creating a core dump file can be done after the process has crashed (as indicated by the flexio_err_status API) and before the process is destroyed by calling the flexio_coredump_create API.

Recommendations for opening DPA core dump file using GDB:

• Use the gdb-multiarch application

• The Program parameter for GDB should be the device-side ELF file
  o Use the dpacc-extract tool (provided with the DPACC package) to extract the device-side ELF file from the application's ELF file

**FlexIO Samples**

This section describes samples based on the FlexIO SDK. These samples illustrate how to use the FlexIO API to configure and execute code on the DPA.

**Running FlexIO Sample**

The FlexIO SDK samples serve as a reference for building and running FlexIO-based DPA applications. They provide a collection of out-of-the-box working DPA applications that encompass the basic functionality of the FlexIO SDK.
**Documentation**

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

- Refer to NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples

**Minimal Requirements**

The user must have the following installed:

- DOCA DPACC package
- DOCA RDMA package
- pkg-config package
- Python3 package
- Gcc with version 7.0 or higher
- Meson package with version 0.53.0 or higher
- Ninja package
- DOCA FlexIO SDK

**Sample Structure**

Each sample is situated in its own directory and is accompanied by a corresponding description in README files. Every sample comprises two applications:

- The first, located in the device directory, is designed for DPA
- The second, found in the host directory, is intended for execution on the DPU or host in a Linux OS environment
Additionally, there is a common directory housing libraries for the examples. These libraries are further categorized into device and host directories to facilitate linking with similar applications. Beyond containing functions and macros, these libraries also serve as illustrative examples for how to use them.

The list of the samples:

- **flexio.rpc** – sample demonstrating how to run RPC functions from DPA
- **packet_processor** – sample demonstrating how to process a package

### Building the Samples

```bash
cd /opt/mellanox/fleio/samples/
./build.sh --check-compatibility --rebuild
```

### Samples

**flexio_rpc**

This sample application executes FlexIO with a remote process call.

The device program calculates the sum of 2 input parameters, prints the result, and copies the result back to the host application.

This sample demonstrates how applications are built (DPA and host), how to create processes and message streams, how to open the IBV device, and how to use RPC from the host to DPA function.

### Compilation

```bash
cd /opt/mellanox/fleio/samples/
./build.sh --check-compatibility --rebuild
```
The output path:

```
/opt/mellanox/flexio/samples/build/flexio_rpc/host/flexio_rpc
```

**Usage**

```
<sample_root>/build/flexio_rpc/host/flexio_rpc <mlx5_device> <arg1> <arg2>
```

Where:

- `mlx5_device` – IBV device with DPA
- `arg1` – first numeric argument
- `arg2` – second numeric argument

**Example:**

```
$ /opt/mellanox/flexio/samples/build/flexio_rpc/host/flexio_rpc mlx5_0 44 55
Welcome to 'Flex IO RPC' sample
Registered on device mlx5_0
/ 2/Calculate: 44 + 55 = 99
Result: 99
Flex IO RPC sample is done
```

**flexio_packet_process**

This example demonstrates packet processing handling.

The device application implements a handler for `flexio_pp_dev` that receives packets from the network, swaps MAC addresses, inserts some text into the packet, and sends it back.
This allows the user to send UDP packets (with a packet length of 65 bytes) and check the content of returned packets. Additionally, the console displays the execution of packet processing, printing each new packet index. Device messaging operates in synchronous mode (i.e., each message from the device received by the host is output immediately).

This sample illustrates how applications work with libraries (DPA and host), how to create SQ, RQ, CQ, memory keys, and doorbell rings, how to create and use DPA memory buffers, how to use UAR, and how to create and run event handlers.

Compilation

```
    cd /opt/mellanox/flexio/samples/
    ./build.sh --check-compatibility --rebuild
```

The output path:

```
    /opt/mellanox/flexio/samples/build/packet_processor/host/flexio_packet_processor
```

Usage

```
    <sample_root>/build/packet_processor/host/flexio_packet_processor <mlx5_device>
```

Where:

- `mlx5_device` – name of IB device with DPA
- `--nic-mode` – optional parameter indicating that the application is run from the host. If the application is run from DPU, then the parameter should not be used.

For example

```
    $sudo /build/packet_processor/host/flexio_packet_processor mlx5_0
```
The application must run with root privileges.

**Running with Traffic**

Run host-side sample:

```
$ cd <sample_root>
$ sudo ./build/packet_processor/host/flexio_packet_processor mlx5_0
```

Use another machine connected to the setup running the application. Bring the interface used as packet generator up:

```
$ sudo ifconfig my_interface up
```

Use `scapy` to run traffic to the device the application is running on:

```
$ python

>>> from scapy.all import *
>>> from scapy.layers.inet import IP, UDP, Ether

>>> sendp(Ether(src="02:42:7e:7f:eb:02", dst="52:54:00:79:db:d3")/IP()/UDP()/Raw(load="===============12345678"), iface="my_interface")
```

**Note**

Source MAC must be same as above as the application defines a steering rule for it. Destination MAC can be anything.
The packets can be viewed using `tcpdump`:

```
$ sudo tcpdump -i my_interface -en host 127.0.0.1 -X
```

Example output:

```
11:53:51.422075 02:42:7e:7f:eb:02 > 52:54:00:12:34:56, ethertype IPv4 (0x0800), length 65:
127.0.0.1.domain > 127.0.0.1.domain: 15677 op7+% [b2&3=0x3d3d] [15677a] [15677q] [15677n]
[15677au][|domain]
 0x0000: 4500 0033 0001 0000 4011 7cb7 7f00 0001 E..3....@.|.....
 0x0010: 7f00 0001 0035 0035 001f 42c6 3d3d 3d3d ....5.5..B.==== <- Original data
 0x0020: 3d3d 3d3d 3d3d 3d3d 3d3d 3d3d 3d3d 3d3d 3233 3435 =========12345
 0x0030: 3637 38 678
```

```
11:53:51.700038 52:54:00:12:34:56 > 02:42:7e:7f:eb:02, ethertype IPv4 (0x0800), length 65:
127.0.0.1.domain > 127.0.0.1.domain: 26144 op8+% [b2&3=0x4576] [29728a] [25966q] [25701n]
[28015au][|domain]
 0x0000: 4500 0033 0001 0000 4011 7cb7 7f00 0001 E..3....@.|.....
 0x0010: 7f00 0001 0035 0035 001f 42c6 6620 4576 ....5.5..B.f.Ev <- Modified data
 0x0020: 656e 7420 6465 2a2a 2a2a 2a2a 2a2a 2a2a ent.demo********
 0x0030: 2a2a 2a 3**
```
DPA Application Authentication

DPA Application Authentication is supported at beta level for BlueField-3.

DPA Application Authentication is currently only supported with statically linked libraries. Dynamically linked libraries are currently not supported.

This section provides instructions for developing, signing, and using authenticated BlueField-3 data-path accelerator (DPA) applications. It includes information on:

- Principles of root of trust and structures supporting it
- Device ownership transfer/claiming flow (i.e., how the user should configure the device so that it will authenticate the DPA applications coming from the user)
- Crypto signing flow and ELF file structure and tools supporting it

Root of Trust Principles

Signing of 3rd Party DPA App Code

NVIDIA® BlueField®-3 introduces the ability for customers/device owners to sign applications running on the DPA with their private key and have it authenticated by a device-embedded certificate chain. This provides the benefit of ensuring that only code permitted by the customer can run on the DPA. The customer can be any party writing code intended to run on the DPA (e.g., a cloud service provider, OEM, etc).

The following figure illustrates the signature of customer code. This signature will allow NVIDIA firmware to authenticate the source of the application's code.

*Example of Customer DPA Code Signed by Customer for Authentication*
The high-level scheme is as follows (see figure "Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device"):

The numbers of these steps correspond to the numbers indicated in the figure below.

1. Customer provides NVIDIA Enterprise Support the public key for device ownership.

2. NVIDIA signs the customer's public key and sends it back to the customer.

3. Customer uploads the NVIDIA-signed public key to the device, enabling "Transfer of Ownership" to the customer (from NVIDIA).

4. Using the private key corresponding to the public key uploaded to the device, the customer can now enable DPA authentication and load the root certificate used for
5. DPA app code crypto-signed by the customer serves to authenticate the source of the app code.

The public key used to authenticate the DPA app is provided as part of the certificate chain (leaf certificate), together with the DPA firmware image.

6. App code and the owner signature serves to authorize the app execution by the NVIDIA firmware (similar to NVIDIA own signature).

**Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device**

The following sections provide more details about this high-level process.

**Verification of Authenticity of DPA App Code**

Authentication of application firmware code before authorization to execute shall consist of validation of the customer certificate chain and customer signature using the customer's public key.

**Public Keys (Infrastructure, Delivery, and Verification)**
For the purposes of the authentication verification of the application firmware, the public key must be securely provided to the hardware. To do so, a secure Management Component Control (MCC) Flow shall be used. Using this, the content of the downloaded certificate is enveloped in an MCC Download Container and signed by NVIDIA Private Key.

The following is an example of how to use the MCC flow describes in detail the procedures, tools and structures supporting this (Section "Loading of CSP CA Certificates and Keys and Provisioning of DPA Firmware to Device" describes the high-level flow for this).

The following command burns the certificate container:

```bash
flint -d <mst device> -i <signed-certificate-container> burn
```

Two use cases are possible:

- The DPA application is developed internally in NVIDIA, and the authentication is based on internal NVIDIA keys and signing infrastructure
- The DPA application is developed by a customer, and the authentication is based on the customer certificate chain

In either case, the customer must download the relevant CA certificate to the device.

**ROT Certificate Chain**
This figure illustrates the build of the certificate chain used for validation of DPA app images. The leaf certificate of these chains is used to validate the DPA application supplied by the customer (with ROT from customer CA). The NVIDIA certificate chain for validation of DPA applications (built internally in NVIDIA) is structured in a very similar way. OEMDpaCert CA is the root CA which can be used by the customer to span their certificate chain up to the customer leaf certificate which is used for validating the signature of the application's image. Similarly, NVDADpaCert CA is the root CA used internally in NVIDIA to build the DPA certificate chain for validation of NVIDIA DPA apps.

Customer private keys must be kept secure and are the sole responsibility of the customer to maintain. It is recommended to have a set of keys ready and usable by customer for redundancy purposes. The whole customer certificate chain, including root CA and leaf, must not exceed 4 certificates.

The NVDA_CACert_DPA and OEM_CACert_DPA certificates are self-signed and trusted because they are loaded by the secure MCC flow and authenticated by the firmware.

The customer certificate chain beyond OEM_CACert_DPA is delivered with the DPA image, including the leaf certificate that is used for validating the cryptographic signature of the DPA firmware (see table "ELF Crypto Data Section Fields Description").
For more details on the certificates and their location in the flash, contact NVIDIA Enterprise Support to obtain the *Flash Application Note*. The rest of the certificate chain used for the DPA firmware authentication includes:

- For NVIDIA-signed images (e.g., figure "ROT Certificate Chain"): NVDA DPA root certificate (*NVDA_CACert_DPA* can be downloaded [here](#))

- For customer-signed images (e.g., figure "ROT Certificate Chain"): Customer CA certificate, customer product, and customer leaf certificates

In both cases (NVIDIA internal and customer-signed) these parts of the certificate chain are attached to the DPA firmware image.

**Loading of CSP CA Certificates and Keys and Provisioning of DPA Firmware to Device**

The figure "Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device" shows, at high-level, the procedures for loading user public keys to the device, signing and loading of customer certificates MCC container, and downloading the DPA firmware images.

For clarity, the hierarchy of ROT validation is as follows:

1. Customer public key to be used for customer TLVs and *CACert_DPA* certificate validation, *PK_TLV* (i.e., *NV_LC_NV_PUBLIC_KEY*):

   1. For a device whose DPA authentication ability the customer wishes to enable for the first time, the customer must get it signed and authenticated by NVIDIA keys by reaching out to NVIDIA Enterprise Support. The complete flow is described in "Device Ownership Claiming Flow".

   2. After *PK_TLV* is loaded, it can be updated by authenticating the update using either the same *PK_TLV*. The complete flow is described in "Device Ownership Claiming Flow".

   3. Authentication of TLV for enabling/disabling DPA authentication is also validated by the *PK_TLV*. The complete flow is described in section "DPA Authentication Enablement".

2. Loading of CA certificate (*CACert_DPA*) to be used for DPA code validation. It is authenticated using the same *PK_TLV*. 
The complete flow is described in "Uploading DPA Root CA Certificate".

3. The public key in the leaf of the certificate chain anchored by CACert_DPA is used for authentication of the DPA firmware Image.

The structure of the ELF file containing the DPA app and the certificate chain is described in "ELF File Structure".

A scalable and reliable infrastructure is required to support many users. The customer must also have an infrastructure to support their own code signing process according to their organization's security policy. Both matters are out of the scope of this document.

Note

Trying to utilize the DPA signing flow in a firmware version prior to DOCA 2.2.0 is not supported.

Device Ownership Claiming Flow

NVIDIA networking devices allow the user of the device to customize the configurations, and in some cases change the behavior of the device. This set of available customizations is controlled by higher level NVIDIA configurations that come either as part of the device firmware or as a separate update file. To allow customers/device owners to change the set of available configurations and allowed behaviors, each device can have a device owner who is allowed to change the default behaviors and configurations of the device, and to change what configurations are exposed to the user.

The items controlled by the customer/device owner are:

- Device configurations: The customer/device owner can change the default value of any configuration available to users. They can also prevent users from changing the value.
• Trusted root certificates: The customer/device owner can control what root
certificates the device trusts. These certificates control various behaviors (e.g., what 3rd
party code the BlueField DPA accepts).

After the device has the public key of the owner, whenever an NVconfig file is signed with
this key, one of two things must be true:

• The `nv_file_id` field in the NVconfig file must have the parameter `keep_same_priority` as
  `True`; or

• The NVconfig file must contain the public key itself (so the public key is rewritten to
  the device)

Otherwise, the public key is removed from the device, and as such will not accept files
signed by the matching private key.

**Detailed Ownership Claiming Flow**

1. Customer generates a private-public key pair, and a UUID for the key pair.

   1. Generating UUID for the key pair:

```
   uuidgen -t
```

   Example output:

```
   77dd4ef0-c633-11ed-9e20-001dd8b744ff
```

   2. Generating an RSA key pair:

```
   openssl genrsa -out OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.pem 4096
```

   Example output:
3. Extracting the public key file from the RSA key pair:

```
```

Output:

```
writing RSA key
```

The public key should look similar to the following:

```
-----BEGIN PUBLIC KEY-----
MIICIdANBgkqhkiG9w0BAQEFAAAOCgAg8AMIICCGKCAAgEASFijde+27A3p7MoZnlm
mtpyUHO1j9AUeKatoHUXkWRiopL9PuswL1KcGfWJ5nzieP5ZRevTHraYlQcru4ofr
W9NBExqlw5s2n7kiFwCCv6FKUqZAuMjTpfuNtv9oC4v0ZiX4TQqWDND8hy+1L
hpF3QLRjvx4G6uHlfvENSwagershuKD0Rl6FaZ1g9S9ldxC0DvTdeUDPqQ0m4
CwEs/3xnksNRLUM+TiPEZoc5MoEoKyJv4GfbGtahbDct5sr9RqAtqTNUSD19B0jr
XoQBHQpQqRgYd31Q31Fhh3G9GjtoAcUQ6i0Gct3DXKFTAdDV3Ly01ijFNRoKUWdhHT
pjDKzNmZAxylIZo/buc24TCgj1yPyFboJtpnHmltyxfm9e+EJsdSlpRiX8YTWwK
alzNj0VSwULwbKow5Gu5FpE/uXDE3cXjLOUNnKihszFv4qksQjKaK4GszXge+
jifEwsDKw5+cuWd9ihnyLrlWF23+OX055jxFDJE8UthOD+3j3GmP3ke1lz2YP
Qvh3lTPRsQqltaiYh+CivqaCHC0voMOP1iAEZ/rW85p6i8EsudNMG2ElRyUl
SznBzI/OxMk4qKx9nGgjaP2YjmcPw2Ffc9zZcwL57ThEOhly56w3E9xwBvZINLe
gMuOIWsui1FK3IiiGxMSCUZQsCAwEAAQ==
-----END PUBLIC KEY-----
```

2. Customer provides NVIDIA Enterprise Support the public key for device ownership with its UUID.
3. NVIDIA generates a signed NVconfig file with this public key and sends it to the customer. This key may only be applied to devices that do not have a device ownership key installed yet.

4. Customer uses mlxconfig to install the OEM key on the needed devices.

```bash
mlxconfig -d /dev/mst/<dev> apply oem_public_key_nvconfig.bin
```

To check if the upload process has been successful, the customer can use mlxconfig to query the device and check if the new public key has been applied. The relevant parameters to query are LC_NV_PUB_KEY_EXP, LC_NV_PUB_KEY_UUID, and LC_NV_PUB_KEY_0_255.

Example of query command and expected response:

```bash
mlxconfig -d <dev>-e q LC_NV_PUB_KEY_0_255
```

### Uploading DPA Root CA Certificate

After uploading a device ownership public key to the device, the owner can upload DPA root CA certificates to the device. There can be multiple DPA root CA certificates on the device at the same time.

If the owner wants to upload authenticated DPA apps developed by NVIDIA, they must upload the NVIDIA DPA root CA certificate found [here](#).

If the owner wants to sign their own DPA apps, they must create another public-private key pair (in addition to the device ownership key pair), create a certificate containing the DPA root CA public key, and create a container with this certificate using mlxdpda.

To upload a signed container with a DPA root CA certificate to the device, mlxdpda must be used. This can be done both for either NVIDIA or customer-created certificates.

### Generating DPA Root CA Certificate
1. Create a DER encoded certificate containing the public key used to validate DPA apps.

1. Generating a certificate and a new key pair:

```
openssl req -x509 -newkey rsa:4096 -keyout OEM-DPA-root-CA-key.pem -outform der -out
OEM-DPA-root-CA-cert.crt -sha256 -nodes -subj
"/C=XX/ST=OEMStateName/L=OEMCityName/O=OEMCompanyName/OU=OEMCompanySect-
days 3650
```

**Note**

Both SHA256 and SHA512 are supported in cert. Only a RSA 4096 key is supported. The size of each certificate in DER format must be less than 1792 bytes.

Output:

```
Generating a 4096 bit RSA private key
......++
........................++
writing new private key to 'OEM-DPA-root-CA-key.pem'
-----
```

2. Create a container for the certificate and sign it with the device ownership private key.

1. To create and add a container:

```
mlxdpa --cert_container_type add -c <cert.der> -o <path to output> --life_cycle_priority
<Nvidia/OEM/User> create_cert_container
```
2. To sign a container:

```
mlxdpa --cert_container <path to container> -p <key file> --keypair_uuid <uuid> --cert_uuid <uuid> --life_cycle_priority <Nvidia/OEM/User> -o <path-to-output> sign_cert_container
```

Certificate container signed successfully!

**Manually Signing Container**

If the server holding the private key cannot run `mlxdpa`, it is possible to manually sign the certificate container and add the signature to the container. In that case, the following process should be followed:

1. Generate unsigned cert container:

```
mlxdpa --cert_container_type add -c <.DER-formatted-certificate> -o <unsigned-container-path> -keypair_uuid <uuid> --cert_uuid <uuid> --life_cycle_priority OEM create_cert_container
```

2. Generate signature field header:

```
echo "90 01 02 0C 10 00 00 00 00 00 00 00" | xxd -r -p - <signature-header-path>
```

3. Generate signature of container (in whatever way, this is an example only):

```
openssl dgst -sha512 -sign <private-key-pem-file> -out <container-signature-path> <unsigned-container-path>
```

4. Concatenate unsigned container, signature header, and signature into one file:
Uploading Certificates

Upload each signed container containing the desired certificates for the device.

```bash
flint -d <dev> -i <signed-container> -y b
```

Output example:

```
-l- Downloading FW ...
FSMST_INITIALIZATION - OK
Writing DIGITAL_CACERT_REMOVAL component - OK
-l- Component FW burn finished successfully.
```

Removing Certificates

To remove root CA certificates from the device, the user must apply a certificate removal container signed by the device ownership private key.

There are two ways to remove certificates, either removing all certificates, or removing all installed certificates:

- Removing all root CA certificates from the device:

  1. Generate a signed container to remove all certificates.

  1. Created certificate container:
mlxdpa --cert_container_type remove --remove_all_certs -o <path-to-container> --life_cycle_priority <Nvidia/OEM/User> create_cert_container

Output example:

Certificate container created successfully!

2. Sign certificate container:

mlxdpa --cert_container <path-to-container> -p <key-file> --keypair_uuid <uuid> --life_cycle_priority <Nvidia/OEM/User> -o <path-to-signed-container> sign_cert_container

Output example:

Certificate container signed successfully!

2. Apply the container to the device.

flint -d <dev> -i <signed-container> -y b

Output example:

- I- Downloading FW ...
  FSMST_INITIALIZE - OK
  Writing DIGITAL_CACERT_REMOVAL component - OK
  - I- Component FW burn finished successfully.

- Removing specific root CA certificates according to their UUID:
1. Generate a signed container to remove certificate based on UUID.

1. Create the container.

```
mlxdpa --cert_container_type remove --cert_uuid <uuid> -o <path-to-container> --life_cycle_priority <Nvidia/OEM/User> create_cert_container
```

Output example:

```
Certificate container created successfully!
```

2. Sign the container:

```
mlxdpa --cert_container <path-to-container> -p <key-file> --keypair_uuid <uuid> --cert_uuid <uuid> --life_cycle_priority <Nvidia/OEM/User> -o <path to output> sign_cert_container
```

Output example:

```
Certificate container signed successfully!
```

2. Apply the container to the device:

```
flint -d <dev> -i <signed container> -y b
```

Output:

```
-\- Downloading FW ...
FSMST_INITIALIZE - OK
Writing DIGITAL_CACERT_REMOVAL component - OK
```
DPA Authentication Enablement

After the device has a device ownership key and DPA root CA certificates installed, the owner of the device can enable DPA authentication. To do this, they must create an NVconfig file, sign it with the device ownership private key, and upload the NVconfig to the device.

Generating NVconfig Enabling DPA Authentication

1. Create XML with TLVs to enable DPA authentication.

   1. Get list of available TLVs for this device:

      mlxconfig -d /dev/mst/<dev> gen_tlvs_file enable_dpa_auth.txt

      Output:

      Saving output...
      Done!

      Example part of the generated text file:

      file_applicable_to 0
      file_comment 0
      file_signature 0
      file_dbg_fw_token_id 0
      file_cs_token_id 0
      file_btc_token_id 0
      file_mac_addr_list 0
      file_public_key 0
      file_signature_4096_a 0
2. Edit the text file to contain the following TLVs:

<table>
<thead>
<tr>
<th>TLV</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_applicable_to</td>
<td>1</td>
</tr>
<tr>
<td>nv_file_id_vendor</td>
<td>1</td>
</tr>
<tr>
<td>nv_dpa_auth</td>
<td>1</td>
</tr>
</tbody>
</table>

3. Convert the .txt file to XML format with another mlxconfig command:

```bash
mlxconfig -a gen_xml_template enable_dpa_auth.txt enable_dpa_auth.xml
```

Output:

```
Saving output...
Done!
```

The generated .xml file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.mellanox.com/config">
  <file_applicable_to ovr_en='1' rd_en='1' writer_id='0'>
    <psid></psid>
    <psid_branch></psid_branch>
  </file_applicable_to>
  <nv_file_id_vendor ovr_en='1' rd_en='1' writer_id='0'>
    <!-- Legal Values: False/True -->
    <disable_override></disable_override>
    <!-- Legal Values: False/True -->
    <keep_same_priority></keep_same_priority>
  </nv_file_id_vendor>
</config>
```
4. Edit the XML file and add the information for each of the TLVs, as seen in the following example XML file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.mellanox.com/config">

<file_applicable_to ovr_en='0' rd_en='1' writer_id='0'>
  <psid>TODO</psid>
  <psid_branch>TODO</psid_branch>
</file_applicable_to>

<nv_file_id_vendor ovr_en='0' rd_en='1' writer_id='0'>
  <disable_override>False</disable_override>
  <keep_same_priority>True</keep_same_priority>
  <per_tlv_priority>False</per_tlv_priority>
  <erase_lower_priority>False</erase_lower_priority>
  <file_version>TODO</file_version>
  <day>TODO</day>
</nv_file_id_vendor>
</config>
```
2. Convert XML file to binary NVconfig file and sign it using mlxconfig:

```
mlxconfig -p OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.pem -u 77dd4ef0-c633-11ed-9e20-001dd8b744ff create_conf enable_dpa_auth.xml enable_dpa_auth.bin
```

Output of `create_conf` command:

```
<month>TODO</month>
<year>TODO</year>
<seconds>TODO</seconds>
<minutes>TODO</minutes>
<hour>TODO</hour>
</nv_file_id_vendor>

<nv_dpa_auth ovr_en='0' rd_en='1' writer_id='0'>
    <dpa_auth_en>True</dpa_auth_en>
</nv_dpa_auth>
</config>

**Note**

In `nv_file_id_vendor`, `keep_same_priority` must be `True` to avoid removing the ownership public key from the device. More information they can be found in section "Device Ownership Claiming Flow".

**Note**

The `ovr_en` should be set to 0. This can ignore user priority changing `nv_dpa_auth`.
3. Upload NVconfig file to device by writing the file to the device:

   mlxconfig -d /dev/mst/<dev> apply enable_dpa_auth.bin

   Output:

   Saving output...
   Done!

4. Verify that the device has DPA authentication enabled by reading the status of DPA authentication from the device:

   mlxconfig -d /dev/mst/<dev> -e q DPA_AUTHENTICATION

   Output:

   Device #1:
   
   Device type: BlueField3
   ...
   ...

   Configurations:                  Default      Current      Next Boot
   RO  DPA_AUTHENTICATION            True(1)      True(1)      True(1)

   The DPU's factory default setting is configured with dpa_auth_en=0 (i.e., DPA applications can run without authentication). To prevent configuration change by any user, it is strongly recommended for the customer to generate and install NVconfig with dpa_auth_en=0/1, according to their preferences, with ovr_en=0.
Manually Signing NVconfig File

If the server holding the private key cannot run mlxconfig, it is possible to manually sign the binary NVconfig file and add the signature to the file. In this case, the following process should be followed instead of step 2:

1. Generate unsigned NVconfig bin file from the XML file:

   mlxconfig create_conf <xml-nvconfig-path> <unsigned-nvconfig-path>

2. Generate random UUID for signature:

   uuidgen -r | xxd -r -p - <signature-uuid-path>

3. Generate signature of NVconfig bin file (in whatever way, this is an example only):

   openssl dgst -sha512 -sign <private-key-pem-file> -out <nvconfig-signature-path> <unsigned-nvconfig-path>

4. Split the signature into two parts:

   head -c 256 <nvconfig-signature-path> > <signature-part-1-path> && tail -c 256 <nvconfig-signature-path> > <signature-part-2-path>

5. Add signing key UUID:

   echo "<signing-key-UUID>" | xxd -r -p - <signing-key-uuid-path>

Use the signing key UUID, which must have a length of exactly 16 bytes, in a format like aa9c8c2f-8b29-4e92-9b76-2429447620e0.
6. Generate headers for signature struct:

```
echo "03 00 01 20 06 00 00 08 00 00 00 00" | xxd -r -p - <signature-1-header-path>
echo "03 00 01 20 06 00 00 0C 00 00 00 00" | xxd -r -p - <signature-2-header-path>
```

7. Concatenate everything:

```
cat <unsigned-nvconfig-path> <signature-1-header-path> <signature-uuid-path> <signing-key-uuid-path> <signature-part-1-path> <signature-2-header-path> <signature-uuid-path> <signing-key-uuid-path> <signature-part-2-path> > <signed-nvconfig-path>
```

**Device Ownership Transfer**

The device owner may change the device ownership key to change the owner of the device or to remove the owner altogether.

**First Installation**

To install the first `OEM_PUBLIC_KEY` on the device, the user must upload an NVCONFIG file signed by NVIDIA. This file would contain the 3 `FILE_OEM_PUBLIC_KEY` TLVs of the current user.

**Removing Device Ownership Key**

Before removing the device ownership key completely, it is recommended that the device owner reverts any changes made to the device since it is not possible to undo them after the key is removed. Mainly, the root CA certificates installed by the owner should be removed.

1. To remove device ownership key completely, follow the steps in section "Generating NVconfig Enabling DPA Authentication" to create an XML file with TLVs.
2. Edit the XML file to contain the following TLVs:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.mellanox.com/config">

<file_applicable_to ovr_en='0' rd_en='1' writer_id='0'>
  <psid>MT_0000000911</psid>
  <psid_branch></psid_branch>
</file_applicable_to>

<nv_file_id_vendor ovr_en='0' rd_en='1' writer_id='0'>
  <disable_override>False</disable_override>
  <keep_same_priority>False</keep_same_priority>
  <per_tlv_priority>False</per_tlv_priority>
  <erase_lower_priority>False</erase_lower_priority>
  <file_version>0</file_version>
  <day>17</day>
  <month>7</month>
  <year>7e7</year>
  <seconds>1</seconds>
  <minutes>e</minutes>
  <hour>15</hour>
</nv_file_id_vendor>
</config>
```

The TLVs in this file are the only TLVs that will have OEM priority after this file is applied, and as the device ownership key will no longer be on the device, the OEM will no longer be able to change the TLVs. To have OEM priority TLVs on the device after removing the device ownership key, add to this XML any TLV that must stay as default on the device.

3. Convert the XML file to a binary NVconfig TLV file signed by the device ownership key as described in section "Generating NVconfig Enabling DPA Authentication".

4. Apply the NVconfig file to the device as described in section "Generating NVconfig Enabling DPA Authentication".

---

**Changing Device Ownership Key**
To transfer ownership of the device to another entity, the previous owner can change the device ownership public key to the public key of the new owner.

To do this, they can use an NVconfig file, and include in it the following TLVs:

```
<nv_ls_nv_public_key_0 ovr_en='0' rd_en='1' writer_id='0'>
  <public_key_exp>65537</public_key_exp>
  <keypair_uuid>77dd4ef0-c633-11ed-9e20-001dd8b744ff</keypair_uuid>
</nv_ls_nv_public_key_0>

<nv_ls_nv_public_key_1 ovr_en='0' rd_en='1' writer_id='0'>
  <key>
    b1:e0:6e:ae:1c:81:70:10:d4:b0:6a:07:ab:b2:1b:
    46:8e:da:00:71:44:3a:97:41:9c:7b:70:d7:28:54:
    c0:00:35:77:2f:2a:35:be:31:4d:ac:e2:94:85:d8:
    53:a6:
  </key>
</nv_ls_nv_public_key_1>

<nv_ls_nv_public_key_2 ovr_en='0' rd_en='1' writer_id='0'>
  <key>
    8a:e0:6b:33:5e:07:be:8d:f8:84:c2:c0:ca:c1:2f:
  </key>
</nv_ls_nv_public_key_2>
```
If the transfer is internal, the owner should set `keep_same_priority=True` in `nv_file_id_vendor TLV` and only include the 3 `nv_ls_nv_public_key_*` TLVs, `file_applicable_to` and `nv_file_id_vendor` TLVs in the NVconfig file.

If the transfer is to another OEM/CSP, the owner should clean the device (similarly to removing the device ownership key) and set `keep_same_priority=False` in `nv_file_id_vendor TLV`.

**ELF File Structure**

For maximal firmware code reuse, the format of the DPA image loaded from driver should be the same as for the file loaded from flash. As for files loaded from the host, ELF is the default file format. This is chosen as the format for the DPA image, both for flash and for files loaded from the host.

The following figure shows, schematically, a generic ELF file structure.

To support DPA Code authentication additional information needs to be presented to firmware. This info must include:

- Cryptographic signature of the DPA code
- Customer certificate chain including a Leaf Certificate with the public key to be used for signature validation (as described in section "Public Keys (Infrastructure, Delivery, and Verification)"

```xml
<key>
  12:e7:18:c5:5c:32:44:f1:4b:61:38:3f:b7:8f:78:
  2f:a0:83:0e:3f:58:a5:00:46:7f:ad:6f:39:a6:2e:
  94:65:0b
</key>
</nv_ls_nv_public_key_2>
```
**ELF File Structure Schematic**

**Crypto Signing Flow**

The host ELF includes parts which run on the host, and those that run on DPA. DPA code files are incorporated in the "big" host ELF as binaries. Each host file may include several DPA applications.

When it is required to sign the DPA applications, the following steps need to be performed by the MFT Signing Tool (also see figure "Crypto Signing Flow"):

1. Using ELF manipulation library APIs of DPACC, extract Apps List Table
   1. Input – host ELF
   2. Output – apps list data table to include:
      1. DPA app index
      2. DPA app name
      3. Offset in host ELF
      4. Size of app
5. Name of corresponding crypto data section

For each DPA application (from i=1 to i=N, N- number of DPA apps in the host ELF) run steps 2 and 3.

2. Fill hash list table:

   ○ Input: Dpa_App_i

   ○ Output: Hash list table

3. Sign the crypto data:

   ○ Input: {Metadata, Hash List Table}, key handle (e.g., UUID from leaf of the Certificate Chain)

   ○ Output: Crypto_Data "Blob", including: Metadata, Hash List Table, Crypto Signature, Certificate Chain

4. Add crypto data section to host ELF:

   ○ Inputs: Host ELF, crypto data section name to use

   ○ Output: File name of host ELF with signature added

The structures used in the flow (hash list table, metadata, etc.) are described in sections "ELF Crypto Data Section Content" and "Hash List Table Layout".

Signing the crypto data may be done using a signing server or a locally stored key.

*Crypto Signing Flow*
**ELF Cryptographic Data Section**

This figure shows, schematically, the layout of the cryptographic data section, and the following subsections provide details about the ELF section header and the rest of the structures.

**ELF Cryptographic Data Section Layout**

- Metadata
- Hash List Table
- Crypto Signature
- Certificate Chain
Crypto Data ELF Section Header

Defined according to the ELF section header format.

**ELF Section Header**

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh_name</td>
<td>0x0</td>
<td>4B</td>
<td>&amp;(&quot;Cryptographic Data Section DPA App X&quot;) An offset to a string (in the .shstrtab section of ELF) which represents the name of this section</td>
</tr>
<tr>
<td>sh_type</td>
<td>0x4</td>
<td>4B</td>
<td>0x70000666 SHT_CRYPTODATA – the section is proprietary and holds crypto information defined in this document</td>
</tr>
<tr>
<td>sh_flags</td>
<td>0x8</td>
<td>8B</td>
<td>0 – no flags</td>
</tr>
<tr>
<td>sh_addr</td>
<td>0x10</td>
<td>8B</td>
<td>Virtual address of the section in memory, for sections that are loaded</td>
</tr>
<tr>
<td>sh_offset</td>
<td>0x18</td>
<td>8B</td>
<td>Offset of the section in the file image</td>
</tr>
<tr>
<td>sh_size</td>
<td>0x20</td>
<td>8B</td>
<td>Size in bytes of the section in the file image. Depends on the content (e.g., presence and type of public key certificate chain and signature).</td>
</tr>
<tr>
<td>sh_link</td>
<td>0x28</td>
<td>4B</td>
<td>0 – =SHN_UNDEF, no link information</td>
</tr>
<tr>
<td>sh_info</td>
<td>0x2c</td>
<td>4B</td>
<td>0 – no extra information about the section</td>
</tr>
<tr>
<td>sh_add</td>
<td>0x30</td>
<td>8B</td>
<td>Contains the required alignment of the section. This field must be a power of two.</td>
</tr>
<tr>
<td>sh_align</td>
<td>0x38</td>
<td>8B</td>
<td>0</td>
</tr>
<tr>
<td>sh_ent</td>
<td>0x40</td>
<td>8B</td>
<td>End of section header (size)</td>
</tr>
</tbody>
</table>

**ELF Crypto Data Section Content**
### ELF Crypto Data Section Fields Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadata_v</td>
<td>0x0</td>
<td>15:0</td>
<td>Version metadata structure format. Initial version is 0.</td>
</tr>
<tr>
<td>Reserved (DPA-fw_type)</td>
<td>0x4</td>
<td>15:8</td>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x8</td>
<td>31:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
<td>0xC</td>
<td>31:0</td>
<td>Reserved. Shall be set to all zeros.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x10</td>
<td>16B</td>
<td>Reserved. Shall be set to all zeros.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x20</td>
<td>4 bytes</td>
<td>Reserved. Shall be set to all zeros.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x24</td>
<td>24B</td>
<td>Reserved. Shall be set to all zeros.</td>
</tr>
<tr>
<td>signature_type</td>
<td>0x3c</td>
<td>15:0</td>
<td>Signature Type. Only relevant for signed firmware:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 0, 1 – Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 2 – RSA_SHA_512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• &gt;3 – Reserved</td>
</tr>
<tr>
<td>Hash List Table</td>
<td>0x40</td>
<td>HashTableLength</td>
<td></td>
</tr>
<tr>
<td>Crypto Signature</td>
<td>0x40 + HashTableLength</td>
<td>Signature_Length</td>
<td>Signature_Length depends on the signature_type.</td>
</tr>
<tr>
<td>Certificate Chain</td>
<td>0x40 + HashTableLength + Signature_Length</td>
<td>CrtChain _Length</td>
<td>Structure given the table under section &quot;Certificate Chain Layout&quot;.</td>
</tr>
<tr>
<td>Padding</td>
<td></td>
<td></td>
<td>FF-padding to align the full size of the data to multiples of DWords (DWs)</td>
</tr>
</tbody>
</table>

The full length of the ELF crypto data section shall be a multiple of DWs (due to firmware legacy implementation). Thus, the MFT (as part of the flow described in figure "Crypto Signing Flow") shall add FF-padding for this structure to align to multiple of DW.
Hash List Table Layout

This table specifies the hash table layout (proposal).

The table contains two parts:

- The 1\textsuperscript{st} part corresponds to the segments of the ELF file, as referenced by the Program Header Table of the ELF file.
- The 2\textsuperscript{nd} part corresponds to the sections of the ELF file, as referenced by the Section Header Table.

The hash algorithm to be used is SHA-256.

\textit{Hash List Table Layout (Proposal)}

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Entries – Segments</td>
<td>0x8</td>
<td>0x8</td>
<td>Number of entries in Hashes Segments part, N_Segments.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x8</td>
<td>0x8</td>
<td>Reserved</td>
</tr>
<tr>
<td>Number of Entries – Sections</td>
<td>0xc</td>
<td>0x8</td>
<td>Number of entries in Hashes Sections part, N_Sections. Minimum – 0</td>
</tr>
<tr>
<td>Reserved</td>
<td>0xc</td>
<td>0x8</td>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x10</td>
<td>0x10</td>
<td>16 bytes Reserved</td>
</tr>
<tr>
<td>DPA Application ELF Hash</td>
<td>0x20</td>
<td>0x20</td>
<td>32 bytes Hash of the full ELF App file</td>
</tr>
<tr>
<td>Name</td>
<td>Offset</td>
<td>Range</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>--------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ELF Header Hash</td>
<td>0x40</td>
<td>32 bytes</td>
<td>Hash of the ELF Header</td>
</tr>
<tr>
<td>Program Header Hash</td>
<td>0x60</td>
<td>32 bytes</td>
<td>Hash of the program header</td>
</tr>
<tr>
<td>Hash of 1&lt;sup&gt;st&lt;/sup&gt; Segment referenced in the Program Header Table</td>
<td>0x80</td>
<td>32 bytes</td>
<td>Hash of 1&lt;sup&gt;st&lt;/sup&gt; segment referenced in the Program Header Table</td>
</tr>
<tr>
<td>Hash of 2&lt;sup&gt;nd&lt;/sup&gt; Segment referenced in the Program Header Table</td>
<td>0xA0</td>
<td>32 bytes</td>
<td>Hash of 2&lt;sup&gt;nd&lt;/sup&gt; Segment referenced in the Program Header Table</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Hash of N_Segments (last) Segment referenced in the Program Header Table</td>
<td>0x60+N_Segments*0x20</td>
<td>32 bytes</td>
<td>Hash of N_Segments (last) Segment referenced in the Program Header Table</td>
</tr>
<tr>
<td>Section Header Table Hash</td>
<td>0x80+N_Segments*0x20</td>
<td>32 bytes</td>
<td>Hash of the Section Header Table</td>
</tr>
<tr>
<td>Hash of 1&lt;sup&gt;st&lt;/sup&gt; Section referenced in the Section Header Table</td>
<td>+ 0x20</td>
<td>32 bytes</td>
<td>Hash of 1&lt;sup&gt;st&lt;/sup&gt; section referenced in the Section Header Table</td>
</tr>
<tr>
<td>Hash of 2&lt;sup&gt;nd&lt;/sup&gt; Section referenced in the Section Header Table</td>
<td>+ 0x20</td>
<td>32 bytes</td>
<td>Hash of 2&lt;sup&gt;nd&lt;/sup&gt; section referenced in the Section Header Table</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Hash of N_Sections (last) Section referenced in the Section Header Table</td>
<td>+ 0x20</td>
<td>32 bytes</td>
<td>Hash of N_Sections (last) section referenced in the Section Header Table</td>
</tr>
</tbody>
</table>

The 32-bytes hash fields of different sections/segments in the previous table shall follow Big-Endian convention, as illustrated here:
Hash Fields (Big Endian) Bytes Alignment

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Type</td>
<td>Chain type. Shall be set to 1. 3rd party code authentication certificate chain.</td>
</tr>
<tr>
<td>0x0</td>
<td>Count</td>
<td>Number of certificates in this chain</td>
</tr>
<tr>
<td>0x0</td>
<td>Length</td>
<td>Total length of the certificate chain, in bytes, including all fields in this table</td>
</tr>
<tr>
<td>0x4</td>
<td>Reserved</td>
<td>31:0 – Reserved</td>
</tr>
<tr>
<td>0x8</td>
<td>CRC</td>
<td>The CRC of the header, for header integrity check, covering DWs in 0x0, 0x4</td>
</tr>
<tr>
<td>0xC-0x1000</td>
<td>Certificates</td>
<td>One or more ASN.1 DER-encoded X509v3 certificates. The ASN.1 DER encoding of each individual certificate can be analyzed to determine its length. The certificates shall be listed in hierarchical order, with the leaf certificate being the last on the list.</td>
</tr>
</tbody>
</table>

Certificate Chain Layout

The following table specifies the certificate chain layout. The leaf (the last certificate) of the chain is used as the public key for authentication of the DPA code. This structure is aligned with the certificate chain layout as defined in the Flash Application Note.

Certificate Chain Layout

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0x0</td>
<td>3:0</td>
<td>Chain type. Shall be set to 1. 3rd party code authentication certificate chain.</td>
</tr>
<tr>
<td>Count</td>
<td>0x0</td>
<td>7:4</td>
<td>Number of certificates in this chain</td>
</tr>
<tr>
<td>Length</td>
<td>0x0</td>
<td>23:8</td>
<td>Total length of the certificate chain, in bytes, including all fields in this table</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x4</td>
<td>31:0</td>
<td>31:0 – Reserved</td>
</tr>
<tr>
<td>CRC</td>
<td>0x8</td>
<td>15:0</td>
<td>The CRC of the header, for header integrity check, covering DWs in 0x0, 0x4</td>
</tr>
<tr>
<td>Certificates</td>
<td>0xC-0x1000</td>
<td></td>
<td>One or more ASN.1 DER-encoded X509v3 certificates. The ASN.1 DER encoding of each individual certificate can be analyzed to determine its length. The certificates shall be listed in hierarchical order, with the leaf certificate being the last on the list.</td>
</tr>
</tbody>
</table>
Known Limitations

Supported Devices
- BlueField-3 based DPUs

Supported Host OS
- Windows is not supported

Supported SDKs
- DOCA FlexIO at beta level
- DOCA DPA at beta level

Toolchain
- DPA image-signing and signature-verification are not currently supported
- Debugger (GDB) is currently not supported

FlexIO
- When `flexio_dev_outbox_config_uar_extension` API is called with a `device_id` parameter different than PF/ECPF ID (i.e., move to SF/VF outbox) and the APIs `flexio_dev_yield()`,
DOCA Libraries

DOCA DPA

Introduction

**Note**

Supported at beta level.

This chapter provides an overview and configuration instructions for DOCA DPA API.

The DOCA DPA library offers a programming model for offloading communication-centric user code to run on the DPA processor on NVIDIA® BlueField®-3 networking platform. DOCA DPA provides a high-level programming interface to the DPA processor.

DOCA DPA offers:

- Full control over DPA threads -
  - The user can control the thread function (kernel) that runs on DPA and their placement on DPA EUs
  - The user can associate a DPA thread with a DPA Completion Context. When the completion context receives a notification, the DPA thread is scheduled.

- Abstraction to allow a DPA thread to issue asynchronous operations

- Abstraction to execute a blocking one-time call from host application to execute the kernel on the DPA from the host application (RPC)

- Abstraction for memory services
• Abstraction for remote communication primitives (integrated with remote event signaling)

• Full control on execution-ordering and notifications/synchronization of the DPA and host/Target BlueField

• A set of debugging APIs that allow diagnosing and troubleshooting any issue on the device, as well as accessing real-time information from the running application

• C API for application developers

DPACC is used to compile and link kernels with DOCA DPA device libraries to get DPA applications that can be loaded from the host program to execute on the DPA (similar to CUDA usage with NVCC). For more information on DPACC, refer to the *NVIDIA DOCA DPACC Compiler*.

**Prerequisites**

DOCA DPA applications can run either on the host or on the Target BlueField. Running on the host machine requires EU pre-configuration using the `dpaeumgmt` tool. For more information, please refer to *NVIDIA DOCA DPA EU Management Tool*.

**Library Changes From Previous Releases**

**Changes in 2.8.0**

The following subsection(s) detail the `doca_dpa` library updates in version 2.8.0.

**Added Features**

- `doca_error_t doca_dpa_device_extend(struct doca_dpa *dpa, struct doca_dev *other_dev, struct doca_dpa **extended_dpa)`
  - Extended DPA with another device/GVMI

- `doca_error_t doca_dpa_get_dpa_handle(struct doca_dpa *dpa, doca_dpa_dev_t *handle)`

- `void doca_dpa_dev_device_set(doca_dpa_dev_t dpa_handle)`

**Development Flow**
DOCA enables developers to program the DPA processor using both DOCA DPA library and a suite of other tools (mainly DPACC).

The following are the main steps to start DPA offload programming:

1. Write DPA device code, or kernels, (.c files) with:
   - The __dpa_global__ keyword before DPA thread function (see "Examples" section)
   - The __dpa_rpc__ keyword before RPC function (see "Examples" section)

2. Use DPACC to build a DPA program (i.e., a host library which contains an embedded device executable). Inputs for DPACC are:
   - Kernels from the previous step
   - DOCA DPA device libraries

3. Build host executable using a host compiler. Inputs for the host compiler are:
   - DPA program from the previous step
   - User host application source files
   - DOCA DPA host library

DPACC is provided by the DOCA SDK installation. For more information, please refer to the NVIDIA DOCA DPACC Compiler.

Software Architecture

Deployment View

DOCA DPA is composed of the following libraries that come with the DOCA SDK installation:

- Host/Target BlueField library and header file (used by user host application)
- `doca_dpa.h`
- `libdoca_dpa.a/libdoca_dpa.so`

- Two device libraries and header files
  - `doca_dpa_dev.h`
  - `doca_dpa_dev_rdma.h`
  - `doca_dpa_dev_sync_event.h`
  - `doca_dpa_dev_buf.h`
  - `libdoca_dpa_dev.a` – DOCA DPA device library for common utilities (e.g., log, trace, completion, sync event, etc.)
  - `libdoca_dpa_dev_comm.a` – DOCA DPA device library for communication utilities (e.g., RDMA)

**DPA Queries**

- Before invoking the DPA API, make sure that DPA is indeed supported on the relevant device. The API which checks whether a device supports DPA is:

  ```c
  doca_error_t doca_devinfo_get_is_dpa_supported(const struct doca_devinfo *devinfo)
  ```

  **Info**

  Only if this call returns `DOCA_SUCCESS` can the user invoke DOCA DPA API on the device.

- To use a valid EU ID for the DPA EU Affinity of a DPA thread, use the following APIs to query EU ID and core valid values:
There is a limitation on the maximum number of DPA threads that can run a single kernel. This can be retrieved by calling the host API:

- `doca_error_t doca_dpa_get_max_threads_per_kernel(struct doca_dpa *dpa, unsigned int *value)`

Each kernel launched into the DPA has a maximum runtime limit. This can be retrieved by calling the host API:

- `doca_error_t doca_dpa_get_kernel_max_run_time(struct doca_dpa *dpa, unsigned long long *value)`

**Note**

If the kernel execution time on the DPA exceeds this maximum runtime limit, it may be terminated and cause a fatal error. To recover, the application must destroy the DPA context and create a new one.

---

**Overview of DOCA DPA Software Objects**

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPA context</td>
<td>Software construct for the host process that encapsulates the state associated with a DPA process (on a specific device).</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DPA Application</td>
<td>Interface with the DPACC compiler to produce a DPA program (app) which is obtained by the DPA context to begin working on DPA.</td>
</tr>
<tr>
<td>Kernel</td>
<td>User function (and its arguments) to be executed on DPA. A kernel may be executed by one or more DPA threads.</td>
</tr>
<tr>
<td>DPA EU Affinity</td>
<td>An object used to control which EU to use for DPA thread.</td>
</tr>
<tr>
<td>DPA Thread</td>
<td>DOCA DPA provides APIs to create/manage DPA thread which runs a given kernel.</td>
</tr>
<tr>
<td>DPA Completion Context</td>
<td>An object used to receive/handle a completion notification. The user can associate a DPA thread with a completion context. When the completion context receives a notification, DPA thread is scheduled.</td>
</tr>
<tr>
<td>DPA Thread Notification</td>
<td>A mechanism for one DPA thread to notify another DPA thread.</td>
</tr>
<tr>
<td>DPA Async Ops</td>
<td>An object used to allow a DPA thread to issue asynchronous operations, like memcpy or post_wait operations.</td>
</tr>
<tr>
<td>DPA RPC</td>
<td>A blocking one-time call from host application to execute a kernel on DPA. RPC is mainly used for control path. The RPC's return value is reported back to the host application.</td>
</tr>
<tr>
<td>DPA Memory</td>
<td>DOCA DPA provides an API to allocate/manage DPA memory, as well as handling host/Target BlueField memory that has been exported to DPA.</td>
</tr>
<tr>
<td>Sync Event</td>
<td>Data structure in either CPU, Target BlueField, GPU, or DPA-heap. An event contains a counter that can be updated and waited on.</td>
</tr>
<tr>
<td>RDMA</td>
<td>Abstraction around a network transport object. Allows executing various RDMA operations.</td>
</tr>
<tr>
<td>DPA Hash Table</td>
<td>DOCA DPA provides an API to create a Hash Table on DPA. This data structure is managed on DPA using relevant device APIs.</td>
</tr>
<tr>
<td>DPA Logger/Tracer</td>
<td>DOCA DPA provides a set of debugging APIs to allow the user to diagnose and troubleshoot any issue on the device, as well as accessing real-time information from the running application.</td>
</tr>
</tbody>
</table>
Initialization

The DPA context encapsulates the DPA device and a DPA process (program). Within this context, the application creates various DPA SDK objects and controls them. After verifying DPA is supported for the chosen device, the DPA context is created.

Use the following host-side APIs to create/set the DPA context and it is expected to be the first programming step:

- To create/destroy DPA context:
  
  ```c
  doca_error_t doca_dpa_create(struct doca_dev *dev, struct doca_dpa **dpa)
  doca_error_t doca_dpa_destroy(struct doca_dpa *dpa)
  ```

- To start/stop DPA context:
  
  ```c
  doca_error_t doca_dpa_start(struct doca_dpa *dpa)
  doca_error_t doca_dpa_stop(struct doca_dpa *dpa)
  ```

Note

The DOCA DPA SDK does not use any means of multi-thread synchronization primitives. All DOCA DPA objects are non-thread-safe. Developers should make sure the user program and kernels are written to avoid race conditions.

Interface to DPACC

DPA Application
To associate a DPA program (app) with a DPA context, use the following host-side APIs:

```c
#define DPA_APP_NAME "dpa_example_app"

extern struct doca_dpa_app *dpa_example_app;

doca_dpa_create(&dpa);
doca_dpa_set_app(dpa, dpa_example_app);
doca_dpa_start(dpa);
```

The app variable name used in `doca_dpa_set_app()` API must be the token passed to DPACC -- `app-name` parameter.

**Example (Pseudo Code)**

For example, when using the following `dpacc` command line:

```bash
dpacc 
   -o dpa_program.a 
   -hostcc=gcc 
   -hostcc-options="..." 
   --devicecc-options="..." 
   -device-libs="-L/opt/mellanox/doca/include -Ldoca_dpa_dev -Ldoca_dpa_dev_comm" 
   --app-name="dpa_example_app"
```

The user must use the following commands to set the `app` of a DPA context:

```c
#define DPA_APP_NAME "dpa_example_app"

extern struct doca_dpa_app *dpa_example_app;

doca_dpa_create(&dpa);
doca_dpa_set_app(dpa, dpa_example_app);
doca_dpa_start(dpa);
```
The user can control which EU to use for a DPA thread using DPA EU affinity object.

A DPA EU affinity object can be configured for one EU ID at a time.

Use the following host-side APIs to manage it:

- To create/destroy DPA EU affinity object:

  ```
  doca_error_t doca_dpa_eu_affinity_create(struct doca_dpa *dpa, struct doca_dpa_eu_affinity **affinity)
  doca_error_t doca_dpa_eu_affinity_destroy(struct doca_dpa_eu_affinity *affinity)
  ```

- To set/clear EU ID in DPA EU affinity object:

  ```
  doca_error_t doca_dpa_eu_affinity_set(struct doca_dpa_eu_affinity *affinity, unsigned int eu_id)
  doca_error_t doca_dpa_eu_affinity_clear(struct doca_dpa_eu_affinity *affinity)
  ```

- To get EU ID of a DPA EU affinity object:

  ```
  doca_error_t doca_dpa_eu_affinity_get(struct doca_dpa_eu_affinity *affinity, unsigned int *eu_id)
  ```

**Threading**

DOCA DPA thread used to run a user function “DPA kernel” on DPA.

User can control on which EU to run DPA kernel by attaching a DPA EU affinity object to the thread.

The thread can be triggered on DPA using two methods:

1. DPA Thread Notification - Notifying one DPA thread from another DPA thread.

2. DPA Completion Context - A completion is arrived at a DPA completion context which is attached to the thread.
DPA Thread

Host-side API

- To create/destroy DPA thread:

  ```c
  doca_error_t doca_dpa_thread_create(struct doca_dpa *dpa, struct doca_dpa_thread **dpa_thread)
  doca_error_t doca_dpa_thread_destroy(struct doca_dpa_thread *dpa_thread)
  ```

- To set/get thread user function and it's argument:

  ```c
  doca_error_t doca_dpa_thread_set_func_arg(struct doca_dpa_thread *thread, doca_dpa_func_t *func, uint64_t arg)
  doca_error_t doca_dpa_thread_get_func_arg(struct doca_dpa_thread *dpa_thread, doca_dpa_func_t **func, uint64_t *arg)
  ```

- To set/get DPA EU Affinity:

  ```c
  doca_error_t doca_dpa_thread_set_affinity(struct doca_dpa_thread *thread, struct doca_dpa_eu_thread_affinity *eu_affinity)
  doca_error_t doca_dpa_thread_get_affinity(struct doca_dpa_thread *dpa_thread, const struct doca_dpa_eu_affinity **affinity)
  ```

- Thread Local Storage (TLS)

  User can ask to store an opaque for a DPA thread in host side application using the following API:

  ```c
  doca_error_t doca_dpa_thread_set_local_storage(struct doca_dpa_thread *dpa_thread, doca_dpa_dev_ulongptr_t dev_ptr)
  ```
dev_ptr is a pre-allocated DPA memory.

In kernel, user can retrieve the stored opaque using the relevant device API (see below API).

This opaque is stored/retrieved using the Thread Local Storage (TLS) mechanism.

- To start/stop DPA thread:

  ```c
  doca_error_t doca_dpa_thread_get_local_storage(struct doca_dpa_thread *dpa_thread, doca_dpa_devuintptr_t *dev_ptr)
  ```

  ```c
  doca_error_t doca_dpa_thread_start(struct doca_dpa_thread *thread)
  ```

  ```c
  doca_error_t doca_dpa_thread_stop(struct doca_dpa_thread *dpa_thread)
  ```

- To run DPA thread:

  ```c
  doca_error_t doca_dpa_thread_run(struct doca_dpa_thread *dpa_thread)
  ```

This API sets the thread to run state.

This function must be called after DPA thread is:

1. Created, set and started.

2. In case of DPA thread is attached to DPA Completion Context, the completion context must be started before, see below pseudo code example:

**Device-side API**

Device APIs are used by user-written kernels.

- Thread Restart APIs
DPA thread can ends its run using one of the following device APIs:

- **Reschedule API:**
  
  ```c
  void doca_dpa_dev_thread_reschedule(void)
  ```
  
  - DPA thread still active.
  - DPA thread resources are back to RTOS.
  - DPA thread can be triggered again.

- **Finish API:**
  
  ```c
  void doca_dpa_dev_thread_finish(void)
  ```
  
  - DPA thread is marked as finished.
  - DPA thread resources are back to RTOS.
  - DPA thread can’t be triggered again.

- **To get TLS:**
  
  ```c
doca_dpa_dev_uintptr_t doca_dpa_dev_thread_get_local_storage(void)
  ```
  
  This function returns DPA thread local storage which was set previously using host API `doca_dpa_thread_set_local_storage()`.

**Example (Host-side Pseudo Code)**

```c
extern doca_dpa_func_t hello_kernel;
```
Completion Context

To tie the user application closely with the DPA native model of event-driven scheduling/computation, we introduced DPA Completion Context.

User associates a DPA Thread with a completion context. When the completion context receives a notification, DPA Thread is triggered.

User can choose not to associate it with DPA Thread and to poll it manually.

User has the option to continue receiving new notifications or ignore them.

DOCA DPA provides a generic completion context that can be shared for Message Queues, RDMA, Ethernet and as well as DPA Async Ops.

```c
// create DPA thread
doca_dpa_thread_create(&dpa_thread);

// set thread kernel
doca_dpa_thread_set_func_arg(dpa_thread, &hello_kernel, func_arg);

// set thread affinity
doca_dpa_eu_affinity_create(&eu_affinity);
doca_dpa_eu_affinity_set(eu_affinity, 10 /* EU ID */);
doca_dpa_thread_set_affinity(dpa_thread, eu_affinity);

// set thread local storage
doca_dpa_mem_alloc(&tls_dev_ptr);
doca_dpa_thread_set_local_storage(dpa_thread, tls_dev_ptr);

// start thread
doca_dpa_thread_start(dpa_thread);

// create and initialize DPA Completion Context
doca_dpa_completion_create(&dpa_comp);
doca_dpa_completion_set_thread(dpa_comp, dpa_thread);
doca_dpa_completion_start(dpa_comp);

// run thread only after both thread is started and the attached completion context is started
doca_dpa_thread_run(dpa_thread);
```
Host-side API

- To create/destroy DPA Completion Context:

```c
#define DOCADPAERROR _doca_error_t

_docadpaerror_t doca_dpa_completion_create(struct doca_dpa *dpa, unsigned int queue_size, struct
doca_dpa_completion **dpa_comp)

_docadpaerror_t doca_dpa_completion_destroy(struct doca_dpa_completion *dpa_comp)
```

- To get queue size:

```c
#define DOCADPAERROR _doca_error_t

_docadpaerror_t doca_dpa_completion_get_queue_size(struct doca_dpa_completion *dpa_comp,
unsigned int *size)
```

- To attach to a DPA Thread:

```c
#define DOCADPAERROR _doca_error_t

_docadpaerror_t doca_dpa_completion_set_thread(struct doca_dpa_completion *dpa_comp, struct
doca_dpa_thread *thread)

_docadpaerror_t doca_dpa_completion_get_thread(struct doca_dpa_completion *dpa_comp, struct
doca_dpa_thread **thread)
```

Attaching to a thread is only required if the user wants triggering of the thread when a completion is arrived at the completion context.

- To start/stop DPA Completion Context:

```c
#define DOCADPAERROR _doca_error_t

_docadpaerror_t doca_dpa_completion_start(struct doca_dpa_completion *dpa_comp)

_docadpaerror_t doca_dpa_completion_stop(struct doca_dpa_completion *dpa_comp)
```

- To get DPA handle:

```c
#define DOCADPAERROR _doca_error_t

```
Use output parameter handle for below device APIs which can be used in thread kernel.

**Device-side API**

Device APIs are used by user-written kernels.

Kernels get doca_dpa_dev_completion_t handle and invoke the following API:

- To get a completion element:

  ```
  int doca_dpa_dev_get_completion(doca_dpa_dev_completion_t dpa_comp_handle,
  doca_dpa_dev_completion_element_t *comp_element)
  ```

  Use the returned comp_element to retrieve completion info using below APIs.

- To get completion element type:

  ```
  typedef enum {
    DOCA_DPA_DEV_COMP_SEND = 0x0, /**< Send completion */
    DOCA_DPA_DEV_COMP_RECV_RDMA_WRITE_IMM = 0x1, /**< Receive RDMA Write with Immediate completion */
    DOCA_DPA_DEV_COMP_RECV_SEND = 0x2, /**< Receive Send completion */
    DOCA_DPA_DEV_COMP_RECV_SEND_IMM = 0x3, /**< Receive Send with Immediate completion */
    DOCA_DPA_DEV_COMP_RECV_ERR = 0xD, /**< Receive Error completion */
    DOCA_DPA_DEV_COMP_RECV_ERR_IMM = 0xE /**< Receive Error completion */
  } doca_dpa_dev_completion_type_t;
  
  doca_dpa_dev_completion_type_t
  doca_dpa_dev_get_completion_type(doca_dpa_dev_completion_element_t comp_element)
  ```
To get completion element user data:

```c
uint32_t doca_dpa_dev_get_completion_user_data(doca_dpa_dev_completion_element_t comp_element)
```

This API returns user data which was set previously in either host APIs:

1. `doca_dpa_async_ops_create(..., user_data, ...)`
   When DPA Completion Context is attached to DPA Async Ops.

2. `doca_ctx_set_user_data(..., user_data)`
   When DPA Completion Context is attached to DOCA context, such as DOCA RDMA context.

To get completion element immediate data:

```c
uint32_t doca_dpa_dev_get_completion_immediate(doca_dpa_dev_completion_element_t comp_element)
```

This API returns immediate data for a completion element of type:

1. `DOCA_DPA_DEV_COMP_RECV_RDMA_WRITE_IMM`
2. `DOCA_DPA_DEV_COMP_RECV_SEND_IMM`

Acknowledge that the completions have been read on DPA Completion Context:

```c
void doca_dpa_dev_completion_ack(doca_dpa_dev_completion_t dpa_comp_handle, uint64_t num_comp)
```

This API releases resources of the acked completion elements in completion context.

This acknowledgment enables receiving new `num_comp` completions.
To request notification on DPA Completion Context:

```c
void doca_dpa_dev_completion_request_notification(doca_dpa_dev_completion_t dpa_comp_handle)
```

This API enables requesting new notifications on DPA Completion Context.

Without calling this function, DPA Completion Context is not being notified on new arrived completion elements, hence new completions are not populated in DPA Completion Context.

**Example (Device-side Pseudo Code)**

```c
__dpa_global__ void hello_kernel(uint64_t arg)
{
    // User is expected to pass in some way the attached completion context handle "dpa_comp_handle" to kernel such as func_arg or a shared memory.
    DOCA_DPA_DEV_LOG_INFO("Hello from kernel\n");

    doca_dpa_dev_completion_element_t comp_element;
    found = doca_dpa_dev_get_completion(dpa_comp_handle, &comp_element);
    if (found) {
        comp_type = doca_dpa_dev_get_completion_type(comp_element);
        // process the completion according to completion type...
        // ack on 1 completion
        doca_dpa_dev_completion_ack(dpa_comp_handle, 1);

        // enable getting more completions and triggering the thread
        doca_dpa_dev_completion_request_notification(dpa_comp_handle);
    }

    // reschedule thread
    doca_dpa_dev_thread_reschedule();
}
```
Thread Notification

Thread Activation is a mechanism for one DPA thread to trigger another DPA Thread.

Thread activation is done without receiving a completion on the attached thread. Therefore it is expected that user of this method of thread activation passes the message in another fashion – such as shared memory.

Thread Activation can be achieved using DPA Notification Completion object.

Host-side API

- To create/destroy DPA Notification Completion:

  ```c
  doca_error_t doca_dpa_notification_completion_create(struct doca_dpa *dpa, struct doca_dpa_thread *dpa_thread, struct doca_dpa_notification_completion **notify_comp)
  
  doca_error_t doca_dpa_notification_completion_destroy(struct doca_dpa_notification_completion *notify_comp)
  ```

  Attaching DPA Notification Completion to a DPA Thread is done using the given parameter `dpa_thread`.

- To get attached DPA Thread:

  ```c
  doca_error_t doca_dpa_notification_completion_get_thread(struct doca_dpa_notification_completion *notify_comp, struct doca_dpa_thread **dpa_thread)
  ```

- To start/stop DPA Notification Completion:

  ```c
  doca_error_t doca_dpa_notification_completion_start(struct doca_dpa_notification_completion *notify_comp)
  
  doca_error_t doca_dpa_notification_completion_stop(struct doca_dpa_notification_completion *notify_comp)
  ```
To get DPA handle:

```
doca_error_t doca_dpa_notify_completion_get_dpa_handle(struct
doca_dpa_notification_completion *notify_comp, doca_dpa_dev_notification_completion_t
*comp_handle)
```

Use output parameter `comp_handle` for below device API which can be used in thread kernel.

**Device-side API**

Device API is used by user-written kernels.

Kernels get `doca_dpa_dev_notification_completion_t handle` and invoke the following API:

```
void doca_dpa_dev_thread_notify(doca_dpa_dev_notification_completion_t comp_handle)
```

Calling this API triggers the attached DPA Thread (the one that is specified in `dpa_thread` parameter in host-side API `doca_dpa_notification_completion_create()`).

**Example (Pseudo Code)**

- **Host-side**

```
extern doca_dpa_func_t hello_kernel;

// create DPA thread
doca_dpa_thread_create(&dpa_thread);

// set thread kernel
doca_dpa_thread_set_func_arg(dpa_thread, &hello_kernel, func_arg);

// start thread
```
## Asynchronous Ops

DPA Async Ops allows DPA Thread to issue asynchronous operations, like memcpy or post_wait.

This feature requires the user to create an “asynchronous ops” context and attach to a completion context.

User is expected to adhere to `queue_size` limit on the device when posting operations.

The completion context can raise activation if it is attached to a DPA Thread.

User can also choose to progress the completion context via polling it manually.

User can provide DPA Async Ops `user_data`, and retrieve this metadata in device using relevant device API.

```c
  doca_dpa_thread_start(dpa_thread);

  // create and start DPA notification completion
  doca_dpa_notification_completion_create(dpa, dpa_thread, &notify_comp);

  doca_dpa_notification_completion_start(notify_comp);

  // get its DPA handle
  doca_dpa_notification_completion_get_dpa_handle(notify_comp, &notify_comp_handle);

  // run thread only after both thread is started and attached notification completion is started
  doca_dpa_thread_run(dpa_thread);

  doca_dpa_dev_thread_notify(notify_comp_handle);
```

- **Device-side**

  Whenever some DPA Thread calls:

  ```c
  doca_dpa_dev_thread_notify(notify_comp_handle);
  ```

  This call triggers `dpa_thread`. 
Host-side API

- To create/destroy DPA Async Ops:

```c
#include <doa.h>

doca_error_t doca_dpa_async_ops_create(struct doca_dpa *dpa, unsigned int queue_size,
  uint64_t user_data, struct doca_dpa_async_ops **async_ops)

doca_error_t doca_dpa_async_ops_destroy(struct doca_dpa_async_ops *async_ops)
```

Please use the following define for valid user_data values:

```c
#define DOCA_DPA_COMPLETION_LOG_MAX_USER_DATA (24)
```

- To get queue size/user_data:

```c
#include <doa.h>

doca_error_t doca_dpa_async_ops_get_queue_size(struct doca_dpa_async_ops *async_ops,
  unsigned int *queue_size)

doca_error_t doca_dpa_async_ops_get_user_data(struct doca_dpa_async_ops *async_ops,
  uint64_t *user_data)
```

- To attach to a DPA Completion Context:

```c
#include <doa.h>

doca_error_t doca_dpa_async_ops_attach(struct doca_dpa_async_ops *async_ops, struct
doca_dpa_completion *dpa_comp)
```

- To start/stop DPA Async Ops:

```c
#include <doa.h>

doca_error_t doca_dpa_async_ops_start(struct doca_dpa_async_ops *async_ops)

doca_error_t doca_dpa_async_ops_stop(struct doca_dpa_async_ops *async_ops)
```
• To get DPA handle:

```c
#include <doa.h>

doca_error_t doca_dpa_async_ops_get_dpa_handle(struct doca_dpa_async_ops *async_ops,
               doca_dpa_dev_async_ops_t *handle)
```

Use output parameter `handle` for below device API which can be used in thread kernel.

**Device-side API**

Device APIs are used by user-written kernels.

Kernels get `doca_dpa_dev_async_ops_t handle` and invoke the following API:

• To post memcpy operation using `doca_buf`:

```c
#include <doa.h>

void doca_dpa_dev_post Buf_memcpy(doca_dpa_dev_async_ops_t async_ops_handle,
               doca_dpa_dev_buf_t dst_buf_handle, doca_dpa_dev_buf_t src_buf_handle, bool
               completion_requested)
```

This API copies data between two DOCA buffers.

The destination buffer, specified by `dst_buf_handle` will contain the copied data after memory copy is complete.

This is a non-blocking routine.

Use `completion_requested` to raise a completion when copy data operation is done (any value greater than 0).

If `completion_requested` was set and the attached DPA Completion Context is attached to a DPA Thread, then the thread is triggered once the memcpy operation is done.

• To post memcpy operation using `doca_mmap` and an explicit addresses:
This API copies data between two DOCA Mmaps.

The destination DOCA Mmap, specified by `dst_mmap_handle`, `dst_addr` will contain the copied data in source DOCA Mmap specified by `src_mmap_handle`, `src_addr` and `length` after memory copy is complete.

This is a non-blocking routine.

Use `completion_requested` to raise a completion when copy data operation is done (any value greater than 0).

If `completion_requested` was set and the attached DPA Completion Context is attached to a DPA thread, then the thread is triggered once the memcpy operation is done.

**Info**

Use this API for memcpy instead of using `doca_buf memcpy API` to gain better performance.

- To post wait greater operation on a DOCA Sync Event:

  ```c
  void doca_dpa_dev_sync_event_post_wait_gt(doca_dpa_dev_async_ops_t async_ops_handle, 
                                      doca_dpa_dev_sync_event_t wait_se_handle, 
                                      uint64_t value)
  ```

  This function posts a wait operation on the DOCA Sync Event using DPA Async Ops to obtain a DPA Thread activation.
Attached thread is activated when value of DOCA Sync Event is greater than a given value.

This is a non-blocking routine.

**Note**

Valid values must be in the range [0, 254] and can be called for event with value in the range [0, 254]. Invalid values leads to undefined behavior.

- To post wait not equal operation on a DOCA Sync Event:

  ```c
  void doca_dpa_dev_sync_event_post_wait_ne(doca_dpa_dev_async_ops_t async_ops_handle, doca_dpa_dev_sync_event_t wait_se_handle, uint64_t value)
  ```

  This function posts a wait operation on the DOCA Sync Event using the DPA Async Ops to obtain a DPA Thread activation.

  Attached thread is activated when value of DOCA Sync Event is not equal to a given value.

  This is a non-blocking routine.

**Example (Host-side Pseudo Code)**

```c
  doca_dpa_thread_create(&dpa_thread);
  doca_dpa_thread_set_func_arg(dpa_thread);
  doca_dpa_thread_start(dpa_thread);

  doca_dpa_completion_create(&dpa_comp);
  doca_dpa_completion_set_thread(dpa_comp, dpa_thread);
  doca_dpa_completion_start(dpa_comp);
```
Thread Group

Thread group is used to aggregate individual DPA threads to a single group.

Please see below host-side APIs for creating/managing thread group.

- To create/destroy DPA Thread Group:

  ```
  doca_error_t doca_dpa_thread_group_create(struct doca_dpa *dpa, unsigned int num_threads, struct doca_dpa_tg **tg)
  ```

  ```
  doca_error_t doca_dpa_thread_group_destroy(struct doca_dpa_tg *tg)
  ```

- To get number of threads:

  ```
  doca_error_t doca_dpa_thread_group_get_num_threads(struct doca_dpa_tg *tg, unsigned int *num_threads);
  ```

- To set DPA Thread at 'rank' in DPA Thread Group:

  ```
  doca_error_t doca_dpa_thread_group_set_thread(struct doca_dpa_tg *tg, struct doca_dpa_thread *thread, unsigned int rank)
  ```
Thread rank is an index of the thread (between 0 and (num_threads - 1)) within the group.

- To start/stop DPA Thread Group:

```c
  doca_error_t doca_dpa_thread_group_start(struct doca_dpa_tg *tg)
  doca_error_t doca_dpa_thread_group_stop(struct doca_dpa_tg *tg)
```

**Memory Subsystem**

The user can allocate (from the host API) and access (from both the host and device API) several memory locations using the relevant DOCA DPA API.

DOCA DPA supports access from the host/Target BlueField to DPA heap memory and also enables device access to host memory (e.g., kernel writes to host heap).

The normal memory usage flow would be to:

1. Allocate memory (Host/Target BlueField/DPA).
2. Register the memory.
3. Get a DPA handle for the registered memory so it can be accessed by DPA kernels.
4. Access/use the memory from the kernel (see relevant device-side APIs).

**Host-side API**

- To allocate DPA heap memory:

```c
  doca_dpa_mem_alloc(doca_dpa_t dpa, size_t size, doca_dpa_dev_uintptr_t *dev_ptr)
```

- To free previously allocated DPA memory:
To copy previously allocated memory from a host pointer to a DPA heap device pointer:

```c
doca_dpa_h2d_memcpym(doca_dpa_t dpa, doca_dpa_dev_uintptr_t src_ptr, void *dst_ptr, size_t size)
```

To copy previously allocated memory from a DOCA Buffer to a DPA heap device pointer:

```c
doca_error_t doca_dpa_h2d_buf_memcpym(struct doca_dpa *dpa, doca_dpa_dev_uintptr_t dst_ptr, struct doca_buf *buf, size_t size)
```

To copy previously allocated memory from a DPA heap device pointer to a host pointer:

```c
doca_dpa_d2h_memcpym(doca_dpa_t dpa, void *dst_ptr, doca_dpa_dev_uintptr_t src_ptr, size_t size)
```

To copy previously allocated memory from a DPA heap device pointer to a DOCA Buffer:

```c
doca_error_t doca_dpa_d2h_buf_memcpym(struct doca_dpa *dpa, struct doca_buf *buf, doca_dpa_dev_uintptr_t src_ptr, size_t size)
```

To set memory:

```c
doca_dpa_memset(doca_dpa_t dpa, doca_dpa_dev_uintptr_t dev_ptr, int value, size_t size)
```
To get a DPA handle to use in kernels, the user must use a DOCA Core Memory Inventory Object in the following manner (refer to "DOCA Memory Subsystem"):

- When the user wants to use device APIs with DOCA Buffer, use the following pseudo code:

```c
doca_buf_arr_create(&buf_arr);
doca_buf_arr_set_target_dpa(buf_arr, doca_dpa);
doca_buf_arr_start(buf_arr);
doca_buf_arr_get_dpa_handle(buf_arr, &handle);
```

Use output parameter `handle` in relevant device APIs in thread kernel.

- When the user wants to use device APIs with DOCA Mmap, use the following pseudo code:

```c
doca_mmap_create(&mmap);
doca_mmap_set_dpa_memrange(mmap, doca_dpa, dev_ptr, dev_ptr_len); // dev_ptr is a pre-allocated DPA memory
doca_mmap_start(mmap);
doca_mmap_dev_get_dpa_handle(mmap, doca_dev, &handle);
```

Use output parameter `handle` in relevant device APIs in thread kernel.

**Device-side API**

Device APIs are used by user-written kernels.

Memory APIs supplied by the DOCA DPA SDK are all asynchronous (i.e., non-blocking).

The user can acquire either:

1. Pre-configured DOCA Buffers (previously configured with `doca_buf_arr_set_params`).
2. Non-configured DOCA Buffers and use below device setters to configure them.
Device-side API operations:

- To obtain a single buffer handle from the buf array handle:

  ```c
  doca_dpa_dev_buf_t doca_dpa_dev_buf_array_get_buf(doca_dpa_dev_buf_arr_t buf_arr, const uint64_t buf_idx)
  ```

- To set/get the address pointed to by the buffer handle:

  ```c
  void doca_dpa_dev_buf_set_addr(doca_dpa_dev_buf_t buf, uintptr_t addr)
  uintptr_t doca_dpa_dev_buf_get_addr(doca_dpa_dev_buf_t buf)
  ```

- To set/get the length of the buffer:

  ```c
  void doca_dpa_dev_buf_set_len(doca_dpa_dev_buf_t buf, size_t len)
  uint64_t doca_dpa_dev_buf_get_len(doca_dpa_dev_buf_t buf)
  ```

- To set/get the DOCA Mmap associated with the buffer:

  ```c
  void doca_dpa_dev_buf_set_mmap(doca_dpa_dev_buf_t buf, doca_dpa_dev_mmap_t mmap)
  doca_dpa_dev_mmap_t doca_dpa_dev_buf_get_mmap(doca_dpa_dev_buf_t buf)
  ```

- To get a pointer to external memory registered on the host using DOCA Buffer:

  ```c
  doca_dpa_dev_uintptr_t doca_dpa_dev_buf_get_external_ptr(doca_dpa_dev_buf_t buf)
  ```

- To get a pointer to external memory registered on the host using an explicit address and DOCA Mmap:

  ```c
  doca_dpa_dev_uintptr_t doca_dpa_dev_mmap_get_external_ptr(doca_dpa_dev_mmap_t
  ```
Sync Events

Sync events fulfill the following roles:

- DOCA DPA execution model is asynchronous and sync events are used to control various threads running in the system (allowing order and dependency)

- DOCA DPA supports remote sync events, so the programmer is capable of invoking remote nodes by means of DOCA sync events

Host-side API

Please refer to "DOCA Sync Event".

Device-side API

- To get the current event value:

  ```c
  doca_dpa_dev_sync_event_get(doca_dpa_dev_sync_event_t event, uint64_t *value)
  ```

- To add/set to the current event value:

  ```c
  doca_dpa_dev_sync_event_update_<add|set>(doca_dpa_dev_sync_event_t event, uint64_t value)
  ```

- To wait until event is greater than threshold:

  ```c
  doca_dpa_dev_sync_event_wait_gt(doca_dpa_dev_sync_event_t event, uint64_t value, uint64_t
  ```
Use mask to apply (bitwise AND) on the DOCA sync event value for comparison with the wait threshold.

**Communication Model**

DOCA DPA communication primitives allow sending data from one node to another.

The object used for the communication between nodes is called an RDMA DPA handle. RDMA DPA handles can be used by kernels only.

RDMAs represent a unidirectional communication pipe between two nodes.

RDMA DPA handles are created when setting a DOCA RDMA context to DPA data path. For more information, please refer to [DOCA RDMA](#).

To track the completion of all communications, the user can attach DOCA RDMA context to a DPA Completion Context.

DPA Completion Context can be associated with a DPA Thread. When the completion context receives a completion on a communication operation, DPA Thread is triggered.

**Info**

The user can choose not to associate it with a DPA Thread and to poll it manually.

**Host-side API**

- To create DOCA RDMA context on DPA, the user must use the following API for the DOCA RDMA context:
To attach a DOCA RDMA context to a DPA Completion Context:

```c
int doca_ctx_set_datapath_on_dpa(struct doca_ctx *ctx, struct doca_dpa *dpa)
```

To obtain a DPA RDMA handle:

```c
int doca_rdma_dpa_completion_attach(struct doca_rdma *rdma, struct doca_dpa_completion *dpa_comp)
```

To read to a local buffer from the remote side buffer:

```c
int doca_rdma_get_dpa_handle(struct doca_rdma *rdma, doca_dpa_dev_rdma_t *dpa_rdma)
```

Use output parameter handle in relevant device APIs in the thread kernel.

**Note**

DPA RDMAs are not thread safe and, therefore, must not be used from different kernels/threads concurrently.

**Device-side API**

DOCA DPA offers two work models for each device RDMA operation:

- An API for RDMA operation using DOCA Buffer
- An API for RDMA operation using DOCA Mmap and an explicit memory address

The user may choose to also raise a completion when the operation is done.

- To read to a local buffer from the remote side buffer:
To write local memory to the remote side buffer:

```c
void doca_dpa_dev_rdma_post_read(doca_dpa_dev_rdma_t rdma,
                                  doca_dpa_dev_mmap_t dst_mmap_handle,
                                  uint64_t dst_addr,
                                  doca_dpa_dev_mmap_t src_mmap_handle,
                                  uint64_t src_addr,
                                  size_t length,
                                  uint32_t completion_requested)

void doca_dpa_dev_rdma_post_buf_read(doca_dpa_dev_rdma_t rdma,
                                      doca_dpa_dev_buf_t dst_buf_handle,
                                      doca_dpa_dev_buf_t src_buf_handle,
                                      uint32_t completion_requested)
```

To write local memory to the remote side buffer with an immediate data which can be retrieved when receiving a completion on this operation:

```c
void doca_dpa_dev_rdma_post_write(doca_dpa_dev_rdma_t rdma,
                                   doca_dpa_dev_mmap_t dst_mmap_handle,
                                   uint64_t dst_addr,
                                   doca_dpa_dev_mmap_t src_mmap_handle,
                                   uint64_t src_addr,
                                   size_t length,
                                   uint32_t completion_requested)

void doca_dpa_dev_rdma_post_buf_write(doca_dpa_dev_rdma_t rdma,
                                      doca_dpa_dev_buf_t dst_buf_handle,
                                      doca_dpa_dev_buf_t src_buf_handle,
                                      uint32_t completion_requested)

void doca_dpa_dev_rdma_post_write_imm(doca_dpa_dev_rdma_t rdma,
                                      doca_dpa_dev_mmap_t dst_mmap_handle,
                                      uint64_t dst_addr,
                                      doca_dpa_dev_mmap_t src_mmap_handle,
                                      uint64_t src_addr,
                                      size_t length,
                                      uint32_t immediate,
                                      uint32_t completion_requested)

void doca_dpa_dev_rdma_post_buf_write_imm(doca_dpa_dev_rdma_t rdma,
• To send local memory:

```c
void doca_dpa_dev_rdma_post_send(doca_dpa_dev_rdma_t rdma,
       doca_dpa_dev_mmap_t mmap_handle,
       uint64_t addr,
       size_t length,
       uint32_t completion_requested)

void doca_dpa_dev_rdma_post_buf_send(doca_dpa_dev_rdma_t rdma,
       doca_dpa_dev_buf_t send_buf_handle,
       uint32_t completion_requested)
```

• To send local memory with an immediate data which can be retrieved when receiving a completion on this operation:

```c
void doca_dpa_dev_rdma_post_send_imm(doca_dpa_dev_rdma_t rdma,
       doca_dpa_dev_mmap_t mmap_handle,
       uint64_t addr,
       size_t length,
       uint32_t immediate,
       uint32_t completion_requested)

void doca_dpa_dev_rdma_post_buf_send_imm(doca_dpa_dev_rdma_t rdma,
       doca_dpa_dev_buf_t send_buf_handle,
       uint32_t immediate,
       uint32_t completion_requested)

// use the following API to retrieve immediate data on completion
uint32_t doca_dpa_dev_get_completion_immediate(doca_dpa_dev_completion_element_t comp_element)
```

• To handle posting RDMA receive operation, use the following APIs:
- To post RDMA receive operation:

```c
void doca_dpa_dev_rdma_post_receive(doca_dpa_dev_rdma_t rdma,
    doca_dpa_dev_mmap_t mmap_handle, uint64_t addr, size_t length)
void doca_dpa_dev_rdma_post_buf_receive(doca_dpa_dev_rdma_t rdma,
    doca_dpa_dev_buf_t receive_buf_handle)
```

- Acknowledge that post receive operations are done (data has been received on associated data buffers). This acknowledgment enables DPA RDMA to repost `#num_acked` new post receive operations.

```c
void doca_dpa_dev_rdma_receive_ack(doca_dpa_dev_rdma_t rdma, uint32_t num_acked)
```

- To perform an atomic add operation on the remote side buffer:

```c
void doca_dpa_dev_rdma_post_atomic_fetch_add(doca_dpa_dev_rdma_t rdma,
    doca_dpa_dev_mmap_t dst_mmap_handle,
    uint64_t dst_addr,
    uint64_t value,
    uint32_t completion_requested)
void doca_dpa_dev_rdma_post_buf_atomic_fetch_add(doca_dpa_dev_rdma_t rdma,
    doca_dpa_dev_buf_t dst_buf_handle,
    uint64_t value,
    uint32_t completion_requested)
```

- To signal a remote event:

```c
doca_dpa_dev_rdma_signal_<add|set>(doca_dpa_dev_rdma_t rdma,
    doca_dpa_dev_sync_event_remote_t remote_sync_event, uint64_t count)
```

---

**Note**
As all DPA RDMA operations are non-blocking, the following API is provided to kernel launch developers to wait until all previous RDMA operations are done (blocking call) to drain the RDMA DPA handle:

```c
doca_dpa_dev_rdma_synchronize(doca_dpa_dev_rdma_t rdma)
```

When this call returns, all previous non-blocking operations on the DPA RDMA have completed (i.e., sent to the remote RDMA). It is expected that the `doca_dpa_dev_rdma_synchronize()` call would use the same thread as the handle calls.

Since DPA RDMAs are non-thread safe, each DPA RDMA must be accessed by a single thread at any given time. If user launches a kernel that should be executed by more than one thread and this kernel includes RDMA communication, it is expected that a user will use array of RDMAs so that each RDMA will be accessed by single thread (each thread can access it's RDMA instance by using `doca_dpa_dev_thread_rank()` as its index in the array of RDMA handles).

When using the Remote Event Exchange API, `void doca_dpa_dev_rdma_signal_<add|set>(..., doca_dpa_dev_event_remote_t event_handle, ...)`, within your kernel, note that event is a remote event. That is, an event created on the remote node and exported to a remote node (`doca_dpa_event_dev_remote_export(event_handle)`).

**Multiple RDMA Contexts**

To support attaching multiple DOCA RDMA contexts to a single DPA Completion Context, DOCA offers the following APIs.

- **RDMAuser_data** which is set using the host API:
And can be retrieved in device using the completion API:

```c
uint32_t doca_dpa_dev_get_completion_user_data(doca_dpa_dev_completion_element_t comp_element)
```

user_data should be used to distinguish which DOCA RDMA context has triggered this completion.

- RDMA work request index using device API for an RDMA completion:

```c
uint32_t doca_dpa_dev_rdma_completion_get_wr_index(doca_dpa_dev_completion_element_t comp_element)
```

work request index should be used to get operation index of DOCA RDMA context which triggered this completion.

### Example (Host-side Pseudo Code)

```c
// create and start DPA Thread
doca_dpa_thread_create(&dpa_thread);
doca_dpa_thread_set_func_arg(dpa_thread);
doca_dpa_thread_start(dpa_thread);

// create and start DPA Completion Context which is attached to DPA Thread
doca_dpa_completion_create(&dpa_comp);
doca_dpa_completion_set_thread(dpa_comp, dpa_thread);
doca_dpa_completion_start(dpa_comp);

doca_dpa_thread_run(dpa_thread);

// create and start DPA RDMA context which is attached to DPA Completion Context
```
DOCA Libraries

Data Structures

Hash Table

DOCA DPA provides an API to create a hash table on DPA. This data structure is managed on DPA using relevant device APIs.

Host-side API

doca_rdma_create(doca_dev, &rdma);
doca_rdma_dpa_completion_attach(rdma, dpa_comp);
doca_rdma_ctx = doca_rdma_as_ctx(rdma);
doca_ctx_set_datapath_on_dpa(doca_rdma_ctx, doca_dpa);
doca_ctx_set_user_data(doca_rdma_ctx, user_data);
doca_ctx_start(doca_rdma_ctx);
doca_rdma_get_dpa_handle(rdma, &handle);

Info

Each completion on an RDMA operation triggers \texttt{dpa\_thread}.

Info

Use output parameter \texttt{handle} in relevant RDMA device APIs in the thread kernel.
• To create a hash table on DPA:

```c
doca_error_t doca_dpa_hash_table_create(struct doca_dpa *dpa, unsigned int num_entries, struct doca_dpa_hash_table **ht)
```

• To destroy a hash table:

```c
doca_error_t doca_dpa_hash_table_destroy(struct doca_dpa_hash_table *ht)
```

• To obtain a DPA handle:

```c
doca_error_t doca_dpa_hash_table_get_dpa_handle(struct doca_dpa_hash_table *ht, doca_dpa_dev_hash_table_t *handle)
```

Use output parameter `handle` in relevant device APIs in the thread kernel.

**Device-side API**

• To add a new entry to the hash table:

```c
void doca_dpa_dev_hash_table_add(doca_dpa_dev_hash_table_t ht_handle, uint32_t key, uint64_t value)
```

**Note**

Adding a new key when the hash table is full causes anomalous behavior.
- To remove an entry from the hash table:

```c
void doca_dpa_dev_hash_table_remove(doca_dpa_dev_hash_table_t ht_handle, uint32_t key)
```

- To return the value to which the specified key is mapped in the hash table:

```c
int doca_dpa_dev_hash_table_find(doca_dpa_dev_hash_table_t ht_handle, uint32_t key, uint64_t *value)
```

---

**RPC and Kernel Launch**

**RPC**

**Host-side API**

A blocking one-time call from the host application to execute a kernel on DPA.

- **Info**

  RPC is mainly used for control path.

The RPC's return value is reported back to the host application.

```c
doca_error_t doca_dpa_rpc(struct doca_dpa *dpa, doca_dpa_func_t *func, uint64_t *retval, ... /* func arguments */)```
Example

- Device-side – DPA device `func` must be annotated with `__dpa_rpc__` annotation, such as:

  ```c
  __dpa_rpc__ uint64_t hello_rpc(int arg)
  {
    ...
  }
  ```

- Host-side:

  ```c
  extern doca_dpa_func_t hello_rpc;
  uint64_t retval;
  doca_dpa_rpc(dpa, &hello_rpc, &retval, 10);
  ```

Kernel Launch

DOCA DPA provides an API which enables full control for launching and monitoring kernels.

Since DOCA DPA libraries are not thread-safe, it is up to the programmer to make sure the kernel is written to allow it to run in a multi-threaded environment. For example, to program a kernel that uses RDMAs with 16 concurrent threads, the user should pass an array of 16 RDMAs to the kernel so that each thread can access its RDMA using its rank `doca_dpa_dev_thread_rank()` as an index to the array.

Host-side API

```c
doca_dpa_kernel_launch_update_<add|set>(struct doca_dpa *dpa, struct doca_sync_event *wait_event, uint64_t wait_threshold, struct doca_sync_event *comp_event, uint64_t comp_count, unsigned int num_threads, doca_dpa_func_t *func, ... /* args */)
```
This function asks DOCA DPA to run `func` in DPA by `num_threads` and give it the supplied list of arguments (variadic list of arguments).

This function is asynchronous so when it returns, it does not mean that `func` started/ended its execution.

To add control or flow/ordering to these asynchronous kernels, two optional parameters for launching kernels are available:

- **wait_event** – the kernel does not start its execution until the event is signaled (if NULL, the kernel starts once DOCA DPA has an available EU to run on it) which means that DOCA DPA would not run the kernel until the event's counter is bigger than `wait_threshold`.

- **comp_event** – once the last thread running the kernel is done, DOCA DPA updates this event (either sets or adds to its current counter value with `comp_count`).

- DOCA DPA takes care of packing (on host/Target BlueField) and unpacking (in DPA) the kernel parameters.

- `func` must be prefixed with the `__dpa_global__` macro for DPACC to compile it as a kernel (and add it to DPA executable binary) and not as part of host application binary.

- The programmer must declare `func` in their application also by adding the line `extern doca_dpa_func_t func`.

**Device-side API**

**Note**

Please note that the valid values for `wait_threshold` and `wait_event` counter and are [0-254]. Values out of this range might cause anomalous behavior.
To retrieve the running thread's rank for a given kernel on the DPA. If, for example, a kernel is launched to run with 16 threads, each thread running this kernel is assigned a rank ranging from 0 to 15 within this kernel. This is helpful for making sure each thread in the kernel only accesses data relevant for its execution to avoid data-races:

```
unsigned int doca_dpa_dev_thread_rank()
```

To return the number of threads running current kernel:

```
unsigned int doca_dpa_dev_num_threads()
```

To yield the thread which runs the kernel:

```
void doca_dpa_dev_yield(void)
```

Examples

**Linear Execution Example**

**Kernel Code**

```
#include "doca_dpa_dev.h"
#include "doca_dpa_dev_sync_event.h"
```
__dpa_global__ void
linear_kernel(doca_dpa_dev_sync_event_t wait_ev, doca_dpa_dev_sync_event_t comp_ev)
{
    if (wait_ev)
        doca_dpa_dev_sync_event_wait_gt(wait_ev, wait_th = 0);

    doca_dpa_dev_sync_event_update_add(comp_ev, comp_count = 1);
}

#include <doca_dev.h>
#include <doca_error.h>
#include <doca_sync_event.h>
#include <doca_dpa.h>

int main(int argc, char **argv)
{
    /*
     A
     | B
     | C
    */

    /* Open DOCA device */
    open_doca_dev(&doca_dev);
    /* Create doca dpa conext */
    doca_dpa_create(doca_dev, dpa_linear_app, &dpa_ctx, 0);

    /* Create event A - subscriber is DPA and publisher is CPU */
    doca_sync_event_create(&ev_a);
    doca_sync_event_add_publisher_location_cpu(ev_a, doca_dev);
    doca_sync_event_add_subscriber_location_dpa(ev_a, dpa_ctx);
    doca_sync_event_start(ev_a);

    /* Create event B - subscriber and publisher are DPA */
doca_sync_event_create(&ev_b);
doca_sync_event_add_publisher_location_dpa(ev_b, dpa_ctx);
doca_sync_event_add_subscriber_location_dpa(ev_b, dpa_ctx);
doca_sync_event_start(ev_b);

/* Create event C - subscriber and publisher are DPA */
doca_sync_event_create(&ev_c);
doca_sync_event_add_publisher_location_dpa(ev_c, dpa_ctx);
doca_sync_event_add_subscriber_location_dpa(ev_c, dpa_ctx);
doca_sync_event_start(ev_c);

/* Create completion event for last kernel - subscriber is CPU and publisher is DPA */
doca_sync_event_create(&comp_ev);
doca_sync_event_add_publisher_location_dpa(comp_ev, dpa_ctx);
doca_sync_event_add_subscriber_location_cpu(comp_ev, doca_dev);
doca_sync_event_start(comp_ev);

/* Export kernel events and acquire their handles */
doca_sync_event_get_dpa_handle(ev_b, dpa_ctx, &ev_b_handle);
doca_sync_event_get_dpa_handle(ev_c, dpa_ctx, &ev_c_handle);
doca_sync_event_get_dpa_handle(comp_ev, dpa_ctx, &comp_ev_handle);

/* Launch kernels */
doca_dpa_kernel_launch_update_add(wait_ev = ev_a, wait_threshold = 1, num_threads = 1,
&linear_kernel, kernel_args: NULL, ev_b_handle);
doca_dpa_kernel_launch_update_add(wait_ev = NULL, num_threads = 1, &linear_kernel, kernel_args:
ev_b_handle, ev_c_handle);
doca_dpa_kernel_launch_update_add(wait_ev = NULL, &linear_kernel, num_threads = 1, kernel_args:
ev_c_handle, comp_ev_handle);

/* Update host event to trigger kernels to start executing in a linear manner */
doca_sync_event_update_set(ev_a, 1)

/* Wait for completion of last kernel */
doca_sync_event_wait_gt(comp_ev, 0);

/* Tear Down... */
teardown_resources();
}
**Diamond Execution Example**

**Kernel Code**

```c
#include "doca_dpa_dev.h"
#include "doca_dpa_dev_sync_event.h"

__dpa_global__ void
diamond_kernel(doca_dpa_dev_sync_event_t wait_ev, uint64_t wait_th, doca_dpa_dev_sync_event_t comp_ev1, doca_dpa_dev_sync_event_t comp_ev2)
{
    if (wait_ev)
        doca_dpa_dev_sync_event_wait_gt(wait_ev, wait_th);

doca_dpa_dev_sync_event_update_add(comp_ev1, comp_count = 1);
    if (comp_ev2) // can be 0 (NULL)
        doca_dpa_dev_sync_event_update_add(comp_ev2, comp_count = 1);
}
```

**Host Application Pseudo Code**

```c
#include <doca_dev.h>
#include <doca_error.h>
#include <doca_sync_event.h>
#include <doca_dpa.h>

int main(int argc, char **argv)
{
    /*
     * A
     * \C B
     * / / D /
     * \ E
     */

    /* Open DOCA device */
```
open_doca_dev(&doca_dev);
/* Create doca dpa conext */
doca_dpa_create(doca_dev, dpa_diamond_app, &dpa_ctx, 0);
/* Create root event A that will signal from the host the rest to start */
doca_sync_event_create(&ev_a);
// set publisher to CPU, subscriber to DPA and start event
/* Create events B,C,D,E */
doca_sync_event_create(&ev_b);
doca_sync_event_create(&ev_c);
doca_sync_event_create(&ev_d);
doca_sync_event_create(&ev_e);
// for events B,C,D,E, set publisher & subscriber to DPA and start event
/* Create completion event for last kernel */
doca_sync_event_create(&comp_ev);
// set publisher to DPA, subscriber to CPU and start event
/* Export kernel events and acquire their handles */
doca_sync_event_get_dpa_handle(&ev_b_handle, &ev_c_handle, &ev_d_handle, &ev_e_handle, &comp_ev_handle);
/* wait threshold for each kernel is the number of parent nodes */
constexpr uint64_t wait_threshold_one_parent {1};
castexpr uint64_t wait_threshold_two_parent {2};
/* launch diamond kernels */
doca_dpa_kernel_launch_update_set(wait_ev = ev_a, wait_threshold = 1, num_threads = 1, &diamond_kernel, kernel_args: NULL, 0, ev_b_handle, ev_c_handle);
doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_b_handle, wait_threshold_one_parent, ev_e_handle, NULL);
doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_c_handle, wait_threshold_one_parent, ev_d_handle, NULL);
doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_d_handle, wait_threshold_one_parent, ev_e_handle, NULL);
doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_e_handle, wait_threshold_two_parent, comp_ev_handle, NULL);
/* Update host event to trigger kernels to start executing in a diamond manner */
doca_sync_event_update_set(ev_a, 1);
/* Wait for completion of last kernel */
doca_sync_event_wait_gt(comp_ev, 0);
Performance Optimizations

- The time interval between a kernel launch call from the host and the start of its execution on the DPA is significantly optimized when the host application calls `doca_dpa_kernel_launch_update_<add|set>()` repeatedly to execute with the same number of DPA threads. So, if the application calls `doca_dpa_kernel_launch_update_<add|set>(..., num_threads = x)`, the next call with `num_threads = x` would have a shorter latency (as low as ~5-7 microseconds) for the start of the kernel's execution.

- Applications calling for kernel launch with a wait event (i.e., the completion event of a previous kernel) also have significantly lower latency in the time between the host launching the kernel and the start of the execution of the kernel on the DPA. So, if the application calls `doca_dpa_kernel_launch_update_<add|set>(..., completion event = m_ev, ...) and then `doca_dpa_kernel_launch_update_<add|set>( wait event = m_ev, ...)`, the latter kernel launch call would have shorter latency (as low as ~3 microseconds) for the start of the kernel's execution.

Limitations

- The order in which kernels are launched is important. If an application launches K1 and then K2, K1 must not depend on K2's completion (e.g., wait on its wait event that K2 should update).

Not following this guideline leads to unpredictable results (at runtime) for the application and might require restarting the DOCA DPA context (i.e., destroying, reinitializing, and rerunning the workload).

- DPA threads are an actual hardware resource and are, therefore, limited in number to 256 (including internal allocations and allocations explicitly requested by the user as part of the kernel launch API)
DOCA DPA does not check these limits. It is up to the application to adhere to this number and track thread allocation across different DPA contexts.

- Each `doca_dpa_dev_rdma_t` consumes one thread.

- The DPA has an internal watchdog timer to make sure threads do not block indefinitely. Kernel execution time must be finite and not exceed the time returned by `doca_dpa_get_kernel_max_run_time`.

- The `num_threads` parameter in the `doca_dpa_kernel_launch` call cannot exceed the maximum allowed number of threads to run a kernel returned by `doca_dpa_get_max_threads_per_kernel`.

---

**Logging and Tracing**

DOCA DPA provides a set of debugging APIs to allow diagnosing and troubleshooting any issues on the device, as well as accessing real-time information from the running application.

Logging in the data path has significant impact on an application's performance. While the tracer provided by the library is of high-frequency and is designed to prevent significant impact on the application's performance.

Therefore it's recommended to use:

- Logging in the control path
- Tracing in the data path

The user is able to control the log/trace file path and device log verbosity.

**Host-side API**

- To set/get the trace file path:

  ```c
  doca_error_t doca_dpa_trace_file_set_path(struct doca_dpa *dpa, const char *file_path)
  ```
To set/get the log file path:

```c
doca_error_t doca_dpa_trace_file_get_path(struct doca_dpa *dpa, char *file_path, uint32_t *file_path_len);

doca_error_t doca_dpa_trace_file_set_path(struct doca_dpa *dpa, const char *file_path);
```

To set/get device log verbosity:

```c
void doca_dpa_dev_log(doca_dpa_dev_log_level_t log_level, const char *format, ...)

typedef enum doca_dpa_dev_log_level {
    DOCA_DPA_DEV_LOG_LEVEL_DISABLE = 10, /**< Disable log messages */
    DOCA_DPA_DEV_LOG_LEVEL_CRIT = 20, /**< Critical log level */
    DOCA_DPA_DEV_LOG_LEVEL_ERROR = 30, /**< Error log level */
    DOCA_DPA_DEV_LOG_LEVEL_WARNING = 40, /**< Warning log level */
    DOCA_DPA_DEV_LOG_LEVEL_INFO = 50, /**< Info log level */
    DOCA_DPA_DEV_LOG_LEVEL_DEBUG = 60, /**< Debug log level */
} doca_dpa_dev_log_level_t;
```

Log macros:

```
DOCA_DPA_DEV_LOG_CRIT(...)
```
To create a trace message entry with arguments:

```c
void doca_dpa_dev_trace(uint64_t arg1, uint64_t arg2, uint64_t arg3, uint64_t arg4, uint64_t arg5)
```

To flush the trace message buffer to host:

```c
void doca_dpa_dev_trace_flush(void)
```

## Error Handling

DPA context can enter an error state caused by the device flow. The application can check this error state by calling the following host API:

```c
doca_error_t doca_dpa_peek_at_last_error(const struct doca_dpa *dpa)
```

If a fatal error core dump and crash occur, data is written to the file path `/tmp/doca_dpa_fatal` or to the file path set by the API `doca_dpa_log_file_set_path()`, with the suffixes `.PID.core` and `.PID.crash` respectively, where PID is the process ID. The data written to the file would include a memory snapshot at the time of the crash, which would contain information instrumental in pinpointing the cause of a crash (e.g., the program’s state, variable values, and the call stack).
Creating core dump files can be done after the DPA application has crashed.

Info

This call does not reset the error state.

Note

If an error occurred, DPA context enters a fatal state and must be destroyed by the user.

Hello World Example

Procedure Outline

1. Write DPA device code (i.e., kernels or .c files).

2. Use DPACC to build a DPA program (i.e., a host library which contains an embedded device executable). Input for DPACC:
   1. Kernels from step 1.
   2. DOCA DPA device library.

3. Build host executable using a host compiler. Input for the host compiler:
   1. DPA program.
2. User host application `.c/.cpp` files.

4. Run host executable.

**Procedure Steps**

The following code snippets show a basic DPA code that eventually prints "Hello World" to stdout.

This is achieved using:

- A DPA Thread which prints the string and signals a DOCA Sync Event to indicate completion to host application
- A DPA RPC to notify DPA Thread

The steps are elaborated in the following subsections.

**Prepare Kernels Code**

```c
#include "doca_dpa_dev.h"
#include "doca_dpa_dev_sync_event.h"

__dpa_global__ void hello_world_thread_kernel(uint64_t arg)
{
    DOCA_DPA_DEV_LOG_INFO("Hello World From DPA Thread!
    ");
    doca_dpa_dev_sync_event_update_set(arg, 1);
    doca_dpa_dev_thread_finish();
}
```
Prepare Host Application Code

```c
#include <stdio.h>
#include <unistd.h>
#include <doca_dev.h>
#include <doca_error.h>
#include <doca_sync_event.h>
#include <doca_dpa.h>

/**
 * A struct that includes all needed info on registered kernels and is initialized during linkage by DPACC.
 * Variable name should be the token passed to DPACC with --app-name parameter.
 */
extern struct doca_dpa_app *dpa_hello_world_app;

/**
 * kernel declaration that the user must declare for each kernel and DPACC is responsible to initialize.
 * Only then, user can use this variable in relevant host APIs
 */
doca_dpa_func_t hello_world_thread_kernel;
doca_dpa_func_t hello_world_thread_notify_rpc;

int main(int argc, char **argv)
{
    struct doca_dev *doca_dev = NULL;
    struct doca_dpa *dpa_ctx = NULL;
    struct doca_sync_event *cpu_se = NULL;
    doca_dpa_dev_sync_event_t cpu_se_handle = 0;
    struct doca_dpa_thread *dpa_thread = NULL;
    struct doca_dpa_notification_completion *notify_comp = NULL;
```
doca_dpa_dev_notification_completion_t notify_comp_handle = 0;
uint64_t retval = 0;

printf("\n----> Open DOCA Device\n");
/* Open appropriate DOCA device doca_dev */

printf("\n----> Initialize DOCA DPA Context\n");
doca_dpa_create(doca_dev, &dpa_ctx);
doca_dpa_set_app(dpa_ctx, dpa_hello_world_app);
doca_dpa_start(dpa_ctx);

printf("\n----> Initialize DOCA Sync Event\n");
doca_sync_event_create(&cpu_se);
doca_sync_event_add_publisher_location_dpa(cpu_se, dpa_ctx);
doca_sync_event_add_subscriber_location_cpu(cpu_se, doca_dev);
doca_sync_event_start(cpu_se);
doca_sync_event_get_dpa_handle(cpu_se, dpa_ctx, &cpu_se_handle);

printf("\n----> Initialize DOCA DPA Thread\n");
doca_dpa_thread_create(dpa_ctx, &dpa_thread);
doca_dpa_thread_set_func_arg(dpa_thread, &hello_world_thread_kernel, cpu_se_handle);
doca_dpa_thread_start(dpa_thread);

printf("\n----> Initialize DOCA DPA Notification Completion\n");
doca_dpa_notification_completion_create(dpa_ctx, dpa_thread, &notify_comp);
doca_dpa_notification_completion_start(notify_comp);
doca_dpa_notification_completion_get_dpa_handle(notify_comp, &notify_comp_handle);

printf("\n----> Run DOCA DPA Thread\n");
doca_dpa_thread_run(dpa_thread);

printf("\n----> Trigger DPA RPC\n");
doca_dpa_rpc(dpa_ctx, &hello_world_thread_notify_rpc, &retval, notify_comp_handle);

printf("\n----> Waiting For hello_world_thread_kernel To Finish\n");
doca_sync_event_wait_gt(cpu_se, 0, 0xFFFFFFFFFFFFFFFF);

printf("\n----> Destroy DOCA DPA Notification Completion\n");
doca_dpa_notification_completion_destroy(notify_comp);

printf("\n----> Destroy DOCA DPA Thread\n");
doca_dpa_thread_destroy(dpa_thread);

printf("\n----> Destroy DOCA DPA event\n");
```c
doca_sync_event_destroy(cpu_se);

printf("\n----> Destroy DOCA DPA context\n");
doca_dpa_destroy(dpa_ctx);

printf("\n----> Destroy DOCA device\n");
doca_dev_close(doca_dev);

printf("\n----> DONE!\n");
return 0;
```

### Build DPA Program

```
/opt/mellanox/doca/tools/dpacc \
   kernel.c \ 
   -o dpa_program.a \ 
   -hostcc=gcc \ 
   -hostcc-options="-Wno-deprecated-declarations -Werror -Wall -Wextra -W" \ 
   --devicecc-options="-D__linux__ -Wno-deprecated-declarations -Werror -Wall -Wextra -W" \ 
   --app-name="dpa_hello_world_app" \ 
   -ldpa \ 
   -I/opt/mellanox/doca/include/
```

### Build Host Application

```
gcc hello_world.c -o hello_world \ 
   dpa_program.a \ 
   -I/opt/mellanox/doca/include/ \ 
   -DDOCA_ALLOW_EXPERIMENTAL_API \ 
   -L/opt/mellanox/doca/lib/x86_64-linux-gnu/ -ldoca_dpa -ldoca_common \ 
   -L/opt/mellanox/flexio/lib -lflexio \ 
   -lstdc++ -libverbs -Ilmlx5
```

### Execution
$ ./hello_world

----> Open DOCA Device

----> Initialize DOCA DPA Context

----> Initialize DOCA Sync Event

----> Initialize DOCA DPA Thread

----> Initialize DOCA DPA Notification Completion

----> Run DOCA DPA Thread

----> Trigger DPA RPC

/ 10/[DOCA][DPA DEVICE][INF] Notifying DPA Thread From RPC

/ 8/[DOCA][DPA DEVICE][INF] Hello World From DPA Thread!

----> Waiting For hello_world_thread_kernel To Finish

----> Destroy DOCA DPA Notification Completion

----> Destroy DOCA DPA Thread

----> Destroy DOCA DPA event

----> Destroy DOCA DPA context

----> Destroy DOCA device

----> DONE!

**Samples**

This section provides DPA sample implementation on top of the BlueField-3 networking platform.
To run DPA samples:

1. Refer to the following documents:
   
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_dpa/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   Info

   The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample (e.g., `dpa_initiator_target`) usage:

   ```
   Usage: doca_dpa_initiator_target [DOCA Flags] [Program Flags]
   
   DOCA Flags:
   -h, --help
   Print a help synopsis
   ```
4. For additional information per sample, use the -h option:

```
/tmp/build/doca_<sample_name> -h
```

---

**Basic Initiator Target**

This sample illustrates how to trigger DPA Thread using DPA Completion Context attached to DOCA RDMA.

This sample consists of initiator and target endpoints.

In the initiator endpoint, a DOCA RDMA executes RDMA post send operation using DPA RPC.

In the target endpoint, a DOCA RDMA, attached to DPA Completion Context, executes RDMA post receive operation using DPA RPC.

Completion on the post receive operation triggers DPA Thread which prints completion info and sets DOCA Sync Event to release the host application that waits on that event before destroying all resources and finish.

The sample logic includes:

1. Allocating DOCA DPA & DOCA RDMA resources for both initiator and target endpoints.

   2. Target: Attaching DOCA RDMA to DPA Completion Context which is attached to DPA Thread.
3. Run DPA Thread.

4. Target: DPA RPC to execute RDMA post receive operation.

5. Initiator: DPA RPC to execute RDMA post send operation.

6. The completion on the post receive operation triggers DPA Thread.

7. Waiting on completion event to be set from DPA Thread.

8. Destroying all resources.

Reference:

- /opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/dpa_basic_initiator_target_main.c
- /opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/host/dpa_basic_initiator_target_samp
- /opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/device/dpa_basic_initiator_target_keri
- /opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/meson.build
- /opt/mellanox/doca/samples/doca_dpa/dpa_common.h
- /opt/mellanox/doca/samples/doca_dpa/dpa_common.c
- /opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh
Advanced Initiator Target

This sample illustrates how to trigger DPA threads using both DPA Notification Completion and DPA Completion Context which is attached to multiple DOCA RDMAs.

This sample consists of initiator and target endpoints.

In the initiator endpoint, two DOCA RDMAs execute an RDMA post send operation using DPA RPC in the following order:

1. RDMA #1 executes the RDMA post send operation on buffer with value 1.
2. RDMA #2 executes the RDMA post send operation on buffer with value 2.
3. RDMA #1 executes the RDMA post send operation on buffer with value 3.
4. RDMA #2 executes the RDMA post send operation on buffer with value 4.

In the target endpoint, two DOCA RDMAs, RDMA #1 with user data 111 and RDMA #2 with user data 222.

Target RDMAs are attached to a single DPA Completion Context which is attached to DPA Thread #1.

Target RDMAs execute the initial RDMA post receive operation using DPA RPC.

Completions on the post receive operations trigger DPA Thread #1 which:

1. Prints completion info including user data.
2. Updates a local data base with the receive buffer value.
3. Repost RDMA receive operation.
4. Ack, request completion and reschedule.

Once target DPA Thread #1 receives all expected values "1, 2, 3, 4", it notify DPA Thread #2 and finish.

Once DPA Thread #2 is triggered, it sets DOCA Sync Event to release the host application that waits on that event before destroying all resources and finishing.
The sample logic includes:

1. Allocating DOCA DPA and DOCA RDMA resources for both initiator and target endpoints.

2. Target: Attaching both DOCA RDMAs to DPA Completion Context which is attached to DPA Thread #1.

3. Target: Attaching DPA Notification Completion to DPA Thread #2.

4. Run DPA threads.

5. Target: DPA RPC to execute the initial RDMA post receive operation.

6. Initiator: DPA RPC to execute all RDMA post send operations.

7. Completions on the post receive operations (4 completions) trigger DPA Thread #1.

8. Once all expected values are received, DPA Thread #1 notifies DPA Thread #2 and finishes.

9. Waiting on completion event to be set from DPA Thread #2.

10. Destroying all resources.

Reference:

- /opt/mellanox/doca/samples/doca_dpa/dpa_initiator_target/dpa_initiator_target_main.c
- /opt/mellanox/doca/samples/doca_dpa/dpa_initiator_target/host/dpa_initiator_target_sample.c
Ping Pong

This sample illustrates the functionality of the following DPA objects:

- DPA Thread
- DPA Completion Context
- DOCA RDMA

This sample consists of ping and pong endpoints which run for 100 iterations. On each iteration, DPA threads (ping and pong) post RDMA receive and send operations for data buffers with values [0-99].

Once all expected values are received on each DPA thread, it sets a DOCA Sync Event to release the host application waiting on that event before destroying all resources and finishes.

To trigger DPA threads, the sample uses a DPA RPC.

The sample logic includes:

1. Allocating DOCA DPA and DOCA RDMA resources.
2. Attaching DOCA RDMA to DPA completion context which is attached to DPA thread.
3. Run DPA threads.
4. DPA RPC to trigger DPA threads.
5. 100 ping pong iterations of RDMA post receive and send operations.

6. Waiting on completion events to be set from DPA threads.

7. Destroying all resources.

Reference:

- /opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/dpa_ping_pong_main.c
- /opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/host/dpa_ping_pong_sample.c
- /opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/device/dpa_ping_pong_kernels_dev.c
- /opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/meson.build
- /opt/mellanox/doca/samples/doca_dpa/dpa_common.h
- /opt/mellanox/doca/samples/doca_dpa/dpa_common.c
- /opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh

**Kernel Launch**
This sample illustrates how to launch a DOCA DPA kernel with wait and completion DOCA sync events.

The sample logic includes:

1. Allocating DOCA DPA resources.
2. Initializing wait and completion DOCA sync events for the DOCA DPA kernel.
3. Running `hello_world` DOCA DPA kernel that waits on the wait event.
4. Running a separate thread that triggers the wait event.
5. `hello_world` DOCA DPA kernel prints "Hello from kernel".
6. Waiting for the completion event of the kernel.
7. Destroying the events and resources.

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/dpa_wait_kernel_launch_main.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/host/dpa_wait_kernel_launch_sample.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/device/dpa_wait_kernel_launch_kernels`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/meson.build`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.h`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.c`
- `/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh`

**DOCA PCC**

This guide provides an overview and configuration instructions for DOCA Programmable Congestion Control (PCC) API.
**Introduction**

The DOCA PCC library provides a high-level programming interface that allows users to implement their own customized congestion control (CC) algorithm, facilitating efficient management of network congestion in their applications.

The DOCA PCC library provides an API to:

- Configure probe packets to send and receive
- Get the CC event/packet and access its fields
- Set a rate limit for a flow
- Maintain a context for each flow
- Initiate and configure CC algorithms
- Obtain request packets arriving from the network and setup response packets in return

This library uses the NVIDIA® BlueField®-3 Platform hardware acceleration for CC management, while providing an API that simplifies hardware complexity, allowing users to focus on the functionality of the CC algorithm.

**Prerequisites**

DOCA PCC-based applications can run either on the host machine or on the NVIDIA® BlueField®-3 Platform (or later) target.

![Info]

Currently, DOCA PCC is only supported for ETHERNET link type.

To enable DOCA PCC RP:

1. Run the following on the host/VM:
2. Perform **graceful shutdown** then power cycle the host.

To enable DOCA PCC NP:

1. Run the following on the host/VM

```
mlxconfig -d <mlx_device> -y s USER_PROGRAMMABLE_CC=1
```

```
mlxconfig -d <mlx_device> -y s PCC_INT_EN=0
```

2. Perform **graceful shutdown** then power cycle the host.

**Note**

Configuring `PCC_INT_EN` to 1 blocks the creation of DOCA PCC NP context and enables legacy NP solution. In addition, it only supports DOCA PCC RP context to set Congestion Control Message After Drop (CCMAD) probe packet format.

If IFA2.0 support is needed, user needs to enable DOCA PCC RP and DOCA PCC NP on all nodes of the cluster.

**Note**

If running from the host in NIC mode, users must have PRIVILIGED permission to configure the above parameters. To check privileging level, run:

```
mlxprivhost -d <mlx_device> q
```
The DPACC tool is used to compile and link user algorithm and device code with the DOCA PCC device library to get applications that can be loaded from the host program.

DPACC is bundled as part of the DOCA SDK installation package. For more information on DPACC, refer to *NVIDIA DOCA DPACC Compiler*.

### Changes From Previous Releases

#### Changes in 2.8.0

**Added**

- `doca_pcc_dump_debug(const struct doca_pcc *pcc)`
- `doca_pcc_enable_debug(const struct doca_pcc *pcc, bool enable)`

**Changed**

- `DOCA_PCC_DEV_MAX_NUM_PARAMS_PER_ALGO (0x1E) → (0x26)`
- `DOCA_PCC_DEV_MAX_NUM_COUNTERS_PER_ALGO (0xF) → (0x3F)`
- `struct mlnx_cc_event_general_attr_t` – port_num split to two fields (see diff below)

```c
struct mlnx_cc_event_general_attr_t { /* Little Endian */
    uint32_t ev_type:8;  /* event type */
    uint32_t ev_subtype:8; /* event subtype */
    - uint32_t port_num:8; /* port id */
    + uint32_t port_num:4; /* port id */
    + uint32_t reserved:4;
        uint32_t flags:8; /* event flags */
};
```

- `struct mlnx_cc_event_t` – added 12B reserved

```c
struct mlnx_cc_event_t { /* Little Endian */
`;```
Dependencies

The library requires firmware version 32.38.1000 and higher.

Architecture

DOCA PCC comprises three main components which are part of the DOCA SDK installation package:

**Host Library**

The host library offers a unified interface for managing the DOCA PCC context configuration.

As part of the control path, the host library integrates passively within the application, orchestrating congestion control activities without directly handling data transmission.

Host/device library and header files:

```
+    uint32_t reserved[3]; /* reserved */
struct mlnx_cc_event_general_attr_t ev_attr; /* event general attributes */
    uint32_t flow_tag; /* unique flow id */
    uint32_t sn; /* serial number */
    uint32_t timestamp; /* event timestamp */
union mlnx_cc_event_spec_attr_t ev_spec_attr; /* attributes which are different for different events */
};
```

**Device Libraries**
The DOCA PCC context assumes one of two roles:

- **Reaction point (RP):** Monitors network conditions actively, dynamically adjusting data transmission rates to alleviate congestion promptly. RP context is global per NIC.

Device library and header files:

- **Notification point (NP):** Passively receives congestion notifications from external sources, processing them intelligently to facilitate informed decisions within the application. NP context is global per e-switch owner.

Device library and header files:

Both RP and NP device libraries share common headers:
Currently, the device library and the user algorithm are implemented and managed over the BlueField's data-path accelerator (DPA) subsystem.

For more info on DPA, refer to [DPA Subsystem](#).

**Development Flow**

DOCA enables developers to program the congestion control algorithm into the system using the DOCA PCC library.

The following are the required steps to start programming:

1. Implement CC algorithms and probe packet handling using the API provided by the device header files.

2. Implement the user callbacks defined by the library for DataPath:
   - For RP: doca_pcc_dev_user_init(), doca_pcc_dev_user_set_algo_params(), doca_pcc_dev_user_algo().
   - For NP: doca_pcc_dev_np_user_packet_handler()

3. Use DPACC to build a DPA application (i.e., a host library which contains an embedded device executable). Input for DPACC are the files containing the implementation of the previous steps.

4. Build host executable using a host compiler. Inputs for the host compiler are the DPA application generated in the previous step and the user application host source files.
5. In the host executable, create and start a DOCA PCC context which is set with the DPA application containing the device code.

For a more descriptive example, refer to *NVIDIA DOCA PCC Application Guide*.

**System Design**

DOCA PCC flow for implementing an RP program:

DOCA PCC flow for implementing an NP program:
API

For the library API reference, refer to PCC API documentation in the *NVIDIA DOCA Library APIs*.

The following sections provide additional details about the library API.

**Host API**

The host library API consists of calls to set the PCC context attributes and observe availability of the process.

**Selecting and Opening DOCA Device**

To perform PCC operations, a device must be selected. To select a device, users may iterate over all DOCA devices using `doca_devinfo_list_create()` and check whether the device supports the desired PCC role either via `doca_devinfo_get_is_pcc_supported()` for RP, or `doca_pcc_np_cap_is_supported()` for NP.

**Setting Up and Starting DOCA PCC Context**

After selecting a DOCA device, a PCC context can be created.
As described in the Architecture section, The DOCA PCC library provides APIs to leverage Reaction Points (RP) and Notification Points (NP) to implement programmable congestion control strategies.

Call `doca_pcc_create()` to create a DOCA PCC RP context, and `doca_pcc_np_create()` to create a DOCA PCC NP context.

Afterwards, the following attributes must be set for the PCC context:

- Context app – the name of the DPA application compiled using DPACC, consisting of the device algorithm and code. This is set using the call `doca_pcc_set_app()`.

- Context threads – the affinity of DPA threads to be used to handle CC events. This is set using the call `doca_pcc_set_thread_affinity()`. The number of threads to be used must be constrained between the minimum and maximum number of threads allowed to run the PCC process (see `doca_pcc_get_min_num_threads()` and `doca_pcc_get_max_num_threads()`). The availability and usage of the threads for PCC is dependent on the complexity of the CC algorithm, link rate, and other potential DPA users.

**Note**

Users can manage DPA threads in the system using EU pre-configuration with the `dpaeumgmt` tool. For more information, refer to *NVIDIA DOCA DPA Execution Unit Management Tool*.

After setting up the context attributes, the context can be started using `doca_pcc_start()`. Starting the context initiates the CC algorithm supplied by the user.

**Configuring Probe Packets**

The DOCA PCC library provides APIs to configure the probe packet settings to tailor congestion control behaviors according to specific network conditions.

The probe packet serves to probe the network for congestion and gather essential feedback for congestion control algorithms.
The DOCA PCC Library supports the following probe packet types:

- **CCMAD** – Provides information about the network's round-trip time so the algorithm can detect and adapt to congestion proactively

- **IFA1** – In-band Flow Analyzer 1 packets provide in-band congestion feedback for proactive congestion control

- **IFA2** – In-band Flow Analyzer 2 packets offer an alternative method for in-band congestion feedback, optimized for specific network environments

### Configuring Dedicated Fields for Different Probe Types

The DOCA PCC library provides APIs to configure specific fields in different supported probe packet types.

- **IFA1** – support to configure probe marker

- **IFA2** – support to configure gns and hop limit

### Configuring Remote NP Handler

To enable Reaction Point contexts to interact with remote Notification Point contexts, the DOCA PCC library provides an API to set the expected remote handler type.

When the DOCA PCC RP process expects CCMAD probe packet responses from a DOCA PCC NP process, it should set it as so using the API `doca_pcc_rp_set_ccmad_remote_sw_handler()`. If not set, the DOCA PCC RP process expects that no remote DOCA PCC NP process is activated, and that responses are handled by the remote node's hardware. Note that if using other probe types than CCMAD, probe packet responses are always expected to be generated from a remote DOCA Notification Point process.

### Debuggability

The DOCA PCC library provides a set of debugging APIs to allow the user to diagnose and troubleshoot any issues on the device, as well as accessing real-time information from
the running application:

- `doca_pcc_set_dev_coredump_file()` – API to set a filename to write crash data and core dump into should a fatal error occur on the device side of the application. The data written into the file would include a memory snapshot at the time of the crash, which would contain information instrumental in pinpointing the cause of a crash (e.g., the program's state, variable values, and the call stack).

- `doca_pcc_set_trace_message()` – API to enable tracing on the device side of the application by setting trace message formats that can be printed from the device. The tracer provided by the library is of high-frequency and is designed to not have significant impact on the application's performance. This API can help the user to monitor and gain insight into the behavior of the running device algorithm, identify performance bottlenecks, and diagnose issues, without incurring any notable performance degradation.

- `doca_pcc_set_print_buffer_size()` – API to set the buffer size to be printed by the print API provided by the device library.

**Host - Device Mailbox**

The DOCA PCC library provides a set of APIs for sending and receiving messages through a mailbox. This service allows communication between the host and device:

- `doca_pcc_set_mailbox()` – API to set the mailbox attributes for the process.

- `doca_pcc_mailbox_get_request_buffer()` and `doca_pcc_mailbox_get_response_buffer()` – API to get the buffers with which the communication will be handled. User can set the request he wants to send to the device, and get a response back.

- `doca_pcc_mailbox_send()` – API to send the mailbox request to the device. This is a blocking call which invokes a callback on the device `doca_pcc_dev_user_mailbox_handle()` which user can handle.

**High Availability**

The DOCA PCC library provides high availability, allowing fast recovery should the running PCC process malfunction. High availability can be achieved by running multiple PCC
processes in parallel.

When calling `doca_pcc_start()`, the library registers the process with the BlueField firmware such that the first PCC process to be registered becomes the ACTIVE PCC process (i.e., actually runs on DPA and handles CC events).

The other processes operate in STANDBY mode. If the ACTIVE process stops processing events or hits an error, the firmware replaces it with one of the standby processes, making it ACTIVE.

The defunct process should call `doca_pcc_destroy()` to free its resources.

The state of the process may be observed periodically using `doca_pcc_get_process_state()`. A change in the state of the process returns the call `doca_pcc_wait()`.

The following values describe the state of the PCC process at any point:

```c
typedef enum {
    DOCA_PCC_PS_ACTIVE = 0,
    /**< The process handles CC events (only one process is active at a given time) */
    DOCA_PCC_PS_STANDBY = 1,
    /**< The process is in standby mode (another process is already ACTIVE)*/
    DOCA_PCC_PS_DEACTIVATED = 2,
    /**< The process was deactivated by NIC FW and should be destroyed */
    DOCA_PCC_PS_ERROR = 3,
    /**< The process is in error state and should be destroyed */
} doca_pcc_process_state_t;
```

**Device API**

The device library API consists of calls to setup the CC algorithm to handle CC events arriving on hardware.

**Counter Sampling**

The device libraries APIs provide an API to sample the NIC bytes counters. These counters help monitor the amount of data transmitted and received through the NIC.
The user can prepare the list of counters to read using `doca_pcc_dev_nic_counters_config()` and sample the new counters values with the call `doca_pcc_dev_nic_counters_sample()`.

**Algorithm Access**

The Reaction Point (RP) device library API provides a set of functions to initiate and identify the different CC algorithms.

The DOCA PCC library is designed to support more than one PCC algorithm. The library comes with a default algorithm which can be used fully or partially by the user using `doca_pcc_dev_default_internal_algo()`, alongside other CC algorithms supplied by the user. This can be useful for fast comparative runs between the different algorithms. Each algorithm can run on a different device port using `doca_pcc_dev_init_algo_slot()`.

The algorithm can supply its own identifier, initiate its parameter (using `doca_pcc_dev_algo_init_param()`), counter (using `doca_pcc_dev_algo_init_counter()`), and metadata base (using `doca_pcc_dev_algo_init_metadata()`).

**Events**

The RP device library API provides a set of optimized CC event access functions. These functions serve as helpers to build the CC algorithm and to provide runtime data to analyze and inspect CC events arriving on hardware.

**Utilities**

The device library APIs provide a set of optimized utility macros that are set to support programming the CC algorithm. Such utilities are composed of fixed-point operations, memory space fences, and more.

**User Callbacks**

The device libraries API consists of specific user callbacks used by the library to initiate and run the CC algorithm and handle input and output packets. These callbacks must be
implemented by the user and, to be part of the DPA application, compiled by DPACC to provide to the DOCA PCC context.

The set of callbacks to be implemented for RP:

- `doca_pcc_dev_user_init()` – called on PCC process load and should initialize the data of all user algorithms
- `doca_pcc_dev_user_algo()` – entry point to the user algorithm handling code
- `doca_pcc_dev_user_set_algo_params()` – called when the parameter change is set externally

The set of callbacks to be implemented for NP:

- `doca_pcc_dev_np_user_packet_handler()` – called on probe packets arrival
DOCA DMA

This guide provides instructions on building and developing applications that require copying memory using Direct Memory Access (DMA).

Introduction

DOCA DMA provides an API to copy data between DOCA buffers using hardware acceleration, supporting both local and remote memory regions.

The library provides an API for executing DMA operations on DOCA buffers, where these buffers reside either in local memory (i.e., within the same host) or host memory accessible by the DPU. See DOCA Core for more information about the memory subsystem.

Using DOCA DMA, complex memory copy operations can be easily executed in an optimized, hardware-accelerated manner.

This document is intended for software developers wishing to accelerate their application's memory I/O operations and access memory that is not local to the host.

Prerequisites

This library follows the architecture of a DOCA Core Context, it is recommended read the following sections before:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

Library Changes From Previous Releases

Changes in 2.8.0
The following subsection(s) detail the `doca_comch` library updates in version 2.8.0.

**API Additions**

- `doca_error_t doca_dma_get_gpu_handle(struct doca_dma *dma, struct doca_gpu_dma **gpu_dma)`
  - Provides the option to export DMA to GPU and use GPUNetIO for DMA datapath on the GPU

**Environment**

DOCA DMA-based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target.

Copying from Host to DPU and vice versa only works with a DPU configured running in DPU mode as described in [NVIDIA BlueField Modes of Operation](#).

**Architecture**

DOCA DMA is a DOCA Context as defined by DOCA Core. See [DOCA Core Context](#) for more information.

DOCA DMA leverages DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

DMA can be used to copy data as follows:

- Copying from local memory to local memory:
• Using DPU to copy memory between host and DPU:

• Using host to copy memory between host and DPU:
Objects

Device and Device Representor

The DMA library needs a DOCA device to operate. The device is used to access memory and perform the actual copy. See DOCA Core Device Discovery.

For same BlueField DPU, it does not matter which device is used (PF/VF/SF), as all these devices utilize the same hardware component. If there are multiple DPUs, then it is possible to create a DMA instance per DPU, providing each instance with a device from a different DPU.

To access memory that is not local (from the host to the DPU or vice versa), the DPU side of the application must select a device with an appropriate representor. See DOCA Core Device Representor Discovery.

The device must stay valid for as long as the DMA instance is not destroyed.

Memory Buffers

The memory copy task requires two DOCA buffers containing the destination and the source. Depending on the allocation pattern of the buffers, refer to the table in the "Inventory Types" section. To find what kind of memory is supported, refer to the table in section "Buffer Support".

Buffers must not be modified or read during the memory copy operation.
Configuration Phase

To start using the library, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

Configurations

The context can be configured to match the application use case.

To find if a configuration is supported, or what the min/max value for it is, refer to section "Device Support".

Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of tasks and/or events in sections "Tasks" and "Events" respectively for information.
- A device with appropriate support must be provided upon creation

Device Support

DOCA DMA requires a device to operate. To picking a device, refer to "DOCA Core Device Discovery".

As device capabilities may change (see DOCA Core Device Support), it is recommended to select your device using the following method:

- doca_dma_cap_task_memcpy_is_supported
Some devices can allow different capabilities as follows:

- The maximum number of tasks
- The maximum buffer size

## Buffer Support

Tasks support buffers with the following features:

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Source Buffer</th>
<th>Destination Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local mmap buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>mmap from PCIe export buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>mmap From RDMA export buffer</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Linked list buffer</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

## Execution Phase

This section describes execution on CPU using **DOCA Core Progress Engine**.

## Tasks

DOCA DMA exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See **DOCA Core Task**.

## Memory Copy Task

The memory copy task allows copying memory from one location to another. Using buffers as described in **Buffer Support**.

## Task Configuration
### Description

<table>
<thead>
<tr>
<th>Description</th>
<th>API to set the configuration</th>
<th>API to query support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_dma_task_memcpy_set_config</code></td>
<td><code>doca_dma_cap_task_memcpy_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_dma_task_memcpy_set_config</code></td>
<td><code>doca_dma_cap_get_max_num_tasks</code></td>
</tr>
<tr>
<td>Maximal buffer size</td>
<td>-</td>
<td><code>doca_dma_cap_task_memcpy_get_max_buf_size</code></td>
</tr>
<tr>
<td>Maximum buffer list size</td>
<td>-</td>
<td><code>doca_dma_cap_task_memcpy_get_max_buf_list_len</code></td>
</tr>
</tbody>
</table>

### Task Input

Common input as described in **DOCA Core Task**.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source buffer</td>
<td>Buffer that points to the memory to be copied</td>
<td>Only the data residing in the data segment is copied</td>
</tr>
<tr>
<td>Destination buffer</td>
<td>Buffer that points to where memory is copied</td>
<td>The data is copied to the tail segment extending the data segment</td>
</tr>
</tbody>
</table>

### Task Output

Common output as described in **DOCA Core Task**.

### Task Completion Success

After the task is completed successfully:

- The data is copied from source to destination
- The destination buffer data segment is extended to include the copied data
**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state, if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

**Task Limitations**

- The operation is not atomic
- Once the task has been submitted, then the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in [DOCA Core Task](#)

**Events**

DOCA DMA exposes asynchronous events to notify on changes that happen unexpectedly, according to [DOCA Core architecture](#).

The only event DMA exposes is common events as described in [DOCA Core Event](#).

**State Machine**
The DOCA DMA library follows the Context state machine as described in DOCA Core Context State Machine.

The following section describes how to move states and what is allowed in each state.

**Idle**

In this state it is expected that application:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all tasks have been freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all tasks are completed and freed</td>
</tr>
</tbody>
</table>

**Starting**

This state cannot be reached.

**Running**

In this state it is expected that application:

- Allocates and submits tasks
• Calls progress to complete tasks and/or receive events

Allowed operations:

• Allocating a previously configured task
• Submitting a task
• Calling stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

**Stopping**

In this state it is expected that application:

• Calls progress to complete all inflight tasks (tasks complete with failure)
• Frees any completed tasks

Allowed operations:

• Call progress

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Call progress and fatal error occurs</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop without freeing all tasks</td>
</tr>
</tbody>
</table>

**Alternative Datapath Options**
DOCA DMA allows data path to be run on the CPU or GPU.

For the CPU data path, see Execution Phase.

**GPU Datapath**

DOCA offers the DOCA GPUNetIO library which provides a programming model for offloading the orchestration of the communication to a GPU CUDA kernel.

The user may run a DMA operation on the GPU data path by configuring the DOCA DMA context used by the application in the following manner:

1. Obtain DOCA CTX by calling `doca_dma_as_ctx()`.
2. Set the datapath of the context to GPU by calling `doca_ctx_set_datapath_on_gpu()`. For additional information, refer to DOCA Core Alternative Data Path.
3. Finish context configuration and start the context by calling `doca_ctx_start()`. For additional information, refer to DOCA Core Context.

After configuring the datapath, the user can obtain a GPU handle for the DOCA RDMA context by calling `doca_dma_get_gpu_handle()`. The GPU handle must be passed to a GPU CUDA kernel so the DOCA GPUNetIO CUDA device functions can execute datapath operations. For additional information, refer to section "GPU Functions – RDMA" under DOCA GPUNetIO library documentation.

**DOCA DMA Samples**

This section describes DOCA DMA samples based on the DOCA DMA library.

The samples illustrate how to use the DOCA DMA API to do the following:

- Copy contents of a local buffer to another buffer
• Use DPU to copy contents of buffer on the host to a local buffer

**Info**

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

## Running the Samples

1. Refer to the following documents:
   - [NVIDIA DOCA Installation Guide for Linux](#) for details on how to install BlueField-related software.
   - [NVIDIA DOCA Troubleshooting Guide](#) for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_dma/dma_local_copy
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary `doca_dma_local_copy` is created under `/tmp/build/`

3. Sample (e.g., `doca_dma_local_copy`) usage:

   ```
   Usage: doca_<sample_name> [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>
   -j, --json <path> Parse all command flags from an input json file
   ```
4. For additional information per sample, use the \texttt{-h} option:

```
/tmp/build/<sample_name> -h
```

## Samples

### DMA Local Copy

This sample illustrates how to locally copy memory with DMA from one buffer to another on the DPU. This sample should be run on the DPU.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Populating DOCA memory map with two relevant buffers.
4. Allocating element in DOCA buffer inventory for each buffer.
5. Initializing DOCA DMA memory copy task object.
6. Submitting DMA task.
7. Handling task completion once it is done.
8. Checking task result.
9. Destroying all DMA and DOCA core structures.

Reference:
This sample illustrates how to copy memory (which contains user defined text) with DMA from the x86 host into the DPU. This sample should be run on the DPU.

The sample logic includes:

1. Locating DOCA device.

2. Initializing needed DOCA core structures.

3. Reading configuration files and saving their content into local buffers.

4. Allocating the local destination buffer in which the host text is to be saved.

5. Populating DOCA memory map with destination buffer.

6. Creating the remote memory map with the export descriptor file.

7. Creating memory map to the remote buffer.

8. Allocating element in DOCA buffer inventory for each buffer.

9. Initializing DOCA DMA memory copy task object.

Note

This sample should run only after DMA Copy Host is run and the required configuration files (descriptor and buffer) have been copied to the DPU.
10. Submitting DMA task.

11. Handling task completion once it is done.

12. Checking DMA task result.

13. If the DMA task ends successfully, printing the text that has been copied to log.

14. Printing to log that the host-side sample can be closed.

15. Destroying all DMA and DOCA core structures.

Reference:

- /opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/dma_copy_dpu_sample.c
- /opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/dma_copy_dpu_main.c
- /opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/meson.build

**DMA Copy Host**

⚠️ Note

This sample should be run first. It is user responsibility to transfer the two configuration files (descriptor and buffer) to the DPU and provide their path to the DMA Copy DPU sample.

This sample illustrates how to allow memory copy with DMA from the x86 host into the DPU. This sample should be run on the host.

The sample logic includes:

1. Locating DOCA device.

2. Initializing needed DOCA core structures.
3. Populating DOCA memory map with source buffer.

4. Exporting memory map.

5. Saving export descriptor and local DMA buffer information into files. These files should be transferred to the DPU before running the DPU sample.

6. Waiting until DPU DMA sample has finished.

7. Destroying all DMA and DOCA core structures.

Reference:

- /opt/mellanox/doca/samples/doca_dma/dma_copy_host/dma_copy_host_sample.c
- /opt/mellanox/doca/samples/doca_dma/dma_copy_host/dma_copy_host_main.c
- /opt/mellanox/doca/samples/doca_dma/dma_copy_host/meson.build
DOCA Comch

**Info**

DOCA Comm Channel API will be deprecated in the next DOCA release (2.9.0).

DOCA Comch API introduces features such as high-performance data path over the consumer-producer API, as well as working with DOCA progress engine and other standard DOCA Core objects.

**Note**

DOCA Comch does not support event-triggered completions.

DOCA Comch – New

This guide provides instructions on building and developing applications that require communication channels between the x86 host and the BlueField Arm cores.

**Introduction**

DOCA Comch provides a communication channel between client applications on the host and servers on the BlueField Arm.

Benefits of using DOCA Comch:

- Security – the communication channel is isolated from the network
• Network independent – the state of the communication channel does not depend on the state and configuration of the network

• Ease of use

DOCA Comch provides two different data path APIs:

• Basic DOCA Comch send/receive for control messages

• High bandwidth, low latency, zero-copy, multi-producer, multi-consumer API

The following table summarizes the differences between the two data path APIs:

<table>
<thead>
<tr>
<th>Features</th>
<th>Basic Send/Receive</th>
<th>Fast Path (using doca_comch_consumer/doca_comch_producer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero-copy</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Takes network bandwidth</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Isolated from network</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Max msg size</td>
<td>Fixed</td>
<td>1GB or more (depends on hardware cap)</td>
</tr>
<tr>
<td>Multi-threaded</td>
<td>Safe for a single thread</td>
<td>Allows creation of consumer/producers per thread.</td>
</tr>
<tr>
<td>Multi-consumer</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Multi-producer</td>
<td>Yes – allows multiple clients per server</td>
<td>Yes – allow multiple producers/consumers per connection</td>
</tr>
<tr>
<td>Requires doca_mmap and doca_buf</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Prerequisites**

This library follows the architecture of a DOCA Core Context, it is recommended to read the following sections before:

• [DOCA Core Execution Model](#)
• **DOCA Core Device**

• **DOCA Core Memory Subsystem** (fast path only)

# Changes From Previous Release

## Modified

**Function name and return type changes**

- `doca_error_t doca_comch_server_get_device_rep(const struct doca_comch_server *comch_server, struct doca_dev_rep **rep)`
  
  - **DOCA 2.7 version:**
    
    ```c
    doca_error_t doca_comch_server_get_device_repr(const struct doca_comch_server *comch_server, struct doca_dev_rep **repr)
    ```

- `doca_comch_server_event_connection_status_changed_register(server, server_connection_status_callback, server_connection_status_callback)`
  
  - **DOCA 2.7 version:**
    
    ```c
    doca_comch_server_event_connection_register(server, server_connection_status_callback, server_connection_status_callback)
    ```

- `doca_error_t doca_comch_consumer_set_dev_max_num_recv(struct doca_comch_consumer *consumer, uint32_t dev_num_recv)`
  
  - **DOCA 2.7 version:**
    
    ```c
    doca_comch_consumer_set_dev_num_recv(struct doca_comch_consumer *consumer, uint32_t dev_num_recv)
    ```

- `doca_error_t doca_comch_producer_set_dev_max_num_send(struct doca_comch_producer *producer, uint32_t dev_num_send)`
  
  - **DOCA 2.7 version:**
    
    ```c
    doca_comch_producer_set_dev_num_send(struct doca_comch_producer *producer, uint32_t dev_num_send)
    ```

- `doca_error_t doca_comch_consumer_completion_get_max_num_consumers(const struct doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)`
  
  - **DOCA 2.7 version:**
    
    ```c
    doca_error_t doca_comch_consumer_completion_get_max_num_consumers(const struct doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)
    ```
- DOCA 2.7 version:
  - `doca_comch_consumer_completion_get_max_num_consumers(struct
doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)`
  - `doca_error_t doca_comch_consumer_completion_get_max_num_consumers(const struct
doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)`

- DOCA 2.7 version:
  - `doca_comch_consumer_completion_get_max_num_recv(struct
doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_recv)`
  - `doca_error_t doca_comch_consumer_completion_get_max_num_recv(const struct
doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_recv)`

-- Adding const to getter API functions

- `doca_comch_consumer_task_post_recv_get_buf(const struct doca_comch_consumer_task_post_recv *task)`
  - DOCA 2.7 version:
    - `doca_comch_consumer_task_post_recv_get_buf(struct doca_comch_consumer_task_post_recv *task)`
  - `doca_comch_consumer_task_post_recv_get_producer_id(const struct
doca_comch_consumer_task_post_recv *task)`
    - DOCA 2.7 version:
      - `doca_comch_consumer_task_post_recv_get_producer_id(struct doca_comch_consumer_task_post_recv *task)`
  - `const uint8_t *doca_comch_consumer_task_post_recv_get_imm_data(const struct
doca_comch_consumer_task_post_recv *task)`
    - DOCA 2.7 version:
- uint8_t *doca_comch_consumer_task_post_recv_get_imm_data(struct doca_comch_consumer_task_post_recv *task)

- doca_comch_consumer_task_post_recv_get_imm_data_len(const struct doca_comch_consumer_task_post_recv *task)

  **DOCA 2.7 version:**

  - doca_comch_consumer_task_post_recv_get_imm_data_len(struct doca_comch_consumer_task_post_recv *task)

- doca_comch_producer_task_send_get_buf(const struct doca_comch_producer_task_send *task)

  **DOCA 2.7 version:**

  - doca_comch_producer_task_send_get_buf(struct doca_comch_producer_task_send *task)

- doca_comch_producer_task_send_get_consumer_id(const struct doca_comch_producer_task_send *task)

  **DOCA 2.7 version:**

  - doca_comch_producer_task_send_get_consumer_id(struct doca_comch_producer_task_send *task)

- doca_comch_producer_task_send_get_imm_data(const struct doca_comch_producer_task_send *task)

  **DOCA 2.7 version:**

  - doca_comch_producer_task_send_get_imm_data(struct doca_comch_producer_task_send *task)

- doca_comch_producer_task_send_get_imm_data_len(const struct doca_comch_producer_task_send *task)

  **DOCA 2.7 version:**

  - doca_comch_producer_task_send_get_imm_data_len(struct doca_comch_producer_task_send *task)

**Environment**
DOCA Comch based applications can run either on the host machine or on the NVIDIA BlueField Arm.

Sending messages between the host and BlueField Arm can only be run with a BlueField configured with a mode as described in NVIDIA BlueField Modes of Operation.

For basic DOCA Comch send and receive, the following configuration is required:

- `doca_comch_server` context must run on the BlueField Arm cores
- `doca_comch_client` context must run on the host machine

**Note**

Producer and consumer objects can run on both the host and BlueField Arm cores. However, there must be a valid client/server connection already established on the channel.

**Architecture**

DOCA Comch is comprised of four DOCA Core Contexts. All DOCA Comch contexts leverage DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

A `doca_comch_server` context runs on the BlueField Arm and listens for incoming connections from the host side. Such host side connections are initiated by a `doca_comch_client` context.

Servers can receive connections from multiple clients in parallel, however, a client can only connect with one server. An established 1-to-1 connection between a client and a server is represented by a `doca_comch_connection`.

Once an established connection exists between a client and a server, the `doca_comch_producer` and `doca_comch_consumer` contexts can be used to run fast path channels.

The following diagram provides examples of the contexts use:
## Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Location</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>doca_comch_server</code></td>
<td>Allows applications on the BlueField Arm cores to listen on a specific server name and accept new incoming connection from the host</td>
<td>BlueField Arm only</td>
<td>Per host PCIe function (doca_dev + doca_dev_rep)</td>
</tr>
<tr>
<td><code>doca_comch_client</code></td>
<td>Allows client applications to connect to a specific server name on the BlueField Arm cores</td>
<td>Host only</td>
<td>Per host PCIe function (doca_dev)</td>
</tr>
<tr>
<td><code>doca_comch_connection</code></td>
<td>A connection handle created on the client side or the server side when a new connection is established. This handle is used to send/receive messages or to create <code>doca_comch_consumer</code>s and <code>doca_comch_producer</code>s.</td>
<td>BlueField Arm and host</td>
<td>Per client server pair</td>
</tr>
<tr>
<td>Description</td>
<td>Location</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------</td>
<td>--------------------------------</td>
<td></td>
</tr>
<tr>
<td>A handle for a FIFO-like send queue that provides a zero-copy API to send</td>
<td>BlueField</td>
<td>Per doca_comch_connection</td>
<td></td>
</tr>
<tr>
<td>messages to a specific doca_comch_consumer on the same doca_comch_connection.</td>
<td>Arm and host</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple doca_comch_producers can be created per doca_comch_connection.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A handle for a FIFO-like receive queue that provides a zero-copy API to</td>
<td>BlueField</td>
<td>Per doca_comch_connection</td>
<td></td>
</tr>
<tr>
<td>receive messages from a doca_comch_producer</td>
<td>Arm and host</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Security Considerations**

- DOCA Comch guarantees:
  - The client is connected to the server by providing the exact server name on the client side
  - Only clients on the PF/VF/SF represented by the doca_dev_rep provided upon server creation can connect to the server
  - The connection requests and data path are isolated from the network

- DOCA Comch does not provide security at the application level:
  - It is up to the user to implement application-level security and verify the identity of the client application
  - A server handles applications from a single PF/VF/SF. If a server application detects a compromised client application, the server app should consider all clients (from that PF/VF/SF) compromised.

**Initialization Flow**
**doca_comch_server Initialization Flow**

1. A `doca_comch_server` is created on a specific `doca_dev` and a specific `doca_dev_rep`.

2. A `doca_comch_server` must have a unique name per `doca_dev/doca_dev_rep` (i.e., two servers on the same `doca_dev` and `doca_dev_rep` cannot have the same name).

3. Once `doca_ctx_start()` is called, the `doca_comch_server` can start receiving new connection requests.

4. For the `doca_comch_server` to process new connection requests and messages, the user must periodically call `doca_pe_progress()`.

5. When a new connection request arrives, `doca_comch_server` calls the connection request handler function and passes a `doca_comch_connection` object.

   The server can now send and receive messages on the connection represented by `doca_comch_connection`.

**doca_comch_client Initialization Flow**

1. A `doca_comch_client` is created on a specific `doca_dev` is targeting a specific `doca_comch_server`.

2. Once `doca_ctx_start()` is called, `doca_comch_client` asynchronously tries to connect to the server.

3. To establish the connection and receive messages, the user must periodically call `doca_pe_progress()`.

4. When the connection is established, `doca_comch_client` calls the state change callback indicating state change to "RUNNING".

   The client can now send a receive messages.

The following diagram describes the initialization of a basic client/server connection on DOCA Comch:
**doca_comch_consumer Initialization Flow**

1. A `doca_comch_consumer` is created on a specific `doca_comch_connection`.

2. `doca_pe_progress()` must be periodically called on the client/server PE to allow registration of the consumer.
3. After the `doca_comch_consumer` moves to "RUNNING" state:

   1. `doca_comch_consumer` notifies its existence to the peer (invoking a new consumer event).

   2. The application can start posting receive tasks.

   3. A `doca_comch_producer` on the peer side can start sending messages to that consumer.

The initialization flow is described in the following diagram:

---

**Teardown Flow**

The teardown flow must be executed in the following order, otherwise errors may occur.
Disconnecting Specific Connection

The proper disconnection process for a specific connection consists of the following steps:

1. Stop all consumers and producers linked to the connection.

2. Server/client:

   1. For server, a connection can be disconnected using `doca_comch_server_disconnect()`. If there are any active producers/consumers linked to the connection, the disconnect would fail. A disconnection notifies the client and initiates teardown on that side too.

   2. For client, since there is only one connection at any given time, the connection can be disconnected by calling `doca_ctx_stop()`. If there are any active producers/consumers, the command would fail. Stopping the client context notifies the server of the disconnection and causes a disconnection of the connection on it.

Tearing Down DOCA Comch

The proper teardown for a DOCA Comch context consists of the following:

1. Stop all consumers and producers linked to the context.

2. Call `doca_ctx_stop()`. If there are any active connections, they would all be disconnected. If there are any active consumers/producers, the command would fail. Disconnecting/stopping the context informs all active peers of the disconnection, and causes teardown (on clients) or disconnection (on server). Calling `doca_ctx_stop()` successfully moves the context to "stopping" state.

3. After moving to stopping state, `doca_pe_progress()` must be called until the context moves to idle state.

MsgQ (DPA Communication)
DOCA Comch MsgQ leverages the existing consumer/producer model to allow communication between host/BlueField and DPA.

Since communication between the host/BlueField and DPA is local, there is no need to create a server, client, or connection. Instead the user can create a MsgQ and use it to create producers and consumers directly.

When creating a consumer/producer using the MsgQ, it becomes possible to use them in the DPA application as well as the CPU application:

- The CPU application can utilize existing consumer/producer APIs for communication
- The DPA application has a different set of APIs that are usable within a DPA application
Communication Direction

Every instance of a MsgQ can only support a single communication direction as follows:

- Communication from host/BlueField to DPA
  - This direction may be specified using `doca_comch_msgq_set_dpa_consumer`
  - Consumers created from this MsgQ are referred to as DPA consumers, while producers are CPU producers

- Communication from DPA to host/BlueField
  - This direction may be specified using `doca_comch_msgq_set_dpa_producer`
  - Consumers created from this MsgQ are referred to as CPU consumers, while producers are DPA producers

To support bidirectional communication in an application, the user has to create 2 MsgQ instances, as shown in the above diagram.

Configuration Phase

To start using the library, users must go through a configuration phase as described in the DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

Configurations

The context can be configured to match the application use case.

To find out if a certain configuration is supported, or what the min/max value for it is, refer to Device Support.

Mandatory Configurations
These configurations are mandatory and must be set by the application before attempting to start the context:

- For a basic send/receive client or server:
  - A send task callback
  - A receive event callback
  - A device with appropriate support must be provided on creation
  - A valid server name must be provided on creation (for clients this is the server to connect to)
  - A connection event callback (server only)

- For fast path producer or consumer:
  - A device with appropriate support must be provided on creation
  - An established client to server connection must be provided on creation
  - A `doca_mmap` with PCIe read/write permissions of where data should be received must be provided on creation (consumer only)
  - A post receive task callback (consumer only)
  - A send task callback (producer only)
  - A new consumer callback (triggered upon creation/destruction of a remove consumer)

- For MsgQ fast path producer or consumer:
  - A started MsgQ must be provided on creation
  - A DPA instance must be provided (DPA consumer/producer only)
  - A DPA consumer completion context must be connected (DPA consumer only)
  - A DPA completion context must be attached (DPA producer only)
• A post receive task callback (CPU consumer only)
• The number of receive operations (DPA consumer only)
• A send task callback (CPU producer only)
• The number of send operations (DPA producer only)

**Optional Configurations**

The following configurations are optional, if they are not set then a default value will be used:

For basic send/receive client:

• `doca_comch_(server|client)_set_max_msg_size` – set the maximum size of message that can be sent. If set, it must be matching between server and client.

• `doca_comch_(server|client)_set_recv_queue_size` – set the size of the queue to receive new messages on

For fast path consumers:

• `doca_comch_consumer_set_imm_data_len` – set the length of immediate data that a consumer can receive.

**Device Support**

DOCA Comch requires a device to operate. For instructions on picking a device, see [DOCA Core Device Discovery](#).

As device capabilities are subject to change (see [DOCA Core Device Support](#)), it is recommended to select a device using the following methods:

• For basic client and server:
  • `doca_comch_cap_server_is_supported`
- `doca_comch_cap_client_is_supported`

- For extended fast path functionality:
  - `doca_comch_producer_cap_is_supported`
  - `doca_comch_consumer_cap_is_supported`

Some devices can allow different capabilities as follows:

- The maximum length server name
- The maximum message size
- The maximum receive queue length
- The maximum clients that can connect to a server
- The maximum number of send tasks or post receive tasks
- The maximum buffer length for fast path
- The maximum immediate data supported by a fast path consumer

**Buffer Support**

Basic send and receive between a client and server does not use DOCA buffers and so has no restrictions on buffer type.

- For producers, supplied buffers need only be from a local mmap
- For consumers, post receive buffers are required to be from a PCIe export mmap

**Note**

Chained buffers are not supported in DOCA Comch.
Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine. For additional execution environments, refer to section "Alternative Datapath Options".

Tasks

DOCA Comch exposes asynchronous tasks that leverage the BlueField hardware according to DOCA Core architecture.

Control Channel Send Task

This task allows the sending of messages between connected client and server objects.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tasks</td>
<td>doca_comch_server_task_send_set_conf</td>
<td></td>
</tr>
<tr>
<td></td>
<td>doca_comch_client_task_send_set_conf</td>
<td>doca_comch_cap_get_max_send_tasks</td>
</tr>
<tr>
<td>Maximal message size</td>
<td>doca_comch_server_set_max_msg_size</td>
<td>doca_comch_server_get_max_msg_size</td>
</tr>
<tr>
<td></td>
<td>doca_comch_client_set_max_msg_size</td>
<td>doca_comch_client_get_max_msg_size</td>
</tr>
</tbody>
</table>

Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer</td>
<td>Established client/server connection</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>Data string to send to remote client/server</td>
<td>The is no requirement for the message to be in DOCA mmap registered memory</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>Length</td>
<td>Number of bytes in the message</td>
<td>Must not exceed configured max size</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in [DOCA Core Task](#).

**Task Completion Success**

After the task completes successfully:

- The message is delivered to the connections remote client/server
- A receive event is triggered on the remote side

**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The message is not delivered to the remote side

**Task Limitations**

- The operation is not atomic
- Once the task has been submitted, then the message should not be updated
- Other limitations are described in [DOCA Core Task](#)
Consumer Post Receive Task

This task allows consumer objects to publish buffers which are available for remote producers to write to.

Tip

A Post Receive task may have a NULL buffer if it only wishes to receive immediate data.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_comch_consumer_task_post_recv_set_conf</td>
<td>doca_comch_consumer_cap_is_supported</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_comch_consumer_task_post_recv_set_conf</td>
<td>doca_comch_consumer_cap_get_max_num_tasks</td>
</tr>
<tr>
<td>Maximal buffer size</td>
<td>_</td>
<td>doca_comch_consumer_cap_get_max_buf_size</td>
</tr>
</tbody>
</table>

Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffer that the consumer can receive data on</td>
<td>Data is appended to the tail of the buffer</td>
</tr>
</tbody>
</table>

Info
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Buffers <code>doca_mmap</code> must have <code>DOCA_ACCESS_FLAG_PCI_READ_WRITE</code> flag set.</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in [DOCA Core Task](#).

**Task Completion Success**

The task only completes once a producer has written to the advertised buffer (or immediate data, or both), not when the post receive has completed.

Upon successful completion, the buffer contains the data written by the producer and its length is updated appropriately.

**Task Completion Failure**

Task failure occurs if a buffer has not been successfully posted to receive data.

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- Producers are not aware of the buffer so would not write to it

**Task Limitations**

- The operation is not atomic
• Once the task has been submitted, the buffer should not be read/written to
• Buffer must come from memory with PCIe read/write access
• Chained buffer lists are not supported
• MsgQ consumer does not support providing `doca_buf`, and can only receive immediate data
• Other limitations are described in DOCA Core Task

**Producer Send Task**

This task allows producer objects to copy buffers for use by remote consumers.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_comch_producer_task_send_set_conf</code></td>
<td><code>doca_comch_producer_cap_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_comch_producer_task_send_set_conf</code></td>
<td><code>doca_comch_producer_cap_get_max_num_tasks</code></td>
</tr>
<tr>
<td>Maximal buffer Size</td>
<td>_</td>
<td><code>doca_comch_producer_cap_get_max_buf_size</code></td>
</tr>
</tbody>
</table>

**Task Input**

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffer that should be copied to a consumer</td>
<td>Only the data residing in the data segment is copied</td>
</tr>
<tr>
<td>Immediate data</td>
<td>Short byte array to add to the post receive completion entry</td>
<td>This is not a zero copy operation but does improve latency for small payloads</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Immediate data length</td>
<td>Length of data immediate data pointed to</td>
<td>Maximum length is determined/set by individual consumers</td>
</tr>
<tr>
<td>Consumer ID</td>
<td>Identifier for the target consumer to write to</td>
<td>Active consumers and their IDs are advertised through consumer events</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in DOCA Core Task.

**Task Completion Success**

After the task is completed successfully:

- The data is copied from the buffer to the next free buffer posted by the given consumer
- Consumers process buffers from a given consumer in the order they are sent

**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination doca_buf objects are not modified
- The destination memory may be modified

**Task Limitations**

- The operation is not atomic
• Once the task has been submitted, the buffer should not be read/written to

• The buffer length should not be greater than consumer post receive buffers (an invalid value is returned otherwise)

• MsgQ producer does not support providing doca_buf, and can only send immediate data

• All limitations described in DOCA Core Task

Events

DOCA Comch exposes asynchronous events to notify about changes that happen out of the blue, according to the DOCA Core architecture. See DOCA Core Event.

Common events as described in DOCA Core Event.

Control Channel Receive Event

This event triggers whenever a remote client/server has sent a message to the local client/server object.

Event Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event</td>
<td>doca_comch_server_event_msg_recv_register</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>doca_comch_client_event_msg_recv_register</td>
<td></td>
</tr>
</tbody>
</table>

Event Trigger Condition

The event is triggered when a remote message is received on any currently active connection associated with the client or server.
Event Output

Upon event detection, the registered callback is triggered, passing the following parameters:

- A pointer to the message data
- The length in bytes of the message
- The active connection on which the message was received

Info

The data is only valid in the context of the callback.

Connection Status Changed Event (Server Only)

This event provides asynchronous updates on the state of any connections associated with a server.

Note

A client object can only connect to a single server, so its connection state can be tracked through its `doca_ctx` state and the generic `doca_ctx_set_state_changed_cb` function.

Event Configuration
Event Trigger Condition

The event is triggered when a new connection is either established or a current connection disconnected on a server.

Event Output

Separate callbacks are registered for connection or disconnection events with the appropriate one triggered based on the specific event.

Both callbacks contain a Boolean indicating if the connection or disconnection was successful.

Consumer Event

This event indicates that a new consumer object has been created or an existing consumer object has been destroyed.

Event Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event</td>
<td>doca_comch_server_event_consumer_register</td>
<td></td>
</tr>
<tr>
<td></td>
<td>doca_comch_client_event_consumer_register</td>
<td></td>
</tr>
</tbody>
</table>

Event Trigger Condition
The event is triggered whenever a new consumer is created or a current consumer destroyed on the remote side of an established DOCA Comch connection.

**Event Output**

The event hits a separate callback for either the creation or destruction of a consumer.

Callback parameters include:

- The established DOCA Comch connection on which the consumer is connected (on the remote side)
- The ID of the consumer (a unique value per Comch connection)

**State Machine**

The DOCA Comch library follows the Context state machine described in DOCA Core Context State Machine.

The following section describes how to move to the state and what is allowed in each state.

**Idle**

In this state it is expected that the application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context
It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all tasks have been freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all tasks are completed and freed</td>
</tr>
</tbody>
</table>

### Starting

In this state it is expected that the application will:

- Call progress to allow transition to next state (e.g., when a connection attempt completes)

Allowed operations:

- Call progress

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

### Running

In this state, it is expected that the application:

- Allocates and submit tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocate a previously configured task
- Submit an allocated task
- Call stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
<tr>
<td>Starting</td>
<td>Call progress until context state transitions</td>
</tr>
</tbody>
</table>

**Stopping**

In this state, it is expected that the application will:

- Free any completed tasks

Allowed operations:

- Allocate previously configured task
- Submit an allocated task
- Call stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Call progress and fatal error occurs</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop without freeing all tasks</td>
</tr>
</tbody>
</table>

**Alternative Datapath Options**

DOCA Comch can be run on as part of DPA data path, using the MsgQ.
DPA

Using the `MsgQ` it is possible to create consumer/producer on the DPA. They follow the definition described in DOCA Core DPA.

Since these objects can be used in DPA, they have DPA APIs that can be used to perform the data path operations expanded on in the following subsections.

**Consumer Ack**

The `doca_dpa_dev_comch_consumer_ack` API prepares the DPA consumer to receive a number of immediate messages from CPU producers.

**Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue Size</td>
<td><code>doca_comch_consumer_set_dev_max_num_recv</code></td>
<td>–</td>
</tr>
</tbody>
</table>

**Input**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Messages</td>
<td>A number describing how many additional immediate messages this consumer can receive</td>
<td>Must not exceed the queue size</td>
</tr>
</tbody>
</table>

**Completion**

Whenever a message is received from the CPU producer a completion element is generated and can be polled using `doca_dpa_dev_comch_consumer_get_completion`.

Using the generated completion, it is possible to get the following outputs:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>A pointer to the immediate message</td>
<td>The message lifetime is the same as the completion element lifetime. That is, once the completion is</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Message</td>
<td>that the CPU producer sent</td>
<td>acked using doca_dpa_dev_comch_consumer_completion_ack, the pointer is no longer valid. To retain the message past the completion lifetime, the user must copy the contents of the message.</td>
</tr>
<tr>
<td>Immediate Message Length</td>
<td>The length in bytes of the immediate message that the CPU producer sent</td>
<td></td>
</tr>
<tr>
<td>Producer ID</td>
<td>The ID of the CPU producer that sent the message</td>
<td>User can find the IDs of each producer by using doca_comch_producer_get_id</td>
</tr>
</tbody>
</table>

**Limitations**

- The maximal immediate message size is 32 bytes

**Producer Post Send Immediate Only**

The `doca_dpa_dev_comch_producer_post_send_imm_only` API sends an immediate message to the CPU consumer. Once the message arrives at the CPU consumer side, the CPU consumer receive task completes.

The CPU producer must have posted a receive task prior to this. The user can verify if the consumer can receive the message using `doca_dpa_dev_comch_producer_is_consumer_empty`. Note, however, that this may add overhead.

**Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue Size</td>
<td><code>doca_comch_producer_set_dev_max_num_send</code></td>
<td>–</td>
</tr>
</tbody>
</table>
**Input**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Message</td>
<td>Short byte array to be sent to the CPU consumer</td>
<td>This is not a zero copy operation but does improve latency for small payloads</td>
</tr>
<tr>
<td>Immediate Message Length</td>
<td>Length of the message the immediate message points to</td>
<td>The maximum length is 32 bytes</td>
</tr>
<tr>
<td>Consumer ID</td>
<td>Identifier for the target CPU consumer to write to</td>
<td>User can find the IDs of each consumer by using <code>doca_comch_consumer_get_id</code></td>
</tr>
<tr>
<td>Completion Requested</td>
<td>Flag indicating whether to generate a completion once the send is completed</td>
<td>This refers to the DPA producer completion which is separate from the completion the CPU consumer receives</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 0 – no completion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 1 – otherwise</td>
</tr>
</tbody>
</table>

**Completion**

Once the message arrives to the CPU consumer, a completion element is generated, indicating that the send is complete (this is separate from the completion the CPU consumer receives) and can be polled using `doca_dpa_dev_get_completion`.

Using the generated completion, it is possible to get the following outputs:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer User Data</td>
<td>Producer user data provided during configuration of the producer</td>
<td>User data previously set using <code>doca_ctx_set_user_data</code> when configuring this producer. User data which is returned belongs to the DPA producer this completion has been generated for, and can be used to identify the specific producer.</td>
</tr>
</tbody>
</table>

**Limitations**
- The maximal immediate message size is 32 bytes

**Producer DMA Copy**

The `doca_dpa_dev_comch_producer_dma_copy` API performs a DMA copy operation and, once the copy operation is done, sends an immediate message to the CPU consumer. Once the message arrives at the CPU consumer side, the CPU consumer receive task completes.

The CPU producer must have posted a receive task prior to this. The user can verify if the consumer can receive the message using `doca_dpa_dev_comch_producer_is_consumer_empty`. Note, however, that this may add overhead.

**Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue Size</td>
<td><code>doca_comch_producer_set_dev_num_recv</code></td>
<td>–</td>
</tr>
</tbody>
</table>

**Input**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination Mmap</td>
<td>Mmap representing the memory to be used as the destination of the copy operation</td>
<td>This mmap must have <code>LOCAL_READ_WRITE</code> access enabled</td>
</tr>
<tr>
<td>Destination Address</td>
<td>The address to be used as the destination of the copy operation</td>
<td>The address and copy length must be within the range of the destination mmap's memory range</td>
</tr>
<tr>
<td>Source Mmap</td>
<td>Mmap representing the memory to be used as the source of the copy operation</td>
<td>This mmap must have <code>LOCAL_READ</code> access enabled</td>
</tr>
<tr>
<td>Source Address</td>
<td>The address to be used as the source of the copy operation</td>
<td>The address and copy length must be within the range of the source mmap's memory range</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Length</td>
<td>The length of the copy operation</td>
<td>Source and destination addresses must not overlap</td>
</tr>
<tr>
<td>Immediate Message</td>
<td>Short byte array to be sent to the CPU consumer once the copy operation is done</td>
<td>This is not a zero copy operation but does improve latency for small payloads</td>
</tr>
<tr>
<td>Immediate Message Length</td>
<td>Length of the message the immediate message points to</td>
<td>The maximum length is 32 bytes</td>
</tr>
<tr>
<td>Consumer ID</td>
<td>Identifier for the target CPU consumer to write to</td>
<td>User can find the IDs of each consumer using <code>doca_comch_consumer_get_id</code></td>
</tr>
<tr>
<td>Completion Requested</td>
<td>Flag indicating whether to generate a completion once the send is completed</td>
<td>This refers to the DPA producer completion which is separate from the completion the CPU consumer receives</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 – no completion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 – otherwise</td>
</tr>
</tbody>
</table>

**Completion**

Once copy is complete and the message arrives to the CPU consumer, a completion element is generated, indicating that the copy is complete (this is separate from the completion the CPU consumer receives) and can be polled using `doca_dpa_dev_get_completion`.

Using the generated completion, it is possible to get the following outputs:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer User Data</td>
<td>Producer user data provided during configuration of the producer</td>
<td>The user data set using <code>doca_ctx_set_user_data</code> when configuring this producer. The user data which is returned belongs to the DPA producer this completion has been generated for, and can be used to identify the specific producer.</td>
</tr>
</tbody>
</table>
Limitations

- The maximal immediate message size is 32 bytes

DOCA Comch Samples

This section describes DOCA Comch samples based on the DOCA Comch library.

The samples illustrate how to use the DOCA Comch API to do the following:

- Set up a client/server between host and BlueField Arm cores and use it to send text messages
- Configure fast path producers and consumers, and send messages between them

Info

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Running the Samples

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:
The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. All DOCA Comch samples accept the same input arguments:

<table>
<thead>
<tr>
<th>Sample</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>doca_comch_ctrl_path_server</td>
<td>-p, --pci-addr</td>
<td>DOCA Comch device PCIe address</td>
</tr>
<tr>
<td>doca_comch_ctrl_path_client</td>
<td>-r, --rep-pci</td>
<td>DOCA Comch device representor PCIe address (required only on BlueField Arm)</td>
</tr>
<tr>
<td>doca_comch_data_path_high_speed_server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>doca_comch_data_path_high_speed_client</td>
<td>-t, --text</td>
<td>Text to be sent to the other side of channel (overwrites default)</td>
</tr>
</tbody>
</table>

4. For additional information per sample, use the `-h` option:

```
/tmp/build/<sample_name> -h
```

**Samples**

**DOCA Comch Control Path Client/Server**

**Note**

`doca_comch_ctrl_path_server` must be run on the BlueField Arm side and started before `doca_comch_ctrl_path_client` is started on the host.
This sample sets up a client server connection between the host and BlueField Arm cores.

The connection is used to pass two messages, the first sent by the client when the connection is established and the second by the server on receipt of the client's message.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the core DOCA structures.
3. Initializing and configuring client/server contexts.
4. Registering tasks and events for sending/receiving messages and tracking connection changes.
5. Allocating and submitting tasks for sending control path messages.
6. Handling event completions for receiving messages.
7. Stopping and destroying client/server objects.

References:

- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_client/comch_ctrl_path_client_main.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_client/comch_ctrl_path_client_sample.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_server/comch_ctrl_path_server_main.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_server/comch_ctrl_path_server_sample.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_common.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_common.h

DOCA Comch Data Path Client/Server

Note
This sample sets up a client server connection between host and BlueField Arm.

The connection is used to create a producer and consumer on both sides and pass a message across the two fastpath connections.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the core DOCA structures.
3. Initializing and configuring client/server contexts.
4. Initializing and configuring producer/consumer contexts on top of an established connection.
5. Submitting post receive tasks for population by producers.
6. Submitting send tasks from producers to write to consumers.
7. Stopping and destroying producer/consumer objects.
8. Stopping and destroying client/server objects.

References:

- /opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_client/comch_data_path_high_client.c
- /opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speedserver/comch_data_path_high_server.c
- /opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_common.c
DOCA Comm Channel – Deprecated

This guide provides instructions on how to use the DOCA Comm Channel API.

**Introduction**

The DOCA Comm Channel (CC) provides a secure, network-independent communication channel between the host and the DPU.

The communication channel allows the host to control services on the DPU or to activate certain offloads.

The DOCA Comm Channel is reliable, message-based, and connecting multiple clients to a single service. The API allows communication between a client using any PF/VF/SF on the host to a service on the DPU.

**Prerequisites**

The CC service can only run on the DPU while the client can only run on a host connected to the DPU.

Refer to [NVIDIA DOCA Release Notes](#) for the supported versions of firmware, OS, and MLNX_OFED.

**API**

**Objects**

`struct doca_comm_channel_ep_t`

Represents a Comm Channel endpoint either on the client or service side. The endpoint is needed for every other Comm Channel API function.
**struct doca_comm_channel_addr_t**

Also referred to as peer_address, represents a connection and can be used to identify the source of a received message. It is required to send a message using \texttt{doca_comm_channel_ep_sendto()}. 

**Query Device Capabilities**

Querying the device capabilities allows users to know the derived Comm Channel limitation (see section Limitations for more information), and to set the properties of an endpoint accordingly.

The capabilities under this section, apart from maximal service name length, may vary between different devices. To select the device you wish to establish a connection upon, you may query each of the devices for its capabilities.

**doca_comm_channel_get_max_service_name_len()**

As each connection requires a name, users must know the maximal length of the name and may use this function to query it. This length includes the null-terminating character, and any name longer than this length is not accepted when trying to establish a connection with Comm Channel.

\begin{verbatim}
doca_error_t doca_comm_channel_get_max_service_name_len(uint32_t *max_service_name_len);
\end{verbatim}

- \texttt{max_service_name_len [out]} – pointer to a parameter that will hold the max service name length on success.
- Returns – \texttt{doca_error_t} value. DOCA_SUCCESS if successful, or an error value upon failure. Possible error values are documented in the header file.

**doca_comm_channel_get_max_message_size()**
Each connection has an upper limit for the messages size. This function returns the maximal value that can be set for this property, for a given device. This limitation is important when trying to set the max message size for an endpoint with `doca_comm_channel_ep_set_max_msg_size()`.

```c
doca_error_t doca_comm_channel_get_max_message_size(struct doca_devinfo *devinfo, uint32_t *max_message_size);
```

- `devinfo [in]` – pointer to a `doca_devinfo` which should be queried for this capability.
- `max_message_size [out]` – pointer to a parameter that on success holds the maximal value that can be set for max message size when communicating on the provided `devinfo`.
- Returns – `doca_error_t` value. `DOCA_SUCCESS` if successful, or an error value upon failure. Possible error values are documented in the header file.

### `doca_comm_channel_get_max_send_queue_size()`

Returns the maximum send queue size that can be set for a given device. This value describes the maximum possible amount of outgoing in-flight messages for a connection. This limitation is important when trying to set the max message size for an endpoint with `doca_comm_channel_ep_set_send_queue_size()`.

```c
doca_error_t doca_comm_channel_get_max_send_queue_size(struct doca_devinfo *devinfo, uint32_t *max_send_queue_size);
```

- `devinfo [in]` – pointer to a `doca_devinfo` which should be queried for this capability.
- `max_send_queue_size [out]` – pointer to a parameter that on success, holds the maximal value that can be set for the send queue size when communicating upon the given `devinfo`.
- Returns – `doca_error_t` value. `DOCA_SUCCESS` if successful, or an error value upon failure. Possible error values are documented in the header file.
**doca_comm_channel_get_max_recv_queue_size()**

Returns the maximum receive queue size that can be set for a given device. This value describes the maximum possible amount of incoming in-flight messages for a connection. This limitation is important when trying to set the max message size for an endpoint with `doca_comm_channel_ep_set_recv_queue_size()`.

```c
doca_error_t doca_comm_channel_get_max_recv_queue_size(struct doca_devinfo *devinfo, uint32_t *max_recv_queue_size);
```

- `devinfo [in]` – pointer to a `doca_devinfo` which should be queried for this capability.
- `max_recv_queue_size [out]` – pointer to a parameter that on success holds the maximal value that can be set for the receive queue size when communicating upon the given `devinfo`.
- Returns – `doca_error_t` value. `DOCA_SUCCESS` if successful, or an error value upon failure. Possible error values are documented in the header file.

**doca_comm_channel_get_service_max_num_connections()**

Returns the maximum amount of connections a service on the DPU can maintain for a given device. If the maximum amount returned is zero, the number of connections is unlimited.

```c
doca_error_t doca_comm_channel_get_service_max_num_connections(struct doca_devinfo *devinfo, uint32_t *max_num_connections);
```

- `devinfo [in]` – pointer to a `doca_devinfo` which should be queried for this capability.
- `max_num_connections [out]` – pointer to a parameter that on success will hold the maximal number of connections the DPU can maintain when communicating upon the given `devinfo`. 
Creating and Configuring an Endpoint

doca_comm_channel_ep_create()

This function is used to create and initialize the endpoint used for all Comm Channel functions.

```c
doca_error_t doca_comm_channel_ep_create(struct doca_comm_channel_ep_t **ep);
```

- ep [out] – pointer to the created endpoint object.

- Returns – doca_error_t value. DOCA_SUCCESS if successful, or an error value upon failure. Possible error values are documented in the header file.

doca_comm_channel_ep_set_*() and doca_comm_channel_ep_get_*(

Use doca_comm_channel_ep_set_*() functions to set the properties of the endpoint, and corresponding doca_comm_channel_ep_get_*( ) functions to retrieve the current properties of the endpoint.

Mandatory Properties

To use the endpoint, the following properties must be set before calling doca_comm_channel_ep_listen() and doca_comm_channel_ep_connect().

doca_comm_channel_ep_set_device()

This function sets the local device through which the communication should be established.
local_ep [in] – pointer to the endpoint for which the property should be set.

device [in] – the doca_dev object which should be used for communication.

Returns – doca_error_t value. DOCA_SUCCESS if successful, or an error value upon failure. Possible error values are documented in the header file.

**doca_comm_channel_ep_set_device_rep()**

This function sets the device representor through which the communication should be established on the service side.

local_ep [in] – a pointer to the endpoint for which the property should be set.

device_rep [in] – the doca_dev_rep object which should be used for communication.

Returns – doca_error_t value. DOCA_SUCCESS if successful, or an error value upon failure. Possible error values are documented in the header file.

**Optional Properties**

The following properties have a default value and may be set as long as the EP is not yet active.

**doca_comm_channel_ep_set_max_msg_size()**
This function sets an upper limit to the size of the messages the application wishes to handle in this EP while communicating with a given endpoint. The actual max_msg_size may be increased by this function. If this property was not set by the user, a default value is used and may be queried using doca_comm_channel_ep_get_max_msg_size() function.

```c
doca_error_t doca_comm_channel_ep_set_max_msg_size(struct doca_comm_channel_ep_t *local_ep,
uint16_t max_msg_size);
```

- local_ep [in] – a pointer to the endpoint for which the property should be set.
- max_msg_size [in] – the preferred maximal message size.
- Returns – doca_error_t value:
  - DOCA_SUCCESS if successful.
  - DOCA_ERROR_INVALID_VALUE if a null pointer to the endpoint has been given or if max_msg_size is equal to 0 or above the maximal value possible for this property.

**doca_comm_channel_ep_set_send_queue_size()**

This function sets the send queue size used when communicating with a given endpoint. The actual send_queue_size may be increased by this function. If this property has not been set by the user, a default value is used which may be queried using the doca_comm_channel_ep_get_send_queue_size() function.

```c
doca_error_t doca_comm_channel_ep_set_send_queue_size(struct doca_comm_channel_ep_t *local_ep,
uint16_t send_queue_size);
```

- local_ep [in] – pointer to the endpoint for which the property should be set.
- send_queue_size [in] – the preferred send queue size.
- Returns – doca_error_t value:
  - DOCA_SUCCESS if successful.
DOCA_ERROR_INVALID_VALUE if a null pointer to the endpoint has been given or if send_queue_size is equal to 0 or above the maximal value possible for this property.

The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_ep_set_recv_queue_size()

This function sets the receive queue size used when communicating with a given endpoint. The actual recv_queue_size may be increased by this function. If this property has not been set by the user, a default value is used which may be queried using doca_comm_channel_ep_get_recv_queue_size() function.

doca_error_t doca_comm_channel_ep_set_recv_queue_size(struct doca_comm_channel_ep_t *local_ep, uint16_t rcv_queue_size);

- local_ep [in] – pointer to the endpoint for which the property should be set.
- rcv_queue_size [in] – the preferred receive queue size.
- Returns – doca_error_t value:
  - DOCA_SUCCESS if successful.
  - DOCA_ERROR_INVALID_VALUE if a null pointer to the endpoint has been given or if rcv_queue_size is equal to 0 or above the maximal value possible for this property.
  - The rest of the error values that may be returned are documented in the header file.
Establishing Connections over Endpoints

The Comm Channel connection is established between endpoints, one on the host and the other on the DPU.

For a client, each connection requires its own EP. On the DPU side, all of the clients with the same service name on a specific representor are connected to a single EP, through which the connections are managed.

The following functions are relevant for the endpoint.

**doca_comm_channel_ep_listen()**

Used to listen on service endpoint, this function can only be called on the DPU. The service listens on the DOCA device representor provided using doca_comm_channel_ep_set_device_rep(). Calling listen allows clients to connect to the service.

```
doca_error_t doca_comm_channel_ep_listen(struct doca_comm_channel_ep_t *local_ep, const char *name);
```

- **local_ep [in]** – pointer to an endpoint to listen on.
- **name [in]** – the name for the service to listen on. Clients must provide the same name to connect to the service.

- **Returns** – *doca_error_t* value:
  - **DOCA_SUCCESS** if successful.
  - **DOCA_ERROR_BAD_STATE** if mandatory properties (doca_dev and doca_dev_rep) have not been set.
  - **DOCA_ERROR_NOT_PERMITTED** if called on the host and not on the DPU.
  - The rest of the error values that may be returned are documented in the header file.
**doxa_comm_channel_ep_connect()**

Used to create a connection between a client and a service. This function can only be called on the host.

```c
int doxa_comm_channel_ep_connect(struct doxa_comm_channel_ep_t *local_ep, const char *name, struct doxa_comm_channel_addr_t **peer_addr);
```

- **local_ep [in]** – a pointer to an endpoint to connect from.
- **name [in]** – the name of the service that the client connects to. Must be the same name the service listens on.
- **peer_addr [out]** – Contains the pointer to the new connection.
- **Returns** – `doxa_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - `DOCA_ERROR_BAD_STATE` if a mandatory property (`doca_dev`) has not been set.
  - `DOCA_ERROR_NOT_PERMITTED` if called on the DPU and not on the host.
  - The rest of the error values that may be returned are documented in the header file.

**Message Event Channel**

Getting notifications for messages sent and received through an EP is managed by the event channel, using the functions listed here.

**doxa_comm_channel_ep_get_event_channel()**

After a connection is established through the EP, this function extracts send/receive handles which can be used to get an interrupt when a new event happens using `epoll()` or
a similar function.

- A send event happens when at least one in-flight message processing ends.
- A receive event happens when a new incoming message is received.

Users may decide to extract only one of the handles and send a NULL parameter for the other.

The event channels are owned by the endpoint and they are released when `doca_comm_channel_ep_destroy()` is called.

```c
doca_error_t doca_comm_channel_ep_get_event_channel(struct doca_comm_channel_ep_t *local_ep,
                                                      doca_event_channel_t *send_event_channel,
                                                      doca_event_channel_t *recv_event_channel);
```

- `local_ep [in]` – pointer to the endpoint for which a handle should be returned.
- `send_event_channel [out]` – pointer that holds a handle for sent messages if successful.
- `recv_event_channel [out]` – pointer that holds a handle for received messages if successful.

Returns – `doca_error_t` value:

- `DOCA_SUCCESS` if successful.
- `DOCA_ERROR_BAD_STATE` if no connection has been established (i.e., `doca_comm_channel_ep_listen()` or `doca_comm_channel_ep_connect()` has not been called beforehand).
- The rest of the error values that may be returned are documented in the header file.

**doca_comm_channel_ep_event_handle_arm_send()**

After an interrupt caused by an event on the handle for sent messages, the handle should be re-armed to enable interrupts on it:
local_ep [in] – pointer to the endpoint from which the handle has been extracted.

Returns – doca_error_t value:

- DOCA_SUCCESS if successful.
- The rest of the error values that may be returned are documented in the header file.

**doca_comm_channel_ep_event_handle_arm_recv()**

After an interrupt caused by an event on the handle for received messages, the handle should be re-armed to enable interrupts on it:

local_ep [in] – pointer to the endpoint from which the handle has been extracted.

Returns – doca_error_t value:

- DOCA_SUCCESS if successful.
- The rest of the error values that may be returned are documented in the header file.

**doca_comm_channel_ep_sendto()**
Used to send a message from one side to the other. The function runs in non-blocking mode. Refer to section "Usage" for more details.

```c

#include <doctrmgr.h>

doca_error_t doca_comm_channel_ep_sendto(struct doca_comm_channel_ep_t *local_ep, const void *msg, size_t len, int flags, struct doca_comm_channel_addr_t *peer_addr);
```

- **local_ep [in]** – pointer to an endpoint to send the message from.
- **msg [in]** – pointer to the buffer that contains the data to be sent.
- **len [in]** – length of data to be sent.
- **flags [in]** – currently, only DOCA_CC_MSG_FLAG_NONE is a valid flag.
- **peer_addr [in]** – Peer address to send the message to (see also `struct doca_comm_channel_addr_t`) that has been returned by `doca_comm_channel_ep_connect()` or `doca_comm_channel_ep_recvfrom()`.

- Returns – *doca_error_t* value:
  - DOCA_SUCCESS if successful.
  - DOCA_ERROR_AGAIN if the send queue is full and this function should be called again.
  - DOCA_ERROR_CONNECTION_RESET if the provided `peer_addr` experienced an error and must be disconnected.
  - The rest of the error values that may be returned are documented in the header file.

**doca_comm_channel_ep_recvfrom()**

Used to receive a packet of data on either the service or the host. The function runs in non-blocking mode. Refer to Usage for more details.
doca_error_t doca_comm_channel_ep_recvfrom(struct doca_comm_channel_ep_t *local_ep, void *msg, size_t *len, int flags, struct doca_comm_channel_addr_t **peer_addr);

- **local_ep [in]** – pointer to an endpoint to receive the message on.
- **msg [out]** – pointer to a buffer that message should be written to.
- **len [in\out]** – the input is the length of the given message buffer (msg). The output is the actual length of the received message.
- **flags [in]** – DOCA_CC_MSG_FLAG_NONE.
- **peer_addr [out]** – handle to peer_addr that represents the connection the message arrived from

**Returns** – doca_error_t value:

- **DOCA_SUCCESS** if successful.
- **DOCA_ERROR_AGAIN** if no message is received.
- **DOCA_ERROR_CONNECTION_RESET** if the message received is from a peer_addr that has an error.
- The rest of the error values that may be returned are documented in the header file.

**Information Regarding Each Connection**

Each connection established over the EP is represented by a doca_comm_channel_addr_t structure, which can also be referred to as a peer_addr. This structure is returned by either doca_comm_channel_ep_connect() when a connection is established or by doca_comm_channel_ep_recvfrom() to identify the connection from which the message has been received.
doaca_comm_channel_peer_addr_set_user_data() and doaca_comm_channel_peer_addr_get_user_data()

Using doaca_comm_channel_peer_addr_set_user_data(), users may give each connection a context, similar to an ID, to identify it later, using doaca_comm_channel_peer_addr_get_user_data(). If a context is not set for a peer_addr, it is given the default value "0".

- peer_addr [in] – pointer to doaca_comm_channel_addr_t structure representing the connection.
- user_context [in] – context that should be set for the connection.
- Returns – doca_error_t value:
  - DOCA_SUCCESS if successful.
  - DOCA_ERROR_INVALID_VALUE if peer_address is NULL.

doaca_error_t doaca_comm_channel_peer_addr_get_user_data(struct doaca_comm_channel_addr_t *peer_addr, uint64_t *user_context);

- peer_addr [in] – pointer to doaca_comm_channel_addr_t structure representing the connection.
- user_context [out] – pointer to a parameter that will hold the context on success.
- Returns – doca_error_t value:
  - DOCA_SUCCESS if successful.
  - DOCA_ERROR_INVALID_VALUE if the parameters is NULL.

Querying Statistics for Connection
Using the peer_addr, users may gather and query the following statistics:

- The number of messages sent.
- The number of bytes sent.
- The number of messages received.
- The number of bytes received.
- The number of outgoing messages yet to be sent.

**doca_comm_channel_peer_addr_update_info()**

Takes a snapshot with the current statistics of the connection. This function should be called prior to any statistics querying function. It is also used to check the connection status. See **Connection Flow** for more.

```c

doca_error_t doca_comm_channel_peer_addr_update_info(struct doca_comm_channel_addr_t *peer_addr);
```

- **peer_addr [in]** – pointer to doca_comm_channel_addr_t structure representing the connection.
- **Returns** – doca_error_t value:
  - **DOCA_SUCCESS** if successful.
  - **DOCA_ERROR_CONNECTION_INPROGRESS** if the connection has yet to be established.
  - **DOCA_ERROR_CONNECTION_ABORTED** if the connection is in an error state.
  - The rest of the error values that may be returned are documented in the header file.

**doca_comm_channel_peer_addr_get_send_messages()**
This function returns the total number of messages sent to a given peer_addr as measured when doca_comm_channel_peer_addr_update_info() has been last called.

```c
doca_error_t doca_comm_channel_peer_addr_get_send_messages(const struct doca_comm_channel_addr_t *peer_addr, uint64_t *send_messages);
```

- **peer_addr [in]** – pointer to `doca_comm_channel_addr_t` structure representing the connection.

- **send_messages [out]** – pointer to a parameter that holds the number of messages sent through the peer_addr on success.

- **Returns** – `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

### doca_comm_channel_peer_addr_get_send_bytes()

This function returns the total number of bytes sent to a given peer_addr as measured when doca_comm_channel_peer_addr_update_info() has been last called.

```c
doca_error_t doca_comm_channel_peer_addr_get_send_bytes(const struct doca_comm_channel_addr_t *peer_addr, uint64_t *send_bytes);
```

- **peer_addr [in]** – pointer to `doca_comm_channel_addr_t` structure representing the connection.

- **send_bytes [out]** – pointer to a parameter that holds the number of bytes sent through the peer_addr on success.

- **Returns** – `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
The rest of the error values that may be returned are documented in the header file.

**doca_comm_channel_peer_addr_get_recv_messages()**

This function returns the total number of messages received from a given peer_addr as measured when doca_comm_channel_peer_addr_update_info() has been last called.

```c
doca_error_t doca_comm_channel_peer_addr_get_recv_messages(const struct doca_comm_channel_addr_t *peer_addr, uint64_t *recv_messages);
```

- **peer_addr [in]** – pointer to doca_comm_channel_addr_t structure representing the connection.

- **recv_messages [out]** – pointer to a parameter that holds the number of messages received from the peer_addr on success.

- **Returns** – doca_error_t value:
  - **DOCA_SUCCESS** if successful.
  - The rest of the error values that may be returned are documented in the header file.

**doca_comm_channel_peer_addr_get_recv_bytes()**

This function will return the total number of bytes received from a given peer_addr as measured when doca_comm_channel_peer_addr_update_info() has been last called.

```c
doca_error_t doca_comm_channel_peer_addr_get_recv_bytes(const struct doca_comm_channel_addr_t *peer_addr, uint64_t *recv_bytes);
```

- **peer_addr [in]** – pointer to doca_comm_channel_addr_t structure representing the connection.
- **recv_bytes [out]** – pointer to a parameter that holds the number of bytes sent through the `peer_addr` on success.

- **Returns** – `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

### `doca_comm_channel_peer_addr_get_send_in_flight_messages()`

This function returns the number of messages still in transmission to a specific `peer_addr` as measured when `doca_comm_channel_peer_addr_update_info()` has been last called. This function can be used to check if all messages are sent before disconnecting.

```c
doca_error_t doca_comm_channel_peer_addr_get_send_in_flight_messages(const struct doca_comm_channel_addr_t *peer_addr,
    uint64_t *send_in_flight_messages);
```

- **peer_addr [in]** – pointer to `doca_comm_channel_addr_t` structure representing the connection.

- **send_in_flight_messages [out]** – pointer to a parameter that holds the number of in-flight messages to the `peer_addr` on success.

- **Returns** – `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.
Service State and Events

The service state and events API provides information about the state of the service including current connected clients, pending connections, and service state. All the functions in this section are relevant and can be run on the service side only.

`doca_comm_channel_ep_get_service_event_channel()`

After a service is created and starts listening, this function extracts a handle which can be used to get an interrupt when a new service event happens using `epoll()` or a similar function.

The currently supported events are service failure, new client connection, and client disconnection. After an event is triggered, the application can call `doca_comm_channel_ep_update_service_state_info()` and the following getter functions to query the service state and connections.

The service event channel is armed automatically when calling `doca_comm_channel_ep_update_service_state_info()`.

```c
doca_error_t doca_comm_channel_ep_get_service_event_channel(struct doca_comm_channel_ep_t *local_ep, doca_event_channel_t *service_event_channel);
```

- `local_ep` [in] – pointer to the service endpoint that should be queried.

- `service_event_channel` [out] – event handle for service events.

- **Returns** – `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

`doca_comm_channel_ep_update_service_state_info()`
**Tip**

This function should be called prior to calling service status get functions.

Takes a snapshot of the current state of the service. The return value may indicate the service state. If the service is in error state, then it is non-recoverable and the endpoint must be destroyed.

**Note**

Calling this function invalidates any array received using `doca_comm_channel_ep_get_peer_addr_list()`.

```c
#include <doca_comm.h>

doca_error_t doca_comm_channel_ep_update_service_state_info(struct doca_comm_channel_ep_t *local_ep);
```

- `local_ep [in]` – pointer to the service endpoint that should be queried.
- **Returns** – `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - `DOCA_ERROR_CONNECTION_RESET` if the service is in error state and cannot be recovered.
  - The rest of the error values that may be returned are documented in the header file.

**doca_comm_channel_ep_get_peer_addr_list()**
This function returns the list of connected `peer_addr`s as present when `doca_comm_channel_ep_update_service_state_info()` was last called.

**Info**

This list includes only active `peer_addr`s which have not been disconnected from the client side or the service side.

The output array is only valid until `doca_comm_channel_ep_update_service_state_info()` is called again.

```c
#include <doaca.h>

doca_error_t doca_comm_channel_ep_get_peer_addr_list(const struct doca_comm_channel_ep_t *local_ep,
                                                     struct doca_comm_channel_addr_t ***peer_addr_array,
                                                     uint32_t *peer_addr_array_len);
```

- `local_ep [in]` – pointer to the service endpoint that should be queried.
- `peer_addr_array [out]` – pointer to array of peer addresses.
- `peer_addr_array_len [out]` – the number of entries in `peer_addr_array`.
- **Returns** – `doca_error_t` value:
  - `DOCA_SUCCESS` if successful.
  - The rest of the error values that may be returned are documented in the header file.

**doca_comm_channel_ep_get_pending_connections()**

This function returns the list of pending connections as present when `doca_comm_channel_ep_update_service_state_info()` was last called. Pending connections are connections that were initiated by the client side but not complete from the service side.
Info

If a pending connection exists, the application is expected to call `doca_comm_channel_ep_recvfrom()` to complete the connection. See section "Connection Flow" for more.

doca_error_t doca_comm_channel_ep_get_pending_connections(const struct doca_comm_channel_ep_t *local_ep,

            uint32_t *pending_connections);

• local_ep [in] – pointer to the service endpoint that should be queried.

• pending_connections [out] – the number of pending connections.

• Returns – `doca_error_t` value:
  
  • `DOCA_SUCCESS` if successful.

  • The rest of the error values that may be returned are documented in the header file.

doca_comm_channel_ep_disconnect()

Disconnects an endpoint from a specific `peer_address`. The disconnection is one-sided and the other side is unaware of it. New connections can be created afterwards. Refer to "Usage" for more details.

doca_error_t doca_comm_channel_ep_disconnect(struct doca_comm_channel_ep_t *local_ep, struct doca_comm_channel_addr_t *peer_addr);

• local_ep [in] – pointer to the endpoint that should be disconnected.
• peer_addr [in] – the connection from which the endpoint should be disconnected.

• Returns – doca_error_t value:
  ○ DOCA_SUCCESS if successful.
  ○ DOCA_ERROR_NOT_CONNECTED if there is no connection between the endpoint and the peer address.

doca_comm_channel_ep_destroy()

Disconnects all connections of the endpoint, destroys the endpoint object, and frees all related resources.

doca_error_t doca_comm_channel_ep_destroy(struct doca_comm_channel_ep_t *ep);

• local_ep [in] – pointer to the endpoint that should be destroyed.

• Returns – doca_error_t value:
  ○ DOCA_SUCCESS if successful.
  ○ The rest of the error values that may be returned are documented in the header file.

Limitations

Endpoint Properties

The maximal values of all endpoint properties can be queried using the proper get functions (see section "Query Device Capabilities"). The max_message_size, send_queue_size, and recv_queue_size attributes may be increased internally. The updated property value can be queried with the proper get functions.
See the following table and section "doca_comm_channel_ep_set_*() and doca_comm_channel_ep_get_*()" for more details.

<table>
<thead>
<tr>
<th>Property</th>
<th>Get Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max message size</td>
<td>doca_comm_channel_get_max_message_size()</td>
</tr>
<tr>
<td>Send queue size</td>
<td>doca_comm_channel_get_max_send_queue_size()</td>
</tr>
<tr>
<td>Receive queue size</td>
<td>doca_comm_channel_get_max_recv_queue_size()</td>
</tr>
<tr>
<td>Service name length</td>
<td>doca_comm_channel_get_max_service_name_len()</td>
</tr>
</tbody>
</table>

**Multi-client**

A single service on the DPU can serve multiple clients but a client can only connect to a single service.

The maximal number of clients connected to a single service can be queried using doca_comm_channel_get_service_max_num_connections().

**Multiple Services**

Multiple endpoints can be created on the same DPU but different services listening on the same representor must have different names. Services listening on different representors can have the same name.

**Threads**

The DOCA Comm Channel is not thread-safe. Using a single endpoint over multiple threads is possible only with the use of locks to prevent parallel usage of the same resources. Different endpoints can be used over different threads with no restriction as each endpoint has its own resources.
Usage

Objects

While working with DOCA Comm Channel, the user must maintain two types of objects:

- struct doca_comm_channel_ep_t (referred to as "endpoint")
- struct doca_comm_channel_addr_t (referred to as "peer_address")

Endpoint

The endpoint object represents the endpoint of the Comm Channel, either on the client or service side. The endpoint is created by calling the doca_comm_channel_ep_create() function. It is required for every other Comm Channel function.

Peer_address

The peer_address structure represents a connection. It is created when a new connection is made (i.e., client calls doca_comm_channel_ep_connect() or a service receives a connection through doca_comm_channel_ep_recvfrom()). Refer to section "Connection Flow" for more details on connections.

The peer_address structure can be used to identify the source of a received message and is necessary to send a message using doca_comm_channel_ep_sendto(). peer_address has an identifier, user_data, which can be set by the user using doca_comm_channel_peer_addr_user_data_set() and retrieved using doca_comm_channel_peer_addr_user_data_get(). The default value for user_data is 0. The user_data field can be used to identify the peer_address object.

Endpoint Initialization

To start using the DOCA Comm Channel, the user must create an endpoint object using the doca_comm_channel_ep_create() function. After creating the endpoint object, the user must
set the mandatory endpoint properties: `doca_dev` for client and service, `doca_dev_rep` for service only. The user may also set the optional endpoint properties.

For further information about endpoint initialization, refer to section "Establishing Connection over Endpoint".

**Connection Flow**

The following diagram illustrates the process of establishing a connection between the host and a service.

1. After initializing the endpoint on the service side, one should call `doca_comm_channel_ep_listen()` with a legal service name (see "Limitations") to start listening.

2. After the service starts listening and the client endpoint is created, the client calls `doca_comm_channel_ep_connect()` with the same service name used for listening.

As part of the connect function, the client starts a handshake protocol with the server, which then waits until the service completes the handshake. If connect is called before the service is listening or the handshake process fails, then the connect function fails.

From the connect function, the client receives a `peer_addr` object representing the new connection to the service:

1. To check whether the connection is complete or not, the client must call `doca_comm_channel_peer_addr_update_info()` with the new `peer_addr`. Depending on the function return code, the client would know whether the connection is complete (`DOCA_SUCCESS`), rejected (`DOCA_ERROR_CONNECTION_ABORTED`) or still in progress (`DOCA_ERROR_CONNECTION_INPROGRESS`).
2. The service receiving new connections is done using `doca_comm_channel_ep_recvfrom()`. No indication is given that a new connection is made. The server keeps waiting to receive packets. If the handshake fails or is done for an existing client, then the receive function fails.

For more information, see section "`doca_comm_channel_ep_listen()`".

**Data Transfer Flow**

After a connection is established between client and service, both sides can send and receive data using the `doca_comm_channel_ep_sendto()` and `doca_comm_channel_ep_recvfrom()` functions, respectively.

If multiple clients are connected to the same service, then the `doca_comm_channel_ep_recvfrom()` function reads the messages in the order of their arrival, regardless of their source.

To send a message, the endpoint must obtain the target's `peer_address` object. This restriction necessitates the client to start the communication (not including the handshake), by sending the first message, for the server to obtain the client's `peer_address` object and send data back.

The `doca_comm_channel_ep_sendto()` function adds the message to an internal send queue where it is processed asynchronously. This means that even if the `doca_comm_channel_ep_sendto()` function returns with `DOCA_SUCCESS`, the message itself may fail to send (e.g., if the other side has been disconnected). If a message fails to send, the relevant `peer_address` moves to `error_state`. See section "Connection Errors" for more.

For more information, see section "`doca_comm_channel_ep_sendto()`".

**Event Channel and Event Handling**

When trying to send or receive messages, the application may face a situation where the resources are not ready—send queue full or no new messages received. In this case, the Comm Channel returns `DOCA_ERROR_AGAIN` for the call. This return value indicates that the function must be called again later in order to complete. To know when to call the send/receive function again, the application can use two approaches:
● Active polling – that is, to use a loop to call the send/receive functions immediately or after a certain time until the DOCA_SUCCESS return code is received.

● Using CC event channel to know when to call the send/receive function again.

The CC event channel is a mechanism that enables getting an event when a new CC event happens. It is divided to send and receive event channels which can be retrieved using doca_comm_channel_ep_get_event_channel(). After retrieving the event channels, the application can use poll in Linux or GetQueuedCompletionStatus in Windows to sleep and wait for events.

When first using the event channels and after each event is received using the event channel, it must be armed using doca_comm_channel_ep_event_handle_arm_send() or doca_comm_channel_ep_event_handle_arm_recv() to receive more events.

For more information, see section "Event Channel".

Connection Errors

In certain cases, for example if a remote peer disconnects and the local endpoint tries sending a message, a peer_addr can move to error state. In such cases, no new messages can be sent to or received from the certain peer_addr.

The Comm Channel indicates a peer_addr is in an error state by returning DOCA_ERROR_CONNECTION_RESET on doca_comm_channel_ep_sendto() if trying to send a message to an errored peer_addr or on doca_comm_channel_ep_recvfrom() when receiving a message from a peer_addr marked as errored, or when calling doca_comm_channel_peer_addr_update_info().

When a peer_addr is in an error state, it is the application's responsibility to disconnect the said peer_addr using doca_comm_channel_ep_disconnect().

Connection Statistics

The peer_addr object provides a statistics mechanism. To get the updated statistics, the application should call doca_comm_channel_peer_addr_update_info() which saves a snapshot of the current statistics.
After calling the update function, the application can query the following statistics which return the data from that snapshot:

<table>
<thead>
<tr>
<th>Statistic Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>doca_comm_channel_peer_addr_get_send_messages()</code></td>
<td>Number of messages sent to the specific <code>peer_addr</code></td>
</tr>
<tr>
<td><code>doca_comm_channel_peer_addr_get_send_bytes()</code></td>
<td>Number of bytes sent to the specific <code>peer_addr</code></td>
</tr>
<tr>
<td><code>doca_comm_channel_peer_addr_get_recv_messages()</code></td>
<td>Number of messages received from the specific <code>peer_addr</code></td>
</tr>
<tr>
<td><code>doca_comm_channel_peer_addr_get_recv_bytes()</code></td>
<td>Number of bytes received from the specific <code>peer_addr</code></td>
</tr>
<tr>
<td><code>doca_comm_channel_peer_addr_get_send_in_flight_messages()</code></td>
<td>Number of messages sent to the specific <code>peer_addr</code> and without returning a confirmation yet</td>
</tr>
</tbody>
</table>

The in-flight messages can be used to make sure all messages have been successfully sent before disconnecting or destroying the endpoint.

For more information, see section "Querying Statistics for Connection".

**Service State and Connections**

DOCA Comm Channel provides an API, `doca_comm_channel_ep_update_service_state_info()`, to query for the service state and connections which an application can call.

The service state is returned as the return value from the update function:

- If the return value is `DOCA_SUCCESS` the service state is operational.
- If the return value is `DOCA_ERROR_CONNECTION_RESET` the service is down and cannot be recovered, and the endpoint should be destroyed.

After calling the update function, the application can query the following functions which return the connection data from that snapshot:
<table>
<thead>
<tr>
<th>Information Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>doca_comm_channel_ep_get_peer_addr_list()</code></td>
<td>Returns the list of connected <code>peer_addrs</code></td>
</tr>
<tr>
<td><code>doca_comm_channel_ep_get_pending_connections()</code></td>
<td>Number of pending connections waiting for the service. If there are pending connections, <code>doca_comm_channel_ep_recvfrom()</code> should be called to handle them.</td>
</tr>
</tbody>
</table>

**Disconnection Flow**

Disconnection can occur specifically by using `doca_comm_channel_ep_disconnect()` or when destroying the whole endpoint.

Disconnection is one-sided, which means that the other side is unaware of the channel being closed and experiences errors when sending data. It is up to the application to synchronize the connection teardown.

Disconnection of a `peer_addr` destroys all of the resources related to it.

It is possible to perform another handshake and establish a new channel connection after disconnection.

For more information, see section "`doca_comm_channel_ep_disconnect()`".

**Endpoint Destruction**

When calling `doca_comm_channel_ep_destroy()`, all resources related to the endpoint are freed immediately which means that if there are any messages in the send queue that have not been sent yet, they are aborted.

To make sure all messages have been successfully sent before disconnection, the application can use the `doca_comm_channel_peer_addr_get_send_in_flight_messages()` statistics function. See section "Connection Statistics" for more information.
DOCA Comm Channel Samples

This section provides Comm Channel sample implementation on top of the BlueField DPU.

Running the Sample

1. Refer to the following documents:
   
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_comm_channel/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   **Note**

   The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample (e.g., `comm_channel_server`) usage:

   ```
   Usage: doca_comm_channel_server [DOCA Flags] [Program Flags]
   
   DOCA Flags:
   -h, --help                     Print a help synopsis
   -v, --version                  Print program version information
   -l, --log-level [<level>]     Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
   ```
4. For additional information per sample, use the -h option:

```
/tmp/build/doca_<sample_name> -h
```

**Samples**

**Comm Channel Server**

- **Note**

  This sample should be run before "Comm Channel Client".

This sample illustrate how to create a simple server on the DPU to communicate with a client on the host.

The sample logic includes:
1. Creating Comm Channel endpoint.

2. Parsing PCIe address.

3. Opening Comm Channel DOCA device based on the PCIe address.

4. Opening Comm Channel DOCA device representor based on the PCIe address.

5. Setting Comm Channel endpoint properties.


7. Waiting until new message arrives.

8. Sending the entered text message as a response.

9. Closing connection and freeing resources.

Reference:

- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_server/comm_channel_server_sample`
- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_server/comm_channel_server_main`
- `/opt/mellanox/doca/samples/doca_comm_channel/comm_channel_server/meson.build`

**Comm Channel Client**

⚠️ **Note**

This sample should be run after "Comm Channel Server".

This sample illustrates how to create a simple client on the host to communicate with a server on the DPU.

The sample logic includes:
1. Creating Comm Channel endpoint.

2. Parsing PCIe address.

3. Opening Comm Channel DOCA device based on the PCIe address.

4. Setting Comm Channel endpoint properties.

5. Connecting current endpoint to server side.

6. Sending the entered text message.

7. Receiving server response.

8. Closing connection and freeing resources.

Reference:

- /opt/mellanox/doca/samples/doca_comm_channel/comm_channel_client/comm_channel_client_sample
- /opt/mellanox/doca/samples/doca_comm_channel/comm_channel_client/comm_channel_client_main.c
- /opt/mellanox/doca/samples/doca_comm_channel/comm_channel_client/meson.build
DOCA UROM

This guide provides an overview and configuration instructions for DOCA Unified Resources and Offload Manager (UROM) API.

Introduction

ℹ️ Note

This library is currently supported at alpha level only.

The DOCA Unified Resource and Offload Manager (UROM) offers a framework for offloading a portion of parallel computing tasks, such as those related to HPC or AI workloads and frameworks, from the host to the NVIDIA DPUs. This framework includes the UROM service which is responsible for resource discovery, coordination between the host and DPU, and the management of UROM workers that execute parallel computing tasks.

When an application utilizes the UROM framework for offloading, it consists of two main components: the host part and the UROM worker on the DPU. The host part is responsible for interacting with the DOCA UROM API and operates as part of the application with the aim of offloading tasks to the DPU. This component establishes a connection with the UROM service and initiates an offload request. In response to the offload request, the UROM service provides network identifiers for the workers, which are spawned by the UROM service. If the UROM service is running as a Kubernetes POD, the workers are spawned within the POD. Each worker is responsible for executing either a single offload or multiple offloads, depending on the requirements of the host application.

Prerequisites
UCX is required for the communication channel between the host and DPU parts of DOCA UROM based on TCP socket transport. This is a mechanism to transfer commands from the host to the UROM service on the DPU and receive responses from the DPU.

By default, UCX scans all available devices on the machine and selects the best ones based on performance characteristics. The environment variable `UCX_NET_DEVICES=<dev1>, <dev2>,...` would restrict UCX to using only the specified devices. For example, `UCX_NET_DEVICES=eth2` uses the Ethernet device `eth2` for TCP socket transport.

For more information about UCX, refer to [DOCA UCX Programming Guide](#).

**Architecture**

**UROM Deployment**

The diagram illustrates a standard UROM deployment where each DPU is required to host both a service process instance and a group of worker processes.

The typical usage of UROM services involves the following steps:

1. Every process in the parallel application discovers the UROM service.
2. UROM handles authentication and provides service details.
3. The host application receives the available offloading plugins on the local DPU through UROM service.
4. The host application picks the desired plugin info and triggers UROM worker plugin instances on the DPU through the UROM service.
5. The application delegates specific tasks to the UROM workers.
6. UROM workers execute these tasks and return the results.

UROM Framework

This diagram shows a high-level overview of the DOCA UROM framework.

A UROM offload plugin is where developers of AI/HPC offloads implement their own offloading logic while using DOCA UROM as the transport layer and resource manager. Each plugin defines commands to execute logic on the DPU and notifications that are...
returned to the host application. Each type of supported offload corresponds to a distinct type of DOCA UROM plugin. For example, a developer may need a UCC plugin to offload UCC functionality to the DPU. Each plugin implements a DPU-side plugin API and exposes a corresponding host-side interface.

A UROM daemon loads the plugin DPU version (.so file) in runtime as part of the discovery of local plugins.

**Plugin Task Offloading Flow**

DOCA UROM installation.

**UROM Installation**

DOCA UROM is an integral part of the DOCA SDK installation package. Depending on your system architecture and enabled offload plugins, UROM is comprised by several components, which can be categorized into two main parts: those on the host and those on the DPU.

- **DOCA UROM library components:**
  - libdoca_urom shared object – contains the DOCA UROM API
libdoca_urom_components_comm_ucp_am – includes the UROM communication channel interface API

<table>
<thead>
<tr>
<th>DOCA UROM Shared Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>libdoca_urom.a</td>
</tr>
<tr>
<td>libdoca_urom.so</td>
</tr>
<tr>
<td>libdoca_urom_components_comm_ucp_am.a</td>
</tr>
<tr>
<td>libdoca_urom_components_comm_ucp_am.so</td>
</tr>
</tbody>
</table>

- **DOCA UROM headers:**

<table>
<thead>
<tr>
<th>DOCA Includes</th>
</tr>
</thead>
<tbody>
<tr>
<td>doca_urom.h</td>
</tr>
<tr>
<td>doca_urom_plugin.h</td>
</tr>
</tbody>
</table>

The header files include definitions for DOCA UROM as described in the following:

- **DOCA UROM host interface (doca_urom.h)** – this header includes three essential components: contexts, tasks, and plugins.
  - **Service context (doca_urom_service)** – this context serves as an abstraction of the UROM service process. Tasks posted within this context include the authentication, spawning, and termination of workers on the DPU.
  - **Worker context (doca_urom_worker)** – this context abstracts the DPU UROM worker, which operates on behalf of host application plugins (offload). Tasks posted within this context involve relaying commands from the host application to the worker on behalf of a specific offload plugin, such as offloaded functionality for communication operations.
  - **Domain context (doca_urom_domain)** – this context encapsulates a group of workers belonging to the same host application. This concept is similar to the MPI (message passing interface) communicator in the MPI programming model or PyTorch's process groups. Plugins are not required to use the UROM Domain.

- **DOCA UROM plugin interface (doca_urom_plugin.h)** – this header includes the main structure and definitions that the user can use to build both the host and DPU components of their own offloading plugins.
- UROM plugin interface structure (urom_plugin_iface) – this interface includes a set of operations to be executed by the UROM worker

- UROM worker command structure (urom_worker_cmd) – this structure defines the worker instance command format

- UROM worker notification structure (urom_worker_notify) – this structure defines the worker instance notification format

The following diagram shows various software components of DOCA UROM:

- DOCA Core – involves DOCA device discovery, DOCA progress engine, DOCA context, etc.

- DOCA UROM Core – includes the UROM library functionality

- DOCA UROM Host SDK – UROM API for the host application to use

- DOCA UROM DPU SDK – UROM API for the NVIDIA® BlueField ® networking platform (DPU or SuperNIC) to use

- DOCA UROM Host Plugin – user plugin host version

- DOCA UROM DPU Plugin – user plugin DPU version

- DOCA UROM App – user UROM host application

- DOCA UROM Worker – the offload functionality component that executes the offloading logic

- DOCA UROM Daemon – is responsible for resource discovery, coordination between the host and DPU, managing the workers on BlueField
The following sections provide additional details about the library API.

**DOCA_UROM_SERVICE_FILE**

This environment variable sets the path to the UROM service file. When creating the UROM service object (see `doca_urom_service_create`), UROM performs a look-up using this file, the hostname where an application is running, and the PCIe address of the...
associated DOCA device to identify the network address, and network devices associated with the UROM service.

This file contains one entry per line describing the location of each UROM service that may be used by UROM. The format of each line must be as follows:

```
<app_hostname> <service_type> <dev_hostname> <dev_pci_addr> <net,devs>
```

Example:

```
app_host1 dpu dpu_host1 03:00.0 dev1:1,dev2:1
```

Fields are described in the following table:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_hostname</td>
<td>Network hostname (or IP address) for the node that this line applies to</td>
</tr>
<tr>
<td>service_type</td>
<td>The UROM service type. Valid type is dpu (used for all DOCA devices).</td>
</tr>
<tr>
<td>dev_hostname</td>
<td>Network hostname (or IP address) for the associated DOCA device</td>
</tr>
<tr>
<td>dev_pci_addr</td>
<td>PCIe address of the associated DOCA device. This must match the PCIe address provided by DOCA.</td>
</tr>
<tr>
<td>net,devs</td>
<td>Comma-separated list of network devices shared between the host and DOCA device</td>
</tr>
</tbody>
</table>

**doca_urom_service**

An opaque structure that represents a DOCA UROM service.

```
struct doca_urom_service;
```
**doca_urom_service_plugin_info**

DOCA UROM plugin info structure. UROM generates this structure for each plugin on the local DPU where the UROM service is running and the service returns an array of available plugins to the host application to pick which plugins to use.

```c
struct doca_urom_service_plugin_info {
    uint64_t id;
    uint64_t version;
    char plugin_name[DOCA_UROM_PLUGIN_NAME_MAX_LEN];
};
```

- **id** – Unique ID to send commands to the plugin, UROM generates this ID
- **version** – Plugin DPU version to verify that the plugin host interface has the same version
- **plugin_name** – The `.so` plugin file name without `".so"`. The name is used to find the desired plugin.

**doca_urom_service_get_workers_by_gid_task**

An opaque structure representing a DOCA service gets workers by group ID task.

```c
struct doca_urom_service_get_workers_by_gid_task;
```

**doca_urom_service_create**

Before performing any UROM service operation (spawn worker, destroy worker, etc.), it is essential to create a `doca_urom_service` object. A service object is created in state
DOCA_CTX_STATE_IDLE. After creation, the user may configure the service using setter methods (e.g., `doca_urom_service_set_dev()`).

Before use, a service object must be transitioned to state `DOCA_CTX_STATE_RUNNING` using the `doca_ctx_start()` interface. A typical invocation looks like `doca_ctx_start(doca_urom_service_as_ctx(service_ctx))`.

```c
doca_error_t doca_urom_service_create(struct doca_urom_service **service_ctx);
```

- `service_ctx [in/out]` – `doca_urom_service` object to be created
- Returns – `DOCA_SUCCESS` on success, error code otherwise

**Info**

Multiple application processes could create different service objects that represent/connect to the same worker on the DPU.

`doca_urom_service_destroy`

Destroy a `doca_urom_service` object.

```c
doca_error_t doca_urom_service_destroy(struct doca_urom_service *service_ctx);
```

- `service_ctx[in]` – `doca_urom_service` object to be destroyed. It is created by `doca_urom_service_create()`.
- Returns – `DOCA_SUCCESS` on success, error code otherwise
**doca_urom_service_set_max_comm_msg_size**

Set the maximum size for a message in the UROM communication channel. The default message size is 4096B.

![Note]

It is important to ensure that the combined size of the plugins' commands and notifications and the UROM structure's size do not exceed this maximum size.

Once the service state is running, users cannot update the maximum size for the message.

```c
doca_error_t doca_urom_service_set_max_comm_msg_size(struct doca_urom_service *service_ctx, size_t msg_size);
```

- **service_ctx[in]** – a pointer to `doca_urom_service` object to set new message size
- **msg_size[in]** – new message size to set
- **Returns** – `DOCA_SUCCESS` on success, error code otherwise

**doca_urom_service_as_ctx**

Convert a `doca_urom_service` object into a DOCA object.

```c
struct doca_ctx *doca_urom_service_as_ctx(struct doca_urom_service *service_ctx);
```

- **service_ctx[in]** – a pointer to `doca_urom_service` object
Returns – a pointer to the doca_ctx object on success, NULL otherwise

doca_urom_service_get_plugins_list
Retrieve the list of supported plugins on the UROM service.

doca_error_t doca_urom_service_get_plugins_list(struct doca_urom_service *service_ctx, const struct doca_urom_service_plugin_info **plugins, size_t *plugins_count);

- service_ctx[in] – a pointer to doca_urom_service object
- plugins[out] – an array of pointers to doca_urom_service_plugin_info object
- plugins_count[out] – number of plugins
- Returns – DOCA_SUCCESS on success, error code otherwise

doca_urom_service_get_cpuset
Get the allowed CPU set for the UROM service on BlueField, which can be used when spawning workers to set processor affinity.

doca_error_t doca_urom_service_get_cpuset(struct doca_urom_service *service_ctx, doca_cpu_set_t *cpuset);

- service_ctx[in] – a pointer to doca_urom_service object
- cpuset[out] – set of allowed CPUs
- Returns – DOCA_SUCCESS on success, error code otherwise
docta_urom_service_get_workers_by_gid_task_allocate_init

Allocate a get-workers-by-GID service task and set task attributes.

```c
docta_error_t docta_urom_service_get_workers_by_gid_task_allocate_init(struct docta_urom_service *service_ctx,
                        uint32_t gid,
                        docta_urom_service_get_workers_by_gid_task_completion_cb_t cb,
                        struct docta_urom_service_get_workers_by_gid_task **task);
```

- `service_ctx[in]` – a pointer to `docta_urom_service` object
- `gid[in]` – group ID to set
- `cb[in]` – user task completion callback
- `task[out]` – a new get-workers-by-GID service task
- Returns – `DOCA_SUCCESS` on success, error code otherwise

docta_urom_service_get_workers_by_gid_task_release

Release a get-workers-by-GID service task and task resources.

```c
docta_error_t docta_urom_service_get_workers_by_gid_task_release(struct docta_urom_service_get_workers_by_gid_task *task);
```

- `task[in]` – service task to release
- Returns – `DOCA_SUCCESS` on success, error code otherwise
**doca_urom_service_get_workers_by_gid_task_as_task**

Convert a `doca_urom_service_get_workers_by_gid_task` object into a DOCA task object.

After creating a service task and configuring it using setter methods (e.g., `doca_urom_service_get_workers_by_gid_task_set_gid()`) or as part of task allocation, the user should submit the task by calling `doca_task_submit`.

A typical invocation looks like

```c
doca_task_submit(doca_urom_service_get_workers_by_gid_task_as_task(task));
```

- **task[in]** – get-workers-by-GID service task
- **Returns** – a pointer to the `doca_task` object on success, `NULL` otherwise

**doca_urom_service_get_workers_by_gid_task_get_workers_count**

Get the number of workers returned for the requested GID.

```c
size_t doca_urom_service_get_workers_by_gid_task_get_workers_count(struct doca_urom_service_get_workers_by_gid_task *task);
```

- **task[in]** – get-workers-by-GID service task
- **Returns** – workers ID's array size

**doca_urom_service_get_workers_by_gid_task_get_workers**

Get service get workers task IDs array.
const uint64_t *doca_urom_service_get_workers_by_gid_task_get_worker_ids(struct doca_urom_service_get_workers_by_gid_task *task);

- task[in] – get-workers-by-GID service task
- Returns – workers ID's array, NULL otherwise

**doca_urom_worker**

An opaque structure representing a DOCA UROM worker context.

```c
struct doca_urom_worker;
```

**doca_urom_worker_cmd_task**

An opaque structure representing a DOCA UROM worker command task context.

```c
struct doca_urom_worker_cmd_task;
```

**doca_urom_worker_cmd_task_completion_cb_t**

A worker command task completion callback type. It is called once the worker task is completed.

```c
typedef void (*doca_urom_worker_cmd_task_completion_cb_t)(struct doca_urom_worker_cmd_task *task,
                                                           union doca_data task_user_data,
                                                           union doca_data ctx_user_data);
```

- task[in] – a pointer to worker command task
- task_user_data[in] – user task data
- ctx_user_data[in] – user worker context data

**doca_urom_worker_create**

This method creates a UROM worker context.

A worker is created in a DOCA_CTX_STATE_IDLE state. After creation, a user may configure the worker using setter methods (e.g., doca_urom_worker_set_service()). Before use, a worker must be transitioned to state DOCA_CTX_STATE_RUNNING using the doca_ctx_start() interface. A typical invocation looks like doca_ctx_start(doca_urom_worker_as_ctx(worker_ctx)).

```c
doca_error_t doca_urom_worker_create(struct doca_urom_worker **worker_ctx);
```

- worker_ctx [in/out] – doca_urom_worker object to be created
- Returns – DOCA_SUCCESS on success, error code otherwise

**doca_urom_worker_destroy**

Destroys a UROM worker context.

```c
doca_error_t doca_urom_worker_destroy(struct doca_urom_worker *worker_ctx);
```

- worker_ctx [in] – doca_urom_worker object to be destroyed. It is created by doca_urom_worker_create().
- Returns – DOCA_SUCCESS on success, error code otherwise
**doca_urom_worker_set_service**

Attaches a UROM service to the worker context. The worker is launched on the DOCA device managed by the provided service context.

```c
doca_error_t doca_urom_worker_set_service(struct doca_urom_worker *worker_ctx, struct doca_urom_service *service_ctx);
```

- **service_ctx [in]** – service context to set
- **Returns** – DOCA_SUCCESS on success, error code otherwise

**doca_urom_worker_set_id**

This method sets the worker context ID to be used to identify the worker. Worker IDs enable an application to establish multiple connections to the same worker process running on a DOCA device.

Worker ID must be unique to a UROM service.

- If `DOCA_UROM_WORKER_ID_ANY` is specified, the service assigns a unique ID for the newly created worker.
- If a specific ID is used, the service looks for an existing worker with matching ID. If one exists, the service establishes a new connection to the existing worker. If a matching worker does not exist, a new worker is created with the specified worker ID.

```c
doca_error_t doca_urom_worker_set_id(struct doca_urom_worker *worker_ctx, uint64_t worker_id);
```

- **worker_ctx [in]** – doca_urom_worker object
- **worker_id [in]** – worker ID
- **Returns** – DOCA_SUCCESS on success, error code otherwise
**doca_urom_worker_set_gid**

Set worker group ID. This ID must be set before starting the worker context.

Through service get workers by GID task, the application can have the list of workers' IDs which are running on DOCA device and that belong to the same group ID.

```c
static doca_error_t doca_urom_worker_set_gid(struct doca_urom_worker *worker_ctx, uint32_t gid);
```

- `gid` [in] – worker group ID
- Returns – `DOCA_SUCCESS` on success, error code otherwise

**doca_urom_worker_set_plugins**

Adds a plugin mask for the supported plugins by the UROM worker on the DPU. The application can use up to 62 plugins.

```c
static doca_error_t doca_urom_worker_set_plugins(struct doca_urom_worker *worker_ctx, uint64_t plugins);
```

- `plugins` [in] – an ORing set of worker plugin IDs
- Returns – `DOCA_SUCCESS` on success, error code otherwise

**doca_urom_worker_set_env**
Set worker environment variables when spawning worker on DPU side by DOCA UROM service. They must be set before starting the worker context.

⚠️ **Info**

This call fails if the worker already spawned on the DPU.

doca_error_t doca_urom_worker_set_env(struct doca_urom_worker *worker_ctx, char const *env[], size_t count);

- env[in] – an array of environment variables
- count[in] – array size
- Returns – DOCA_SUCCESS on success, error code otherwise

**doca_urom_worker_as_ctx**

Convert a doca_urom_worker object into a DOCA object.

struct doca_ctx *doca_urom_worker_as_ctx(struct doca_urom_worker *worker_ctx);

- worker_ctx[in] – a pointer to doca_urom_worker object
- Returns – a pointer to the doca_ctx object on success, NULL otherwise

**doca_urom_worker_cmd_task_allocate_init**
Allocate worker command task and set task attributes.

```c
doca_error_t doca_urom_worker_cmd_task_allocate_init(struct doca_urom_worker *worker_ctx,
uint64_t plugin, struct doca_urom_worker_cmd_task **task);
```

- `worker_ctx [in]` – a pointer to `doca_urom_worker` object
- `plugin [in]` – task plugin ID
- `task [out]` – set worker command new task
- Returns – `DOCA_SUCCESS` on success, error code otherwise

**doca_urom_worker_cmd_task_release**

Release worker command task.

```c
doca_error_t doca_urom_worker_cmd_task_release(struct doca_urom_worker_cmd_task *task);
```

- `task [in]` – worker task to release
- Returns – `DOCA_SUCCESS` on success, error code otherwise

**doca_urom_worker_cmd_task_set_plugin**

Set worker command task plugin ID. The plugin ID is created by the UROM service and the plugin host interface should hold it to create UROM worker command tasks.

```c
void doca_urom_worker_cmd_task_set_plugin(struct doca_urom_worker_cmd_task *task, uint64_t plugin);
```
- task [in] – worker task
- plugin [in] – task plugin to set

**doca_urom_worker_cmd_task_set_cb**

Set worker command task completion callback.

```c
void doca_urom_worker_cmd_task_set_cb(struct doca_urom_worker_cmd_task *task,
                                      doca_urom_worker_cmd_task_completion_cb_t cb);
```

- task [in] – worker task
- plugin [in] – task callback to set

**doca_urom_worker_cmd_task_get_payload**

Get worker command task payload. The plugin interface populates this buffer by plugin command structure. The payload size is the maximum message size in the DOCA UROM communication channel (the user can configure the size by calling `doca_urom_service_set_max_comm_msg_size()`). To update the payload buffer, the user should call `doca_buf_set_data()`.

```c
struct doca_buf *doca_urom_worker_cmd_task_get_payload(struct doca_urom_worker_cmd_task *task);
```

- task [in] – worker task
- Returns – a `doca_buf` that represents the task's payload

**doca_urom_worker_cmd_task_get_response**
Get worker command task response. To get the response's buffer, the user should call 
doca_buf_get_data().

```
struct doca_buf *doca_urom_worker_cmd_task_get_response(struct doca_urom_worker_cmd_task *
task);
```

- task [in] – worker task
- Returns – a doca_buf that represents the task's response

**doca_urom_worker_cmd_task_get_user_data**

Get worker command user data to populate. The data refers to the reserved data inside the task that the user can get when calling the completion callback. The maximum data size is 32 bytes.

```
void *doca_urom_worker_cmd_task_get_user_data(struct doca_urom_worker_cmd_task *task);
```

- task [in] – worker task
- Returns – a pointer to user data memory

**doca_urom_worker_cmd_task_as_task**

Convert a doca_urom_worker_cmd_task object into a DOCA task object.

After creating a worker command task and configuring it using setter methods (e.g., 
doca_urom_worker_cmd_task_set_plugin()) or as part of task allocation, the user should submit the task by calling doca_task_submit.

A typical invocation looks like doca_task_submit(doca_urom_worker_cmd_task_as_task(task)).
struct doca_task *doca_urom_worker_cmd_task_as_task(struct doca_urom_worker_cmd_task *task);

- task[in] – worker command task
- Returns – a pointer to the doca_task object on success, NULL otherwise

**doca_urom_domain**

An opaque structure representing a DOCA UROM domain context.

struct doca_urom_domain;

**doca_urom_domain_allgather_cb_t**

A callback for a non-blocking all-gather operation.

typedef doca_error_t (*doca_urom_domain_allgather_cb_t)(void *sbuf, void *rbuf, size_t msglen, void *coll_info, void **req);

- sbuf [in] – local buffer to send to other processes
- rbuf [in] – global buffer to include other process's source buffer
- msglen [in] – source buffer length
- coll_info [in] – collection info
- req [in] – allgather request data
- Returns – DOCA_SUCCESS on success, error code otherwise
**doca_urom_domain_req_test_cb_t**

A callback to test the status of a non-blocking allgather request.

```c
typedef doca_error_t (*doca_urom_domain_req_test_cb_t)(void *req);
```

- `req [in]` – allgather request data to check status
- Returns – `DOCA_SUCCESS` on success, `DOCA_ERROR_IN_PROGRESS` otherwise

**doca_urom_domain_req_free_cb_t**

A callback to free a non-blocking allgather request.

```c
typedef doca_error_t (*doca_urom_domain_req_free_cb_t)(void *req);
```

- Returns – `DOCA_SUCCESS` on success, error code otherwise

**doca_urom_domain_oob_coll**

Out-of-band communication descriptor for domain creation.

```c
struct doca_urom_domain_oob_coll {
    doca_urom_domain_allgather_cb_t allgather;
    doca_urom_domain_req_test_cb_t req_test;
    doca_urom_domain_req_free_cb_t req_free;
    void *coll_info;
    uint32_t n_oob_indexes;
    uint32_t oob_index;
};
```
- allgather – non-blocking allgather callback
- req_test – request test callback
- req_free – request free callback
- coll_info – context or metadata required by the OOB collective
- n_oob_indexes – number of endpoints participating in the OOB operation (e.g., number of client processes representing domain workers)
- oob_index – an integer value that represents the position of the calling processes in the given OOB operation. The data specified by src_buf is placed at the offset "oob_index*size" in the recv_buf.

**Note**

oob_index must be unique at every calling process and should be in the range [0:n_oob_indexes].

doca_urom_domain_create

Creates a UROM domain context. A domain is created in state DOCA_CTX_STATE_IDLE. After creation, a user may configure the domain using setter methods (e.g., doca_urom_domain_set_workers()). Before use, a domain must be transitioned to state DOCA_CTX_STATE_RUNNING using the doca_ctx_start() interface. A typical invocation looks like doca_ctx_start(doca_urom_domain_as_ctx(worker_ctx)).

doca_error_t doca_urom_domain_create(struct doca_urom_domain **domain_ctx);
- **domain_ctx [in/out]** – doca_urom_domain object to be created
- **Returns** – **DOCA_SUCCESS** on success, error code otherwise

### doca_urom_domain_destroy

Destroys a UROM domain context.

```c
struct doca_urom_domain *domain_ctx);
doca_error_t doca_urom_domain_destroy(struct doca_urom_domain *domain_ctx);
```

- **domain_ctx [in]** – doca_urom_domain object to be destroyed; it is created by doca_urom_domain_create()
- **Returns** – **DOCA_SUCCESS** on success, error code otherwise

### doca_urom_domain_set_workers

Sets the list of workers in the domain.

```c
struct doca_urom_domain *domain_ctx, uint64_t *domain_worker_ids, struct doca_urom_worker **workers, size_t workers_cnt);
doca_error_t doca_urom_domain_set_workers(struct doca_urom_domain *domain_ctx, uint64_t *
```

- **domain_ctx [in]** – doca_urom_domain object
- **domain_worker_ids [in]** – list of domain worker IDs
- **workers [in]** – an array of UROM worker contexts that should be part of the domain
- **workers_cnt [in]** – the number of workers in the given array
- **Returns** – **DOCA_SUCCESS** on success, error code otherwise
**doca_urom_domain_add_buffer**

Attaches local buffer attributes to the domain. It should be called after calling `doca_urom_domain_set_buffers_count()`.

The local buffer will be shared with all workers belonging to the domain.

```c
int doca_urom_domain_add_buffer(struct doca_urom_domain *domain_ctx, void *buffer, size_t buf_len, void *memh, size_t memh_len, void *mkey, size_t mkey_len);
```

- **domain_ctx [in]** – doca_urom_domain object
- **buffer [in]** – buffer ready for remote access which is given to the domain
- **buf_len [in]** – buffer length
- **memh [in]** – memory handle for the exported buffer. (should be packed)
- **memh_len [in]** – memory handle size
- **mkey [in]** – memory key for the exported buffer. (should be packed)
- **mkey_len [in]** – memory key size
- **Returns** – DOCA_SUCCESS on success, error code otherwise

**doca_urom_domain_set_oob**

Sets OOB communication info to be used for domain initialization.

```c
int doca_urom_domain_set_oob(struct doca_urom_domain *domain_ctx, struct doca_urom_domain_oob_coll *oob);
```

- **domain_ctx [in]** – doca_urom_domain object
- `oob [in]` – OOB communication info to set
- `Returns` – DOCA_SUCCESS on success, error code otherwise

### doca_urom_domain_as_ctx

Convert a `doca_urom_domain` object into a DOCA object.

```c
struct doca_ctx *doca_urom_domain_as_ctx(struct doca_urom_domain *domain_ctx);
```

- `domain_ctx[in]` – a pointer to `doca_urom_domain` object
- `Returns` – a pointer to the `doca_ctx` object on success, NULL otherwise

### Execution Model

DOCA UROM uses the DOCA Core Progress Engine as an execution model for service and worker contexts and tasks progress. For more details about it please refer to this [guide](#).

### UROM Building Blocks

This section explains the general concepts behind the fundamental building blocks to use when creating a DOCA UROM application and offloading functionality.

### Program Flow

**DPU**

**Launching DOCA UROM Service**

DOCA UROM service should be run before running the application on the host to offload commands to BlueField. For more information, refer to the [NVIDIA DOCA UROM Service Guide](#).
Host

Initializing UROM Service Context

1. Create service context: Establish a service context within the control plane alongside the progress engine.

2. Set service context attributes: Specific attributes of the service context are configured. The required attribute is `doca_dev`.

3. Start the service context: The service context is initiated by invoking the `doca_ctx_start` function.
   
   1. Discover BlueField availability: The UROM library identifies the available BlueField device.
   
   2. Connect to UROM service: The library establishes a connection to the UROM service. The connection process is synchronized, meaning that the host process and the BlueField service process are blocked until the connection is established.
   
   3. Perform lookup using UROM service file: A lookup operation is executed using the UROM service file. The path to this file should be specified in the `DOCA_UROM_SERVICE_FILE` environment variable. More information can be found in `doca_urom.h`.

4. Switch context state to `DOCA_CTX_STATE_RUNNING`: The context state transitions to `DOCA_CTX_STATE_RUNNING` at this point.

5. Service context waits for worker bootstrap requests: The service context is now in a state where it awaits and handles worker bootstrap requests.

```c
/* Create DOCA UROM service instance */
doca_urom_service_create(&service);

/* Connect service context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_service_as_ctx(service));
```
Picking UROM Worker Offload Functionality

Once the service context state is running, the application can call `doca_urom_service_get_plugins_list()` to get the available plugins on the local BlueField device where the UROM service is running.

The UROM service generates an identifier for each plugin and the application is responsible for forwarding this ID to the host plugin interface for sending commands and receiving notifications by calling `urom_<plugin_name>_init(<plugin_id>, <plugin_version>).`
Initializing UROM Worker Context

1. Create a service context and connect the worker context to DOCA Progress Engine (PE).

2. Set worker context attributes (in the example below worker plugin is RDMO).

3. Start worker context, submitting internally spawns worker requests on the service context.

4. Worker context state changes to DOCA_CTX_STATE_STARTING (this process is asynchronous).

5. Wait until the worker context state changes to DOCA_CTX_STATE_RUNNING:
   1. When calling `doca_pe_progress`, check for a response from the service context that the spawning worker on BlueField is done.
   2. If the worker is spawned on BlueField, connect to it and change the status to running.

```c
const struct doca_urom_service_plugin_info *rdmo_info;

/* Create DOCA UROM worker instance */
doca_urom_worker_create(&worker);

/* Connect worker context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_worker_as_ctx(worker));

/* Set worker attributes */
doca_urom_worker_set_service(worker, service);
doca_urom_worker_set_id(worker, worker_id);
doca_urom_worker_set_max_inflight_tasks(worker, nb_tasks);
doca_urom_worker_set_plugins(worker, rdmo_info->id);
doca_urom_worker_set_cpuset(worker, cpuset);

/* Start UROM worker context */
```
Offloading Plugin Task

Once the worker context state is `DOCA_CTX_STATE_RUNNING`, the worker is ready to execute offload tasks. The example below is for offloading an RDMO command.

1. Prepare RDMO task arguments (e.g., completion callback).

2. Call the task function from the plugin host interface.

3. Poll for completion by calling `doca_pe_progress`.

4. Get completion notification through the user callback.

```c
int ret;
struct doca_urom_worker *worker;
struct rdmo_result res = {0};
union doca_data cookie = {0};
size_t server_worker_addr_len;
ucp_address_t *server_worker_addr;

cookie.ptr = &res;
res.result = DOCA_SUCCESS;

ucp_worker_create(*ucp_context, &worker_params, server_ucp_worker);
ucp_worker_get_address(*server_ucp_worker, &server_worker_addr, &server_worker_addr_len);
/* Create and submit RDMO client init task */
urm_rdm0_task_client_init(worker, cookie, 0, server_worker_addr, server_worker_addr_len,
                         urm_rdm0_client_init_finished);
/* Wait for completion */
do {
    ret = doca_pe_progress(pe);
    ucp_worker_progress(*server_ucp_worker);
```
Initializing UROM Domain Context

1. Create a domain context on the control plane PE.
2. Set domain context attributes.
3. Start the domain context by calling `doca_ctx_start`.
   1. Exchange memory descriptors between all workers.
4. Wait until the domain context state is running.

```c
/* Create DOCA UROM domain instance */
doca_urom_domain_create(&domain);

/* Connect domain context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_domain_as_ctx(domain));

/* Set domain attributes */
doca_urom_domain_set_oob(domain, oob);
doca_urom_domain_set_workers(domain, worker_ids, workers, nb_workers);
doca_urom_domain_set_buffers_count(domain, nb_buffers);
for each buffer:
    doca_urom_domain_add_buffer(domain);

/* Start domain context */
doca_ctx_start(doca_urom_domain_as_ctx(domain));

/* Loop till domain state changes to running */
do {
    doca_pe_progress(pe);
    result = doca_ctx_get_state(doca_urom_domain_as_ctx(domain), &state);
} while (state == DOCA_CTX_STATE_STARTING && result == DOCA_SUCCESS);
```
Destroying UROM Domain Context

1. Request the domain context to stop by calling `doca_ctx_stop`.

2. Clean up resources by destroying the domain context.

```c
/* Request domain context stop */
doca_ctx_stop(doca_urom_domain_as_ctx(domain));

/* Destroy domain context */
doca_urom_domain_destroy(domain);
```

Destroying UROM Worker Context

1. Request the worker context stop by calling `doca_ctx_stop` and posting the destroy command on the service context.

2. Wait until a completion for the destroy command is received.
   1. Change worker state to idle.

3. Clean up resources.

```c
/* Stop worker context */
doca_ctx_stop(doca_urom_worker_as_ctx(worker));

/* Progress till receiving a completion */
do {
    doca_pe_progress(pe);
    doca_ctx_get_state(doca_urom_worker_as_ctx(worker), &state);
} while (state != DOCA_CTX_STATE_IDLE);

/* Destroy worker context */
doca_urom_worker_destroy(worker);
```
Destroying UROM Service Context

1. Wait for the completion of the UROM worker context commands.

2. Once all UROM workers have been successfully destroyed, initiate service context stop by invoking `doca_ctx_stop`.

3. Disconnect from the UROM service.

4. Perform resource cleanup.

```c
/* Handling workers teardown requests*/
do {
    doca_pe_progress(pe);
} while (!are_all_workers_exited);

/* Stop service context */
doca_ctx_stop(doca_urom_service_as_ctx(service));

/* Destroy service context */
doca_urom_service_destroy(service);
```

Plugin Development

Developing Offload Plugin on DPU

1. Implement `struct urom_plugin_iface` methods.

   1. The `open()` method initializes the plugin connection state and may create an endpoint to perform communication with other processes/workers.

```c
static doca_error_t urom_worker_rdmo_open(struct urom_worker_ctx *ctx)
{
    ucp_context_h ucp_context;
    ucp_worker_h ucp_worker;
```
2. The `addr()` method returns the address of the plugin endpoint generated during `open()` if it exists (e.g., UCX endpoint to communicate with other UROM workers).

3. The `worker_cmd()` method is used to parse and start work on incoming commands to the plugin.

```c
static doca_error_t urom_worker_rdmo_worker_cmd(struct urom_worker_ctx *ctx, ucs_list_link_t *cmd_list)
{
    struct urom_worker_rdmo_cmd *rdmo_cmd;
    struct urom_worker_cmd_desc *cmd_desc;
    struct urom_worker_rdmo *rdmo_worker = (struct urom_worker_rdmo *)ctx->plugin_ctx;

    struct urom_worker_rdmo *rdmo_worker;
    rdmo_worker = calloc(1, sizeof(*rdmo_worker));
    if (rdmo_worker == NULL)
        return DOCA_ERROR_NO_MEMORY;

    ctx->plugin_ctx = rdmo_worker;
    /* UCX transport layer initialization */
    .
    .
    .

    /* Create UCX worker Endpoint */
    ucp_worker_create(ucp_context, &worker_params, &ucp_worker);
    ucp_worker_get_address(ucp_worker, &rdmo_worker->ucp_data.worker_address,
                           &rdmo_worker->ucp_data.ucp_addrlen);

    /* Resources initialization */
    rdmo_worker->clients = kh_init(client);
    rdmo_worker->eps = kh_init(ep);
    /* Init completions list, UROM worker checks completed requests by calling progress() method */
    ucs_list_head_init(&rdmo_worker->completed_reqs);

    return DOCA_SUCCESS;
}
```
The progress() method is used to give CPU time to the plugin code to advance asynchronous tasks.

```c
while (ucs_list_is_empty(cmd_list)) {
    /* Get new RDMO command from the list */
    cmd_desc = ucs_list_extract_head(cmd_list, struct urom_worker_cmd_desc, entry);

    /* Unpack and deserialize RDMO command */
    urom_worker_rdmo_cmd_unpack(&cmd_desc->worker_cmd, cmd_desc->worker_cmd.len, &cmd);
    rdmo_cmd = (struct urom_worker_rdmo_cmd *)cmd->plugin_cmd;
    /* Handle command according to it's type */
    switch (rdmo_cmd->type) {
    case UROM_WORKER_CMD_RDMO_CLIENT_INIT:
        /* Handle RDMO client init command */
        status = urom_worker_rdmo_client_init_cmd(rdmo_worker, cmd_desc);
        break;
    case UROM_WORKER_CMD_RDMO_RQ_CREATE:
        /* Handle RDMO RQ create command */
        status = urom_worker_rdmo_rq_create_cmd(rdmo_worker, cmd_desc);
        break;
    .
    .
    .
    default:
        DOCA_LOG_INFO("Invalid RDMO command type: %lu", rdmo_cmd->type);
        status = DOCA_ERROR_INVALID_VALUE;
        break;
    }
    free(cmd_desc);
    if (status != DOCA_SUCCESS)
        return status;
}
return status;
}
```

4. The progress() method is used to give CPU time to the plugin code to advance asynchronous tasks.

```c
static doca_error_t urom_worker_rdmo_progress(struct urom_worker_ctx *ctx,
                                             ucs_list_link_t *notif_list)
{
    struct urom_worker_notif_desc *nd;
```
struct urom_worker_rdomo *rdomo_worker = (struct urom_worker_rdomo *)ctx->plugin_ctx;
/* RDMO UCP worker progress */
ucp_worker_progress(rdomo_worker->ucp_data.ucp_worker);
/* Check if completion list is empty */
if (ucs_list_is_empty(&rdomo_worker->completed_reqs))
    return DOCA_ERROR_EMPTY;
/* Pop completed commands from the list */
while (!ucs_list_is_empty(&rdomo_worker->completed_reqs)) {
    nd = ucs_list_extract_head(&rdomo_worker->completed_reqs, struct
        urom_worker_notif_desc, entry);
    ucs_list_add_tail(notif_list, &nd->entry);
}
return DOCA_SUCCESS;

5. The `notif_pack()` method is used to serialize notifications before they are sent back to the host.

2. Implement and expose the following symbols:

1. `doca_error_t urom_plugin_get_version(uint64_t *version);`

   Returns a compile-time constant value stored within the `.so` file and is used to verify that the host and DPU plugin versions are compatible.

2. `doca_error_t urom_plugin_get_iface(struct urom_plugin_iface *iface);`

   Get the `urom_plugin_iface` struct with methods implemented by the plugin.

3. Compile the user plugin as an `.so` file and place it where the UROM service can access it. For more details, refer to section "Plugin Discovery and Reporting" under the NVIDIA DOCA UROM Service Guide.

**Creating Plugin Host Task**

1. Allocate and init worker command task.

2. Populate payload buffer by task command.
3. Pack and serialize the command.

4. Set user data.

5. Submit the task.

doca_error_t urom_rdmo_task_client_init(struct doca_urom_worker *worker_ctx, union
doca_data cookie, uint32_t id, void *addr, uint64_t addr_len, urom_rdmo_client_init_finished cb)
{
    doca_error_t result;
    size_t pack_len = 0;
    struct doca_buf *payload;
    struct doca_urom_worker_cmd_task *task;
    struct doca_rdmo_task_data *task_data;
    struct urom_worker_rdmo_cmd *rdmo_cmd;

    /* Allocate task */
    doca_urom_worker_cmd_task_allocate_init(worker_ctx, rdmo_id, &task);
    /* Get payload buffer */
    payload = doca_urom_worker_cmd_task_get_payload(task);
    doca_buf_get_data(payload, (void **)&rdmo_cmd);
    doca_buf_get_data_len(payload, &pack_len);
    /* Populate commands attributes */
    rdmo_cmd->type = UROM_WORKER_CMD_RDMO_CLIENT_INIT;
    rdmo_cmd->client_init.id = id;
    rdmo_cmd->client_init.addr = addr;
    rdmo_cmd->client_init.addr_len = addr_len;
    /* Pack and serialize the command */
    urom_worker_rdmo_cmd_pack(rdmo_cmd, &pack_len, (void *)rdmo_cmd);
    /* Update payload data size */
    doca_buf_set_data(payload, rdmo_cmd, pack_len);
    /* Set user data */
    task_data = (struct doca_rdmo_task_data *
    doca_urom_worker_cmd_task_get_user_data(task);
    task_data->client_init_cb = cb;
    task_data->cookie = cookie;
    /* Set task plugin callback */
    doca_urom_worker_cmd_task_set_cb(task, urom_rdmo_client_init_completed);

    /* Submit task */
    doca_task_submit(doca_urom_worker_cmd_task_as_task(task));

    return DOCA_SUCCESS;
DOCA UROM Samples

This section provides DOCA UROM library sample implementations on top of BlueField.

The samples illustrate how to use the DOCA UROM API to do the following:

- Define and create a UROM plugin host and DPU versions for offloading HPC/AI tasks
- Build host applications that use the plugin to execute jobs on BlueField by the DOCA UROM service and workers

Info

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Sample Prerequisite

<table>
<thead>
<tr>
<th>Sample</th>
<th>Type</th>
<th>Prerequisite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandbox</td>
<td>Plugin</td>
<td>A plugin which offloads the UCX tagged send/receive API</td>
</tr>
<tr>
<td>Graph</td>
<td>Plugin</td>
<td>The plugin uses UCX data structures and UCX endpoint</td>
</tr>
<tr>
<td>UROM Ping Pong</td>
<td>Program</td>
<td>The sample uses the Open MPI package as a launcher framework to launch two processes in parallel</td>
</tr>
</tbody>
</table>

Running the Sample
1. Refer to the following documents:

- [NVIDIA DOCA Installation Guide for Linux](#) for details on how to install BlueField-related software
- [NVIDIA DOCA Troubleshooting Guide](#) for any issue you may encounter with the installation, compilation, or execution of DOCA samples

2. To build a given sample:

```bash
cd /opt/mellanox/doca/samples/doca_urom/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

### Info

The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. UROM Sample arguments:

<table>
<thead>
<tr>
<th>Sample</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UROM multi-workers bootstrap</td>
<td>-d, --device &lt;IB device name&gt;</td>
<td>IB device name</td>
</tr>
<tr>
<td>UROM Ping Pong</td>
<td>-d, --device &lt;IB device name&gt;</td>
<td>IB device name</td>
</tr>
<tr>
<td></td>
<td>-m, --message</td>
<td>Specify ping pong message</td>
</tr>
</tbody>
</table>

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```
UROM Plugin Samples

DOCA UROM plugin samples have two components. The first one is the host component which is linked with UROM host programs. The second is the DPU component which is compiled as an .so file and is loaded at runtime by the DOCA UROM service (daemon, workers).

To build a given plugin:

```
  cd /opt/mellanox/doca/samples/doca_urom/plugins/worker_<plugin_name>
  meson /tmp/build
  ninja -C /tmp/build
```

**Info**

The binary worker_<sample_name>.so file is created under /tmp/build/.

Graph

This plugin provides a simple example for creating a UROM plugin interface. It exposes only a single command loopback, sending a specific value in the command, and expects to receive the same value in the notification from UROM worker.

References:

- /opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/meson.build
- /opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/worker_graph.c
- /opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/worker_graph.h
**Sandbox**

This plugin provides a set of commands for using the offloaded ping pong communication operation.

References:

- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/meson.build`
- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/urom_sandbox.h`
- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/worker_sandbox.c`
- `/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/worker_sandbox.h`

**UROM Program Samples**

DOCA UROM program samples can run only on the host side and require at least one DOCA UROM service instance to be running on BlueField.

The environment variable should be set ` DOCA_UROM_SERVICE_FILE` to the path to the UROM service file.

**UROM Multi-worker Bootstrap**

This sample illustrates how to properly initialize DOCA UROM interfaces and use the API to spawn multiple workers on the same application process.

The sample initiates four threads as UROM workers to execute concurrently, alongside the main thread operating as a UROM service. It divides the workers into two groups based on their IDs, with odd-numbered workers in one group and even-numbered workers in the other.

Each worker executes the data loopback command by using the Graph plugin, sends a specific value, and expects to receive the same value in the notification.

The sample logic includes:

1. Opening DOCA IB device.
2. Initializing needed DOCA core structures.
3. Creating and starting UROM service context.
4. Initiating the Graph plugin host interface by attaching the generated plugin ID.
5. Launching 4 threads and for each of them:
   1. Creating and starting UROM worker context.
   2. Once the worker context switches to running, sending the loopback graph command to wait until receiving a notification.
   3. Verifying the received data.
   4. Waiting until an interrupt signal is received.
6. The main thread checking for pending jobs of spawning workers (4 jobs, one per thread).
7. Waiting until an interrupt signal is received.
8. The main thread checking for pending jobs of destroying workers (4 jobs, one per thread) for exiting.
9. Cleaning up and exiting.

References:

- /opt/mellanox/doca/samples/doca_urom/urom_multi_workers_bootstrap/urom_multi_workers_bootstrap
- /opt/mellanox/doca/samples/doca_urom/urom_common.c
- /opt/mellanox/doca/samples/doca_urom/urom_common.h

**UROM Ping Pong**
This sample illustrates how to properly initialize the DOCA UROM interfaces and use its API to create two different workers and run ping pong between them by using Sandbox plugin-based UCX.

The sample is using Open MPI to launch two different processes, one process as server and the second one as client, the flow is decided according to process rank.

The sample logic per process includes:

1. Initializing MPI.
2. Opening DOCA IB device.
3. Creating and starting UROM service context.
4. Initiating the Sandbox plugin host interface by attaching the generated plugin id.
5. Creating and starting UROM worker context.
6. Creating and starting domain context.
7. Through domain context, the sample processes exchange the worker's details to communicate between them on the BlueField side for ping pong flow.
8. Starting ping pong flow between the processes, each process offloading the commands to its worker on the BlueField side.
9. Verifying that ping pong is finished successfully.
10. Destroying the domain context.
11. Destroying the worker context.
12. Destroying the service context.

References:

- /opt/mellanox/doca/samples/doca_urom/urom_ping_pong/urom_ping_pong_sample.c
- /opt/mellanox/doca/samples/doca_urom/urom_ping_pong/urom_ping_pong_main.c
- /opt/mellanox/doca/samples/doca_urom/urom_ping_pong/meson.build
- /opt/mellanox/doca/samples/doca_urom/urom_common.c
- /opt/mellanox/doca/samples/doca_urom/urom_common.h
DOCA RDMA

This guide provides an overview and configuration instructions for the DOCA RDMA API.

Introduction

ℹ️ Note

This library is currently supported at beta level only.

DOCA RDMA enables direct access to the memory of remote machines, without interrupting the processing of their CPUs or operating systems. Avoiding CPU interruptions reduces context switching for I/O operations, leading to lower latency and higher bandwidth compared to traditional network communication methods.

DOCA RDMA library provides an API to execute the various RDMA operations.

This document is intended for software developers wishing to improve their applications by utilizing RDMA operations.

⚠️ Warning

RDMA operations should be executed over a secure channel in a production deployment, given the sensitivity that arises from the nature of the protocol.

Prerequisites
This library follows the architecture of a DOCA Core Context, it is recommended read the following sections before proceeding:

- **DOCA Core Execution Model**
- **DOCA Core Device**
- **DOCA Core Memory Subsystem**

**Environment**

DOCA RDMA-based applications can run either on the host machine or on the NVIDIA® BlueField® networking platform (DPU or SuperNIC).

**Architecture**

DOCA RDMA is a DOCA Context as defined by DOCA Core. See NVIDIA DOCA Core Context for more information.

DOCA RDMA consists of two connected sides, passing data between one another. This includes the option for one side to access the remote side's memory if the granted permissions allow it.

The connection between the two sides can either be based on InfiniBand (IB) or based on Ethernet using RoCE. Currently, only reliable connection (RC) transport type is supported.

DOCA RDMA leverages the Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

The supported operations that may be executed between the two sides, using DOCA RDMA, are:

- **Receive**
- **Send**
- **Send with immediate**
- **Write**
- **Write with immediate**
- Read
- Atomic compare and swap
- Atomic fetch and add
- Get remote DOCA Sync Event
- Set remote DOCA Sync Event
- Add remote DOCA Sync Event

**Objects**

**Device**

The RDMA library requires a DOCA device to operate. This device is used to utilize the connection between the peers in RDMA, access memory, and perform the different operations.

**Note**

The device must stay valid until the RDMA instance is destroyed.

**Memory Map**

Executing any DOCA RDMA operation in which data is passed between the peers requires creating a memory map (mmap) on each side.

- The mmap's permissions must include the relevant RDMA permission, according to the required RDMA operations. Tasks fail in case of insufficient permissions.
To allow the peer to execute RDMA operations, the mmap must be exported, using `doca_mmap_export_rdma()`, and passed to the peer (i.e., the side requesting the RDMA operation) where the remote mmap is created and used to access the memory.

**Buffer Inventory and Buffers**

Executing any DOCA RDMA operation, in which data is passed between the peers, requires using buffers, and thus requires a buffer inventory as well.

Each operation calls for a different set-up for the buffers in use, this is explicitly explained in the "Tasks" section.

**Configuration Phase**

To start using the library you need to first go through a configuration phase as described in [DOCA Core Context Configuration Phase](#).

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

**Configurations**

The context can be configured to match the application use case.

**Mandatory Configurations**

These configurations are mandatory and must be set by the application before attempting to start the context:

**Task Configurations**
At least one task/event type must be configured. See configuration of Tasks and/or Events.

**Permissions**

Different tasks require different permission to be set for both the RDMA and the mmap in use.

The following table summarizes the necessary RDMA and mmap permissions for each RDMA operation:

<table>
<thead>
<tr>
<th>DOCA RDMA task Types</th>
<th>Minimal Permissions</th>
<th>Should Export MMAP? (a)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>The Side Submitting the Task</strong></td>
<td><strong>The Peer</strong></td>
<td></td>
</tr>
<tr>
<td><strong>RDMA</strong></td>
<td><strong>MMAP</strong></td>
<td><strong>RDMA</strong></td>
</tr>
<tr>
<td>Read</td>
<td>Local read write</td>
<td>RDMA read</td>
</tr>
<tr>
<td>Get Remote Sync Event</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Write</td>
<td>Local read write</td>
<td>RDMA write</td>
</tr>
<tr>
<td>Write with Immediate</td>
<td>Local read write</td>
<td>RDMA write</td>
</tr>
<tr>
<td>Set Remote Sync Event</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Atomic</td>
<td>Local read write</td>
<td>RDMA atomic</td>
</tr>
<tr>
<td>Compare and Swap</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Atomic Fetch and Add</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Add Remote Sync Event</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Send</td>
<td>Local read write</td>
<td>-</td>
</tr>
<tr>
<td>DOCA RDMA task Types</td>
<td>Minimal Permissions</td>
<td>Should Export MMAP?</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Receive</td>
<td>Depending on the received task</td>
<td>Local read write</td>
</tr>
</tbody>
</table>

Note

(a) Refers to the peer. A side that only submits tasks is never required to export an mmap.

Optional Configurations

If these configurations are not set, a default value is used.

Users may edit the default properties of the RDMA instance using the `doca_rdma_set_<property>()`. The user may also query the default/set properties using `doca_rdma_cap_get_<property>(struct doca_rdma *, ...)` functions.

Info

The number of tasks that can be submitted in bulk is dependent on the properties `max_send_buf_list_len` and `send_queue_size`.

Refer to `Library Capability` for querying valid property values when configuring the library context.
Device Support

DOCA RDMA requires a device to operate. For picking a device, see DOCA Core Device Discovery.

As device capabilities may change in the future, it is recommended to query each `doca_devinfo` for its capabilities relevant to RDMA operations, using `doca_rdma_cap_*(struct doca_devinfo *, ...)` functions, and check whether the device is suitable for the required RDMA task types, using `doca_rdma_task_<task_type>_is_supported()`.

BlueField-2 and higher devices are supported:

- On the host, any `doca_dev` is supported
- On the BlueField Platform, applications must provide the library with SFs as a `doca_dev`. See NVIDIA OpenvSwitch Acceleration (OVS in DOCA) and BlueField DPU Scalable Function to see how to create SFs and connect them to the appropriate ports.

Info

An exception to this is when running RDMA on the DPA datapath, which currently only supports PFs.

Buffer Support

The DOCA RDMA library utilizes different buffer types, depending on the task and the buffer's purpose:

- Local mmap buffer
- Mmap from RDMA export buffer
- Mmap from PCIe export buffers

![Info]

This type of buffer can be used in an equivalent manner to local mmap buffers.

- Linked list buffer

For task-specific information, refer to section "Tasks".

**Establishing RDMA Connections**

To establish the communication between the peers and allow the execution of different DOCA RDMA tasks, the RDMA instances must be connected.

![Note]

This should be executed after `doca_ctx_start()` is called.

![Info]

Refer to section "State Machine" for more information.

There are two methods to establish RDMA connections as detailed in the following subsections.

**Exporting and Connecting RDMA**
Connecting the RDMA instances can be done by exporting each RDMA instance to the remote side to a blob by using `doca_rdma_export()`, transferring the blob to the opposite side, out-of-band (OOB), and providing it as input to the `doca_rdma_connect()` function on that side.

All in all, the configuration flow should be as presented in the following image:
Connecting Using RDMA CM Connection Flow

⚠️ Warning
The exported data contains sensitive information. Make sure to pass this data through a secure channel!

⚠️ Note
This connection method is not currently available for DPA/GPU data paths.

The RDMA CM (communication manager) flow uses the server/client scheme where one of the RDMA instances acts as a server for the second RDMA instance (client). The process for both RDMA instances is non-blocking, event driven, and governed by the progress engine (PE). The connection process is reported to both instances by callbacks which should be set with `doca_rdma_set_connection_state_callbacks()`.

There are four state callbacks:

- **Connection request callback** – This function is called by `doca_pe_progress()` when a connection request is received by an RDMA instance acting as server

- **Connection established callback** – This function is called by `doca_pe_progress()` when a connection is successfully established between server/client RDMA instances

- **Connection failure callback** – This function is called by `doca_pe_progress()` when a connection fails to be established between server/client RDMA instances

- **Connection disconnection callback** – This function is called by `doca_pe_progress()` when a connection disconnects either server/client RDMA instances
A typical connection flow would be as follows:

1. Prior to initiating a connection, the RDMA instance acting as server (i.e., RDMA server) must start active listening for a connection from a remote RDMA peer (using RDMA CM) to a specific port using `doca_rdma_start_listen_to_port()`. An RDMA server can stop listening for a connection from a remote RDMA peer (using RDMA CM) by using `doca_rdma_stop_listen_to_port()`.

2. The RDMA CM instance acting as client (i.e., RDMA client) can now perform an RDMA connection to the RDMA server. As first step it must create an address object by using `doca_rdma_addr_create()`. The parameters to this function correspond to the RDMA server details required to perform a connection. This object can be destroyed by using `doca_rdma_addr_destroy()`, and retrieve it from a connection with `doca_rdma_connection_get_addr()`.

3. The RDMA client can set the connection user data to include in each connection using `doca_rdma_connection_set_user_data()`, and retrieve it from a connection using `doca_rdma_connection_get_user_data()`.

4. The RDMA client can now perform a connection to the RDMA server using `doca_rdma_connect_to_addr()`. Depending on the network topology and configuration, the connection can be established with IPv4, IPv6, or GID.

5. The RDMA server receives a notification with a connection request through the previously set connection request callback function. The RDMA server can decide to accept the connection with `doca_rdma_connection_accept()` or reject the connection with `doca_rdma_connection_reject()`.

   - If the RDMA server rejects the connection or the connection cannot be successfully established, the RDMA server and RDMA client receive a notification through the connection failure callback function.
   - If the RDMA server accepts the connection and the connection can be successfully established, the RDMA server and RDMA client receive a notification through the connection established callback function.

6. After the RDMA operation is complete, either side can perform the disconnection process using `doca_rdma_connection_disconnect()`. The RDMA instance that did not initiate the disconnection process receives a notification through the disconnection callback function.
Using Bridge Functions to Accept CM Connection

DOCA RDMA offers connection functionality for user RDMA CM applications acting as a server that maintains a CM event channel and performs the listen process itself (i.e., not using DOCA RDMA connection flow functions).

The functionality must be executed as follows:

1. Server user application, using RDMA CM, must create an RDMA CM event channel, start active listening for a connection from a remote RDMA peer, and monitor the created CM event channel. These functions are performed without the use of the DOCA RDMA connection flow functions explained in section "Connecting Using RDMA CM Connection Flow".
2. Once the server user application received a connection request from a remote RDMA peer acting as client (using RDMA CM), it can call `doca_rdma_bridge_accept()`. This method acts as a bridge to accept a connection request from an application that performs the listen process by itself. The previously explained `doca_rdma_connection_accept()` cannot be used for this connection step as the user application needs to provide the RDMA CM id to accept the connection.

3. After the server side calls `doca_rdma_bridge_accept()` and confirms the client connection is successfully established, it should call `doca_rdma_bridge_established()` to finish the connection process from the server side. Only after a connection is established can DOCA RDMA tasks be allocated and submitted.

**Step 1:** Initiate the RDMA instance, server user application creates CM event channel and start listening for a remote RDMA peer connection request

```
RDMA Server

doca_rdma_create();
doca_rdma_as_ctx();
(set properties)
...
doca_ctx_start();

(Create and monitor RDMA CM event channel and start listening)
(Wait for connection request)
```

**Step 2:** Server receives connection request from a remote RDMA peer

```
RDMA Server

(connection request)
doca_rdma_bridge_accept();
```

**Step 3:** Server confirms connection establish and finalize bridge connection process

```
RDMA Server

(confirm connection established with peer)
doca_rdma_bridge_established();

(Finish configuration and start executing RDMA operations)
```

**Execution Phase**
This section describes execution on CPU using DOCA Core PE. For additional execution environments refer to section "Alternative Datapath Options".

**Tasks**

DOCA RDMA exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

**Note**

Most DOCA RDMA operations are not atomic and therefore it is imperative that the application handle synchronization appropriately. Moreover, successful completion of a write task, with or without immediate, does not guarantee data has been fully written to the remote address.

**Note**

All buffers used in DOCA RDMA tasks must remain valid until the task result is retrieved.

**Receive Task**

This task should be submitted prior to an expected submission of a send/send with immediate/write with immediate task on the remote side.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_rdma_task_receive_set_conf</code></td>
<td><code>doca_rdma_cap_task_receive_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_rdma_task_receive_set_conf</code></td>
<td>–</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Destination buffer  | Buffer pointing to a local memory address. The data is written to the buffer upon successful completion of the task. | • Linked list buffers are supported  
• The given destination buffer/list of buffers (given in dst_buf) must have a total length sufficient for the expected message size or the task would fail  
• The destination buffer is not mandatory and may be NULL when the requested DOCA RDMA task on the remote side is "write with immediate" or when the remote side is sending an empty message, with or without immediate (these tasks are presented later on in the "Tasks" section)  
• For the DOCA RDMA receive task, the length of each buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to in each buffer. The data length is increased in each buffer if data is written to it once the task is successfully completed. |

**Task Output**

Common output as described in DOCA Core Task.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result length</td>
<td>The length of data received by the task</td>
<td>Valid only on successful completion of the task</td>
</tr>
<tr>
<td>Result opcode</td>
<td>The opcode of the operation executed by the peer and received by the task</td>
<td>Valid only after task completion, irrespective of success</td>
</tr>
</tbody>
</table>
| Result immediate data | The immediate data received by the task                                     | • Valid only on successful completion of the task  
• Valid only when an immediate value was received (i.e. when the result opcode is DOCA_RDMA_OPCODE_RECV_SEND_WITH_IMM or DOCA_RDMA_OPCODE_RECV_WRITE_WITH_IMM) – may be retrieved using doca_rdma_task_receive_get_result_opcode() |

**Task Completion Success**

After the task completes successfully, the following happens:

- The received data is copied to the tail segment extending the original data segment
- The data length is increased by the received data length

**Task Completion Failure**

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated. Some buffers may be updated and some may remain unchanged.
Task Limitations

- The operation is not atomic and therefore it is imperative that the application handle synchronization appropriately
- The destination buffer must remain valid until task is completed
- The total length of the message must not exceed the `max_message_size` device capability
- The buffer list length must not exceed the `dst_buf_list_len` property of the DOCA RDMA receive task
- Other limitations are described in DOCA Core Task

Send Task

This task should be submitted to transfer a message to the remote side, and while the remote side is expecting a message and had submitted a receive task beforehand.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_rdma_task_send_set_conf</code></td>
<td><code>doca_rdma_cap_task_send_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_rdma_task_send_set_conf</code></td>
<td>–</td>
</tr>
<tr>
<td>Source buffer list length</td>
<td><code>doca_rdma_set_max_send_buf_list_len</code></td>
<td><code>doca_rdma_cap_get_max_send_buf_list_len</code></td>
</tr>
</tbody>
</table>

1. This configuration affects other tasks as well.

Task Input
Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Source buffer   | Buffer pointing to a local memory address and holds the data to be sent to the remote peer | ● Linked list buffers are supported  
● The total length of the given source buffer/list of buffers (in src_buf) may not exceed the expected message size on the remote side or the task fails  
● The source buffer is not mandatory and may be NULL when wishing to send an empty message  
● For the DOCA RDMA send task, the length of each buffer is considered as its data length |

**Task Output**

Common output as described in DOCA Core Task.

**Task Completion Success**

After the task completes successfully, the following happens:

- On successful completion of the task, the data in the source buffer will be sent to the remote side.

- It doesn't indicate that the data is received by the remote side.

**Task Completion Failure**

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated

**Task Limitations**

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The source buffer must remain valid until the task completes
- The total length of the message must not exceed the `max_message_size` device capability
- The buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in **DOCA Core Task**

**Send With Immediate Task**

This task should be submitted to transfer a message to the remote side with immediate data (a 32-bit value sent to the remote side, out-of-band), and while the remote side is expecting a message and had submitted a receive task beforehand.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_rdma_task_send_imm_set_conf</code></td>
<td><code>doca_rdma_cap_task_send_imm_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_rdma_task_send_imm_set_conf</code></td>
<td>_</td>
</tr>
<tr>
<td>Source buffer list length</td>
<td><code>doca_rdma_set_max_send_buf_list_len</code></td>
<td><code>doca_rdma_cap_get_max_send_buf_list_len</code></td>
</tr>
</tbody>
</table>

1. This configuration affects other tasks as well.
**Task Input**

Common input as described in **DOCA Core Task**.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Source buffer  | Buffer pointing to a local memory address and holding the data to be sent to the remote peer | • Linked list buffers are supported.  
• The total length of the given source buffer/list of buffers (in src_buf) may not exceed the expected message size on the remote side or the task fails.  
• The source buffer is not mandatory and may be NULL when wishing to send an empty message (may be relevant when wishing to keep a connection alive)  
• For the DOCA RDMA send task, the length of each buffer is considered as its data length |
| Immediate data | 32-bit value sent to the remote side, out-of-band                           | • The immediate_data field should be in Big-Endian format. This value is received by the remote side only once a receive task is completed successfully.                                                   |

**Task Output**

Common output as described in **DOCA Core Task**.

**Task Completion Success**

After the task completes successfully, the following happens:

- The data in the source buffer is sent to the remote side
- It does not indicate that the data is received by the remote side
**Task Completion Failure**

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user.

- If a non-fatal error occurs, the task status is updated.

**Task Limitations**

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.

- The source buffer must remain valid until the task completes.

- The total length of the message must not exceed the `max_message_size` device capability.

- The buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance.

- Other limitations are described in [DOCA Core Task](#).

**Read Task**

This task should be submitted when wishing to read data from remote memory (i.e., the memory on the remote side of the connection).

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_rdma_task_read_set_conf</code></td>
<td><code>doca_rdma_cap_task_read_is_supported</code></td>
</tr>
</tbody>
</table>
1. This configuration affects other tasks as well.

### Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source buffer</td>
<td>Points to a remote memory address and holds the data to be read</td>
<td>• Linked list buffers are not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The source buffer (src_buf) is not mandatory and may be NULL when</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wishing to read zero bytes (may be relevant when wishing to keep a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>connection alive)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The data is read only from the data section of the source buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The length of the source buffer is considered its data length. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td>length of data read from the source buffer depends on its data length</td>
</tr>
<tr>
<td></td>
<td></td>
<td>yet can not exceed the total length of the given destination buffer/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list of buffers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>That is, the actual length read depends on the minimal length</td>
</tr>
<tr>
<td></td>
<td></td>
<td>between the source and destination.</td>
</tr>
<tr>
<td>Destination buffer</td>
<td>Points to a local memory address. The data is written to the buffer upon</td>
<td>• Linked list buffers are supported</td>
</tr>
<tr>
<td></td>
<td>successful completion of the task</td>
<td>• The length of each destination buffer is considered as the length</td>
</tr>
<tr>
<td></td>
<td></td>
<td>from the end of the data section until the end of the buffer, as this</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is the available memory that can be written to in each buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• May be NULL if the source buffer has been set to NULL</td>
</tr>
</tbody>
</table>
**Task Output**

Common output as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result length</td>
<td>The length of data read by the task</td>
<td>Valid only on successful completion of the task</td>
</tr>
</tbody>
</table>

**Task Completion Success**

After the task completes successfully, the following happens:

- The read data is appended after the data section in the destination buffer, as it was prior to the task submission
- The data length is increased by the read data length

**Task Completion Failure**

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated and some may remain unchanged.

**Task Limitations**

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The task buffers must remain valid until task is completed
• The given source buffer length must not exceed the `max_message_size` device capability

• The destination buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance

• Other limitations are described in DOCA Core Task

Write Task

This task should be submitted when wishing to write data to remote memory (i.e., the memory on the remote side of the connection).

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_rdma_task_write_set_conf</code></td>
<td><code>doca_rdma_cap_task_write_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_rdma_task_write_set_conf</code></td>
<td>–</td>
</tr>
<tr>
<td>Source buffer list length</td>
<td><code>doca_rdma_set_max_send_buf_list_len</code></td>
<td><code>doca_rdma_cap_get_max_send_buf_list_len</code></td>
</tr>
</tbody>
</table>

1. This configuration affects other tasks as well. 

Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Source buffer      | Buffer pointing to a local memory address and holding the data to be written to the remote peer. | • Linked list buffers are supported  
<p>|                    |                                                                             | • The source buffer should point to a local memory address from which the data should be read. The data is read only from the data section of the source buffer. |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Destination buffer | Points to a remote memory address. The data is written to the buffer upon successful completion of the task. | - The source buffer (src_buf) is not mandatory and may be NULL when wishing to write zero bytes (may be relevant when wishing to keep a connection alive)  
   - The length of the buffer is considered as its data length  
   - Linked list buffers are not supported  
   - The destination buffer (dst_buf) should point to a remote memory address  
   - The length of the buffer is considered as its data length  
   - The length of the destination buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to  
   - The length of data written to the destination buffer depends on the total length of the given source buffer/list of buffers  
   - May be NULL if the source buffer was set to NULL |

**Task Output**

Common output as described in [DOCA Core Task](#).

**Task Completion Success**

After the task completes successfully, the following happens:

- The written data is appended after the data section in the destination buffer, as it was prior to the task submission.
- The data length is increased by the written data length
Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user.
- If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated and some may remain unchanged.

Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The task buffers must remain valid until task is completed.
- The total length of the given source buffer/list of buffers must be not exceed the max_message_size device capability.
- The source buffer list length must not exceed the max_send_buf_list_len property of the DOCA RDMA instance.
- Other limitations are described in DOCA Core Task.

Write With Immediate Task

This task should be submitted when wishing to write data to remote memory (i.e., the memory on the remote side of the connection).

Task Configuration
<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_rdma_task_write_imm_set_conf</code></td>
<td><code>doca_rdma_cap_task_write_imm_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_rdma_task_write_imm_set_conf</code></td>
<td>_</td>
</tr>
<tr>
<td>Source buffer list length</td>
<td><code>doca_rdma_set_max_send_buf_list_len</code></td>
<td><code>doca_rdma_cap_get_max_send_buf_list_len</code></td>
</tr>
</tbody>
</table>

1. This configuration affects other tasks as well. _

**Task Input**

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source buffer</td>
<td>Buffer pointing to a local memory address and holding the data to be written to the remote peer</td>
<td>• Linked list buffers are supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The source buffer should point to a local memory address from which the data should be read. The data is read only from the data section of the source buffer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The source buffer (<code>src_buf</code>) is not mandatory and may be NULL when wishing to write zero bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The length of the buffer is considered as its data length</td>
</tr>
<tr>
<td>Destination buffer</td>
<td>Points to a remote memory address. The data is written to the buffer upon successful completion of the task.</td>
<td>• Linked list buffers are not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The destination buffer (<code>dst_buf</code>) should point to a remote memory address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The length of the buffer is considered as its data length</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The length of the destination buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Immediate data</td>
<td>32-bit value sent to the remote side, out-of-band</td>
<td>• Should be in a Big-Endian format</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Value is received by the remote side only once a receive task completes successfully</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in [DOCA Core Task](#).

**Task Completion Success**

A write with immediate task succeeds only if the remote side is expecting the immediate and had submitted a receive task beforehand.

After the task completes successfully, the following happens:

- The written data is appended after the data section in the destination buffer, as it was prior to the task submission
- The data length is increased by the written data length.

**Task Completion Failure**

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user.
• If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated and some may remain unchanged.

**Task Limitations**

• The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.

• The tasks buffers must remain valid until task is completed

• The total length of the given source buffer/list of buffers must be not exceed the `max_message_size` device capability

• The source buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance

• Other limitations are described in **DOCA Core Task**

---

**Atomic Compare and Swap Task**

This task should be submitted when wishing to execute an 8-byte atomic read-modify-write operation on the remote memory (i.e., the memory on the remote side of the connection), in which the remote value is retrieved and updated if it is equal to a given value.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_rdma_task_atomic_cmp_swpr_set_conf</code></td>
<td><code>doca_rdma_cap_task_atomic_cmp_swpr_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_rdma_task_atomic_cmp_swpr_set_conf</code></td>
<td>-</td>
</tr>
</tbody>
</table>
Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination buffer</td>
<td>Buffer pointing to a remote memory address</td>
<td>• Linked list buffers are not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The destination buffer's data section must begin in a memory address aligned to 8-bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Only the first 8-bytes following the data address are considered for atomic operations</td>
</tr>
<tr>
<td>Compare data</td>
<td>64-bit value to be compared with the value in the destination buffer</td>
<td></td>
</tr>
<tr>
<td>Swap data</td>
<td>64-bit value to be swapped with the value in the destination buffer</td>
<td>• The value in the destination buffer is only swapped if the compared data value is equal to the value in the destination buffer. Otherwise, the destination buffer remains unchanged.</td>
</tr>
<tr>
<td>Result buffer</td>
<td>Buffer pointing to a local memory address. The original value of the destination buffer (before executing the atomic operation) is written to the buffer upon success.</td>
<td>• Linked list buffers are not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The result is written to the first 8-bytes following the data address</td>
</tr>
</tbody>
</table>

Task Output

Common output as described in DOCA Core Task.

Task Completion Success
After the task completes successfully, the following happens:

- If the compared values are equal, the value in the destination is swapped with the 64-bit value in the task's swap data field (swap_data)
- If the compared values are not equal, the value in the destination value remains unchanged
- The original value of the destination buffer (before executing the atomic operation) is written to the result buffer

**Task Completion Failure**

If the task fails midway:

- The context is stopped, and the task should be freed by the user

**Task Limitations**

- Task buffers must remain valid until task is completed
- Other limitations are described in DOCA Core Task

**Atomic Fetch and Add Task**

This task should be submitted when wishing to execute an 8-byte atomic read-modify-write operation on the remote memory (i.e., the memory on the remote side of the connection), in which the remote value is retrieved and increased by a given value.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_rdma_task_atomic_fetch_add_set_conf</td>
<td>doca_rdma_cap_task_atomic_fetch_add_is_supported</td>
</tr>
<tr>
<td>Description</td>
<td>API to Set the Configuration</td>
<td>API to Query Support</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_rdma_task_atomic_fetch_add_set_conf</td>
<td>-</td>
</tr>
</tbody>
</table>

**Task Input**

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destinaton buffer</td>
<td>Buffer that points to a remote memory address</td>
<td>• Linked list buffers are not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The destination buffer's data section must begin in a memory address aligned to 8-bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Only the first 8-bytes following the data address are considered for atomic operations</td>
</tr>
<tr>
<td>Add data</td>
<td>64-bit value to be added to the value in the destination buffer</td>
<td></td>
</tr>
<tr>
<td>Result buffer</td>
<td>Buffer pointing to a local memory address. The original value of the destination buffer (before executing the atomic operation) is written to the buffer upon success.</td>
<td>• Linked list buffers are not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The result is written to the first 8-bytes following the data address</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in DOCA Core Task.

**Task Completion Success**
After the task completes successfully, the following happens:

- The value in the destination is increased by the 64-bit value in the task's add data field
- The original value of the destination buffer (before executing the atomic operation) is written to the result buffer

**Task Completion Failure**

If the task fails midway:

- The context is stopped, and the task should be freed by the user

**Task Limitations**

- Task buffers must remain valid until task is completed
- Other limitations are described in DOCA Core Task

**Get Remote Sync Event Task**

This task should be submitted when wishing to get the value of a remote sync event.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_rdma_task_remote_net_sync_event_get_set_conf</code></td>
<td><code>doca_rdma_cap_task_remote_net_sync_event_get_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_rdma_task_remote_net_sync_event_get_set_conf</code></td>
<td>-</td>
</tr>
<tr>
<td>Destination buffer list length</td>
<td><code>doca_rdma_set_max_send_buf_list_len 6</code></td>
<td><code>doca_rdma_cap_get_max_send_buf_list_len</code></td>
</tr>
</tbody>
</table>
1. This configuration affects other tasks as well.

**Task Input**

Common input as described in **DOCA Core Task**.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync Event</td>
<td>The remote DOCA Sync Event to get its value</td>
<td></td>
</tr>
</tbody>
</table>
| Destination buffer | Points to a local memory address. The Sync Event value is written to the buffer upon successful completion of the task. | - Linked list buffers are supported  
- The length of the each buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to in each buffer |

**Task Output**

Common output as described in **DOCA Core Task**.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result length</td>
<td>The length of data received by the task</td>
<td>Valid only on successful completion of the task</td>
</tr>
</tbody>
</table>

**Task Completion Success**

After the task completes successfully, the following happens:

- The remote Sync Event value is appended after the data section in the destination buffer, as it was prior to the task submission
• The data length is increased by the retrieved data length

**Task Completion Failure**

If the task fails midway:

• If a fatal error occurs, the context is stopped, and the task should be freed by the user.

• If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated and some may remain unchanged.

**Task Limitations**

• The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.

• The destination buffer must remain valid until the task is completed.

• The destination buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance.

• Other limitations are described in [DOCA Core Task](#).

**Set Remote Sync Event Task**

This task should be submitted when wishing to set a remote sync event to a given value.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_rdma_task_remote_net_sync_event_notify_set_set_conf</code></td>
<td><code>doca_rdma_cap_task_remote_net_sync_event_notify_set_is_supported</code></td>
</tr>
<tr>
<td>Description</td>
<td>API to Set the Configuration</td>
<td>API to Query Support</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_rdma_task_remote_net_sync_event_notify_set_set_conf</code></td>
<td><code>doca_rdma_cap_get_max_send_buf_list_len</code></td>
</tr>
<tr>
<td>Source buffer list length</td>
<td><code>doca_rdma_set_max_send_buf_list_len</code> 7</td>
<td><code>doca_rdma_cap_get_max_send_buf_list_len</code></td>
</tr>
</tbody>
</table>

1. This configuration affects other tasks as well.

**Task Input**

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Source buffer | Points to a local memory address from which the Sync Event should be retrieved | • Linked list buffers are supported  
• The data is retrieved only from the buffer data section, until 8-bytes  
• The length of the source buffer is considered its data length. The length of data retrieved from the source buffer will not exceed the Sync Event value length (8-bytes). Thus, the actual length retrieved depends on the minimal length between the source buffer and Sync Event value length. |
| Sync Event    | The remote DOCA Sync Event to get its value                                 |                                                                                           |

**Task Output**

Common output as described in DOCA Core Task.
**Task Completion Success**

After the task completes successfully, the following happens:

- The remote sync event value is set to the data in the source buffer

**Task Completion Failure**

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated, and the Sync Event value is undefined

**Task Limitations**

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The source buffer must remain valid until the task completes
- The source buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in [DOCA Core Task](#)

**Add Remote Sync Event Task**

This task should be submitted when wishing to atomically increase a remote sync event by a given value.

**Task Configuration**
<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_rdma_task_remote_net_sync_event_notify_add_set_conf</code></td>
<td><code>doca_rdma_cap_task_remote_net_sync_event_notify_add_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_rdma_task_remote_net_sync_event_notify_add_set_conf</code></td>
<td>–</td>
</tr>
</tbody>
</table>

**Task Input**

Common input as described in [DOCA Core Task](#).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync event</td>
<td>A remote Sync Event</td>
<td></td>
</tr>
<tr>
<td>Add data</td>
<td>64-bit value that is added to the Sync Event value</td>
<td></td>
</tr>
</tbody>
</table>
| Result buffer    | Buffer pointing to a local memory address. The original Sync Event value of the destination buffer (before executing the atomic operation) is written to the buffer upon success. | • Linked list buffers are not supported  
• The result is written to the first 8-bytes following the data address |

**Task Output**

Common output as described in [DOCA Core Task](#).

**Task Completion Success**

After the task completes successfully, the following happens:
The value of the remote sync event is increased by the 64-bit value in the task's add data field

- The original value of the remote sync event (before executing the operation) is written to the result buffer

### Task Completion Failure

If the task fails midway:

- The context is stopped, and the task should be freed by the user

### Task Limitations

- Result buffer must remain valid until task is completed
- Other limitations are described in [DOCA Core Task](#)

### Events

DOCA RDMA exposes asynchronous events to notify about changes that happen unexpectedly, according to DOCA Core architecture.

The only event DOCA RDMA exposes is common events as described in [DOCA Core Event](#).

### State Machine

The DOCA RDMA library follows the Context state machine as described in [DOCA Core Context State Machine](#).

The following section describes how to move states and what is allowed in each state.
Idle

In this state, it is expected that application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all tasks have been freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all tasks are completed and freed</td>
</tr>
</tbody>
</table>

Starting

This state cannot be reached.

Running

In this state, it is expected that application:

1. Connects the RDMA instances on both peers. Refer to section "Establishing RDMA Connections" for more information.

2. Performs an RDMA instance disconnection process if the connection was established using the RDMA CM flow. Refer to section "Connecting Using RDMA CM Connection Flow" for more information.
3. Performs a new connection of the RDMA instances on both peers after an RDMA instance disconnection process if the connection was established using the RDMA CM flow. Refer to section "Connecting Using RDMA CM Connection Flow" for more information.

4. Accepts and indicates an established RDMA connection if the listening and CM channel monitoring was done by the user application. Refer to section "Connecting Using RDMA CM Connection Flow" for more information.

5. Allocates and submits tasks.

6. Calls progress to complete tasks and/or receive events.

Allowed operations:

- Performing a connection between 2 peers
- Allocating previously configured task
- Submitting an allocated task
- Calling stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

**Stopping**

In this state, it is expected that application:

1. Calls progress to complete all inflight tasks (tasks complete with failure)

2. Frees any completed tasks

3. Performs an RDMA instance disconnection process if the connection was established using the RDMA CM flow. Refer to section "Connecting Using RDMA CM Connection Flow" for more information.
Allowed operations:

- Call progress

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Call progress and fatal error occurs</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop without freeing all tasks</td>
</tr>
</tbody>
</table>

Alternative Datapath Options

DOCA RDMA allows data path to be run on DPA or GPU.

DPA Datapath

DOCA offers the DOCA DPA library which provides a programming model for offloading communication-centric user code to run on the DPA processor on the BlueField DPU. For additional information on the DOCA DPA library.

Note

DOCA RDMA on DPA datapath supports local networks only (i.e., cross-network or routing is not supported).

The user can choose to run an RDMA operation on the DPA datapath by configuring the DOCA RDMA context used by the application in the following manner:

1. Obtain DOCA CTX by calling `doca_rdma_as_ctx()`.

2. Set the datapath of the context to DPA by calling `doca_ctx_set_datapath_on_dpa()`. For additional information, refer to DOCA Core Alternative Data Path.
3. Finish context configuration and start the context by calling `doca_ctx_start()`. For additional information, refer to DOCA Context.

After configuring the datapath, the user can obtain a DPA handle for the DOCA RDMA context by calling `doca_rdma_get_dpa_handle()`.

The DPA handle can be used by the DOCA DPA library for datapath operations. For additional information, refer to DOCA DPA Communication Model.

**GPU Datapath**

DOCA offers the DOCA GPUNetIO library which provides a programming model for offloading the orchestration of the communication to a GPU CUDA kernel. For additional information on the DOCA GPUNetIO library.

The user can choose to run an RDMA operation on the GPU datapath by configuring the DOCA RDMA context used by the application in the following manner:

1. Obtain DOCA CTX by calling `doca_rdma_as_ctx()`.

2. Set the datapath of the context to GPU by calling `doca_ctx_set_datapath_on_gpu()`. For additional information, refer to DOCA Core Alternative Data Path.

3. Finish context configuration and start the context by calling `doca_ctx_start()`. For additional information, refer to DOCA Core Context.

After configuring the datapath, the user can obtain a GPU handle for the DOCA RDMA context by calling `doca_rdma_get_gpu_handle()`.

The GPU handle must be passed to a GPU CUDA kernel so the DOCA GPUNetIO CUDA device functions can execute datapath operations. For additional information, refer to DOCA GPUNetIO device functions.

**DOCA RDMA Samples**

These samples illustrate how to use the DOCA RDMA API to execute DOCA RDMA operations.
Running the Samples

1. Refer to the following documents:

   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_rdma/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample usage:

   - Common arguments
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d, --device</td>
<td>IB device name (optional). If not provided, a random IB device is assigned.</td>
</tr>
<tr>
<td>-ld, --local-descriptor-path</td>
<td>Local descriptor file path that includes the local connection information to be copied to the remote program</td>
</tr>
<tr>
<td>-re, --remote-descriptor-path</td>
<td>Remote descriptor file path that includes the remote connection information to be copied from the remote program</td>
</tr>
<tr>
<td>-m, --mmap-descriptor-path</td>
<td>Remote descriptor file path that includes the remote mmap connection information to be copied from the remote program</td>
</tr>
<tr>
<td>-g, --gid-index</td>
<td>GID index for DOCA RDMA (optional)</td>
</tr>
</tbody>
</table>

- **Sample-specific arguments**

<table>
<thead>
<tr>
<th>Sample</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDMA Read Responder</td>
<td>-r, --read-string</td>
<td>String to read (optional). If not provided, &quot;Hi DOCA RDMA!&quot; is defined.</td>
</tr>
<tr>
<td>RDMA Send RDMA Send Immediate</td>
<td>-s, --send-string</td>
<td>String to read (optional). If not provided, &quot;Hi DOCA RDMA!&quot; is defined.</td>
</tr>
<tr>
<td>RDMA Write Requester</td>
<td>-w, --write-string</td>
<td>String to read (optional). If not provided, &quot;Hi DOCA RDMA!&quot; is defined.</td>
</tr>
</tbody>
</table>

4. For additional information per sample, use the -h option:

```
/tmp/build/<sample_name> -h
```

**Samples**
Each sample presents a connection between two peers, transferring data from one to another, using a different RDMA operation in each sample. For more information on the available RDMA operations, refer to section "Tasks".

Each sample is comprised of two executables, each running on a peer.

The samples can run on either DPU or host, as long as the chosen peers have a connection between them.

**Note**

Prior to running the samples, ensure that the chosen devices, selected by the device name and the GID index, are set correctly and have a connection between one another. In each sample, it is the user's responsibility to copy the descriptors between the peers.

Most of the samples follow the following main basic steps:

1. Allocating resources:
   1. Locating and opening a device. The chosen device is one that supports the tasks relevant for the sample. If the sample requires no task, any device may be chosen.
   2. Creating a local MMAP and configuring it (including setting the MMAP memory range and relevant permissions)
   3. Creating a DOCA PE
   4. Creating an RDMA instance and configuring it (including setting the relevant permissions)
   5. Connecting the RDMA context to the PE

2. Sample-specific configurations:
   1. Configuring the tasks relevant to the sample, if any. Including:
1. Setting the number of tasks for each task type.

2. Setting callback functions for each task type, with the following logic:

   1. Successful completion callback:
      
      1. Verifying the data received from the remote, if any, is valid.
      2. Printing the transferred data.
      3. Freeing the task and task-specific resources (such as source/destination buffers).
      4. If an error occurs in steps a. and b., update the error that was encountered.

   2. Failed completion callback:
      
      1. Update the error that was encountered.

   Note
   
   If the context is not in idle state, only the first error in the flow is saved.

5. Decreasing the number of remaining tasks and stopping the context once it reaches 0.
2. Freeing the task and task-specific resources (such as source/destination buffers).

3. Decreasing the number of remaining tasks and stopping the context once it reaches 0.

2. Setting a state change callback function, with the following logic:

- Once the context moves to Starting state (can only be reached from Idle), export and connect the RDMA and, in some samples, export the local mmap or the sync event.

**Note**

During this step, the user is responsible for copying the descriptors between the two peers.

**Note**

The descriptors are to be read and used only by the peer, using the relevant DOCA functions (the descriptors contain encoded data).

- Once the context moves to Running state (can only be reached from Starting state in RDMA samples):
  - In some samples, only print a log and wait for the peer, or synchronize events
  - In other samples, prepare and submit a task:
    1. If needed, create an mmap from the received exported mmap descriptor, passed from the peer.
2. Request the required buffers from the buffer inventory.

3. Allocate and initiate the required task, together with setting the number of remaining tasks parameter as the task’s user data.

4. Submit the task.
   - Once the context moves to Stopping state, print a relevant log.
   - Once the context moves to Idle state:
     1. Print a relevant log.
     2. Send update that the main loop may be stopped.

3. Setting the program’s resources as the context user data to be used in callbacks.

4. Creating a buffer inventory and starting it.

5. Starting the context.

**Info**

After starting the context, the state change callback function is called by the PE which executes the relevant steps.

**Info**

In a successful run, each section is executed in the order they are presented in section 2.b.

6. Progressing the PE until the context returns to Idle state and the main loop may be stopped, either because of a run in which all tasks have been completed, or due to a
fatal error.

7. Cleaning up the resources.

RDMA Read

RDMA Read Requester

This sample illustrates how to read from a remote peer (the responder) using DOCA RDMA.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.

2. A read task is configured for this sample.

3. In this sample, data is read from the peer, verified to be valid, and printed in the successful task completion callback.

4. The local mmap is not exported as the peer does not intend to access it.

5. To read from the peer, a remote mmap is created from the peer's exported mmap.

Reference:

- /opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/rdma_read_requester_sample.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/rdma_read_requester_main.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/meson.build

RDMA Read Responder

This sample illustrates how to set up a remote peer for a DOCA RDMA read request.

The sample logic is as presented in the General Sample Steps, with attention to the following:
1. The permissions for both the local mmap and the RDMA instance in this sample allow for RDMA read.

2. No tasks are configured for this sample, and thus no tasks are prepared and submitted, nor are there task completion callbacks.

3. The local mmap is exported to the remote memory to allow it to be used by the peer for RDMA read.

4. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- /opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/rdma_read_responder_sample.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/rdma_read_responder_main.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/meson.build

**RDMA Write**

**RDMA Write Requester**

This sample illustrates how to write to a remote peer (the responder) using DOCA RDMA.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.

2. A write task is configured for this sample.

3. In this sample, data is written to the peer and printed in the successful task completion callback.

4. The local mmap is not exported as the peer does not intend to access it.

5. To write to the peer, a remote mmap is created from the peer's exported mmap.
Reference:

- /opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/rdma_write_requester_sample.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/rdma_write_requester_main.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/meson.build

**RDMA Write Responder**

This sample illustrates how to set up a remote peer for a DOCA RDMA write request.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for both the local mmap and the RDMA instance in this sample allow for RDMA write.

2. No tasks are configured for this sample, and thus no tasks are prepared and submitted, nor are there task completion callbacks. In this sample, the data written to the memory of the responder is printed once the context state is changed to Running, using the state change callback. This is done only after receiving input from the user, indicating that the requester had finished writing.

3. The local mmap is exported to the remote memory to allow it to be used by the peer for RDMA write.

4. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- /opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/rdma_write_responder_sample.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/rdma_write_responder_main.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/meson.build
RDMA Write Immediate

**RDMA Write Immediate Requester**

This sample illustrates how to write to a remote peer (the responder) using DOCA RDMA along with a 32-bit immediate value which is sent OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.

2. A write with immediate task is configured for this sample.

3. In this sample, data is written to the peer and printed in the successful task completion callback.

4. The local mmap is not exported as the peer does not intend to access it.

5. To write to the peer, a remote mmap is created from the peer's exported mmap.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/rdma_write_immediate_req`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/rdma_write_immediate_req`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/meson.build`

**RDMA Write Immediate Responder**

This sample illustrates how the set up a remote peer for a DOCA RDMA write request whilst receiving a 32-bit immediate value from the peer's OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for both the local mmap and the RDMA instance in this sample allow for RDMA write.
2. A receive task is configured for this sample to retrieve the immediate value. Failing to submit a receive task prior to the write with immediate task results in a fatal failure.

3. In this sample, the successful task completion callback also includes:

   1. Checking the result opcode, to verify that the receive task has completed after receiving a write with immediate request.

   2. Verifying the data written to the memory of the responder is valid and printing it, along with the immediate data received.

4. The local mmap is exported to the remote memory, to allow it to be used by the peer for RDMA write.

5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- /opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_responder/rdma_write_immediate_responder
- /opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_responder/rdma_write_immediate_responder
- /opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_responder/meson.build

**RDMA Send and Receive**

**RDMA Send**

This sample illustrates how to send a message to a remote peer using DOCA RDMA.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.

2. A send task is configured for this sample.
3. In this sample, the data sent is printed during the task preparation, not in the successful task completion callback.

4. The local mmap is not exported as the peer does not intend to access it.

5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- /opt/mellanox/doca/samples/doca_rdma/rdma_send/rdma_send_sample.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_send/rdma_send_main.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_send/meson.build

**RDMA Receive**

This sample illustrates how the remote peer can receive a message sent by the peer (the sender).

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.

2. A receive task is configured for this sample to retrieve the sent data. Failing to submit a receive task prior to the send task results in a fatal failure.

3. In this sample, data is received from the peer verified to be valid and printed in the successful task completion callback.

4. The local mmap is not exported as the peer does not intend to access it.

5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- /opt/mellanox/doca/samples/doca_rdma/rdma_receive/rdma_receive_sample.c
RDMA Send and Receive with Immediate

RDMA Send with Immediate

This sample illustrates how to send a message to a remote peer using DOCA RDMA along with a 32-bit immediate value which is sent OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.

2. A send with immediate task is configured for this sample.

3. In this sample, the data sent is printed during the task preparation, not in the successful task completion callback.

4. The local mmap is not exported as the peer does not intend to access it.

5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- /opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/rdma_send_immediate_sample.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/rdma_send_immediate_main.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/meson.build

RDMA Receive with Immediate
This sample illustrates how the remote peer can receive a message sent by the peer (the sender) while also receiving a 32-bit immediate value from the peer's OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.

2. A receive task is configured for this sample to retrieve the sent data and the immediate value. Failing to submit a receive task prior to the send with immediate task results in a fatal failure.

3. In this sample, the successful task completion callback also includes:
   1. Checking the result opcode, to verify that the receive task has completed after receiving a sent message with an immediate.
   2. Verifying the data received from the peer is valid and printing it along with the immediate data received.

4. In this sample, data is received from the peer verified to be valid and printed in the successful task completion callback.

5. The local mmap is not exported as the peer does not intend to access it.

6. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- /opt/mellanox/doca/samples/doca_rdma/rdma_receiveImmediate/rdma_receiveImmediate_sample.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_receiveImmediate/rdma_receiveImmediate_main.c
- /opt/mellanox/doca/samples/doca_rdma/rdma_receiveImmediate/meson.build

**RDMA Remote Sync Event**
This sample illustrates how to synchronize between local sync event and a remote sync event DOCA RDMA.

**RDMA Remote Sync Event Requester**

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.

2. A "remote net sync event notify set" task is configured for this sample.
   - For this task, the successful task completion callback has the following logic:
     1. Printing an info log saying the task was successfully completed and a specific successful completion log for the task.
     2. Decreasing the number of remaining tasks. Once 0 is reached:
        1. Freeing the task and task-specific resources.
        2. Stopping the context.
   - For this task, the failed task completion callback stops the context even when the number of remaining tasks is different than 0 (since the synchronization between the peers would fail).

3. A "remote net sync event get" task is configured for this sample.
   - For this task, the successful task completion callback also includes:
     1. Resubmitting the task, until a value greater than or equal to the expected value is retrieved.
     2. Once such value is retrieved, submitting a "remote net sync event notify set" task to signal sample completion, including:
        1. Updating the successful completion message accordingly.
        2. Increasing the number of submitted tasks.
3. If an error was encountered, and the "remote net sync event notify set" task was not submitted, the task and task resources are freed.
   - For this task, the failed task completion callback also includes freeing the "remote net sync event notify set" task and task resources.

4. The local mmap is not exported as the peer does not intend to access it.

5. No remote mmap is created as there is no intention to access the remote memory in this sample.

6. To synchronize events with the peer, a sync event remote net is created from the peer’s exported sync event.

7. Both tasks are prepared and submitted in the state change callback, once the context moves from starting to running.

8. The user data of the "remote net sync event get" task points to the "remote net sync event notify set" task.

Reference:

- /opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/rdma_sync_event_requester_sample
- /opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/rdma_sync_event_requester_main
- /opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/meson.build

**RDMA Remote Sync Event Responder**

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample is set to local read and write.

2. This sample includes creating a local sync event and exporting it to the remote memory to allow the peer to create a remote handle.

3. No tasks are configured for this sample, and thus no tasks are prepared and submitted, nor are there task completion callbacks. In this sample, the following
steps are executed once the context moves from starting to running, using the state change callback:

1. Waiting for the sync event to be signaled from the remote side.
2. Notifying the sync event from the local side.
3. Waiting for completion notification from the remote side.

Reference:

- /opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/rdma_sync_event_responder_sar
- /opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/rdma_sync_event_responder_man
- /opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/meson.build
DOCA Ethernet

This guide provides an overview and configuration instructions for the DOCA ETH API.

Introduction

**Note**

The DOCA Ethernet library is supported at alpha level.

DOCA ETH comprises of two APIs, DOCA ETH RXQ and DOCA ETH TXQ. The control path is always handled on the host/DPU CPU side by the library. The datapath can be managed either on the CPU by the DOCA ETH library or on the GPU by the GPUNetIO library.

DOCA ETH RXQ is an RX queue. It defines a queue for receiving packets. It also supports receiving Ethernet packets on any memory mapped by `doca_mmap`.

The memory location to which packets are scattered is agnostic to the processor which manages the datapath (CPU/DPU/GPU). For example, the datapath may be managed on the CPU while packets are scattered to GPU memory.

DOCA ETH TXQ is an TX queue. It defines a queue for sending packets. It also supports sending Ethernet packets from any memory mapped by `doca_mmap`.

To free the CPU from managing the datapath, the user can choose to manage the datapath from the GPU. In this mode of operation, the library collects user configurations and creates a receive/send queue object on the GPU memory (using the DOCA GPU sub-device) and coordinates with the network card (NIC) to interact with the GPU processor.

Prerequisites

This library follows the architecture of a DOCA Core Context. It is recommended to read the following sections:
• **DOCA Core Execution Model**

• **DOCA Core Device**

• **DOCA Core Memory Subsystem**

• **DOCA Flow Programming Guide**

• **OpenvSwitch Offload**

• **BlueField DPU Scalable Function** (for using SF on DPU)

• **DOCA GPUNetIO** (for GPU datapath)

### Changes From Previous Releases

#### Changes in 2.8.0

The following subsection(s) detail the doca_eth library updates in version 2.8.0.

**Added**

- `doca_error_t doca_eth_rxq_set_notification_moderation(struct doca_eth_rxq *eth_rxq, uint16_t period_usec, uint16_t comp_count)`

- `doca_error_t doca_eth_txq_task_send_num_expand(struct doca_eth_txq *eth_txq, uint32_t task_send_num)`

- `doca_error_t doca_eth_txq_task_lso_send_num_expand(struct doca_eth_txq *eth_txq, uint32_t task_lso_send_num)`

- `doca_error_t doca_eth_txq_task_batch_send_num_expand(struct doca_eth_txq *eth_txq, uint16_t task_batches_num)`

- `doca_error_t doca_eth_txq_task_batch_lso_send_num_expand(struct doca_eth_txq *eth_txq, uint16_t task_batches_num)`

**Changed**

- `doca_eth_rxq_task_recv_allocate_init(struct doca_eth_rxq *eth_rxq, union doca_data user_data, struct doca_buf *pkt, struct doca_eth_rxq_task_recv **task_recv)`
docta_eth_rxq_task_recv_allocate_init(struct doca_eth_rxq *eth_rxq, struct doca_buf *pkt, union doca_data user_data, struct doca_eth_rxq_task_recv **task_recv)

Environment

DOCA ETH based applications can run either on the Linux host machine or on the NVIDIA® BlueField® DPU target. The following is required:

- Applications should run with root privileges
- To run DOCA ETH on the DPU, applications must supply the library with SFs as a doca_dev. See OpenvSwitch Offload and BlueField DPU Scalable Function to see how to create SFs and connect them to the appropriate ports.
- Applications need to use DOCA Flow to forward incoming traffic to DOCA ETH RXQ's queue. See DOCA Flow and DOCA ETH RXQ samples for reference.

ℹ️ Info

Make sure the system has free huge pages for DOCA Flow.

Architecture

DOCA ETH is comprised of two parts: DOCA ETH RXQ and DOCA ETH TXQ.

DOCA ETH RXQ

Operating Modes

DOCA ETH RXQ can operate in the three modes, each exposing a slightly different control/datapath.

Regular Receive
In this mode, the received packet buffers are managed by the user. To receive a packet, the user should submit a receive task containing a `doca_buf` to write the packet into.

The application uses this mode if it wants to:

- Run on CPU
- Manage the memory of received packet and the packet's exact place in memory
- Forward the received packets to other DOCA libraries

## Cyclic Receive

**Info**

This mode is supported only for CPU datapath.
In this mode, the library scatters packets to the packet buffer (supplied by the user as `doca_mmap`) in a cyclic manner. Packets acquired by the user may be overwritten by the library if not processed fast enough by the application.

In this mode, the user must provide DOCA ETH RXQ with a packet buffer to be managed by the library (see `doca_eth_rxq_set_pkt_buf()`). The buffer should be large enough to avoid packet loss (see `doca_eth_rxq_estimate_packet_buf_size()`).

The application uses this mode if:

- It wants to run on GPU
- It has a deterministic packet processing time, where a packet is guaranteed to be processed before the library overwrites it with a new packet
- It wants best performance

**Managed Memory Pool Receive**

- Create DOCA ETH RXQ with opened DOCA device
- Configure DOCA ETH RXQ, this includes:
  1. Setting its mode
  2. Setting the created DOCA mmap as its packet buffer
  3. Set DOCA CTX data path on GPU
- Start DOCA CTX
- Get GPU handle of DOCA ETH RXQ using `doca_eth_rxq_get_gpu_handle()`
- Use acquired GPU handle in DOCA GPU netIO to receive packets
- Open a DOCA device using capability functions
- Calculate required mmap size using `doca_eth_rxq_get_pkt_buffer_size()`
- Allocate memory with required size
- Create mmap and set its memory range to the allocated memory

**This mode is supported only for GPU datapath.**
In this mode, the library uses various optimizations to manage the packet buffers. Packets acquired by the user cannot be overwritten by the library unless explicitly freed by the application. Thus, if the application does not release the packet buffers fast enough, the library would run out of memory and packets would start dropping.

Unlike Cyclic Receive mode, the user can pass the packet to other libraries in DOCA with the guarantee that the packet is not overwritten while being processed by those libraries.

In this mode, the user must provide DOCA ETH RXQ with a packet buffer to be managed by the library (see `doca_eth_rxq_set_pkt_buf()`). The buffer should be large enough to avoid packet loss (see `doca_eth_rxq_estimate_packet_buf_size()`).

The application uses this mode if:

- It wants to run on CPU
- It has a deterministic packet processing time, where a packet is guaranteed to be processed before the library runs out of memory and packets start dropping
- It wants to forward the received packets to other DOCA libraries
- It wants best performance

**Info**

This mode is supported only for CPU datapath.
Working with DOCA Flow

In order to route incoming packets to the desired DOCA ETH RXQ, applications need to use DOCA Flow. Applications need to do the following:

- Create and start DOCA Flow on the appropriate port (device)
- Create pipes to route packets into
- Get the queue ID of the queue (inside DOCA ETH RXQ) using `doca_eth_rxq_get_flow_queue_id()`
- Add an entry to a pipe which routes packets into the RX queue (using the queue ID we obtained)
DOCA ETH TXQ

Operating Modes

DOCA ETH TXQ can only operate in one mode.

Regular Send

For the CPU datapath, the user should submit a send task containing a `doca_buf` of the packet to send.

For information regarding the datapath on the GPU, see DOCA GPUNetIO.
Offloads

DOCA ETH TXQ supports:

- **Large Segment Offloading (LSO)** – the hardware supports LSO on transmitted TCP packets over IPv4 and IPv6. LSO enables the software to prepare a large TCP message for sending with a header template (the application should provide this header to the library) which is updated automatically for every generated segment. The hardware segments the large TCP message into multiple TCP segments. Per each such segment, device updates the header template accordingly (see LSO Send Task).

- **L3/L4 checksum offloading** – the hardware supports calculation of checksum on transmitted packets and validation of received packet checksum. Checksum calculation is supported for TCP/UDP running over IPv4 and IPv6. (In case of tunneling, the hardware calculates the checksum of the outer header.) The hardware does not require any pseudo header checksum calculation, and the value placed in TCP/UDP checksum is ignored when performing the calculation. See doca_eth_txq_set_l3_chksum_offload()/doca_eth_txq_set_l4_chksum_offload().
Objects

- **doca_mmap** – in Cyclic Receive and Managed Memory Pool Receive modes, the user must configure DOCA ETH RXQ with packet buffer to write the received packets into as a `doca_mmap` (see DOCA Core Memory Subsystem).

- **doca_buf** – in Regular Receive mode, the user must submit receive tasks that includes a buffer to write the received packet into as a `doca_buf`. Also, in Regular Send mode, the user must submit send tasks that include a buffer of the packet to send as a `doca_buf` (see DOCA Core Memory Subsystem).

Configurations Phase

To start using the library, the user must first go through a configuration phase as described in [DOCA Core Context Configuration Phase](#).

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

- **Note**

  DOCA ETH in GPU datapath does not need to be associated with a DOCA PE (since the datapath is not on the CPU).

Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or the min/max value for it, refer to [Device Support](#).

Mandatory Configurations
These configurations are mandatory and must be set by the application before attempting to start the context.

**DOCA ETH RXQ**

- At least one task/event/event_batch type must be configured. Refer to Tasks/Events/Event Batch for more information.

- Max packet size (the maximum size of packet that can be received) must be provided at creation time of the DOCA ETH RXQ context

- Max burst size (the maximum number of packets that the library can handle at the same time) must be provided at creation time of the DOCA ETH RXQ context

- A device with appropriate support must be provided upon creation

- When in Cyclic Receive or Managed Memory Pool Receive modes, a doca_mmap must be provided in-order write the received packets into (see doca_eth_rxq_set_pkt_buf())

- In case of a GPU datapath, A DOCA GPU sub-device must be provided using doca_ctx_set_datapath_on_gpu()

**DOCA ETH TXQ**

- At least one task/task_batch type must be configured. Refer to Tasks/Task Batch for more information.

- Max burst size (the maximum number of packets that the library can handle at the same time) must be provided at creation time of the DOCA ETH TXQ context

- A device with appropriate support must be provided on creation

- In case of a GPU datapath, a DOCA GPU sub-device must be provided using doca_ctx_set_datapath_on_gpu()

**Optional Configurations**
The following configurations are optional. If they are not set, then a default value is used.

**DOCA ETH RXQ**

- RXQ mode – User can set the working mode using `doca_eth_rxq_set_type()`. The default type is Regular Receive.

- Max receive buffer list length – User can set the maximum length of buffer list/chain as a receive buffer using `doca_eth_rxq_set_max_recv_buf_list_len()`. The default value is 1.

**DOCA ETH TXQ**

- TXQ mode – User can set the working mode using `doca_eth_txq_set_type()`. The default type is Regular Send.

- Max send buffer list length – User can set the maximum length of buffer list/chain as a send buffer using `doca_eth_rxq_set_max_send_buf_list_len()`. The default value is 1.

- L3/L4 offload checksum – User can enable/disable L3/L4 checksum offloading using `doca_eth_txq_set_l3_chksum_offload()` or `doca_eth_txq_set_l4_chksum_offload()`. They are disabled by default.

- MSS – User can set MSS (maximum segment size) value for LSO send task/task_batch using `doca_eth_txq_set_mss()`. The default value is 1500.

- Max LSO headers size – User can set the maximum LSO headers size for LSO send task/task_batch using `doca_eth_txq_set_max_lso_header_size()`. The default value is 74.

**Device Support**

DOCA ETH requires a device to operate. For picking a device, see [DOCA Core Device Discovery](#).

To check if a device supports a specific mode, use the type capabilities functions (see `doca_eth_rxq_cap_is_type_supported()` and `doca_eth_txq_cap_is_type_supported()`).
Devices can allow the following capabilities:

- The maximum burst size
- The maximum buffer chain list (only for Regular Receive/Regular Send modes)
- The maximum packet size (only for DOCA ETH RXQ)
- L3/L4 checksum offloading capability (only for DOCA ETH TXQ)
- Maximum LSO message/header size (only for DOCA ETH TXQ)
- Wait-on-time offloading capability (only for DOCA ETH TXQ in GPU datapath)

**Buffer Support**

DOCA ETH support buffers (doca_mmap or doca_buf) with the following features:

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Send Task</th>
<th>LSO Send Task</th>
<th>Receive Task</th>
<th>Managed Receive Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local mmap buffer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from PCIe export buffer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from RDMA export buffer</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Linked list buffer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

For buffer support in the case of GPU datapath, see [DOCA GPUNetIO Programming Guide](#).

**Execution Phase**

This section describes execution on CPU (unless stated otherwise) using DOCA Core Progress Engine.
**Tasks**

DOCA ETH exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

**DOCA ETH RXQ**

**Receive Task**

This task allows receiving packets from a `doca_dev`.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>calling doca_eth_rxq_task_recv_set_conf()</td>
<td>doca_eth_rxq_cap_is_type_supported() checking support for Regular Receive mode</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>task_recv_num in doca_eth_rxq_task_recv_set_conf()</td>
<td>-</td>
</tr>
<tr>
<td>Max receive buffer list length</td>
<td>doca_eth_rxq_set_max_recv_buf_list_len() (default value is 1)</td>
<td>doca_eth_rxq_cap_get_max_recv_buf_list_len()</td>
</tr>
<tr>
<td>Maximal packet size</td>
<td>max_packet_size in doca_eth_rxq_create()</td>
<td>doca_eth_rxq_cap_get_max_packet_size()</td>
</tr>
</tbody>
</table>

**Task Input**

Common input as described in DOCA Core Task.

---

**Note**

For information regarding GPU datapath, see DOCA GPUNetIO.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet buffer</td>
<td>Buffer pointing to the memory where received packet are to be written</td>
<td>The received packet is written to the tail segment extending the data segment</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in [DOCA Core Task](#).

Additionally:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3 checksum result</td>
<td>Value indicating whether the L3 checksum of the received packet is valid or not</td>
<td>Can be queried using <code>doca_eth_rxq_task_recv_get_l3_ok()</code></td>
</tr>
<tr>
<td>L4 checksum result</td>
<td>Value indicating whether the L4 checksum of the received packet is valid or not</td>
<td>Can be queried using <code>doca_eth_rxq_task_recv_get_l4_ok()</code></td>
</tr>
</tbody>
</table>

**Task Completion Success**

After the task is completed successfully the following will happen:

- The received packet is written to the packet buffer
- The packet buffer data segment is extended to include the received packet

**Task Completion Failure**

If the task fails midway:

- The context enters stopping state
- The packet buffer `doca_buf` object is not modified
- The packet buffer contents may be modified

**Task Limitations**

All limitations described in [DOCA Core Task](#)

Additionally:

- The operation is not atomic.
- Once the task has been submitted, then the packet buffer should not be read/written to.

### DOCA ETH TXQ

**Send Task**

This task allows sending packets from a `doca_dev`.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>calling <code>doca_eth_txq_task_send_set_conf()</code></td>
<td><code>doca_eth_txq_cap_is_type_supported()</code> checking support for Regular Send mode</td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>task_send_num in doca_eth_txq_task_send_set_conf()</code></td>
<td>-</td>
</tr>
<tr>
<td>Max send buffer list length</td>
<td><code>doca_eth_txq_set_max_send_buf_list_len()</code> (default value is 1)</td>
<td><code>doca_eth_txq_cap_get_max_send_buf_list_len()</code></td>
</tr>
<tr>
<td>Description</td>
<td>API to Set the Configuration</td>
<td>API to Query Support</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>L3/L4 offload checksum</td>
<td><code>doca_eth_txq_set_l3_chksum_offload()</code>&lt;br&gt;<code>doca_eth_txq_set_l4_chksum_offload()</code>&lt;br&gt;Disabled by default.</td>
<td><code>doca_eth_txq_cap_is_l3_chksum_offload_supported()</code>&lt;br&gt;<code>doca_eth_txq_cap_is_l4_chksum_offload_supported()</code></td>
</tr>
</tbody>
</table>

**Task Input**

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet buffer</td>
<td>Buffer pointing to the packet to send</td>
<td>The sent packet is the memory in the data segment</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in DOCA Core Task.

**Task Completion Success**

The task finishing successfully does not guarantee that the packet has been transmitted onto the wire. It only signifies that the packet has successfully entered the device’s TX hardware and that the packet buffer `doca_buf` is no longer in the library's ownership and it can be reused by the application.

**Task Completion Failure**

If the task fails midway:

- The context enters stopping state
- The packet buffer `doca_buf` object is not modified
**Task Limitations**

- The operation is not atomic
- Once the task has been submitted, the packet buffer should not be written to
- Other limitations are described in [DOCA Core Task](#)

**LSO Send Task**

This task allows sending "large" packets (larger than MTU) from a `doca_dev` (hardware splits the packet into several packets smaller than the MTU and sends them).

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>calling <code>doca_eth_txq_task_lso_send_set_conf()</code></td>
<td><code>doca_eth_txq_cap_is_type_supported()</code> checking support for Regular Send mode</td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>task_lso_send_num in doca_eth_txq_task_lso_send_set_conf()</code></td>
<td>-</td>
</tr>
<tr>
<td>Max send buffer list length</td>
<td><code>doca_eth_txq_set_max_send_buf_list_len()</code> (default value is 1)</td>
<td><code>doca_eth_txq_cap_get_max_send_buf_list_len()</code></td>
</tr>
<tr>
<td>L3/L4 offload checksum</td>
<td><code>doca_eth_txq_set_l3_chksum_offload()</code></td>
<td><code>doca_eth_txq_cap_is_l3_chksum_offload_supported()</code></td>
</tr>
<tr>
<td></td>
<td><code>doca_eth_txq_set_l4_chksum_offload()</code> (disabled by default)</td>
<td><code>doca_eth_txq_cap_is_l4_chksum_offload_supported()</code></td>
</tr>
<tr>
<td>MSS</td>
<td><code>doca_eth_txq_set_mss()</code> (default value is 1500)</td>
<td>-</td>
</tr>
<tr>
<td>Max LSO headers size</td>
<td><code>doca_eth_txq_set_max_lso_header_size()</code> (default value is 74)</td>
<td><code>doca_eth_txq_cap_get_max_lso_header_size()</code></td>
</tr>
</tbody>
</table>
**Task Input**

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet payload buffer</td>
<td>Buffer that points to the &quot;large&quot; packet's payload (does not include headers) to send</td>
<td>The sent packet is the memory in the data segment</td>
</tr>
<tr>
<td>Packet headers buffer</td>
<td>Gather list that when combined includes the &quot;large&quot; packet's headers to send</td>
<td>See struct doca_gather_list</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in DOCA Core Task.

**Task Completion Success**

The task finishing successfully does not guarantee that the packet has been transmitted onto the wire. It only means that the packet has successfully entered the device's TX hardware and that the packet payload buffer and the packet headers buffer is no longer in the library's ownership and it can be reused by the application.

**Task Completion Failure**

If the task fails midway:

- The context enters stopping state
- The packet payload buffer `doca_buf` object and the packet header buffer `doca_gather_list` are not modified
**Task Limitations**

- The operation is not atomic
- Once the task has been submitted, the packet payload buffer and the packet headers buffer should not be written to
- Other limitations are described in **DOCA Core Task**

**Events**

DOCA ETH exposes asynchronous events to notify about changes that happen asynchronously, according to the DOCA Core architecture. See **DOCA Core Event**.

In addition to common events as described in **DOCA Core Event**, DOCA ETH exposes an extra events:

**DOCA ETH RXQ**

**Managed Receive Event**

This event allows receiving packets from a `doca_dev` (without requiring the application to manage the memory the packets are written to).

**Event Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the</td>
<td><code>doca_eth_rxq_event_managed_recv_register()</code></td>
<td><code>doca_eth_rxq_cap_is_type_supported()</code> checking support for Managed Memory Pool Receive mode</td>
</tr>
<tr>
<td>event</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Event Trigger Condition**

The event is triggered every time a packet is received.

**Event Success Handler**

The success callback (provided in the event registration) is invoked and the user is expected to perform the following:

- Use the `pkt` parameter to process the received packet
- Use `event_user_data` to get the application context
- Query L3/L4 checksum results of the packet
- Free the `pkt` (a `doca_buf` object) and return it to the library

![Note]

Not freeing the `pkt` may cause scenario where packets are lost.

**Event Failure Handler**

The failure callback (provided in the event registration) is invoked, and the following happens:

- The context enters stopping state
- The `pkt` parameter becomes NULL
- The `event_user_data` parameter contains the value provided by the application when registering the event
**DOCA ETH TXQ**

**Error Send Packet**

This event is relevant when running DOCA ETH on GPU datapath (see DOCA GPUNetIO). It allows detecting failure in sending packets.

**Event Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event</td>
<td><code>doca_eth_txq_gpu_event_error_send_packet_register()</code></td>
<td>Always supported</td>
</tr>
</tbody>
</table>

**Event Trigger Condition**

The event is triggered when sending a packet fails.

**Event Handler**

The callback (provided in the event registration) is invoked and the user can:

- Get the position (index) of the packet that TXQ failed to send

**Notify Send Packet**

This event is relevant when running DOCA ETH on GPU datapath (see DOCA GPUNetIO). It notifies user every time a packet is sent successfully.
**Event Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event</td>
<td>doca_eth_txq_gpu_event_notify_send_packet_register()</td>
<td>Always supported</td>
</tr>
</tbody>
</table>

**Event Trigger Condition**

The event is triggered when sending a packet fails.

**Event Handler**

The callback (provided in the event registration) is invoked and the user can:

- Get the position (index) of the packet was sent
- Timestamp of sending the packet

**Task Batch**

DOCA ETH exposes asynchronous task batches that leverage the BlueField Platform hardware according to the DOCA Core architecture.

**DOCA ETH RXQ**

There are no task batches in ETH RXQ at the moment.

**DOCA ETH TXQ**
Send Task Batch

This is an extended task batch for Send Task which allows batched sending of packets from a doca_dev.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task batch</td>
<td>calling doca_eth_txq_task_batch_send_set_conf()</td>
<td>doca_eth_txq_cap_is_type_supported() checking support for Regular Send mode</td>
</tr>
<tr>
<td>Number of task batches</td>
<td>num_task_batches in</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>doca_eth_txq_task_batch_send_set_conf()</td>
<td>-</td>
</tr>
<tr>
<td>Max number of tasks per task</td>
<td>max_tasks_number in</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>doca_eth_txq_task_batch_send_set_conf()</td>
<td>-</td>
</tr>
<tr>
<td>Max send buffer list length</td>
<td>doca_eth_txq_set_max_send_buf_list_len() (default value is 1)</td>
<td>doca_eth_txq_cap_get_max_send_buf_list_len()</td>
</tr>
<tr>
<td>L3/L4 offload checksum</td>
<td>doca_eth_txq_set_l3_chksum_offload() doca_eth_txq_set_l4_chksum_offload()</td>
<td>Disabled by default. doca_eth_txq_cap_is_l3_chksum_offload_supported() doca_eth_txq_cap_is_l4_chksum_offload_supported()</td>
</tr>
</tbody>
</table>

Task Input

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks number</td>
<td>Number of send tasks &quot;behind&quot; the task batch</td>
<td>This number equals the number of packets to send</td>
</tr>
<tr>
<td>Batch user data</td>
<td>User data associated for the task batch</td>
<td>-</td>
</tr>
<tr>
<td>Packets array</td>
<td>Pointer to an array of buffers pointing at the packets to send per task</td>
<td>The sent packet is the memory in the data segment of each buffer</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>User data array</td>
<td>Pointer to an array of user data per task</td>
<td>–</td>
</tr>
</tbody>
</table>

**Task Output**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status array</td>
<td>Pointer to an array of statuses per task of the finished task batch</td>
</tr>
</tbody>
</table>

**Task Completion Success**

A task batch is complete if all the send tasks finished successfully and all the packets entered the device's TX hardware. All packets in the "Packet array" are now in the ownership of the user.

**Task Completion Failure**

If a task batch fails, then one (or more) of the tasks associated with the task batch failed. The user can look at "Status array" to see which task/packet caused the failure.

Also, the following behavior is expected:

- The context enters stopping state
- The packet's `doca_buf` objects are not modified

**Task Limitations**

In addition to all the Send Task Limitations:

- Task batch completion occurs only when all the tasks are completed (no partial completion)
LSO Send Task Batch

This is an extended task batch for LSO Send Task which allows batched sending of LSO packets from a doca_dev.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task batch</td>
<td>Calling doca_eth_txq_task_batch_lso_send_set_conf()</td>
<td>doca_eth_txq_cap_is_type_supported() checking support for Regular Send mode</td>
</tr>
<tr>
<td>Number of task batches</td>
<td>num_task_batches in doca_eth_txq_task_batch_lso_send_set_conf()</td>
<td>-</td>
</tr>
<tr>
<td>Max number of tasks per task batch</td>
<td>num_task_batches in doca_eth_txq_task_batch_lso_send_set_conf()</td>
<td>-</td>
</tr>
<tr>
<td>Max send buffer list length</td>
<td>doca_eth_txq_set_max_send_buf_list_len() (default value is 1)</td>
<td>doca_eth_txq_cap_get_max_send_buf_list_len()</td>
</tr>
<tr>
<td>L3/L4 offload checksum</td>
<td>doca_eth_txq_set_l3_chksum_offload()</td>
<td>doca_eth_txq_cap_is_l3_chksum_offload_supported()</td>
</tr>
<tr>
<td></td>
<td>doca_eth_txq_set_l4_chksum_offload()</td>
<td>doca_eth_txq_cap_is_l4 chksum_offload_supported()</td>
</tr>
<tr>
<td></td>
<td>Disabled by default.</td>
<td></td>
</tr>
<tr>
<td>MSS</td>
<td>doca_eth_txq_set_mss() (default value is 1500)</td>
<td>-</td>
</tr>
<tr>
<td>Max LSO headers size</td>
<td>doca_eth_txq_set_max_lso_header_size() (default value is 74)</td>
<td>doca_eth_txq_cap_get_max_lso_header_size()</td>
</tr>
</tbody>
</table>

Task Input
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks number</td>
<td>Number of send tasks &quot;behind&quot; the task batch</td>
<td>This number equals the number of packets to send</td>
</tr>
<tr>
<td>Batch user data</td>
<td>User data associated for the task batch</td>
<td>-</td>
</tr>
<tr>
<td>Packets payload array</td>
<td>Pointer to an array of buffers pointing at the &quot;large&quot; packet's payload to send per task</td>
<td>The sent packet payload is the memory in the data segment of each buffer</td>
</tr>
<tr>
<td>Packets headers array</td>
<td>Pointer to an array of gather lists, each of which when combined assembles a &quot;large&quot; packet's headers to send per task</td>
<td>See struct doca_gather_list</td>
</tr>
<tr>
<td>User data array</td>
<td>Pointer to an array of user data per task</td>
<td>-</td>
</tr>
</tbody>
</table>

**Task Output**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status array</td>
<td>Pointer to an array of status per task of the finished task batch</td>
</tr>
</tbody>
</table>

**Task Completion Success**

A task batch is complete if all the LSO send tasks finished successfully and all the packets entered the device's TX hardware. All packet payload in "Packets payload array" and packet headers in "Packets headers array" are now in the ownership of the user.

**Task Completion Failure**

If a task batch fails, then one (or more) of the tasks associated with the task batch failed, and the user can look at the "Status array" to try and figure out which task/packet caused the failure.
Also, the following behavior is expected:

- The context enters stopping state
- The packets payload \texttt{doca\_buf} objects are not modified
- The packets headers \texttt{doca\_gather\_list} objects are not modified

**Task Limitations**

In addition to all the LSO Send Task Limitations:

- Task batch completion happens only when all the tasks are completed (no partial completion)

**Event Batch**

DOCA ETH exposes asynchronous \texttt{event batches} to notify about changes that happen asynchronously.

**DOCA ETH RXQ**

**Managed Receive Event Batch**

This is an extended event batch for Managed Receive Event which allows receiving packets from a \texttt{doca\_dev} (without requiring the application to manage the memory the packets are written to).

**Event Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event batch</td>
<td>Calling \texttt{doca_eth_rxq_event_batc}</td>
<td>\texttt{doca_eth_rxq_cap_is_type_supported} (checking support for)</td>
</tr>
<tr>
<td>Description</td>
<td>API to Set the Configuration</td>
<td>API to Query Support</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>Max events number: Equal to the maximum number of completed events per event batch completion</td>
<td><code>h_managed_recv_register()</code></td>
<td>Managed Memory Pool Receive mode</td>
</tr>
<tr>
<td>Min events number: Equal to the minimum number of completed events per event batch completion</td>
<td><code>events_number_max in doca_eth_rxq_event_batch_manged_recv_register()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>events_number_min in doca_eth_rxq_event_batch_manged_recv_register()</code></td>
<td></td>
</tr>
</tbody>
</table>

**Event Trigger Condition**

The event batch is triggered every time a number of packets (number between "Min events number" and "Max events number") are received.

**Event Batch Success Handler**

The success callback (provided in the event of batch registration) is invoked and the user is expected to perform the following:

1. Identify the number of received packets by `events_number`.

2. Use the `pkt_array` parameter to process the received packets.

3. Use `event_batch_user_data` to get the application context.

4. Query the L3/L4 checksum results of the packets using `l3_ok_array` and `l4_ok_array`.

5. Free the buffers from `pkt_array` (a `doaca_buf` object) and return it to the library. This can be done in two ways:
   - Iterating over the buffers in `pkt_array` and freeing them using `doaca_buf_dec_refcount()`.
- Freeing all the buffers in pkt_array together (gives better performance) using doca_eth_rxq_event_batch_managed_recv_pkt_array_free().

**Event Batch Failure Handler**

The failure callback (provided in the event batch registration) is invoked, and the following happens:

- The context enters stopping state
- The pkt_array parameter is NULL
- The l3_ok_array parameter is NULL
- The l4_ok_array parameter is NULL
- The event_batch_user_data parameter contains the value provided by the application when registering the event

**DOCA ETH TXQ**

There are no event batches in ETH TXQ at the moment.

**State Machine**

The DOCA ETH library follows the Context state machine as described in DOCA Core Context State Machine.

The following section describes how to move to the state and what is allowed in each state.

**Idle**
In this state it is expected that application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to *Configurations*
- Starting the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Creating the context</td>
</tr>
</tbody>
</table>
| Running        | Calling stop after:
|                | • All tasks are completed and freed
|                | • All `doca_buf` objects returned by Managed Receive Event callback are freed |
| Stopping       | Calling progress until:
|                | • All tasks are completed and freed
|                | • All `doca_buf` objects returned by Managed Receive Event callback are freed |

**Starting**

This state cannot be reached.

**Running**
In this state it is expected that application will do the following:

- Allocate and submit tasks
- Call progress to complete tasks and/or receive events

Allowed operations:

- Allocate previously configured task
- Submit a task
- Call `doca_eth_rxq_get_flow_queue_id()` to connect the RX queue to DOCA Flow
- Call stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

**Stopping**

In this state, it is expected that application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks
- Frees `doca_buf` objects returned by Managed Receive Event callback

Allowed operations:

- Call progress

It is possible to reach this state as follows:
### Alternative Datapath Options

In addition to the CPU datapath (mentioned in Execution Phase), DOCA ETH supports running on GPU datapath. This allows applications to release the CPU from datapath management and allow low latency GPU processing of network traffic.

To export the handles, the application should call `doca_ctx_set_datapath_on_gpu()` before `doca_ctx_start()` to program the library to set up a GPU operated context.

To get the GPU context handle, the user should call `doca_rxq_get_gpu_handle()` which returns a pointer to a handle in the GPU memory space.

#### Note

The datapath cannot be managed concurrently for the GPU and the CPU.

The DOCA ETH context is configured on the CPU and then exported to the GPU:
The following example shows the expected flow for a GPU-managed datapath with packets being scattered to GPU memory (for doca_eth_rxq):

1. Create a DOCA GPU device handler.
2. Create doca_eth_rxq and configure its parameters.
3. Set the datapath of the context to GPU.
4. Start the context.
5. Get a GPU handle of the context.
For more information regarding the GPU datapath see DOCA GPUNetIO.

**DOCA ETH Samples**

This section describes DOCA ETH samples based on the DOCA ETH library.

The samples illustrate how to use the DOCA ETH API to do the following:

- Send "regular" packets (smaller than MTU) using DOCA ETH TXQ
- Send "large" packets (larger than MTU) using DOCA ETH TXQ
- Receive packets using DOCA ETH RXQ in Regular Receive mode
- Receive packets using DOCA ETH RXQ in Managed Memory Pool Receive mode

ℹ️ **Info**
All of the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Running the Samples

1. Refer to the following documents:

   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample (e.g., `eth_txq_send_ethernet_frames`):

   ```
   cd /opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary `eth_txq_send_ethernet_frames` is created under `/tmp/build/`.

3. Sample (e.g., `eth_txq_send_ethernet_frames`) usage:

   ```
   Usage: doca_eth_txq_send_ethernet_frames [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
   ```
4. For additional information per sample, use the -h option:

```
/top/build/<sample_name> -h
```

**Samples**

**Note**

The following samples are for the CPU datapath. For GPU datapath samples, see [DOCA GPUNetIO](#).

**ETH TXQ Send Ethernet Frames**

This sample illustrates how to send a "regular" packet (smaller than MTU) using DOCA ETH TXQ.

The sample logic includes:

1. Locating DOCA device.

2. Initializing the required DOCA Core structures.

3. Populating DOCA memory map with one buffer to the packet's data.

4. Writing the packet's content into the allocated buffer.

5. Allocating elements from DOCA buffer inventory for the buffer.
6. Initializing and configuring DOCA ETH TXQ context.

7. Starting the DOCA ETH TXQ context.

8. Allocating DOCA ETH TXQ send task.

9. Submitting DOCA ETH TXQ send task into progress engine.

10. Retrieving DOCA ETH TXQ send task from the progress engine.

11. Handling the completed task using the provided callback.

12. Stopping the DOCA ETH TXQ context.

13. Destroying DOCA ETH TXQ context.

14. Destroying all DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/eth_txq_send_ethernet_frames_s`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/eth_txq_send_ethernet_frames_r`
- `/opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/meson.build`

**ETH TXQ LSO Send Ethernet Frames**

This sample illustrates how to send a "large" packet (larger than MTU) using DOCA ETH TXQ.

The sample logic includes:

1. Locating DOCA device.

2. Initializing the required DOCA Core structures.

3. Populating DOCA memory map with one buffer to the packet's payload.

4. Writing the packet's payload into the allocated buffer.
5. Allocating elements from DOCA Buffer inventory for the buffer.

6. Allocating DOCA gather list consisting of one node to the packet's headers.

7. Writing the packet's headers into the allocated gather list node.

8. Initializing and configuring DOCA ETH TXQ context.

9. Starting the DOCA ETH TXQ context.

10. Allocating DOCA ETH TXQ LSO send task.

11. Submitting DOCA ETH TXQ LSO send task into progress engine.

12. Retrieving DOCA ETH TXQ LSO send task from the progress engine.

13. Handling the completed task using the provided callback.

14. Stopping the DOCA ETH TXQ context.

15. Destroying DOCA ETH TXQ context.

16. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/eth_txq_lso_send_ethernet_frames
- /opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/eth_txq_lso_send_ethernet_frames
- /opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/meson.build

**ETH TXQ Batch Send Ethernet Frames**

This sample illustrates how to send a batch of "regular" packets (smaller than MTU) using DOCA ETH TXQ.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.

3. Populating DOCA memory map with multiple buffers, each representing a packet's data.

4. Writing the packets' content into the allocated buffers.

5. Allocating elements from DOCA Buffer inventory for the buffers.

6. Initializing and configuring DOCA ETH TXQ context.

7. Starting the DOCA ETH TXQ context.

8. Allocating DOCA ETH TXQ send task batch.

9. Copying all buffers' pointers to task batch's pkt_arr.

10. Submitting DOCA ETH TXQ send task batch into the progress engine.

11. Retrieving DOCA ETH TXQ send task batch from the progress engine.

12. Handling the completed task batch using the provided callback.

13. Stopping the DOCA ETH TXQ context.


15. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/eth_txq_batch_send_ether
- /opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/eth_txq_batch_sendEtherFiles
- /opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/meson.build

**ETH TXQ Batch LSO Send Ethernet Frames**

This sample illustrates how to send a batch of "large" packets (larger than MTU) using DOCA ETH TXQ.
The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with multiple buffers, each representing a packet's payload.
4. Writing the packets' payload into the allocated buffers.
5. Allocating elements from DOCA Buffer inventory for the buffers.
6. Allocating DOCA gather lists each consisting of one node for the packet's headers.
7. Writing the packets' headers into the allocated gather list nodes.
8. Initializing and configuring DOCA ETH TXQ context.
9. Starting the DOCA ETH TXQ context.
10. Allocating DOCA ETH TXQ LSO send task.
11. Copying all buffers' pointers to task batch's pkt_payload_arr.
12. Copying all gather lists' pointers to task batch's headers_arr.
13. Submitting DOCA ETH TXQ LSO send task batch into the progress engine.
14. Retrieving DOCA ETH TXQ LSO send task batch from the progress engine.
15. Handling the completed task batch using the provided callback.
16. Stopping the DOCA ETH TXQ context.
17. Destroying DOCA ETH TXQ context.
18. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_txq_batch_lso_send_ethernet_frames/eth_txq_batch_lso_sen
ETH RXQ Regular Receive

This sample illustrates how to receive a packet using DOCA ETH RXQ in Regular Receive mode.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with one buffer to the packet's data.
4. Allocating element from DOCA Buffer inventory for each buffer.
5. Initializing DOCA Flow.
6. Initializing and configuring DOCA ETH RXQ context.
7. Starting the DOCA ETH RXQ context.
8. Starting DOCA Flow.
9. Creating a pipe connecting to DOCA ETH RXQ's RX queue.
10. Allocating DOCA ETH RXQ receive task.
11. Submitting DOCA ETH RXQ receive task into the progress engine.
12. Retrieving DOCA ETH RXQ receive task from the progress engine.
13. Handling the completed task using the provided callback.
15. Stopping the DOCA ETH RXQ context.
16. Destroying DOCA ETH RXQ context.

17. Destroying DOCA Flow.

18. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/eth_rxq_regular_receive_sample.c
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/eth_rxq_regular_receive_main.c
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/meson.build

**ETH RXQ Managed Receive**

This sample illustrates how to receive packets using DOCA ETH RXQ in Managed Memory Pool Receive mode.

The sample logic includes:

1. Locating DOCA device.

2. Initializing the required DOCA Core structures.

3. Calculating the required size of the buffer to receive the packets from DOCA ETH RXQ.

4. Populating DOCA memory map with a packets buffer.

5. Initializing DOCA Flow.

6. Initializing and configuring DOCA ETH RXQ context.

7. Registering DOCA ETH RXQ managed receive event.

8. Starting the DOCA ETH RXQ context.


10. Creating a pipe connecting to DOCA ETH RXQ's RX queue.
11. Retrieving DOCA ETH RXQ managed receive events from the progress engine.

12. Handling the completed events using the provided callback.


14. Stopping the DOCA ETH RXQ context.

15. Destroying DOCA ETH RXQ context.


17. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/eth_rxq_managed_mempool_receive
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/eth_rxq_managed_mempool_receive
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/meson.build

**ETH RXQ Batch Managed Receive**

This sample illustrates how to receive batches of packets using DOCA ETH RXQ in Managed Memory Pool Receive mode.

The sample logic includes:

1. Locating DOCA device.

2. Initializing the required DOCA Core structures.

3. Calculating the required size of the buffer to receive the packets from DOCA ETH RXQ.

4. Populating DOCA memory map with a packets buffer.

5. Initializing DOCA Flow.

6. Initializing and configuring DOCA ETH RXQ context.
7. Registering DOCA ETH RXQ managed receive event batch.

8. Starting the DOCA ETH RXQ context.


10. Creating a pipe connecting to DOCA ETH RXQ's RX queue.

11. Retrieving DOCA ETH RXQ managed receive event batches from the progress engine.

12. Handling the completed event batches using the provided callback.


14. Stopping the DOCA ETH RXQ context.

15. Destroying DOCA ETH RXQ context.


17. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/eth_rxq_batch_managed_mempool_receive
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/eth_rxq_batch_managed_mempool_receive
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/meson.build
DOCA GPUNetIO

This document provides an overview and configuration instructions for DOCA GPUNetIO API.

**Introduction**

Real-time GPU processing of network packets is a technique useful for application domains involving signal processing, network security, information gathering, input reconstruction, and more. These applications involve the CPU in the critical path (CPU-centric approach) to coordinate the network card (NIC) for receiving packets in the GPU memory (GPUDirect RDMA) and notifying a packet-processing CUDA kernel waiting on the GPU for a new set of packets. In lower-power platforms, the CPU can easily become the bottleneck, masking GPU value. The aim is to maximize the zero-packet-loss throughput at the the lowest latency possible.

A CPU-centric approach may not be scalable when increasing the number of clients connected to the application as the time between two receive operations on the same queue (client) would increase with the number of queues. The new DOCA GPUNetIO library allows developers to orchestrate these kinds of applications while optimizing performance, combining GPUDirect RDMA for data-path acceleration, GDRCopy library to give the CPU direct access to GPU memory, and GPUDirect async kernel-initiated network (GDAKIN) communications to allow a CUDA kernel to directly control the NIC.

CPU-centric approach:
GPU-centric approach:

DOCA GPUNetIO enables GPU-centric solutions that remove the CPU from the critical path by providing the following features:

- **GPUDirect async kernel-initiated technology** – a GPU CUDA kernel can directly control other hardware components like the network card or NVIDIA® BlueField®'s DMA engine
  - **GDarkin communications** – a GPU CUDA kernel can control network communications to send or receive data
    - GPU can control Ethernet communications
- GPU can control RDMA communications (InfiniBand or RoCE are supported)
- CPU intervention is not needed in the application critical path
  - DMA engine – a GPU CUDA kernel can trigger a memory copy using BlueField's DMA engine
- **GPUDirect RDMA** – use a contiguous GPU memory to send or receive RDMA data or Ethernet packets without CPU memory staging copies
- Semaphores – provide a standardized low-latency message passing protocol between two CUDA kernels or a CUDA kernel and a CPU thread
- Smart memory allocation – allocate aligned GPU memory buffers, possibly exposing them to direct CPU access
  - Combination of CUDA, DPDK gpudev library and GDRCopy library already embedded in the DPDK released with DOCA
- Accurate send scheduling – schedule Ethernet packets' send in the future according to a user-provided timestamp

**Aerial 5G SDK, Morpheus, and Holoscan Advanced Network Operator** are examples of NVIDIA applications actively using DOCA GPUNetIO.

For a deep dive into the technology and motivations, please refer to the NVIDIA blog posts **Inline GPU Packet Processing with NVIDIA DOCA GPUNetIO** and **Unlocking GPU-Accelerated RDMA with NVIDIA DOCA GPUNetIO**. Another NVIDIA blog post **Realizing the Power of Real-time Network Processing with NVIDIA DOCA GPUNetIO** has been published to provide more use-case examples where DOCA GPUNetIO has been useful to improve the execution.

⚠️ **Warning**

RDMA on DOCA GPUNetIO is currently supported at alpha level.
Changes From Previous Releases

Changes in 2.8

The following subsection(s) detail the doca_gpunetio library updates in version 2.8.0.

**Added**

- `__device__ doca_error_t doca_gpu_dev_buf_get_mkey(const struct doca_gpu_buf *buf, uint32_t *mkey)`
- `__device__ doca_error_t doca_gpu_dev_buf_create(uintptr_t addr, uint32_t mkey, struct doca_gpu_buf **buf)`
- `__device__ doca_error_t doca_gpu_dev_dma_memcpyp(struct doca_gpu_dma *dma, struct doca_gpu_buf *src_buf, uint64_t src_offset, struct doca_gpu_buf *dst_buf, uint64_t dst_offset, size_t length)`
- `__device__ doca_error_t doca_gpu_dev_dma_commit(struct doca_gpu_dma *dma)`
- `__device__ doca_error_t doca_gpu_dev_rdma_wait_all(struct doca_gpu_dev_rdma *rdma, uint32_t *num_ops)`

**Changed**

- `struct doca_gpu_buf` – Added `uint32_t mkey` field after `size` field
- `struct doca_gpu_eth_rxq` – Added `bool need_flush`
- `__device__ doca_error_t doca_gpu_dev_rdma_recv_weak(struct doca_gpu_dev_rdma_r *rdma_r, size_t recv_length, uint64_t recv_offset, const uint32_t flags_bitmask, uint32_t position,uint64_t *hdl)`;
- `__device__ doca_error_t doca_gpu_dev_rdma_recv_strong(struct doca_gpu_dev_rdma_r *rdma_r, struct doca_gpu_buf *recv_buf, size_t recv_length, uint64_t recv_offset, const uint32_t flags_bitmask,uint64_t *hdl)`;
- `__device__ doca_error_t doca_gpu_dev_rdma_recv_wait_all(struct doca_gpu_dev_rdma_r *rdma_r)`
  - `__device__ doca_error_t doca_gpu_dev_rdma_recv_wait_all(struct doca_gpu_dev_rdma_r *rdma_r, uint64_t *hdl, const enum doca_gpu_dev_rdma_recv_wait_flags flags, enum doca_rdma_opcode *opcode, uint32_t *imm)`
System Configuration

DOCA GPUNetIO requires a properly configured environment which depends on whether the application should run on the x86 host or DPU Arm cores. The following subsections describe the required configuration in both scenarios, assuming DOCA, CUDA Toolkit and NVIDIA driver are installed on the system (x86 host or BlueField Arm) where the DOCA GPUNetIO is built and executed.

DOCA GPUNetIO is available for all DOCA for host and BFB packages downloadable here.

Assuming the DOCA package has been downloaded and installed on the system, to install all DOCA GPUNetIO components, run:

- For Ubuntu/Debian:

  ```
  apt install doca-all doca-sdk-gpunetio libdoca-sdk-gpunetio-dev
  ```

- For RHEL:

  ```
  yum install doca-all doca-sdk-gpunetio doca-sdk-gpunetio-devel
  ```

Internal hardware topology of the system should be GPUDirect-RDMA-friendly to maximize the internal throughput between the GPU and the NIC.

As DOCA GPUNetIO is present in both DOCA-for-Host and DOCA BFB (for BlueField Arm), a GPUNetIO application can be executed either on the host CPU or on the BlueField's Arm cores. The following subsections provide a description of both scenarios.

Note

DOCA GPUNetIO has been tested on bare-metal and in docker but never in a virtualized environment. Using KVM is discouraged for now.
Application on Host CPU

Assuming the DOCA GPUNetIO application is running on the host x86 CPU cores, it is highly recommended to have a dedicated PCIe connection between the GPU and the NIC. This topology can be realized in two ways:

- Adding an additional PCIe switch to one of the PCIe root complex slots and attaching to this switch a GPU and a NVIDIA® ConnectX® adapter
- Connecting an NVIDIA® Converged Accelerator DPU to the PCIe root complex and setting it to NIC mode (i.e., exposing the GPU and NIC devices to the host)

You may check the topology of your system using `lspci -tvvv` or `nvidia-smi topo -m`.

Option 1: ConnectX Adapter in Ethernet Mode

**Note**

NVIDIA® ConnectX® firmware must be 22.36.1010 or later. It is highly recommended to only use NVIDIA adapter from ConnectX-6 Dx and later.
DOCA GPUNetIO allows a CUDA kernel to control the NIC when working with Ethernet protocol. For this reason, the ConnectX must be set to Ethernet mode.

To do that, follow these steps:

1. Start MST, check the status, and copy the MST device name:

```
# Start MST
mst start
mst status -v
```

MST modules:
-------------
MST PCI module is not loaded
MST PCI configuration module loaded
PCI devices:
-------------

```
DEVICE_TYPE               MST              PCI    RDMA            NET                      NUMA
ConnectX6DX(rev:0)        /dev/mst/mt4125_pciconf0.1  b5:00.1  mlx5_1          net-ens6f1                0
ConnectX6DX(rev:0)        /dev/mst/mt4125_pciconf0     b5:00.0  mlx5_0          net-ens6f0                0
```

2. Configure the NIC to Ethernet mode and enable Accurate Send Scheduling (if required on the send side):

```
Info

The following example assumes that the adapter is dual-port. If single port, only P1 options apply.

mlxconfig -d <mst_device> s KEEP_ETH_LINK_UP_P1=1 KEEP_ETH_LINK_UP_P2=1
KEEP_IB_LINK_UP_P1=0 KEEP_IB_LINK_UP_P2=0
mlxconfig -d <mst_device> --yes set ACCURATE_TX_SCHEDULER=1 REAL_TIME_CLOCK_ENABLE=1
```

3. Perform cold reboot to apply the configuration changes:
Option 2: DPU Converged Accelerator in NIC mode

To expose and use the GPU and the NIC on the converged accelerator DPU to an application running on the Host x86, configure the DPU to operate in NIC mode.

To do that, follow these steps:

1. Start MST, check the status, and copy the MST device name:

```
# Enable MST
sudo mst start
sudo mst status

MST devices:
------------
/dev/mst/mi41686_pciconf0   - PCI configuration cycles access.
   domain:bus:dev.fn=0000:b8:00.0 addr.reg=88 data.reg=92
cr_bar.gw_offset=-1
   Chip revision is: 01
```

2. Expose the GPU on the converged accelerator DPU to the host.

   - For BlueField-2, the `PCI_DOWNSTREAM_PORT_OWNER` offset must be set to 4:

```
ipmitool power cycle
```
For BlueField-3, the PCI_DOWNSTREAM_PORT_OWNER offset must be set to 8:

```
sudo mlxconfig -d <mst_device> --yes s PCI_DOWNSTREAM_PORT_OWNER[4]=0x0
```

```
sudo mlxconfig -d <mst_device> --yes s PCI_DOWNSTREAM_PORT_OWNER[8]=0x0
```

3. Set BlueField to Ethernet mode, enable Accurate Send Scheduling (if required on the send side), and set it to NIC mode:

```
sudo mlxconfig -d <mst_device> --yes set LINK_TYPE_P1=2 LINK_TYPE_P2=2
INTERNAL_CPU_MODEL=1 INTERNAL_CPU_PAGE_SUPPLIER=1
INTERNAL_CPU_ESWITCH_MANAGER=1 INTERNAL_CPU_IB_VPORT0=1
INTERNAL_CPU_OFFLOAD_ENGINE=DISABLED
```

```
sudo mlxconfig -d <mst_device> --yes set ACCURATE_TX_SCHEDULER=1
REAL_TIME_CLOCK_ENABLE=1
```

4. Perform cold reboot to apply the configuration changes:

```
ipmitool power cycle
```

5. Verify configuration:

```
sudo mlxconfig -d <mst_device> q LINK_TYPE_P1 LINK_TYPE_P2 INTERNAL_CPU_MODEL
INTERNAL_CPU_PAGE_SUPPLIER INTERNAL_CPU_ESWITCH_MANAGER INTERNAL_CPU_IB_VPORT0
INTERNAL_CPU_OFFLOAD_ENGINE ACCURATE_TX_SCHEDULER REAL_TIME_CLOCK_ENABLE
```

| LINK_TYPE_P1 | ETH(2) |
| LINK_TYPE_P2 | ETH(2) |
| INTERNAL_CPU_MODEL | EMBEDDED_CPU(1) |
| INTERNAL_CPU_PAGE_SUPPLIER | EXT_HOST_PF(1) |
| INTERNAL_CPU_ESWITCH_MANAGER | EXT_HOST_PF(1) |
| INTERNAL_CPU_IB_VPORT0 | EXT_HOST_PF(1) |
| INTERNAL_CPU_OFFLOAD_ENGINE | DISABLED(1) |
| ACCURATE_TX_SCHEDULER | True(1) |
Application on BlueField Converged Arm CPU

In this scenario, the DOCA GPUNetIO is running on the CPU Arm cores of the BlueField using the GPU and NIC on the same BlueField.

The converged accelerator DPU must be set to CPU mode after flashing the right BFB image (refer to NVIDIA DOCA Installation Guide for Linux for details). From the x86 host, configure the DPU as detailed in the following steps:

- **Info**

  Valid for both BlueField-2 and BlueField-3 converged accelerator DPUs.
1. Start MST, check the status, and copy the MST device name:

```bash
# Enable MST
sudo mst start
sudo mst status

MST devices:
-------------
/dev/mst/mt41686_pciconf0 - PCI configuration cycles access.
  domain:bus:dev.fn=0000:b8:00.0 addr.reg=88 data.reg=92
cr_bar.gw_offset=-1
  Chip revision is: 01
```

2. Set the DPU as the GPU owner.

   1. For BlueField-2 the `PCI_DOWNSTREAM_PORT_OWNER` offset must be set to 4:

      ```bash
      sudo mlxconfig -d <mst_device> --yes s PCI_DOWNSTREAM_PORT_OWNER[4]=0xF
      ```

   2. For BlueField-3 the `PCI_DOWNSTREAM_PORT_OWNER` offset must be set to 8:

      ```bash
      sudo mlxconfig -d <mst_device> --yes s PCI_DOWNSTREAM_PORT_OWNER[8]=0xF
      ```

3. Set BlueField to Ethernet mode and enable Accurate Send Scheduling (if required on the send side):

   ```bash
   sudo mlxconfig -d <mst_device> --yes set LINK_TYPE_P1=2 LINK_TYPE_P2=2
   INTERNAL_CPU_MODEL=1 INTERNAL_CPU_PAGE_SUPPLIER=0
   INTERNAL_CPU_ESWITCH_MANAGER=0 INTERNAL_CPU_IB_VPORT0=0
   INTERNAL_CPU_OFFLOAD_ENGINE=ENABLED
   sudo mlxconfig -d <mst_device> --yes set ACCURATE_TX_SCHEDULER=1
   REAL_TIME_CLOCK_ENABLE=1
   ```

4. Perform cold reboot to apply the configuration changes:
5. Verify configuration:

```bash
mlxconfig -d <mst_device> q LINK_TYPE_P1 LINK_TYPE_P2 INTERNAL_CPU_MODEL
INTERNAL_CPU_PAGE_SUPPLIER INTERNAL_CPU_ESWITCH_MANAGER INTERNAL_CPU_IB_VPORT0
INTERNAL_CPU_OFFLOAD_ENGINE ACCURATE_TX_SCHEDULER REAL_TIME_CLOCK_ENABLE
...
```

<table>
<thead>
<tr>
<th>Configurations:</th>
<th>Next Boot</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINK_TYPE_P1</td>
<td>ETH(2)</td>
</tr>
<tr>
<td>LINK_TYPE_P2</td>
<td>ETH(2)</td>
</tr>
<tr>
<td>INTERNAL_CPU_MODEL</td>
<td>EMBEDDED_CPU(1)</td>
</tr>
<tr>
<td>INTERNAL_CPU_PAGE_SUPPLIER</td>
<td>ECPF(0)</td>
</tr>
<tr>
<td>INTERNAL_CPU_ESWITCH_MANAGER</td>
<td>ECPF(0)</td>
</tr>
<tr>
<td>INTERNAL_CPU_IB_VPORT0</td>
<td>ECPF(0)</td>
</tr>
<tr>
<td>INTERNAL_CPU_OFFLOAD_ENGINE</td>
<td>ENABLED(0)</td>
</tr>
<tr>
<td>ACCURATE_TX_SCHEDULER</td>
<td>True(1)</td>
</tr>
<tr>
<td>REAL_TIME_CLOCK_ENABLE</td>
<td>True(1)</td>
</tr>
</tbody>
</table>

At this point, it should be possible to SSH into BlueField to access the OS installed on it. Before installing DOCA GPUNetIO as previously described, CUDA Toolkit (and NVIDIA driver) must be installed.

**PCIe Configuration**

On some x86 systems, the Access Control Services (ACS) must be disabled to ensure direct communication between the NIC and GPU, whether they reside on the same converged accelerator DPU or on different PCIe slots in the system. The recommended solution is to disable ACS control via BIOS (e.g., Supermicro or HPE). Alternatively, it is also possible to disable it via command line, but it may not be as effective as the BIOS option. Assuming system topology **Option 2**, with a converged accelerator DPU as follows:

```bash
$lspci -tvvv...+-[0000:b0]-+-00.0  Intel Corporation Device 09a2
  |  +00.1  Intel Corporation Device 09a4
  |  +00.2  Intel Corporation Device 09a3
```
The PCIe switch address to consider is b2:00.0 (entry point of the DPU). ACSCtl must have all negative values:

```
setpci -s b2:00.0 ECAP_ACS+6.w=0:fc
```

To verify that the setting has been applied correctly:

```
$ sudo lspci -s b2:00.0 -vvvv | grep -i ACSCtl
ACSCtl: SrcValid- TransBlk- ReqRedir- CmpltRedir- UpstreamFwd- EgressCtrl- DirectTrans-
```

Please refer to this page and this page for more information.

If the application still does not report any received packets, try to disable IOMMU. On some systems, it can be done from the BIOS looking for the VT-d or IOMMU from the NorthBridge configuration and change that setting to Disable and save it. The system may also require adding intel_iommu=off or amd_iommu=off to the kernel options. That can be done through the grub command line as follows:

```
$ sudo vim /etc/default/grub
# GRUB_CMDLINE_LINUX_DEFAULT="iommu=off intel_iommu=off <more options>"
$ sudo update-grub
$ sudo reboot
```
Hugepages

A DOCA GPUNetIO application over Ethernet uses typically DOCA Flow to set flow steering rules to the Ethernet receive queues. Flow-based programs require an allocation of huge pages and it can be done temporarily as explained in the DOCA Flow or permanently via grub command line:

```
$ sudo vim /etc/default/grub
# GRUB_CMDLINE_LINUX_DEFAULT="default_hugepagesz=1G hugepagesz=1G hugepages=4 <more options>"
$ sudo update-grub
$ sudo reboot

# After rebooting, check huge pages info
$ grep -i huge /proc/meminfo
AnonHugePages: 0 kB
ShmemHugePages: 0 kB
FileHugePages: 0 kB
HugePages_Total: 4
HugePages_Free: 4
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 1048576 kB
Hugetlb: 4194304 kB
```

GPU Configuration

CUDA Toolkit 12.1 or newer must be installed on the host. It is also recommended to enable persistence mode to decrease initial application latency `nvidia-smi -pm 1`.

To allow the CPU to access the GPU memory directly without the need for CUDA API, DPDK and DOCA require the GDRCopy kernel module to be installed on the system:

```
# Run nvidia-peermem kernel module
sudo modprobe nvidia-peermem

# Install GDRCopy
sudo apt install -y check kmod
```
BlueField-3 Specific Configuration

To run a DOCA GPUNetIO application on the BlueField Arm cores in a BlueField-3 converged card (section "Application on DPU Converged Arm CPU"), it is mandatory to set an NVIDIA driver option at the end of the driver configuration file:

```
cat <<EOF | sudo tee /etc/modprobe.d/nvidia.conf
options nvidia NVreg_RegistryDwords="RmDmaAdjustPeerMmioBF3=1;"
EOF
```

To make sure the option has been detected by the NVIDIA driver, run:

```
$ grep RegistryDwords /proc/driver/nvidia/params
RegistryDwords: "RmDmaAdjustPeerMmioBF3=1;"
RegistryDwordsPerDevice: ""
```

GPU Memory Mapping (nvidia-peermem vs. dmabuf)
To allow the NIC to send and receive packets using GPU memory, it is required to launch the NVIDIA kernel module `nvidia-peermem` (using `modprobe nvidia-peermem`). It is shipped by default with the CUDA Toolkit installation.

Mapping buffers through the `nvidia-peermem` module is the legacy mapping mode.

Alternatively, DOCA offers the ability to map GPU memory through the `dmabuf` providing a set high-level functions. Prerequisites are DOCA installed on a system with:

- Linux Kernel ≥ 6.2
- libibverbs ≥ 1.14.44
- CUDA Toolkit installed with the `-m=kernel-open` flag (which implies the NVIDIA driver in Open Source mode)

**Note**

Installing DOCA on kernel 6.2 to enable the `dmabuf` is experimental.

An example can be found in the DOCA GPU Packet Processing application:

```c
/* Get from CUDA the dmabuf file-descriptor for the GPU memory buffer */
result = doca_gpu_dmabuf_fd(gpu_dev, gpu_buffer_addr, gpu_buffer_size, &(dmabuf_fd));
if (result != DOCA_SUCCESS) {
    /* If it fails, create a DOCA mmap for the GPU memory buffer with the nvidia-peermem legacy method */
    doca_mmap_set_memrange(gpu_buffer_mmap, gpu_buffer_addr, gpu_buffer_size);
} else {
    /* If it succeeds, create a DOCA mmap for the GPU memory buffer using the dmabuf method */
    doca_mmap_set_dmabuf_memrange(gpu_buffer_mmap, dmabuf_fd, gpu_buffer_addr, 0, gpu_buffer_size);
}
```

If the function `doca_gpu_dmabuf_fd` fails, it probably means the NVIDIA driver is not installed with the open-source mode.
Later, when calling the `doxa_mmap_start`, the DOCA library tries to map the GPU memory buffer using the `dmabuf` file descriptor. If it fails (something incorrectly set on the Linux system), it fallbacks trying to map the GPU buffer with the legacy mode (`nvidia-peermem`). If it fails, an informative error is returned.

**GPU BAR1 Size**

Every time a GPU buffer is mapped to the NIC (e.g., buffers associated with send or receive queues), a portion of the GPU BAR1 mapping space is used. Therefore, it is important to check that the BAR1 mapping is large enough to hold all the bytes the DOCA GPUNetIO application is trying to map. To verify the BAR1 mapping space of a GPU you can use `nvidia-smi`:

```
$ nvidia-smi -q

==============NVSMI LOG==============
......
Attached GPUs : 1
GPU 00000000:CA:00.0
    Product Name : NVIDIA A100 80GB PCIe
    Product Architecture : Ampere
    Persistence Mode : Enabled
......
BAR1 Memory Usage
    Total : 131072 MiB
    Used : 1 MiB
    Free : 131071 MiB
```

By default, some GPUs (e.g. RTX models) may have a very small BAR1 size:

```
$ nvidia-smi -q | grep -i bar -A 3
BAR1 Memory Usage
    Total : 256 MiB
    Used : 6 MiB
    Free : 250 MiB
```
If the BAR1 size is not enough, DOCA GPUNetIO applications may exit with errors because DOCA mmap fails to map the GPU memory buffers to the NIC (e.g., Failed to start mmap DOCA Driver call failure). To overcome this issue, the GPU BAR1 must be increased from the BIOS. The system should have "Resizable BAR" option enabled. For further information, refer to this NVIDIA forum post.

**Architecture**

A GPU packet processing network application can be split into two fundamental phases:

- Setup on the CPU (devices configuration, memory allocation, launch of CUDA kernels, etc.)
- Main data path where GPU and NIC interact to exercise their functions

DOCA GPUNetIO provides different building blocks, some of them in combination with the DOCA Ethernet or DOCA RDMA library, to create a full pipeline running entirely on the GPU.

During the setup phase on the CPU, applications must:

1. Prepare all the objects on the CPU.
2. Export a GPU handler for them.
3. Launch a CUDA kernel passing the object's GPU handler to work with the object during the data path.

For this reason, DOCA GPUNetIO is composed of two libraries:

- **libdoca_gpunetio** with functions invoked by CPU to prepare the GPU, allocate memory and objects
- **libdoca_gpunetio_device** with functions invoked by GPU within CUDA kernels during the data path

**Note**
The pkgconfig file for the DOCA GPUNetIO shared library is `doca-gpunetio.pc`. However, there is no pkgconfig file for the DOCA GPUNetIO CUDA device's static library `/opt/mellanox/doca/lib/x86_64-linux-gnu/libdoca_gpunetio_device.a`, so it must be explicitly linked to the CUDA application if DOCA GPUNetIO CUDA device functions are required.

The following diagram presents the typical flow:
Refer to the NVIDIA DOCA GPU Packet Processing Application Guide for an example of using DOCA GPUNetIO to send and receive Ethernet packets.

**API**
This section details the specific structures and operations related to the main DOCA GPUNetIO API on CPU and GPU. GPUNetIO headers are:

- `doca_gpunetio.h` – CPU functions
- `doca_gpunetio_dev_buf.cuh` – GPU functions to manage a DOCA buffer array
- `doca_gpunetio_dev_eth_rxq.cuh` – GPU functions to manage a DOCA Ethernet receive queue
- `doca_gpunetio_dev_eth_txq.cuh` – GPU functions to manage a DOCA Ethernet send queue
- `doca_gpunetio_dev_sem.cuh` – GPU functions to manage a DOCA GPUNetIO semaphore
- `doca_gpunetio_dev_rdma.cuh` – GPU functions to manage a DOCA RDMA queue
- `doca_gpunetio_dev_dma.cuh` – GPU functions to manage a DOCA DMA queue

This section lists the main functions of DOCA GPUNetIO. To better understand their usage, refer to section "Building Blocks" which includes several code examples.

**Tip**

To better understand structures, objects, and functions related to Ethernet send and receive, please refer to the DOCA Ethernet.

**Tip**

To better understand structures, objects, and functions related to RDMA operations, please refer to the DOCA RDMA.

**Tip**
To better understand structures, objects, and functions related to DMA operations, please refer to the DOCA DMA.

**Tip**

To better understand DOCA core objects like `doca_mmap` or `doca_buf_array`, please refer to the DOCA Core.

All DOCA Core and Ethernet object used with GPUNetIO have a GPU export function to obtain a GPU handler for that object. The following are a few examples:

- *doca_buf_array is exported as* `doca_gpu_buf_arr`:

  ```c
  struct doca_mmap *mmap;
  struct doca_buf_arr *buf_arr_cpu;
  struct doca_gpu_buf_arr *buf_arr_gpu;
  doca_mmap_create(&(mmap));
  /* Populate and start mmap */
  doca_buf_arr_create(mmap, &buf_arr_cpu);
  /* Populate and start buf arr attributes. Set datapath on GPU */
  /* Export the buf array CPU handler to a buf array GPU handler */
  doca_buf_arr_get_gpu_handle(buf_arr_cpu, &buf_arr_gpu);
  /* To use the GPU handler, pass it as parameter of the CUDA kernel */
  cuda_kernel<<<...>>>(buf_arr_gpu, ...);
  ```

- *doca_eth_rxq is exported as* `doca_gpu_eth_rxq`:

  ```c
  struct doca_mmap *mmap;
  struct doca_eth_rxq *eth_rxq_cpu;
  struct doca_gpu_eth_rxq *eth_rxq_gpu;
  struct doca_dev *ddev;
  ```
CPU Functions

In this section there is the list of DOCA GPUNetIO functions that can be used on the CPU only.

**doca_gpu_mem_type**

This enum lists all the possible memory types that can be allocated with GPUNetIO.

```c
enum doca_gpu_mem_type {
    DOCA_GPU_MEM_TYPE_GPU        = 0,
    DOCA_GPU_MEM_TYPE_GPU_CPU    = 1,
    DOCA_GPU_MEM_TYPE_CPU_GPU    = 2,
};
```

**Note**

With regards to the syntax, the text string after the `DOCA_GPU_MEM_TYPE_` prefix signifies `<where-memory-resides>_<who-has-access>`.

- **DOCA_GPU_MEM_TYPE_GPU** – memory resides on the GPU and is accessible from the GPU only
- **DOCA_GPU_MEM_TYPE_GPU_CPU** – memory resides on the GPU and is accessible also by the CPU
- **DOCA_GPU_MEM_TYPE_CPU_GPU** – memory resides on the CPU and is accessible also by the GPU

Typical usage of the **DOCA_GPU_MEM_TYPE_CPU_GPU** memory type is to send a notification from the CPU to the GPU (e.g., a CUDA kernel periodically checking to see if the exit condition set by the CPU is met).

**doca_gpu_create**

This is the first function a GPUNetIO application must invoke to create an handler on a GPU device. The function initializes a pointer to a structure in memory with type `struct doca_gpu *`.

```c
doca_error_t doca_gpu_create(const char *gpu_bus_id, struct doca_gpu **gpu_dev);
```

- `gpu_bus_id` – `<PCIe-bus>:<device>.<function>` of the GPU device you want to use in your application
- `gpu_dev [out]` – GPUNetIO handler to that GPU device

To get the PCIe address, users can use the commands `lspci` or `nvidia-smi`.

**doca_gpu_mem_alloc**

This CPU function allocates different flavors of memory.

```c
doca_error_t doca_gpu_mem_alloc(struct doca_gpu *gpu_dev, size_t size, size_t alignment, enum doca_gpu_mem_type mtype, void **memptr_gpu, void **memptr_cpu)
```

- `gpu_dev` – GPUNetIO device handler
- `size` – Size, in bytes, of the memory area to allocate
- `alignment` – Memory address alignment to use. If 0, default one will be used
- mtype – Type of memory to allocate
- memptr_gpu [out] – GPU pointer to use to modify that memory from the GPU if memory is allocated on or is visible by the GPU
- memptr_cpu[out] – CPU pointer to use to modify that memory from the CPU if memory is allocated on or is visible by the CPU. Can be NULL if memory is GPU-only

⚠️ **Warning**

Make sure to use the right pointer on the right device! If an application tries to access the memory using the memptr_gpu address from the CPU, a segmentation fault will result.

---

**doca_gpu_semaphore_create**

Creates a new instance of a DOCA GPUNetIO semaphore. A semaphore is composed by a list of items each having, by default, a status flag, number of packets, and the index of a doca_gpu_buf in a doca_gpu_buf_arr.

For example, a GPUNetIO semaphore can be used in applications where a CUDA kernel is responsible for receiving packets in a doca_gpu_buf_arr array associated with an Ethernet receive queue object, doca_gpu_eth_rxq (see section "doca_gpu_dev_eth_rxq_receive_*"), and dispatching packet info to a second CUDA kernel which processes them.

Another way to use a GPUNetIO semaphore is to exchange data across different entities like two CUDA kernels or a CUDA kernel and a CPU thread. The reason for this scenario may be that the CUDA kernel needs to provide the outcome of the packet processing to the CPU which would in turn compile a statistics report. Therefore, it is possible to associate a custom application-defined structure to each item in the semaphore. This way, the semaphore can be used as a message passing object.

Both situations are illustrated in the "Receive and Process" section.
Entities communicating through a semaphore must adopt a poll/update mechanism according to the following logic:

- **Update:**
  1. Populate the next item of the semaphore (packets' info and/or custom application-defined info).
  2. Set status flag to READY.

- **Poll:**
  1. Wait for the next item to have a status flag equal to READY.
  2. Read and process info.
  3. Set status flag to DONE.
DOCA Libraries 710

DoCA Libraries 710

gpu_dev – GPUNetIO handler

semaphore [out] – GPUNetIO semaphore handler associated to the GPU device

doca_gpu_semaphore_set_memory_type

This function defines the type of memory for the semaphore allocation.

doca_error_t doca_gpu_semaphore_set_memory_type(struct doca_gpu_semaphore *semaphore, enum doca_gpu_mem_type mtype)

semaphore – GPUNetIO semaphore handler

mtype – Type of memory to allocate the custom info structure

○ If the application must share packet info only across CUDA kernels, then DoCA_GPU_MEM_GPU is the suggested memory type.

○ If the application must share info from a CUDA kernel to a CPU (e.g., to report statistics or output of the pipeline computation), then DoCA_GPU_MEM_CPU_GPU is the suggested memory type

doca_gpu_semaphore_set_items_num

This function defines the number of items in a semaphore.

doca_error_t doca_gpu_semaphore_set_items_num(struct doca_gpu_semaphore *semaphore, uint32_t num_items)
- **semaphore** – GPUNetIO semaphore handler
- **num_items** – Number of items to allocate

### `doca_gpu_semaphore_set_custom_info`

This function associates an application-specific structure to semaphore items as explained under "`doca_gpu_semaphore_create`".

```c
doca_error_t doca_gpu_semaphore_set_custom_info(struct doca_gpu_semaphore *semaphore, uint32_t nbytes, enum doca_gpu_mem_type mtype)
```

- **semaphore** – GPUNetIO semaphore handler
- **nbytes** – Size of the custom info structure to associate
- **mtype** – Type of memory to allocate the custom info structure

- If the application must share packet info only across CUDA kernels, then `DOCA_GPU_MEM_GPU` is the suggested memory type
- If the application must share info from a CUDA kernel to a CPU (e.g., to report statistics or output of the pipeline computation), then `DOCA_GPU_MEM_CPU_GPU` is the suggested memory type

### `doca_gpu_semaphore_get_status`

From the CPU, query the status of a semaphore item. If the semaphore is allocated with `DOCA_GPU_MEM_GPU`, this function results in a segmentation fault.

```c
doca_error_t doca_gpu_semaphore_get_status(struct doca_gpu_semaphore *semaphore_cpu, uint32_t idx, enum doca_gpu_semaphore_status *status)
```

- **semaphore_cpu** – GPUNetIO semaphore CPU handler
- `idx` – Semaphore item index
- `status [out]` – Output semaphore status

**doca_gpu_semaphore_get_custom_info_addr**

From the CPU, retrieve the address of the custom info structure associated to a semaphore item. If the semaphore or the custom info is allocated with `DOCA_GPU_MEM_GPU` this function results in a segmentation fault.

```c
docta_error_t doca_gpu_semaphore_get_custom_info_addr(struct doca_gpu_semaphore *semaphore_cpu, uint32_t idx, void **custom_info)
```

- `semaphore_cpu` – GPUNetIO semaphore CPU handler
- `idx` – Semaphore item index
- `custom_info [out]` – Output semaphore custom info address

**DOCA PE**

A DOCA Ethernet Txq context, exported for GPUNetIO usage, can be tracked via DOCA PE on the CPU side to check if there are errors when sending packets or to retrieve notification info after sending a packet with any of the `doca_gpu_dev_eth_txq_*_enqueue_*` functions on the GPU. An example can be found in the DOCA GPU packet processing application with ICMP traffic.

**Strong Mode vs. Weak Mode**

Some Ethernet and RDMA GPU functions present two modes of operation: Weak and strong.
• In weak mode, the application calculates the next available position in the queue. With the help of functions like doca_gpu_eth_txq_get_info, doca_gpu_rdma_get_info, or doca_gpu_dev_rdma_recv_get_info it is possible to know the next available position in the queue and the mask of the number of total entries in the queue (so the incremental descriptor index can be wrapped). In this mode, the developer must specify a queue descriptor number for where to enqueue the packet, ensuring that no descriptor in the queue is left empty. It’s a bit more complex to manage but it should result in better performance and developer can emphasize GPU memory coalescing enqueuing sequential operations using sequential memory locations.

• In strong mode, the GPU function enqueues the Ethernet/RDMA operation in the next available position in the queue. It is simpler to manage as developer does not have to worry about operation's position, but it may introduce an extra latency to atomically guarantee the access of multiple threads to the same queue. Moreover, it does not guarantee that sequential operations refer to sequential memory locations.

Note

All strong mode functions work at the CUDA block level. That is, it is not possible to access the same Eth/RDMA queue at the same time from two different CUDA blocks.

In sections "Produce and Send" and "CUDA Kernel for RDMA Write", there are a few examples about how to use the weak mode API.

GPU Functions – Ethernet

This section provides a list of DOCA GPUNetIO functions that can be used for Ethernet network operations on the GPU only within a CUDA kernel.

doca_gpu_dev_eth_rxq_receive_*

To acquire packets in a CUDA kernel, DOCA GPUNetIO offers different flavors of the receive function for different scopes: per CUDA block, per CUDA warp, and per CUDA
thread.

- **eth_rxq** – Ethernet receive queue GPU handler

- **max_rx_pkts** – Maximum number of packets to receive. It ensures the number of packets returned by the function is lower or equal to this number.

- **timeout ns** – Nanoseconds to wait for packets before returning

- **num_rx_pkts [out]** – Effective number of received packets. With CUDA block or warp scopes, this variable should be visible in memory by all the other threads (shared or global memory).

- **doca_gpu_buf_idx [out]** – DOCA buffer index of the first packet received in this function. With CUDA block or warp scopes, this variable should be visible in memory by all the other threads (shared or global memory).

**Note**

If both max_rx_pkts and timeout ns are 0, the function never returns.

CUDA threads in the same scope (thread, warp, or block) must invoke the function on the same receive queue. The output parameters num_rx_pkts and doca_gpu_buf_idx must be visible by all threads in the scope (e.g., CUDA shared memory for warp and block).

Each packet received by this function goes to the doca_gpu_buf_arr internally created and associated with the Ethernet queues (see section "Building Blocks").
The function exits when timeout_ns is reached or when the maximum number of packets is received.

**Note**

For CUDA block scope, the block invoking the receive function must have at least 32 CUDA threads (i.e., one warp).

The output parameters indicate how many packets have been received (num_rx_pkts) and the index of the first received packet in the doca_gpu_buf_arr internally associated with the Ethernet receive queue. Packets are stored consecutively in the doca_gpu_buf_arr so if the function returns num_rx_pkts=N and doca_gpu_buf_idx=X, this means that all the doca_gpu_buf in the doca_gpu_buf_arr within the range [X, ...X + (N-1)] have been filled with packets.

The DOCA buffer array is treated in a circular fashion so that once the last DOCA buffer is filled by a packet, the queue circles back to the first DOCA buffer. There is no need for the application to lock or free doca_gpu_buf_arr buffers.

**Note**

It is the application's responsibility to consume packets before they are overwritten when circling back, properly dimensioning the DOCA buffer array size and scaling across multiple receive queues.
**doca_gpu_send_flags**

This enum lists all the possible flags for the txq functions. The usage of those flags makes sense if a DOCA PE has been attached to the DOCA Ethernet Txq context with GPU data path and a CPU thread, in a loop, keeps invoking `doca_pe_progress`.

⚠️ **Warning**

If no DOCA PE has been attached to the DOCA Ethernet Txq context, it’s mandatory to use the `DOCA_GPU_SEND_FLAG_NONE` flag.

```c
enum doca_gpu_mem_type {
    DOCA_GPU_SEND_FLAG_NONE     = 0,
    DOCA_GPU_SEND_FLAG_NOTIFY   = 1 << 0,
};
```

- **DOCA_GPU_SEND_FLAG_NONE** (default) – send is executed and no notification info is returned. If an error occurs, an event is generated. This error can be detected from the CPU side using DOCA PE.

- **DOCA_GPU_SEND_FLAG_NOTIFY** – once the send (or wait) is executed, return a notification with packet info. This notification can be detected from the CPU side using DOCA PE.

**doca_gpu_dev_eth_txq_send_** *

To send packets from a CUDA kernel, DOCA GPUNetIO offers a strong and weak modes for enqueuing a packet in the Ethernet TXQ. For both modes, the scope is the single CUDA thread each populating and enqueuing a different `doca_gpu_buf` from a `doca_gpu_buf_arr` in the send queue.

```c
__device__ doca_error_t doca_gpu_dev_eth_txq_get_info(struct doca_eth_txq *eth_txq, uint32_t
```
*curr_position, uint32_t *mask_max_position)

- **eth_txq** – Ethernet send queue GPU handler
- **curr_position** – Next available position in the queue
- **mask_max_position** – Mask of the total number of positions in the queue

```c
__device__ doca_error_t doca_gpu_dev_eth_txq_send_enqueue_strong(struct doca_gpu_eth_txq *eth_txq, const struct doca_gpu_buf *buf_ptr, const uint32_t nbytes, const uint32_t flags_bitmask)
```

- **eth_txq** – Ethernet send queue GPU handler
- **buf_ptr** – DOCA buffer from a DOCA GPU buffer array to be sent
- **nbytes** – Number of bytes to be sent in the packet
- **flags_bitmask** – One of the flags in the doca_gpu_send_flags enum

```c
__device__ doca_error_t doca_gpu_dev_eth_txq_send_enqueue_weak(const struct doca_gpu_eth_txq *eth_txq, const struct doca_gpu_buf *buf_ptr, const uint32_t nbytes, const uint32_t ndescr, const uint32_t flags_bitmask)
```

- **eth_txq** – Ethernet send queue GPU handler
- **buf_ptr** – DOCA buffer from a DOCA GPU buffer array to be sent
- **nbytes** – Number of bytes to be sent in the packet
- **ndescr** – Position in the queue to place the packet. Range: 0 - mask_max_position.
- **flags_bitmask** – One of the flags in the doca_gpu_send_flags enum

`doca_gpu_dev_eth_txq_wait_*`
To enable Accurate Send Scheduling, the "wait on time" barrier (based on timestamp) must be set in the send queue before enqueuing more packets. Like `doca_gpu_dev_eth_txq_send_*`, `doca_gpu_dev_eth_txq_wait_*` also has a strong and weak mode.

```c
__device__ doca_error_t doca_gpu_dev_eth_txq_wait_time_enqueue_strong(struct doca_gpu_eth_txq *eth_txq, const uint64_t wait_on_time_value, const uint32_t flags_bitmask)
```

- `eth_txq` – Ethernet send queue GPU handler
- `wait_on_time_value` – Timestamp to specify when packets must be sent after this barrier
- `flags_bitmask` – One of the flags in the `doca_gpu_send_flags` enum

```c
__device__ doca_error_t doca_gpu_dev_eth_txq_wait_time_enqueue_weak(struct doca_gpu_eth_txq *eth_txq, const uint64_t wait_on_time_value, const uint32_t ndescr, const uint32_t flags_bitmask)
```

- `eth_txq` – Ethernet send queue GPU handler
- `wait_on_time_value` – Timestamp to specify when packets must be sent after this barrier
- `ndescr` – Position in the queue to place the packet. Range: 0 - `mask_max_position`
- `flags_bitmask` – One of the flags in the `doca_gpu_send_flags` enum

Please refer to section "GPUNetIO Samples" to understand how to enable and use Accurate Send Scheduling.

**doca_gpu_dev_eth_txq_commit_***

After enqueuing all the packets to be sent and time barriers, a commit function must be invoked on the txq queue. The right commit function must be used according to the type of enqueue mode (i.e., strong or weak) used in `doca_gpu_dev_eth_txq_send_*` and `doca_gpu_dev_eth_txq_wait_*`.
__device__ doca_error_t doca_gpu_dev_eth_txq_commit_strong(struct doca_gpu_eth_txq *eth_txq)

- **eth_txq** – Ethernet send queue GPU handler

__device__ doca_error_t doca_gpu_dev_eth_txq_commit_weak(struct doca_gpu_eth_txq *eth_txq, const uint32_t descr_num)

- **eth_txq** – Ethernet send queue GPU handler
- **descr_num** – Number of queue items enqueued thus far

Only one CUDA thread in the scope (CUDA block or CUDA warp) can invoke this function on the send queue after several enqueue operations. Typical flow is as follows:

1. All threads in the scope enqueue packets in the send queue.

2. Synchronization point.

3. Only one thread in the scope performs the send queue commit.

**doca_gpu_dev_eth_txq_push**

After committing, the items in the send queue must be actually pushed to the network card.

__device__ doca_error_t doca_gpu_dev_eth_txq_push(struct doca_gpu_eth_txq *eth_txq)

- **eth_txq** – Ethernet send queue GPU handler

Only one CUDA thread in the scope (CUDA block or CUDA warp) can invoke this function on the send queue after several enqueue or commit operations. Typical flow is as follows:

1. All threads in the scope enqueue packets in the send queue.
2. Synchronization point.

3. Only one thread in the scope does the send queue commit.

4. Only one thread in the scope does the send queue push.

Section "Produce and Send" provides an example where the scope is a block (e.g., each CUDA block operates on a different Ethernet send queue).

### GPU Functions – RDMA

This section provides a list of DOCA GPUNetIO functions that can be used on the GPU only within a CUDA kernel to execute RDMA operations. These functions offer a strong and a weak mode.

- **rdma** – RDMA queue GPU handler
- **curr_position** – Next available position in the queue
- **mask_max_position** – Mask of the total number of positions in the queue

```c
__device__ doca_error_t __device__ doca_error_t doca_gpu_dev_rdma_get_info(struct doca_gpu_dev_rdma *rdma, uint32_t *curr_position, uint32_t *mask_max_position)
```

```c
__device__ doca_error_t __device__ doca_error_t doca_gpu_dev_rdma_recv_get_info(struct doca_gpu_dev_rdma_r *rdma_r, uint32_t *curr_position, uint32_t *mask_max_position)
```

- **rdma_r** – RDMA receive queue GPU handler
- **curr_position** – Next available position in the queue
- **mask_max_position** – Mask of the total number of positions in the queue

**doca_gpu_dev_rdma_write_**
To RDMA write data onto a remote memory location from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

- **rdma** – RDMA queue GPU handler
- **remote_buf** – Remote DOCA buffer from a DOCA GPU buffer array to write data to
- **remote_offset** – Offset, in bytes, to write data to in the remote buffer
- **local_buf** – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to write
- **local_offset** – Offset, in bytes, to fetch data from in the local buffer
- **length** – Number of bytes to write
- **imm** – Immediate value uint32_t
- **flags** – One of the flags in the doca_gpu_dev_rdma_write_flags enum

```c
__device__ doca_error_t doca_gpu_dev_rdma_write_strong(struct doca_gpu_dev_rdma *rdma,
                                                        struct doca_gpu_buf *remote_buf,
                                                        uint64_t remote_offset,
                                                        struct doca_gpu_buf *local_buf, uint64_t
                                                        local_offset,
                                                        size_t length, uint32_t imm,
                                                        const enum
                                                        doca_gpu_dev_rdma_write_flags flags)
```

```c
__device__ doca_error_t doca_gpu_dev_rdma_write_weak(struct doca_gpu_dev_rdma *rdma,
                                                     struct doca_gpu_buf *remote_buf, uint64_t
                                                     remote_offset,
                                                     struct doca_gpu_buf *local_buf, uint64_t
                                                     local_offset,
                                                     size_t length, uint32_t imm,
                                                     const enum doca_gpu_dev_rdma_write_flags
                                                     flags,
                                                     uint32_t position);
```
- `rdma` – RDMA queue GPU handler
- `remote_buf` – Remote DOCA buffer from a DOCA GPU buffer array to write data to
- `remote_offset` – Offset, in bytes, to write data to in the remote buffer
- `local_buf` – Local DOCA buffer from a DOCA GPU buffer array where to fetch data to write
- `local_offset` – Offset, in bytes, to fetch data in the local buffer
- `length` – Number of bytes to write
- `imm` – Immediate value `uint32_t`
- `flags` – One of the flags in the `doca_gpu_dev_rdma_write_flags` enum
- `position` – Position in the queue to place the RDMA operation. Range: 0 - `mask_max_position`.

**doca_gpu_dev_rdma_read_***

To RDMA read data onto a remote memory location from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes to enqueue operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```c
__device__ doca_error_t doca_gpu_dev_rdma_read_strong(struct doca_gpu_dev_rdma *rdma,
struct doca_gpu_buf *remote_buf,
uint64_t remote_offset,
struct doca_gpu_buf *local_buf, uint64_t
local_offset,
size_t length,
const uint32_t flags_bitmask)
```

- `rdma` – RDMA queue GPU handler
- `remote_buf` – Remote DOCA buffer from a DOCA GPU buffer array where to read data
- `remote_offset` – Offset in bytes to read data to in the remote buffer
- `local_buf` – Local DOCA buffer from a DOCA GPU buffer array where to store remote data
- `local_offset` – Offset in bytes to store data in the local buffer
- `length` – Number of bytes to be read
- `flags_bitmask` – Must be 0; reserved for future use

```c
__device__ doca_error_t doca_gpu_dev_rdma_read_weak(
    struct doca_gpu_dev_rdma *rdma,
    struct doca_gpu_buf *remote_buf, uint64_t remote_offset,
    struct doca_gpu_buf *local_buf, uint64_t local_offset,
    size_t length,
    const uint32_t flags_bitmask,
    uint32_t position);
```

- `rdma` – RDMA queue GPU handler
- `remote_buf` – Remote DOCA buffer from a DOCA GPU buffer array where to read data
- `remote_offset` – Offset in bytes to read data to in the remote buffer
- `local_buf` – Local DOCA buffer from a DOCA GPU buffer array where to store remote data
- `local_offset` – Offset in bytes to store data in the local buffer
- `length` – Number of bytes to be read
- `flags_bitmask` – Must be 0; reserved for future use
- `position` – Position in the queue to place the RDMA operation. Range: 0 - mask_max_position.

`doca_gpu_dev_rdma_send_*`
To RDMA send data from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>device</em> doca_error_t doca_gpu_dev_rdma_send_strong(</td>
<td></td>
</tr>
<tr>
<td>struct doca_gpu_dev_rdma *rdma,</td>
<td></td>
</tr>
<tr>
<td>struct doca_gpu_buf *local_buf, uint64_t</td>
<td></td>
</tr>
<tr>
<td>local_offset,</td>
<td></td>
</tr>
<tr>
<td>size_t length, uint32_t imm,</td>
<td></td>
</tr>
<tr>
<td>doca_gpu_dev_rdma_write_flags flags)</td>
<td></td>
</tr>
<tr>
<td>const enum</td>
<td></td>
</tr>
</tbody>
</table>

- **rdma** – RDMA queue GPU handler
- **local_buf** – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- **local_offset** – Offset in bytes to fetch data in the local buffer
- **length** – Number of bytes to send
- **imm** – Immediate value uint32_t
- **flags** – One of the flags in the doca_gpu_dev_rdma_write_flags enum

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>device</em> doca_error_t doca_gpu_dev_rdma_send_weak(</td>
<td></td>
</tr>
<tr>
<td>struct doca_gpu_dev_rdma *rdma,</td>
<td></td>
</tr>
<tr>
<td>struct doca_gpu_buf *local_buf, uint64_t</td>
<td></td>
</tr>
<tr>
<td>local_offset,</td>
<td></td>
</tr>
<tr>
<td>size_t length, uint32_t imm,</td>
<td></td>
</tr>
<tr>
<td>doca_gpu_dev_rdma_write_flags flags,</td>
<td></td>
</tr>
<tr>
<td>const enum</td>
<td></td>
</tr>
<tr>
<td>uint32_t position);</td>
<td></td>
</tr>
</tbody>
</table>

- **rdma** – RDMA queue GPU handler
- **local_buf** – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- **local_offset** – Offset in bytes to fetch data in the local buffer
• length – Number of bytes to send

• imm – Immediate value uint32_t

• flags – One of the flags in the doca_gpu_dev_rdma_write_flags enum

• position – Position in the queue to place the RDMA operation. Range: 0 - mask_max_position.

doca_gpu_dev_rdma_commit_*

Once all RDMA write, send or read requests have been enqueue in the RDMA queue, a synchronization point must be reached to consolidate and execute those requests. Only 1 CUDA thread can invoke this function at a time.

__device__ doca_error_t doca_gpu_dev_rdma_commit_strong(struct doca_gpu_dev_rdma *rdma)

• rdma – RDMA queue GPU handler

__device__ doca_error_t doca_gpu_dev_rdma_commit_weak(struct doca_gpu_dev_rdma *rdma, uint32_t num_ops)

• rdma – RDMA queue GPU handler

• num_ops – Number of RDMA requests enqueued since the last commit

doca_gpu_dev_rdma_wait_all

After a commit, RDMA requests are executed by the network card as applications move forward doing other operations. If the application needs to verify all RDMA operations have been done by the network card, this "wait all" function can be used to wait for all previous posted operations. Only 1 CUDA thread can invoke this function at a time.
To receive data from an RDMA send, send with immediate, or write with immediate, the destination peer should post a receive operation. DOCA GPUNetIO RDMA receive operations must be done with a `doca_gpu_dev_rdma_r` handler. This handler can be obtained with the function `doca_gpu_dev_rdma_get_recv`.

### Info

This function is optional.

### doca_gpu_dev_rdma_get_recv *

To receive data from an RDMA send, send with immediate, or write with immediate, the destination peer should post a receive operation. DOCA GPUNetIO RDMA receive operations must be done with a `doca_gpu_dev_rdma_r` handler. This handler can be obtained with the function `doca_gpu_dev_rdma_get_recv`.

### Note

All receive operations must use this object.

### `__device__ doca_error_t doca_gpu_dev_rdma_get_recv(struct doca_gpu_dev_rdma *rdma, struct doca_gpu_dev_rdma_r **rdma_r)`

- `rdma` – RDMA queue GPU handler
- `rdma_r` – RDMA receive queue GPU handler
Even for the receive side, in this case, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```c
__device__ doca_error_t doca_gpu_dev_rdma_recv_strong(  
    struct doca_gpu_dev_rdma_r *rdma_r,  
    struct doca_gpu_buf *recv_buf,  
    size_t recv_length,  
    uint64_t recv_offset,  
    const uint32_t flags_bitmask)
```

- **rdma_r** – RDMA receive queue GPU handler
- **recv_buf** – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- **recv_length** – Number of bytes to send
- **recv_offset** – Offset in bytes to fetch data in the local buffer
- **flags_bitmask** – Must be 0; reserved for future use

```c
__device__ doca_error_t doca_gpu_dev_rdma_recv_weak(  
    struct doca_gpu_dev_rdma_r *rdma_r,  
    struct doca_gpu_buf *recv_buf,  
    size_t recv_length,  
    uint64_t recv_offset,  
    const uint32_t flags_bitmask,  
    uint32_t position);
```

- **rdma_r** – RDMA receive queue GPU handler
- **recv_buf** – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- **recv_length** – Number of bytes to send
- **recv_offset** – Offset in bytes to fetch data in the local buffer
- **flags_bitmask** – Must be 0; reserved for future use
- **position** – Position in the queue to place the RDMA operation. Range: 0 - mask_max_position.

### `doca_gpu_dev_rdma_recv_commit_*`

After posting a number of RDMA receive, a commit function must be invoked to activate the receive in the queue. Only 1 CUDA thread can invoke this function at a time.

```c
__device__ doca_error_t doca_gpu_dev_rdma_recv_commit_strong(struct doca_gpu_dev_rdma_r *rdma_r)
```

- **rdma_r** – RDMA receive queue GPU handler

```c
__device__ doca_error_t doca_gpu_dev_rdma_recv_commit_weak(struct doca_gpu_dev_rdma_r *rdma_r, uint32_t num_ops)
```

- **rdma_r** – RDMA receive queue GPU handler
- **num_ops** – Number of RDMA receive requests enqueued since the last commit

### `doca_gpu_dev_rdma_recv_wait_all`

This function waits for the completion of all previously posted RDMA receive operation. Only 1 CUDA thread can invoke this function at a time. It works in blocking or non-blocking mode.

```c
enum doca_gpu_dev_rdma_recv_wait_flags {
    DOCA_GPU_RDMA_RECV_WAIT_FLAG_NB = 0, /**< Non-Blocking mode: the wait receive function checks if the receive operation happened (data has been received) */
    DOCA_GPU_RDMA_RECV_WAIT_FLAG_B = 1, /**< Blocking mode: the receive function is blocked until the receive operation has happened and data is available */
}
```

- * checks if the receive operation happened (data has been received)
- * and exit from the function. If nothing has been received, the function doesn't block the execution.
Function:

```c
__device__ doca_error_t doca_gpu_dev_rdma_recv_wait_all(
    struct doca_gpu_dev_rdma_r *rdma_r,
    const enum doca_gpu_dev_rdma_recv_wait_flags flags,
    uint32_t *num_ops,
    uint32_t *imm_val)
```

- `rdma_r` – RDMA receive queue GPU handler
- `flags` – receive flags
- `num_ops` – Output parameter. Function reports number of completed operations.
- `imm_val` – Output parameter. Application-provided buffer where the function can store received immediate values, if any (or 0xFFFFFFFF if no immediate value is received). If `nullptr`, the function ignores this parameter.

**GPU Functions – DMA**

This section provides a list of DOCA GPUNetIO functions that can be used on the GPU only within a CUDA kernel to execute DMA operations.

**doca_gpu_dev_dma_memcopy**

This function allows a CUDA kernel to trigger a DMA memory copy operation through the DMA GPU engine. There is no strong/weak mode here, the DMA is assuming the strong behavior by default.


- **dma** – DMA queue GPU handler

- **src_buf** – memcpy source buffer

- **src_offset** – fetch data starting from this source buffer offset

- **dst_buf** – memcpy destination buffer

- **dst_offset** – copy data starting from this destination buffer offset

- **length** – number of bytes to copy

---

### `doca_gpu_dev_dma_memcpy`

After posting several DMA memory copies, a commit function must be invoked to execute the operations enqueued in the DMA queue. Only 1 CUDA thread can invoke this function at a time.

```c
__device__ doca_error_t doca_gpu_dev_dma_memcpy(struct doca_gpu_dma *dma, struct doca_gpu_buf *src_buf, uint64_t src_offset, struct doca_gpu_buf *dst_buf, uint64_t dst_offset, size_t length);
```

---

- **dma** – DMA queue GPU handler

---

### `doca_gpu_dev_dma_commit`

```c
__device__ doca_error_t doca_gpu_dev_dma_commit(struct doca_gpu_dma *dma);
```

---

### Building Blocks

This section explains general concepts behind the fundamental building blocks to use when creating a DOCA GPUNetIO application.

### Initialize GPU and NIC
When DOCA GPUNetIO is used in combination with the NIC to send or receive Ethernet traffic, the following must be performed to properly set up the application and devices:

```c
uint16_t dpdk_port_id;
struct doca_dev *ddev;
struct doca_gpu *gdev;
char *eal_param[3] = {"", "-a", "00:00.0");

/* Initialize DPDK with empty device. DOCA device will hot-plug the network card later. */
rte_eal_init(3, eal_param);
/* Create DOCA device on a specific network card */
doca_dpdk_port_probe(&ddev);
get_dpdk_port_id_doca_dev(&ddev, &dpdk_port_id);
/* Create GPUNetIO handler on a specific GPU */
doca_gpu_create(gpu_pcie_address, &gdev);
```

The application would may have to enable different items depending on the task at hand.

### Semaphore

If the DOCA application must dispatch some packets' info across CUDA kernels or from the CUDA kernel and some CPU thread, a semaphore must be created.

A semaphore is a list of items, allocated either on the GPU or CPU (depending on the use case) visible by both the GPU and CPU. This object can be used to discipline communication across items in the GPU pipeline between CUDA kernels or a CUDA kernel and a CPU thread.

By default, each semaphore item can hold info about its status (FREE, READY, HOLD, DONE, ERROR), the number of received packets, and an index of a `doca_gpu_buf` in a `doca_gpu_buf_arr`.

If the semaphore must be used to exchange data with the CPU, a preferred memory layout would be `DOCA_GPU_MEM_CPU_GPU`. Whereas, if the semaphore is only needed across CUDA kernels, `DOCA_GPU_MEM_GPU` is the best memory layout to use.

As an optional feature, if the application must pass more application-specific info through the semaphore items, it is possible to attach a custom structure to each item of the semaphore.
Ethernet Queue with GPU Data Path

Receive Queue

If the DOCA application must receive Ethernet packets, receive queues must be created. The receive queue works in a circular way: At creation time, each receive queue is
associated with a DOCA buffer array allocated on the GPU by the application. Each DOCA buffer of the buffer array has a maximum fixed size.

```c
/* Start DPDK device */
rte_eth_dev_start(dpdk_port_id);
/* Initialise DOCA Flow */
struct doca_flow_port_cfg port_cfg;
port_cfg.port_id = port_id;
doca_flow_init(port_cfg);
doca_flow_port_start();

struct doca_dev *ddev;
struct doca_eth_rxq *eth_rxq_cpu;
struct doca_gpu_eth_rxq *eth_rxq_gpu;
struct doca_mmap *mmap;
void *gpu_buffer;

/* Create DOCA Ethernet receive queues */
doca_eth_rxq_create(ddev, MAX_PACKETS_NUM, MAX_PACKETS_SIZE, &eth_rxq_cpu);

/* Set Ethernet receive queue properties */
/* ... */

/* Create DOCA mmap in GPU memory to be used for the DOCA buffer array associated to this Ethernet queue */
doca_mmap_create(&mmap);
/* Set DOCA mmap properties */
doca_gpu_mem_alloc(gdev, buffer_size, alignment, DOCA_GPU_MEM_GPU, (void **)&gpu_buffer, NULL);
doca_mmap_start(mmap);
doca_eth_rxq_set_pkt_buffer(eth_rxq_cpu, mmap, 0, buffer_size);
/* This DOCA Ethernet Rxq object will be managed by the GPU */
doca_ctx_set_datapath_on_gpu();
/* Start the Ethernet queue object */
/* Export GPU handle for the receive queue */
doca_eth_rxq_get_gpu_handle(eth_rxq_cpu, &eth_rxq_gpu);
```

It is mandatory to associate DOCA Flow pipe(s) to the receive queues. Otherwise, the application cannot receive any packet.
Send Queue

If the DOCA application must send Ethernet packets, send queues must be created in combination with `doca_gpu_buf_arr` to prepare and send packets from GPU memory.

```c
struct doca_dev *ddev;
struct doca_eth_txq *eth_txq_cpu;
struct doca_gpu_eth_txq *eth_txq_gpu;

/* Create DOCA Ethernet send queues */
doca_eth_txq_create(ddev, QUEUE_DEPTH, &eth_txq_cpu);
/* Set properties to send queues */

/* This DOCA Ethernet Rxq object will be managed by the GPU */
doca_ctx_set_datapath_on_gpu();
/* Start the Ethernet queue object */
/* Export GPU handle for the send queue */
doca_eth_txq_get_gpu_handle(eth_txq_cpu, &eth_txq_gpu);

/* Create DOCA mmap to define memory layout and type for the DOCA buf array */
struct doca_mmap *mmap;
doca_mmap_create(&mmap);
/* Set DOCA mmap properties */

/* Create DOCA buf arr and export it to GPU */
struct doca_buf_arr *buf_arr;
struct doca_gpu_buf_arr *buf_arr_gpu;
doca_buf_arr_create(mmap, &buf_arr);
/* Set DOCA buf array properties */
...
/* Export GPU handle for the buf arr */
doca_buf_arr_get_gpu_handle(buf_arr, &buf_arr_gpu);
```

Receive and Process

At this point, the application has created and initialized all the objects required by the GPU to exercise the data path to send or receive packets with GPUNetIO.

In this example, the application must receive packets from different queues with a receiver CUDA kernel and dispatch packet info to a second CUDA kernel responsible for
packet processing.

The CPU launches the CUDA kernels and waits on the semaphore for output:

```c
#define CUDA_THREADS 512
#define CUDA_BLOCKS 1
int semaphore_index = 0;
enum doca_gpu_semaphore_status status;
struct custom_info *gpu_info;

/* On the CPU */
cuda_kernel_receive_dispatch<<<CUDA_THREADS, CUDA_BLOCKS, ..., stream_0>>>(eth_rxq_gpu, sem_to_gpu_gpu)
cuda_kernel_process<<<CUDA_THREADS, CUDA_BLOCKS, ..., stream_1>>>(eth_rxq_gpu, sem_to_cpu_gpu, sem_to_gpu_gpu)
while(/* condition */) {
    doca_gpu_semaphore_get_status(sem_to_cpu, semaphore_index, &status);
    if (status == DOCA_GPU_SEMAPHORE_STATUS_READY) {
        doca_gpu_semaphore_get_custom_info_addr(sem_to_cpu, semaphore_index, (void **)&(gpu_info));
        report_info(gpu_info);
        doca_gpu_semaphore_set_status(sem_to_cpu, semaphore_index, DOCA_GPU_SEMAPHORE_STATUS_FREE);
        semaphore_index = (semaphore_index+1) % SEMAPHORE_ITEMS;
    }
}
```

On the GPU, the two CUDA kernels are running on different streams:

```c
cuda_kernel_receive_dispatch(eth_rxq_gpu, sem_to_gpu_gpu) {
    __shared__ uint32_t rx_pkt_num;
    __shared__ uint64_t rx_buf_idx;
    int semaphore_index = 0;

    while (/* exit condition */) {
        doca_gpu_dev_eth_rxq_receive_block(eth_rxq_gpu, MAX_NUM_RECEIVE_PACKETS, TIMEOUT_RECEIVE_NS, &rx_pkt_num, &rx_buf_idx);
        if (threadIdx.x == 0 && rx_pkt_num > 0) {
```
doca_gpu_dev_sem_set_packet_info(sem_to_gpu_gpu, semaphore_index, DOCA_GPU_SEMAPHORE_STATUS_READY, rx_pkt_num, rx_buf_idx);
    semaphore_index = (semaphore_index+1) % SEMAPHORE_ITEMS;
}
}
}

cuda_kernel_process(eth_rxq_gpu, sem_to_cpu_gpu, sem_to_gpu_gpu) {
    __shared__ uint32_t rx_pkt_num;
    __shared__ uint64_t rx_buf_idx;
    int semaphore_index = 0;
    int thread_buf_idx = 0;
    struct doca_gpu_buf *buf_ptr;
    uintptr_t buf_addr;
    struct custom_info *gpu_info;

    while (/* exit condition */) {
        if (threadIdx.x == 0) {
            do {
                result = doca_gpu_dev_sem_get_packet_info_status(sem_to_gpu_gpu, semaphore_index, DOCA_GPU_SEMAPHORE_STATUS_READY, &rx_pkt_num, &rx_buf_idx);
            } while (result != DOCA_ERROR_NOT_FOUND /* && other exit condition */);
            __syncthreads();
            thread_buf_idx = threadIdx.x;
        }
        while (thread_buf_idx < rx_pkt_num) {
            /* Get DOCA GPU buffer from the GPU buffer in the receive queue */
            doca_gpu_dev_eth_rxq_get_buf(eth_rxq_gpu, rx_buf_idx + thread_buf_idx, &buf_ptr);
            /* Get DOCA GPU buffer memory address */
            doca_gpu_dev_buf_get_addr(buf_ptr, &buf_addr);
            /*
            * Atomic here is has the entire CUDA block accesses the same semaphore to CPU.
            * Smarter implementation can be done at warp level, with multiple semaphores, etc.. to
            avoid this atomic
            */
            int semaphore_index_tmp = atomicAdd_block(&semaphore_index, 1);
            semaphore_index_tmp = semaphore_index_tmp % SEMAPHORE_ITEMS;
            doca_gpu_dev_sem_get_custom_info_addr(sem_to_cpu_gpu, semaphore_index_tmp, (void **)&gpu_info);
            populate_custom_info(buf_addr, gpu_info);
            doca_gpu_dev_sem_set_status(sem_to_cpu_gpu, semaphore_index_tmp, DOCA_GPU_SEMAPHORE_STATUS_READY);
            thread_buf_idx = threadIdx.x;
        }
    }
}
This code can be represented with the following diagram when multiple queues and/or semaphores are used:

Please note that receiving and dispatching packets to another CUDA kernel is not required. A simpler scenario can have a single CUDA kernel receiving and processing packets:
The drawback of this approach is that the time between two receives depends on the time taken by the CUDA kernel to process received packets.

The type of pipeline that must be built heavily depends on the specific use case.

**Produce and Send**

In this example, the GPU produces some data, stores it into packets and then sends them over the network. The CPU launches the CUDA kernels and continues doing other work:

```c
#define CUDA_THREADS 512
#define CUDA_BLOCKS 1
int semaphore_index = 0;
enum doca_gpu_semaphore_status status;
struct custom_info *gpu_info;

/* On the CPU */
cuda_kernel_produce_send<<<CUDA_THREADS, CUDA_BLOCKS, ..., stream_0>>>(eth_txq_gpu, buf_arr_gpu)

/* do other stuff */
```

On the GPU, the CUDA kernel fills the packets with meaningful data and sends them. In the following example, the scope is CUDA block so each block uses a different DOCA Ethernet send queue:

```c
cuda_kernel_produce_send(eth_txq_gpu, buf_arr_gpu) {
    uint64_t doca_gpu_buf_idx = threadIdx.x;
    struct doca_gpu_buf *buf;
    uintptr_t buf_addr;
    uint32_t packet_len;
    uint32_t curr_position;
    uint32_t mask_max_position;
    uint32_t num_pkts_per_send = blockDim.x;

    /* Get last occupied position in the Tx queue */
    doca_gpu_dev_eth_txq_get_info(eth_txq_gpu, &curr_position, &mask_max_position);
    __syncthreads();
```
RDMA Queue with GPU Data Path

To execute RDMA operations from a GPU CUDA kernel, in the setup phase, the application must first create a DOCA RDMA queue, export the RDMA as context, and then set the datapath of the context on the GPU (as shown in the following code snippet).

The following is a pseudo-code to serve as a guide. Please refer to real function signatures in header files (*.h) and documentation for a complete overview of the functions.
At this point, the application has an RDMA queue usable from a GPU CUDA kernel. The next step would be to establish a connection using some OOB (out-of-band) mechanism (e.g., Linux sockets) to exchange RDMA queue info so each peer can connect to the other's queues.

To exchange data, users must create DOCA GPU buffer arrays to send or receive data. If the application also requires read or write, then the GPU memory associated with the buffer arrays must be exported and exchanged with the remote peers using the OOB mechanism.
Please refer to the "RDMA Client Server" sample as a basic layout to implement all the steps described in this section.

### CUDA Kernel for RDMA Write

Assuming the RDMA queues and buffer arrays are correctly created and exchanged across peers, the application can launch a CUDA kernel to remotely write data. As typically applications use strong mode, the following code snippet shows how to use weak mode to post multiple writes from different CUDA threads in the same CUDA block.

```c
__global__ void rdma_write_bw(struct doca_gpu_dev_rdma *rdma_gpu, struct doca_gpu_buf_arr *local_buf_arr, struct doca_gpu_buf_arr *remote_buf_arr)
{
    struct doca_gpu_buf *remote_buf;
    struct doca_gpu_buf *local_buf;
    struct doca_gpu_dev_rdma *rdma_gpu;
    struct doca_gpu_buf_arr *server_local_buf_arr;
    struct doca_gpu_buf_arr *server_remote_buf_arr;
    uint32_t curr_position;
    uint32_t mask_max_position;
    uint32_t num_ops;

    doca_gpu_dev_buf_get_buf(server_local_buf_arr, threadIdx.x, &local_buf);
    doca_gpu_dev_buf_get_buf(server_remote_buf_arr, threadIdx.x, &remote_buf);
    /* Get RDMA queue current available position and mask of the max position number */
    doca_gpu_dev_rdma_get_info(rdma_gpu, &curr_position, &mask_max_position);
```
This section contains two samples that show how to enable simple GPUNetIO features. Be sure to correctly set the following environment variables:

doca_gpu_dev_rdma_write_weak(rdma_gpu,
   /* Write into this remote buffer at offset 0 */
   remote_buf, 0,
   /* Fetch data from this local buffer at offset 0 */
   local_buf, 0,
   /* Number of bytes to write */
   msg_size,
   /* Don't use immediate */
   0, DOCA_GPU_RDMA_WRITE_FLAG_NONE,
   /* Position in the RDMA queue to post the write */
   (curr_position + threadIdx.x) & mask_max_position);

   /* Wait all CUDA threads to post their RDMA Write */
   __syncthreads();

   if (threadIdx.x == 0) {
      /* Only 1 CUDA thread can push the write op just posted */
      doca_gpu_dev_rdma_commit_weak(rdma_gpu, blockDim.x);
      doca_gpu_dev_rdma_commit_all(rdma_gpu, &num_ops);
   }
   __syncthreads();

Info

The code in the "RDMA Client Server" sample shows how to use write and send with immediate flag set.

GPUNetIO Samples
Ethernet Send Wait Time

The sample shows how to enable Accurate Send Scheduling (or wait-on-time) in the context of a GPUNetIO application. Accurate Send Scheduling is the ability of an NVIDIA NIC to send packets in the future according to application-provided timestamps.
This DOCA GPUNetIO sample provides a simple application to send packets with Accurate Send Scheduling from the GPU.

**Synchronizing Clocks**

Before starting the sample, it is important to properly synchronize the CPU clock with the NIC clock. This way, timestamps provided by the system clock are synchronized with the time in the NIC.

For this purpose, at least the `phc2sys` service must be used. To install it on an Ubuntu system:

```
sudo apt install linuxptp
```

To start the `phc2sys` service properly, a config file must be created in `/lib/systemd/system/phc2sys.service`:

```
[Unit]
Description=Synchronize system clock or PTP hardware clock (PHC)
Documentation=man:phc2sys

[Service]
Restart=always
RestartSec=5s
Type=simple
```

**Info**

This NVIDIA blog post offers an example for how this feature has been used in 5G networks.
Now \texttt{phc2sys} service can be started:

\begin{verbatim}
sudo systemctl stop systemd-timesyncd
sudo systemctl disable systemd-timesyncd
sudo systemctl daemon-reload
sudo systemctl start phc2sys.service
\end{verbatim}

To check the status of \texttt{phc2sys}:

\begin{verbatim}
$ sudo systemctl status phc2sys.service

phc2sys.service - Synchronize system clock or PTP hardware clock (PHC)
 Loaded: loaded (/lib/systemd/system/phc2sys.service; disabled; vendor preset: enabled)
 Active: active (running) since Mon 2023-04-03 10:59:13 UTC; 2 days ago
   Docs: man:phc2sys
 Main PID: 337824 (sh)
    Tasks: 2 (limit: 303788)
   Memory: 560.0K
    CPU: 52min 8.199s
   CGroup: /system.slice/phc2sys.service
           337824 /bin/sh -c "taskset -c 15 /usr/sbin/phc2sys -s /dev/ptp$(ethtool -T ens6f0 | grep PTP | awk '{print $4}') -c CLOCK_REALTIME -n 24 -O 0 -R 256 -u 256"
           337829 /usr/sbin/phc2sys -s /dev/ptp3 -c CLOCK_REALTIME -n 24 -O 0 -R 256 -u 256

Apr 05 16:35:52 doca-vr-045 phc2sys[337829]: [457395.040] CLOCK_REALTIME rms  8 max  18 freq
+110532 +/- 27 delay  770 +/-  3
Apr 05 16:35:53 doca-vr-045 phc2sys[337829]: [457396.071] CLOCK_REALTIME rms  8 max  20 freq
+110513 +/- 30 delay  769 +/-  3
Apr 05 16:35:54 doca-vr-045 phc2sys[337829]: [457397.102] CLOCK_REALTIME rms  8 max  18 freq
+110527 +/- 30 delay  769 +/-  3
Apr 05 16:35:55 doca-vr-045 phc2sys[337829]: [457398.130] CLOCK_REALTIME rms  8 max  18 freq
+110517 +/- 31 delay  769 +/-  3
\end{verbatim}
At this point, the system and NIC clocks are synchronized so timestamps provided by the CPU are correctly interpreted by the NIC.

⚠️ **Warning**

The timestamps you get may not reflect the real time and day. To get that, you must properly set the `ptp4l` service with an external grand master on the system. Doing that is out of the scope of this sample.

### Running the Sample

The sample is shipped with the source files that must be built:

```bash
# Ensure DOCA and DPDK are in the pkgconfig environment variable
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_send_wait_time
meson build
ninja -C build
```
The sample sends 8 bursts of 32 raw Ethernet packets or 1kB to a dummy Ethernet address, 10:11:12:13:14:15, in a timed way. Program the NIC to send every $t$ nanoseconds (command line option `-t`).

The following example programs a system with GPU PCIe address ca:00.0 and NIC PCIe address 17:00.0 to send 32 packets every 5 milliseconds:

```bash
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
$ sudo ./build/doca_gpunetio_send_wait_time -n 17:00.0 -g ca:00.0 -t 5000000

[1316878][DOCA][INF][gpunetio_send_wait_time_main.c:195][main] Starting the sample
[09:22:54:438260][1316878][DOCA][INF][gpunetio_send_wait_time_main.c:224][main] Sample configuration:
  GPU ca:00.0
  NIC 17:00.0
  Timeout 5000000ns
EAL: Detected CPU lcores: 128
...
EAL: Probe PCI driver: mlx5_pci (15b3:a2d6) device: 0000:17:00.0 (socket 0)
[09:22:54:819996][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:607]
  [gpunetio_send_wait_time] Wait on time supported mode: DPDK
EAL: Probe PCI driver: gpu_cuda (10de:20b5) device: 0000:ca:00.0 (socket 1)
[09:22:54:830212][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:252][create_tx_buf]
  Mapping send queue buffer (0x0x7f48e32a0000 size 262144B) with legacy nvidia-peermem mode
[09:22:54:832462][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:657]
  [gpunetio_send_wait_time] Launching CUDA kernel to send packets
[09:22:54:842945][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:664]
  [gpunetio_send_wait_time] Waiting 10 sec for 256 packets to be sent
[09:23:04:883309][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:684]
  [gpunetio_send_wait_time] Sample finished successfully
[09:23:04:883339][1316878][DOCA][INF][gpunetio_send_wait_time_main.c:239][main] Sample finished successfully

To verify that packets are actually sent at the right time, use a packet sniffer on the other side (e.g., tcpdump):

```bash
$ sudo tcpdump -i enp23s0f1np1 -A -s 64

17:12:23.480318 IP5 (invalid)
Sent from DOCA GPUNetIO.........................
```
The output should show a jump of approximately 5 milliseconds every 32 packets.

**Note**

tcpdump may increase latency in sniffing packets and reporting the receive timestamp, so the difference between bursts of 32 packets reported may be less than expected, especially with small interval times like 500 microseconds (\(-t 500000\)).

**Ethernet Simple Receive**

This simple application shows the fundamental steps to build a DOCA GPUNetIO receiver application with one queue for UDP packets and one CUDA kernel receiving those packets from the GPU, printing packet info to the console.

**Warning**

Invoking a printf from a CUDA kernel is not good practice for release software and should be used only to print debug information as it
To build and run the application:

```
# Ensure DOCA and DPDK are in the pkgconfig environment variable
cd /opt/mellanox/doa/samples/doa_gpunetio/gpunetio_simple_receive
meson build
ninja -C build
```

To test the application, this guide assumes the usual setup with two machines: one with the DOCA receiver application and the second one acting as packet generator. As UDP packet generator, this example considers the `nping` application that can be easily installed easily on any Linux machine.

The command to send 10 UDP packets via `nping` on the packet generator machine is:

```
$ nping --udp -c 10 -p 2090 192.168.1.1 --data-length 1024 --delay 500ms
```

Starting Nping 0.7.80 (https://nmap.org/nping) at 2023-11-20 11:05 UTC
SENT (0.0018s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (0.5018s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (1.0025s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (1.5025s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (2.0032s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (2.5033s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (3.0040s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (3.5040s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (4.0047s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (4.5048s) UDP packet with 1024 bytes to 192.168.1.1:2090

Max rtt: N/A | Min rtt: N/A | Avg rtt: N/A
UDP packets sent: 10 | Rcvd: 0 | Lost: 10 (100.00%)
Nping done: 1 IP address pinged in 5.50 seconds

Assuming the DOCA Simple Receive sample is waiting on the other machine at IP address 192.168.1.1.
The DOCA Simple Receive sample is launched on a system with NIC at 17:00.1 PCIe address and GPU at ca:00.0 PCIe address:

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
$ sudo ./build/doca_gpunetio_simple_receive -n 17:00.1 -g ca:00.0
[11:00:30:397080][2328673][DOCA][INF][gpunetio_simple_receive_main.c:159][main] Starting the sample
[11:00:30:652622][2328673][DOCA][INF][gpunetio_simple_receive_main.c:189][main] Sample configuration:
  GPU ca:00.0
  NIC 17:00.1

EAL: Detected CPU lcores: 128
EAL: Detected NUMA nodes: 2
EAL: Detected shared linkage of DPDK
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
EAL: Selected IOVA mode 'PA'
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: mlx5_pci (15b3:a2d6) device: 0000:17:00.1 (socket 0)
[11:00:31:036760][2328673][DOCA][WRN][engine_model.c:72][adapt_queue_depth] adapting queue depth to 128.
[11:00:31:928926][2328673][DOCA][WRN][engine_port.c:321][port_driver_process_properties] detected representor used in VNF mode (driver port id 0)
EAL: Probe PCI driver: gpu_cuda (10de:20b5) device: 0000:ca:00.0 (socket 1)
[11:00:31:977261][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:425][create_rxq] Creating Sample Eth Rxq

[11:00:31:977841][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:466][create_rxq] Mapping receive queue buffer (0x0x7f86cc000000 size 33554432B) with nvidia-peermem mode
[11:00:32:043182][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:610] [gpunetio_simple_receive] Launching CUDA kernel to receive packets
[11:00:32:055193][2328673][DOCA][INF][gpunetio_simple_receive_sample.c:614] [gpunetio_simple_receive] Waiting for termination
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
 Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth dst 10:70:fd:fa:77:e9
```

RDMA Client Server

This sample exhibits how to use the GPUNetIO RDMA API to receive and send/write with immediate using a single RDMA queue.

The server has a GPU buffer array A composed by GPU_BUF_NUM doca_gpu_buf elements, each 1kB in size. The client has two GPU buffer arrays, B and C, each composed by GPU_BUF_NUM doca_gpu_buf elements, each 512B in size.

The goal is for the client to fill a single server buffer of 1kB with two GPU buffers of 512B as illustrated in the following figure:

To show how to use RDMA write and send, even buffers are sent from the client with write immediate, while odd buffers are sent with send immediate. In both cases, the server must pre-post the RDMA receive operations.
For each buffer, the CUDA kernel code repeats the handshake:

```
For each buffer, the CUDA kernel code repeats the handshake:
```

Once all buffers are filled, the server double checks that all values are valid. The server output should be as follows:

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
$ cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_rdma_client_server_write
$ ./build/doca_gpunetio_rdma_client_server_write -gpu 17:00.0 -d mlx5_0

[14:11:43:000930][1173110][DOCA][INF][gpunetio_rdma_client_server_write_main.c:250][main] Starting the sample
EAL: Detected CPU lcores: 64
EAL: Detected NUMA nodes: 2
EAL: Detected shared linkage of DPDK
EAL: Selected IOVA mode 'VA'
EAL: No free 2048 kB hugepages reported on node 0
EAL: No free 2048 kB hugepages reported on node 1
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: gpu_cuda (10de:2331) device: 0000:17:00.0 (socket 0)
[14:11:43:686581][1173110][DOCA][INF][rdma_common.c:64][oob_connection_server_setup] Socket created successfully
[14:11:43:686598][1173110][DOCA][INF][rdma_common.c:83][oob_connection_server_setup] Done with binding
[14:11:43:686610][1173110][DOCA][INF][rdma_common.c:91][oob_connection_server_setup] Listening for incoming connections
```
Client connected at IP: 192.168.2.28 and port: 46274
Send connection details to remote peer size 216
Receive remote connection details
Connect DOCA RDMA to remote RDMA
Create local server mmap A context
Send exported mmap A to remote client
Receive client mmap F export
Create local DOCA buf array context A
Create local DOCA buf array context F
Create remote DOCA buf array context F
Before launching CUDA kernel, buffer array A is:
RDMA Recv 2 ops completed with immediate values 0 and 1!
RDMA Recv 2 ops completed with immediate values 1 and 2!
RDMA Recv 2 ops completed with immediate values 2 and 3!
RDMA Recv 2 ops completed with immediate values 3 and 4!
After launching CUDA kernel, buffer array A is:
RDMA Recv 2 ops completed with immediate values 0 and 1!
On the other side, assuming the server is at IP address 192.168.2.28, the client output should be as follows:

```plaintext
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable

$ cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_rdma_client_server_write
$ ./build/doca_gpunetio_rdma_client_server_write -gpu 17:00.0 -d mlx5_0 -c 192.168.2.28

EAL: Detected CPU lcores: 64
EAL: Detected NUMA nodes: 2
EAL: Detected shared linkage of DPDK
EAL: Selected IOVA mode 'PA'
EAL: No free 2048 kB hugepages reported on node 0
EAL: No free 2048 kB hugepages reported on node 1
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: gpu_cuda (10de:2331) device: 0000:17:00.0 (socket 0)
```

Client waiting on flag 7f6596735000 for server to post RDMA Recvs
Thread 0 post rdma write imm 0
Thread 1 post rdma write imm 0
Client waiting on flag 7f6596735001 for server to post RDMA Recvs
Thread 0 post rdma send imm 1
Thread 1 post rdma send imm 1
Client waiting on flag 7f6596735002 for server to post RDMA Recvs
Thread 0 post rdma write imm 2
GPU DMA Copy

This sample exhibits how to use the DOCA DMA and DOCA GPUNetIO libraries to DMA copy a memory buffer from the CPU to the GPU (with DOCA DMA CPU functions) and from the GPU to the CPU (with DOCA GPUNetIO DMA device functions) from a CUDA kernel. This sample requires a DPU as it uses the DMA engine on it.

Note

With RDMA, the network device must be specified by name (e.g., mlx5_0) instead of the PCIe address (as is the case for Ethernet).

Warning

Printing from a CUDA kernel is not recommended for performance. It may make sense for debugging purposes and for simple samples like this one.

$ cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_dma_memcpy

# Build the sample and then execute

$ ./build/doca_gpunetio_dma_memcpy -g 17:00.0 -n ca:00.0
Starting the sample
EAL: Detected CPU cores: 64
EAL: Detected NUMA nodes: 2
EAL: Detected shared linkage of DPDK
EAL: Selected IOVA mode 'VA'
EAL: No free 2048 kB hugepages reported on node 0
EAL: No free 2048 kB hugepages reported on node 1
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: gpu_cuda (10de:2331) device: 0000:17:00.0 (socket 0)
The CPU source buffer value to be copied to GPU memory: This is a sample piece of text from CPU
The GPU source buffer value to be copied to CPU memory: This is a sample piece of text from GPU
CUDA KERNEL INFO: The GPU destination buffer value after the memcpy: This is a sample piece of text from CPU
CPU received message from GPU: This is a sample piece of text from GPU
Cleanup DMA ctx with GPU data path
Cleanup DMA ctx with CPU data path
Sample finished successfully
DOCA App Shield

This guide provides instructions on using the DOCA App Shield API.

Introduction

DOCA App Shield API offers a solution for strong intrusion detection capabilities using the DPU services to collect and analyze data from the host's (or a VM on the host) memory in real time. This solution provides intrusion detection and forensics investigation in a way that is:

- Robust against attacks on a host machine
- Able to detect a wide range of attacks (including zero-day attacks)
- Least disruptive to the execution of host application (where current detection solutions hinder the performance of host applications)
- Transparent to the host, such that the host does not need to install anything (other than providing some files obtained from the `doca_apsh_config.py` tool)

App Shield uses a DMA device to access the host's memory and analyze it.

The App Shield API provides multiple functions that help with gathering data extracted from system's memory (e.g., processes list, modules list, connections). This data helps with detecting attacks on critical services or processes in a system (e.g., services that enforce integrity or privacy of the execution of different applications).

Prerequisites

1. Configure the NVIDIA® BlueField® networking platform's (DPU or SuperNIC) firmware.

   1. On BlueField, configure the PF base address register and NVMe emulation.
      Run:
If working with VFs, configure NVME emulation, SR-IOV, and number of VFs. Run:

```
dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 s PF_BAR2_SIZE=2 PF_BAR2_ENABLE=1

dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 s NVME_EMULATION_ENABLE=1 SRIOV_EN=1 NUM_OF_VFS=<vf-number>
```

2. Perform **graceful shutdown** and a cold boot from the host.

**Info**

These configurations can be checked using the following command:

```
dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 q | grep -E "NVME|BAR|SRIOV|NUM_OF_VFS"
```

2. Download target system (host/VM) symbols.

   - For Ubuntu:

```
host> sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
deb http://ddebs.ubuntu.com/ $(lsb_release -cs) main restricted universe multiverse
deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-updates main restricted universe multiverse
deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-proposed main restricted universe multiverse
EOF

host> sudo apt install ubuntu-dbgsym-keyring
host> sudo apt-get update
```
3. Perform IOMMU passthrough. This stage is only necessary if IOMMU is not enabled by default (e.g., when the host is using an AMD CPU).

**Note**

Skip this step if you are not sure whether it is needed. Return to it only if DMA fails with a message similar to the following in `dmesg`:

```
host> dmesg
[ 3839.822897] mlx5_core 0000:81:00.0: AMD-Vi: Event logged
[IO_PAGE_FAULT domain=0x0047 address=0x2a0aff8 flags=0x0000]
```

1. Locate your OS’s grub file (most likely `/boot/grub/grub.conf`, `/boot/grub2/grub.cfg`, or `/etc/default/grub`) and open it for editing. Run:

```
host> vim /etc/default/grub
```

2. Search for the line defining `GRUB_CMDLINE_LINUX_DEFAULT` and add the argument `iommu=pt`. For example:
3. Run:

![Info]

**Note**

Prior to performing a power cycle, make sure to do a **graceful shutdown**.

- **For Ubuntu:**

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

- **For CentOS:**

  ```
  host> grub2-mkconfig -o /boot/grub2/grub.cfg
  ```

4. Prepare target:

1. Install DOCA on the target system.

2. Create the ZIP and JSON files. Run:

```bash
target-system> cd /opt/mellanox/doca/tools/
target-system> python3 doca_apsh_config.py --pid <pid-of-process-to-monitor> --os <windows/linux> --path <path to dwarf2json executable or pdbparse-to-json.py>
target-system> cp /opt/mellanox/doca/tools/*/.<shared-folder-with-baremetal>
dpu> scp <shared-folder-with-baremetal>/* <path-to-app-shield-binary>
```
If the target system does not have DOCA installed, the script can be copied from BlueField.

The required `dwaf2json` and `pdbparse-to-json.py` are not provided with DOCA.

**Note**

If the kernel and process `.exe` have not changed, there is no need to redo this step.

**Dependencies**

The library requires firmware version 24.32.1010 or higher.

**API**

For the library API reference, refer to the DOCA APSH API documentation in the NVIDIA DOCA Library APIs.

**Note**

The pkg-config (*.pc file) for the APSH library is `doca-apsh`.

The following subsections provide more details about the library API.

**doca_apsh_dma_dev_set**

To attach a DOCA DMA device to App Shield, calling this function is mandatory and must be done before calling `doca_apsh_start`.

```
doca_apsh_dma_dev_set(doca_apsh_ctx, doca_dev)
```
- `doca_apsh_ctx [in]` – App Shield opaque context struct
- `doca_dev [in]` – struct for DOCA Device with DMA capabilities

## Capabilities Per System

For each initialized system, App Shield retrieves an array of the requested object according to the getter's name:

<table>
<thead>
<tr>
<th>Getter Function Name</th>
<th>Functions Information</th>
<th>Functions Signature</th>
<th>Return Type</th>
</tr>
</thead>
</table>
| Get modules          | Returns an array with information about the system modules (drivers) loaded into the kernel of the OS. | ```

doca_error_t
doca_apsh_modules_get(struct
doca_apsh_system *system,
struct doca_apsh_module
**modules, int
*modules_size);
``` | - Array of struct doca_apsh_module
- int – size of the returned array
- `doca_error_status` |
<table>
<thead>
<tr>
<th>Getter Function Name</th>
<th>Functions Information</th>
<th>Functions Signature</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get processes</td>
<td>Returns an array with information about each process running on the system.</td>
<td>doca_error_t doca_apsh_processes_get(struct doca_apsh_system *system, struct doca_apsh_process ***processes, int *processes_size);</td>
<td>• Array of struct doca_apsh_process</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• int – size of the returned array</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• doca_error_status</td>
</tr>
<tr>
<td>Get library</td>
<td>For a specified process, this function returns an array with information about each library loaded into this process.</td>
<td>doca_error_t doca_apsh_libs_get(struct doca_apsh_process *process, struct doca_apsh_lib ***libs, int *libs_size);</td>
<td>• Array of struct doca_apsh_lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• int – size of the returned array</td>
</tr>
<tr>
<td>Getter Function Name</td>
<td>Functions Information</td>
<td>Functions Signature</td>
<td>Return Type</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------</td>
<td>---------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Get threads</td>
<td>For a specified process, this function returns an array with information about each thread running within this process.</td>
<td>doca_error_t doca_apsh_threads_get(struct doca_apsh_process *process, struct doca_apsh_thread ***threads, int *threads_size);</td>
<td>d array • doca_error_status</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Array of struct doca_apsh_thread • int – size of the returned array • doca_error_status</td>
</tr>
<tr>
<td>Getter Function Name</td>
<td>Functions Information</td>
<td>Functions Signature</td>
<td>Return Type</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------</td>
<td>---------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| Get virtual memory areas/virtual address description | For a specified process, this function returns an array with information about each virtual memory area within this process. | doca_error_t
doca_apsh_vads_get(struct
doca_apsh_process *process,
struct doca_apsh_vad ***vads,
int *vads_size); | • Array of struct doca_apsh_vm 
• int – size of the returned array
• doca_error_status |
<table>
<thead>
<tr>
<th>Getter Function Name</th>
<th>Functions Information</th>
<th>Functions Signature</th>
<th>Return Type</th>
</tr>
</thead>
</table>
| Get privileges        | For a specified process, this function returns an array with information about each possible privilege for this process, as described here. | doca_error_t
doca_apsh_privileges_get(struct doca_apsh_process *process,
struct doca_apsh_privilege ***privileges, int
*privileges_size); | • Array of struct doca_apsh_privilege
• int – size of the returned array
• doca_error_status |

**Note**
Available on a Windows host only.
<table>
<thead>
<tr>
<th>Getter Function Name</th>
<th>Functions Information</th>
<th>Functions Signature</th>
<th>Return Type</th>
</tr>
</thead>
</table>
| Get environment variables | For a specified process, this function returns an array with information about each environment variable within this process. | doca_error_t
doca_apsh_envvars_get(struct doca_apsh_process *process, struct doca_apsh_envvar ***envars, int *envars_size); | • Array of struct doca_apsh_envar  
• int – size of the returned array  
• doca_error_stat us |

**Note**  
Available on a Windows host only.
<table>
<thead>
<tr>
<th>Getter Function Name</th>
<th>Functions Information</th>
<th>Functions Signature</th>
<th>Return Type</th>
</tr>
</thead>
</table>
| Get handles          | For a specified process, this function returns an array with information about each handle this process holds. | doca_error_t  
doca_apsh_handles_get(struct  
doca_apsh_process *process,  
struct doca_apsh_handle  
***handles, int *handles_size); | - Array of struct doca_apsh_handle  
- int – size of the returned array  
- doca_error_status |

**Note**
Available on a Windows host only.
<table>
<thead>
<tr>
<th>Getter Function Name</th>
<th>Functions Information</th>
<th>Functions Signature</th>
<th>Return Type</th>
</tr>
</thead>
</table>
| Get LDR modules      | For a specified process, this function returns an array with information about each loaded module within this process. | `doca_error_t
doca_apsh_ldrmodules_get(struct doca_apsh_process *process, struct doca_apsh_ldrmodule ***ldrmodules, int *ldrmodules_size);` | - Array of struct doca_apsh_ldrmodule |
|                      |                        |                      | - int – size of the returned array |
|                      |                        |                      | - doca_error_stat_t |
|                      | Note Available on a Windows host only. | | |
| Process attestation  | For a specified process, this function attests the memory pages of the process according to a precomputed golden hash file given as an input. | `doca_error_t
doca_apsh_attestation_get(struct doca_apsh_process *process, const char *exec_hash_map_path, struct doca_apsh_attestation ***attestation, int *attestation_size);` | - Array of struct doca_apsh_attestation |
<p>|                      | Note Single-threaded processes are | | - int – size of the |</p>
<table>
<thead>
<tr>
<th>Getter Function Name</th>
<th>Functions Information</th>
<th>Functions Signature</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>supported at beta level.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attestation refresh</td>
<td>Refreshes a single attestation handler of a process with a new snapshot.</td>
<td>doca_error_t doca_apsh_attst_refresh(struct doca_apsh_attestation ***attestation, int *attestation_size);</td>
<td>Array of struct doca_apsh_attestation, int size of the returned array, doca_error_status</td>
</tr>
<tr>
<td>Getter Function Name</td>
<td>Functions Information</td>
<td>Functions Signature</td>
<td>Return Type</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------</td>
<td>---------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Get NetS...</td>
<td>This function...</td>
<td><code>doca_error_t</code></td>
<td>• Array of <code>doca_apsh_netscan</code></td>
</tr>
<tr>
<td>Getter Function Name</td>
<td>Functions Information</td>
<td>Functions Signature</td>
<td>Return Type</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------</td>
<td>---------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>

ℹ️ **Note**
This feature is currently supported at beta level.
<table>
<thead>
<tr>
<th>Getter Function Name</th>
<th>Functions Information</th>
<th>Functions Signature</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get process parameters</td>
<td>For a specified process, this function returns a struct object (not an array) with information about the process' parameters (ones not included in the &quot;get processes&quot; capability).</td>
<td><code>doca_error_t</code> <code>doca_apsh_process_parameters_get(struct doca_apsh_process *process, struct doca_apsh_process_parameters **process_parameters);</code></td>
<td>- An object of struct <code>doca_apsh_process_parameters</code> - <code>doca_error_t</code> status</td>
</tr>
<tr>
<td>Get security identifier (SID)</td>
<td>For a specified process, this function returns an array with information about each SID (security identifier) included in the process's security context.</td>
<td><code>doca_error_t</code> <code>doca_apsh_sids_get(struct doca_apsh_process *process, struct doca_apsh_sid ***sids, int *sids_size);</code></td>
<td>- Array of struct <code>doca_apsh_sid</code> - int – size of the return</td>
</tr>
</tbody>
</table>

**Note**
Available on a Windows host only.

**Note**
This feature is currently supported at beta level.
<table>
<thead>
<tr>
<th>Getter Function Name</th>
<th>Functions Information</th>
<th>Functions Signature</th>
<th>Return Type</th>
</tr>
</thead>
</table>
| Perform Yara scan    | For a specified process, this function returns an array with information about each Yara rule match found in the process's memory. | doca_error_t  
doca_apsh_yara_get(struct  
doca_apsh_process *process,  
enum doca_apsh_yara_rule  
yara_rules_arr, uint32_t  
yara_rules_arr_size,  
uint64_t scan_type, struct  
doca_apsh_yara  
***yara_matches, int  
*yara_matches_size); | • Array of struc  
t doca_apsh_yara  
• int – size of the retu  
rne d array  
• doca _error  
status |

**Note**
Available on a Windows host and Ubuntu 22.04 DPU.

**Note**
To get a better understanding of the arguments, refer to document
<table>
<thead>
<tr>
<th>Getter Function Name</th>
<th>Functions Information</th>
<th>Functions Signature</th>
<th>Return Type</th>
</tr>
</thead>
</table>
| Get containers        | Returns an array with information about each container running on the system. | `doca_error_t doca_apsh_containers_get(struct doca_apsh_system *system, struct doca_apsh_container ***containers, int *containers_size);` | - Array of struct doca_apsh_container info 
- `int` - size of the returned array 
- doca_error_status |
<p>| Get container's processes | For a specified container, this function returns an array with information about each process running within this container. | <code>doca_error_t doca_apsh_container_processes_get(struct doca_apsh_container *container, struct</code> | - Array of struct doca_aps |</p>
<table>
<thead>
<tr>
<th>Getter Function Name</th>
<th>Functions Information</th>
<th>Functions Signature</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Note</strong></td>
<td>Available on a Linux host only.</td>
<td>doca_apsh_process ***processes, int *processes_size);</td>
<td>h_process</td>
</tr>
</tbody>
</table>
| **Note** | Only available for containers on the following runtimes:  
  - runc  
  - containerd | doca_apsh_module_info_get(struct doca_apsh_module *module, enum doca_apsh_module_attr attr);  
doca_apsh_lib_info_get(struct doca_apsh_lib *lib, enum doca_apsh_lib_attr attr);  
doca_apsh_thread_info_get(struct doca_apsh_thread *thread, enum doca_apsh_lib_attr attr);  
doca_apsh_vad_info_get(struct doca_apsh_vad *vad, enum doca_apsh_vad_attr attr);  
doca_apsh_privilege_info_get(struct doca_apsh_privilege *privilege, enum doca_apsh_privilege_attr attr);  
doca_apsh_envar_info_get(struct doca_apsh_envar *envar, enum doca_apsh_envar_attr attr);  
doca_apsh_handle_info_get(struct doca_apsh_handle *handle, enum doca_apsh_handle_attr attr); | int – size of the returned array  
  - doca_error  
  - status |
App Shield Initialization and Teardown

To use App Shield, users must initialize and configure two main structs. This section presents these structs and explains how to interact with them.

**doca_apsh_ctx**

doca_apsh_ctx is the basic struct used by App Shield which defines the DMA device used to perform the memory forensics techniques required to run App Shield.

### Note

The same doca_apsh_ctx struct may be used to run multiple App Shield instances over different systems (e.g., two different VMs on the host).
1. To acquire an instance of the `doca_apsh_ctx` struct, use the following function:

```c
struct doca_apsh_ctx *doca_apsh_create(void);
```

2. To configure the `doca_apsh_ctx` instance with DMA device to use:

```c
doca_error_t doca_apsh_dma_dev_set(struct doca_apsh_ctx *ctx, struct doca_dev *dma_dev);
```

3. To start the `doca_apsh_ctx` instance, call the following function:

```c
doca_error_t doca_apsh_start(struct doca_apsh_ctx *ctx);
```

4. To destroy the `doca_apsh_ctx` instance when it is no longer needed, call:

```c
void doca_apsh_destroy(struct doca_apsh_ctx *ctx);
```

**doca_apsh_system**

The `doca_apsh_system` struct is built on the `doca_apsh_ctx` instance. This struct is created per system running App Shield. `doca_apsh_system` defines multiple attributes used by App Shield to perform memory analysis over the specific system successfully.

1. To acquire an instance of the `doca_apsh_system` struct, use the following function:

```c
const uint pid = doca_apsh_process_info_get(processes[i], DOCA_APSH_PROCESS_PID);
const char *proc_name = doca_apsh_process_info_get(processes[i], DOCA_APSH_PROCESS_COMM);
```
- OS type – specifies the system's OS type.

```c
#include <doapsh_api.h>
doca_error_t doca_apsh_sys_os_type_set(struct doca_apsh_system *ctx, enum doca_apsh_system_os os_type);
```

**Note**

Currently supported types: Windows or Linux.

- System representor – specifies the representor of the device connected to the system for App Shield to run on (which can be a representor of VF/PF). For information on querying the DOCA device, refer to the DOCA Core.

After acquiring the DOCA device, use the following function to configure it into the system instance:

```c
#include <doapsh_api.h>
doca_error_t doca_apsh_sys_dev_set(struct doca_apsh_system *system, struct doca_dev_rep *dev);
```

- System symbols map – includes information about the OS that App Shield is attempting to run on (e.g., Window 10 Build 18363) and the size and fields of the OS structures, which helps App Shield with the memory forensic techniques it uses to access and analyze these structures in the system's memory. This can be obtained by running the `doca_apsh_config.py` on the system machine.

After obtaining it, run:

```c
#include <doapsh_api.h>
doca_error_t doca_apsh_sys_os_symbol_map_set(struct doca_apsh_system *system, const char *system_os_symbol_map_path);
```
- Memory regions – includes the physical addresses of the memory regions which are mapped for system memory RAM. This is needed to prevent App Shield from accessing other memory regions, such as memory mapped I/O regions. This can be obtained by running the `doca_apsh_config.py` tool on the system machine.

  After obtaining it, run:

  ```c
  doca_error_t doca_apsh_sys_mem_region_set(struct doca_apsh_system *system, const char *system_mem_region_path);
  ```

- KPGD file (optional and relevant only for Linux OS) – contains the KPGD physical address and the virtual address of `init_task`. This information is required since App Shield extracts data from the kernel struct in the physical memory. Thus, the kernel page directory table must translate the virtual addresses of these structs. This can be obtained by running the `doca_apsh_config.py` tool on the system machine with the flag `find_kpgd=1`. Since setting this attribute is optional, App Shield can work without it, but providing it speeds up App Shield's initialization process.

  After obtaining it, run:

  ```c
  doca_error_t doca_apsh_sys_kpgd_file_set(struct doca_apsh_system *system, const char *system_kpgd_file_path);
  ```

3. To start the `doca_apsh_system`:

  ```c
  doca_error_t doca_apsh_system_start(struct doca_apsh_system *system);
  ```

4. To destroy the `doca_apsh_system` instance when it is no longer needed, call:

  ```c
  void doca_apsh_system_destroy(struct doca_apsh_system *system);
  ```
**doca_apsh_config.py Tool**

The `doca_apsh_config.py` tool is a python3 script which can be used to obtain all the attributes needed to run `doca_apsh_system` instance.

The following parameters are necessary to use the tool:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pid</strong> (optional)</td>
<td>The process ID of the process we want to run attestation capability on</td>
</tr>
<tr>
<td><strong>os</strong> (mandatory)</td>
<td>The OS type of the machine (i.e., Linux or Windows)</td>
</tr>
<tr>
<td><strong>find_kpgd</strong> (optional)</td>
<td>Relevant for Linux OS only, AS flag to enable/disable creating kpgd_file.conf. Default 0.</td>
</tr>
<tr>
<td><strong>files</strong> (mandatory)</td>
<td>A list of files for the tool to create. File options: hash, symbols, memregions, kpgd_file (only relevant for Linux).</td>
</tr>
</tbody>
</table>

**Note**
Make sure that the value set is appropriate for your setup.

- **path** (mandatory)
  - Linux – path to the dwarf2json executable. Default `/dwarf2json`. This file can be obtained by compiling the following project using Go.
  - Windows – path to pdbparse-to-json.py. Default `./pdbparse-to-json.py`. This file can be found here.

**Note**
Make sure that the value set is appropriate for your setup.
The tool creates the following files:

- Symbol map – this file changes once the system kernel is updated or a kernel module is installed. The file does not change on system reboot.
- Memory regions – this file changes when adding or removing hardware or drivers that affect the system's memory map (e.g., when adding register addresses). The file does not change on system reboot.
- hash.zip – this file is required for attestation but is unnecessary for all other capabilities. The ZIP file contains the required data to attest to a single process. The file changes on library or executable update.
- kpgd_file.conf (relevant for Linux OS only) – helps with faster initialization of the library. The file changes on system reboot.

DOCA App Shield Samples

This section provides DOCA App Shield library sample implementations on top of BlueField.

Info

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Sample Prerequisites

Follow the guidelines in section "Prerequisites" then copy the generated JSON files, symbols.json and mem_regions.json, to the /tmp/ directory.

Running the Sample
1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

```bash
    cd /opt/mellanox/doca/samples/doca_apsh/<sample_name>
    meson /tmp/build
    ninja -C /tmp/build
```

**Note**

The binary `doca_<sample_name>` will be created under `/tmp/build/`.

3. Sample (e.g., `apsh_libs_get`) usage:

```
Usage: doca_apsh_libs_get [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, --pid` Process ID of process to be analyzed
- `-f, --vuid` VUID of the System device
- `-d, --dma` DMA device name
```
4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

## Samples

### Apsh Libs Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of loadable libraries of a specific process.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCI device via given vendor unique identifier (VUID).
5. Creating DOCA Apsh system handler.
7. Getting the list of system process using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process-loadable libraries using `doca_apsh_libs_get` Apsh API call.
9. Querying the libraries for 3 selected fields using `doca_apsh_lib_info_get` Apsh API call.
10. Printing libraries' attributes to the terminal.
11. Cleaning up.
References:

- /opt/mellanox/doca/samples/doca_apsh/apsh_libs_get/apsh_libs_get_sample.c
- /opt/mellanox/doca/samples/doca_apsh/apsh_libs_get/apsh_libs_get_main.c
- /opt/mellanox/doca/samples/doca_apsh/apsh_libs_get/meson.build
- /opt/mellanox/doca/samples/doca_apsh/apsh_common.c;
  /opt/mellanox/doca/samples/doca_apsh/apsh_common.h

**Apsh Modules Get**

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of installed modules on a monitored system.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCI device via given VUID.
5. Creating DOCA Apsh system handler.
7. Getting the list of system-installed modules using `doca_apsh_modules_get` Apsh API call.
8. Querying the names of modules using `doca_apsh_module_info_get` Apsh API call.
9. Printing the attributes of up to 5 modules attributes to the terminal.
10. Cleaning up.

References:
Apsh Pslist

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of running processes on a monitored system.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCI device via given VUID.
5. Creating DOCA Apsh system handler.
7. Getting the list of processes running on the system using `doca_apsh_processes_get` Apsh API call.
8. Querying the processes for 4 chosen attributes using `doca_apsh_proc_info_get` Apsh API call.
9. Printing the attributes of up to 5 processes to the terminal.
10. Cleaning up.

References:

- /opt/mellanox/doca/samples/doca_apsh/apsh_pslist/apsh_pslist_sample.c
Apsh Threads Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of threads of a specific process.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCI device via given VUID.
5. Creating DOCA Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process threads using `doca_apsh_threads_get` Apsh API call.
9. Querying the threads for up to 3 selected fields using `doca_apsh_thread_info_get` Apsh API call.
10. Printing thread attributes to the terminal.
11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_threads_get/apsh_threads_get_sample.c`
Apsh Vads Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of virtual address descriptors (VADs) of a specific process.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and start the Apsh context.
4. Opening DOCA remote PCI device via given VUID.
5. Creating DOCA Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process VADs using `apsh_vads_get` Apsh API call.
9. Querying the VADs for 3 selected fields using `apsh_vad_info_get` Apsh API call.
10. Printing the attributes of up to 5 VADs to the terminal.
11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_vads_get/apsh_vads_get_sample.c`
Apsh Envars Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of environment variables of a specific process.

Note

This sample works only on target systems with Windows OS.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process envars using `doca_apsh_envars_get` Apsh API call.
9. Querying the envars for 2 selected fields using `doca_apsh_envar_info_get` Apsh API call.
10. Printing the envars attributes to the terminal.

11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_envars_get/apsh_envvars_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_envars_get/apsh_envvars_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_envars_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

**Apsh Privileges Get**

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of privileges of a specific process.

**Note**

This sample works only on target systems with Windows OS.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.

8. Getting the list of process privileges using the `doca_apsh_privileges_get` Apsh API call.

9. Querying the privileges for 5 selected fields using the `doca_apsh_privilege_info_get` Apsh API call.

10. Printing the privileges attributes to the terminal.

11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_privileges_get/apsh_privileges_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_privileges_get/apsh_privileges_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_privileges_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

**Apsh Containers Get**

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of running containers on a monitored system, as well as getting a list of processes for each container.

⚠️ **Note**

This sample works only on target systems with Linux OS.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.

3. Setting and starting the Apsh context.

4. Opening DOCA remote PCIe device using specific VUID.

5. Creating DOCA Apsh system handler.


7. Getting the list of containers running on the system using `doca_apsh_containers_get` Apsh API call.

8. Querying the containers for container ID attribute using `doca_apsh_container_info_get` Apsh API call.

9. Getting list of processes for each container using `doca_apsh_container_processes_get` Apsh API call.

10. Printing the attributes of up to 5 processes to the terminal.

11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_containers_get/apsh_containers_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_containers_get/apsh_containers_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_containers_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`
DOCA Compress

This guide provides instructions on how to use the DOCA Compress API.

Introduction

DOCA Compress library provides an API to compress and decompress data using hardware acceleration, supporting both host and NVIDIA® BlueField® DPU memory regions.

The library provides an API for executing compress operations on DOCA buffers, where these buffers reside in either the DPU memory or host memory.

Using DOCA Compress, compress and decompress memory operations can be easily executed in an optimized, hardware-accelerated manner.

This document is intended for software developers wishing to accelerate their application's compress memory operations.

Prerequisites

The DOCA Compress library follows the architecture of a DOCA Core Context. It is recommended to read the following sections before proceeding:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

Changes From Previous Releases

Changes in 2.8

The following subsection(s) detail the doca_compress library updates in version 2.8.0.
Removed

- doca_compress_cap_task_decompress_lz4_is_supported
- doca_compress_cap_task_decompress_lz4_get_max_buf_size
- doca_compress_cap_task_decompress_lz4_get_max_buf_list_len
- doca_compress_task_decompress_lz4_set_conf
- doca_compress_task_decompress_lz4_alloc_init
- doca_compress_task_decompress_lz4_as_task
- doca_compress_task_decompress_lz4_set_src
- doca_compress_task_decompress_lz4_get_src
- doca_compress_task_decompress_lz4_set_dst
- doca_compress_task_decompress_lz4_get_dst
- doca_compress_task_decompress_lz4_get_crc_cs
- doca_compress_task_decompress_lz4_get_adler_cs

Environment

DOCA Compress-based applications can run either on the host machine or on the BlueField DPU target.

Compress can only be run with a DPU configured with DPU mode as described in NVIDIA BlueField Modes of Operation.

Architecture

DOCA Compress is a DOCA Context as defined by DOCA Core. See NVIDIA DOCA Core Context for more information.
DOCA Compress leverages DOCA Core architecture to expose asynchronous tasks that are offloaded to hardware.

**Compress operation:**

```
src_buf  
data  
\[\text{compress}\]  
doca_compress  
LIB  
dst_buf  
data  
+  
Checksum of src_buf
```

**Decompress operation:**

```
src_buf  
data  
\[\text{decompress}\]  
doca_compress  
LIB  
dst_buf  
data  
+  
Checksum of dst_buf buffer after decompress
```

**Supported Compress/Decompress Algorithms**

For BlueField-2 devices, this library supports:

- Compress operation using the deflate algorithm
- Decompress operation using the deflate algorithm

For BlueField-3 devices, this library supports:

- Decompress operation using the deflate algorithm
- Decompress operation using the LZ4 algorithm

**Supported Checksum Methods**

Depending on the task type, the following checksum methods are produced and may be retrieved using the relevant getter functions:

- Adler – produced by the deflate compress and decompress tasks
- CRC – produced by all tasks
- xxHash – produced by the LZ4 decompress tasks
Refer to "Tasks" section for more information.

**Objects**

**Device and Device Representor**

The library requires a DOCA device to operate, the device is used to access memory and perform the actual copy. See DOCA Core Device Discovery for information.

For same BlueField DPU, it does not matter which device is used (PF/VF/SF), as all these devices utilize the same hardware component. If there are multiple DPUs, it is possible to create a Compress instance per DPU, providing each instance with a device from a different DPU.

To access memory that is not local (from the host to the DPU or vice versa), then the DPU side of the application must pick a device with an appropriate representor. See DOCA Core Device Representor Discovery.

The device must stay valid as long as the Compress instance is not destroyed.

**Memory Buffers**

All compress/decompress tasks require two DOCA buffers containing the destination and the source. Depending on the allocation pattern of the buffers, refer to the Inventory Types table.

Buffers must not be modified or read during the compress/decompress operation.

**Source and Destination Location**

DOCA Compress can process DOCA buffers that reside on the host, the DPU, or both.

**Local Host**
Source and destination buffers reside on the host and the compress library runs on the host.

**Local DPU**

Source and destination buffers reside on the DPU and the compress library runs on the DPU.

**Remote**

**Source at Host, Destination at DPU**

- The source resides on the host and is exported (DOCA mmap export) to the DPU
- The destination resides on the DPU
- The compress library runs on the DPU and compresses/decompresses the host source to the DPU destination

**Source at DPU, Destination at Host**

- The source resides on the DPU
- The destination resides on the host and is exported (DOCA mmap export) to the DPU
- Compress library runs on the DPU and compresses/decompresses the DPU source to the host destination

**Configuration Phase**
To start using the library, the user must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

**Configurations**

The context can be configured to match the use case of the application.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

**Mandatory Configurations**

The following configurations must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of Tasks.
- A device with appropriate support must be provided upon creation

**Device Support**

DOCA Compress requires a device to operate. To pick a device, see DOCA Core Device Discovery.

As device capabilities may change in the future (see DOCA Core Device Support), it is recommended to select your device using the following APIs:

**Supported Tasks**

- doca_compress_cap_task_compress_deflate_is_supported
- doca_compress_cap_task_decompress_deflate_is_supported
- doca_compress_cap_task_decompress_lz4_stream_is_supported
- `doca_compress_cap_task_decompress_lz4_block_is_supported`

### Supported Buffer Size

- `doca_compress_cap_task_compress_deflate_get_max_buf_size`
- `doca_compress_cap_task_decompress_deflate_get_max_buf_size`
- `doca_compress_cap_task_decompress_lz4_stream_get_max_buf_size`
- `doca_compress_cap_task_decompress_lz4_block_get_max_buf_size`

### Buffer Support

Tasks support buffers with the following features:

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Source Buffer</th>
<th>Destination Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List Buffer</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Local mmap Buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>mmap From PCI Export Buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>mmap From RDMA Export Buffer</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Execution Phase

This section describes execution on CPU or DPU using [DOCA Core Progress Engine](#).

### Tasks

**Compress Deflate Task**
This task facilitates compressing memory, with the deflate algorithm, using buffers as described in section "Buffer Support".

**Note**

DOCA compress returns only the payload. To create a compressed file, (e.g., gzip), the developer must add a gzip header/trailer.

### Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to set the configuration</th>
<th>API to query support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_compress_task_compress_deflate_set_conf</td>
<td>doca_compress_cap_task_compress_deflate_is_supported</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_compress_task_compress_deflate_set_conf</td>
<td>doca_compress_get_max_num_tasks (max total num tasks)</td>
</tr>
<tr>
<td>Maximal buffer size</td>
<td>–</td>
<td>doca_compress_cap_task_compress_deflate_get_max_buf_size</td>
</tr>
<tr>
<td>Maximum buffer list size</td>
<td>–</td>
<td>doca_compress_cap_task_compress_deflate_get_max_buf_list_len</td>
</tr>
</tbody>
</table>

### Task Input

Common input as described in **DOCA Core Task**.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source buffer</td>
<td>Buffer pointing to the memory to be compressed</td>
<td>Only the data residing in the data segment is compressed</td>
</tr>
<tr>
<td>Destination buffer</td>
<td>Buffer pointing to where compressed memory will be stored</td>
<td>The data is compressed to the tail segment extending the data segment</td>
</tr>
</tbody>
</table>
Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The source data is compressed to destination
- The destination buffer data segment is extended to include the compressed data
- Adler can be retrieved by calling `doca_compress_task_compress_deflate_get_adler_cs`
- CRC can be retrieved by calling `doca_compress_task_compress_deflate_get_crc_cs`

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task
Decompress Deflate Task

This task facilitates decompressing memory, with the deflate algorithm, using buffers as described in section "Buffer Support".

**Note**

DOCA decompress expects the payload alone. To decompress a file (e.g. gzip), the developer must strip the header/trailer.

### Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_compress_task_decompress_deflate_set_conf</td>
<td>doca_compress_cap_task_decompress_deflate_is_supported</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_compress_task_decompress_deflate_set_conf</td>
<td>doca_compress_get_max_num_tasks (max-total-num-tasks)</td>
</tr>
<tr>
<td>Maximal buffer size</td>
<td>-</td>
<td>doca_compress_cap_task_decompress_deflate_get_max_buf_size</td>
</tr>
<tr>
<td>Maximum buffer list size</td>
<td>-</td>
<td>doca_compress_cap_task_decompress_deflate_get_max_buf_list_len</td>
</tr>
</tbody>
</table>

### Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>source buffer</td>
<td>Buffer pointing to the memory to be decompressed</td>
<td>Only the data residing in the data segment is decompressed</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>destination buffer</td>
<td>Buffer pointing to where decompressed memory will be stored</td>
<td>The data is decompressed to the tail segment extending the data segment</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in DOCA Core Task.

**Task Completion Success**

After the task completes successfully, the following happens:

- The source data is decompressed to destination
- The destination buffer data segment is extended to include the decompressed data
- Adler can be retrieved by calling `doca_compress_task_decompress_deflate_get_adler_cs`
- CRC can be retrieved by calling `doca_compress_task_decompress_deflate_get_crc_cs`

**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

**Task Limitations**

- The operation is not atomic
Once the task has been submitted, the source and destination should not be read/written to

- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

### Decompress LZ4 Tasks

These tasks facilitate decompressing memory with the LZ4 algorithm, using buffers as described in section "Buffer Support".

The main differences between the tasks is the input data format –

- The decompress LZ4 stream task expects a stream of one or more blocks, without the frame (i.e., the magic number, frame descriptor, and content checksum)
- The decompress LZ4 block task expects a single, compressed, data-only block (i.e., without block size or block checksum)

### Decompress LZ4 Stream Task

This task facilitates decompressing memory with the LZ4 algorithm, using buffers as described in section "Buffer Support".

**Note**

The decompress LZ4 stream task expects a stream of one or more blocks without the frame (i.e., the magic number, frame descriptor, and content checksum).

### Task Configuration
### Task Input

Common input as described in [DOCA Core Task](#).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Has block checksum Flag</td>
<td>A flag to indicate whether or not the blocks in the stream have a checksum</td>
<td>1 if the task should expect blocks in the stream to have a checksum; 0 otherwise</td>
</tr>
<tr>
<td>Are blocks independent flag</td>
<td>A flag to indicate whether or not each block depends on previous blocks in the stream</td>
<td>1 the task should expect blocks to be independent; 0 otherwise (dependent blocks)</td>
</tr>
<tr>
<td>Source buffer</td>
<td>Buffer pointing to the memory to be decompressed</td>
<td>Only the data residing in the data segment is decompressed</td>
</tr>
<tr>
<td>Destination buffer</td>
<td>Buffer pointing to where decompressed memory will be stored</td>
<td>The data is decompressed to the tail segment extending the data segment</td>
</tr>
</tbody>
</table>

### Task Output

Common output as described in [DOCA Core Task](#).
Task Completion Success

After the task completes successfully:

- The source data is decompressed to destination
- The destination buffer data segment is extended to include the decompressed data
- CRC can be retrieved by calling `doca_compress_task_decompress_lz4_stream_get_crc_cs`
- xxHash can be retrieved by calling `doca_compress_task_decompress_lz4_stream_get_xxh_cs`

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

Decompress LZ4 Block Task
This task facilitates decompressing memory with the LZ4 algorithm, using buffers as described in section "Buffer Support".

⚠️ Note

The decompress LZ4 block task expects a single, compressed, data-only block (i.e., without block size or block checksum).

### Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_compress_task_decompress_lz4_block_set_conf</code></td>
<td><code>doca_compress_cap_task_decompress_lz4_block_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_compress_task_decompress_lz4_block_set_conf</code></td>
<td><code>doca_compress_get_max_num_tasks (max total num tasks)</code></td>
</tr>
<tr>
<td>Maximal buffer size</td>
<td>—</td>
<td><code>doca_compress_cap_task_decompress_lz4_block_get_max_buf_size</code></td>
</tr>
<tr>
<td>Maximum buffer list size</td>
<td>—</td>
<td><code>doca_compress_cap_task_decompress_lz4_block_get_max_buf_list_len</code></td>
</tr>
</tbody>
</table>

### Task Input

Common input as described in **DOCA Core Task**.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source buffer</td>
<td>Buffer pointing to the memory to be decompressed</td>
<td>Only the data residing in the data segment will be decompressed</td>
</tr>
<tr>
<td>Destination buffer</td>
<td>Buffer pointing to where decompressed memory will be stored</td>
<td>The data is decompressed to the tail segment extending the data segment</td>
</tr>
</tbody>
</table>
Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully:

- The source data is decompressed to destination
- The destination buffer data segment is extended to include the decompressed data
- CRC can be retrieved by calling `doca_compress_task_decompress_lz4_block_get_crc_cs`
- xxHash can be retrieved by calling `doca_compress_task_decompress_lz4_block_get_xxh_cs`

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination should not be read/written to
- Source and destination must not overlap
Events

DOCA Compress exposes asynchronous events to notify about changes that happen unexpectedly according to DOCA Core architecture.

The only events DOCA Compress expose are common events (DOCA CTX state changed). See more info in DOCA Core Event.

State Machine

The DOCA Compress library follows the Context state machine described in DOCA Core Context State Machine.

This section describes how to move states and what is allowed in each state.

States

Idle

In this state, it is expected that application:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:
### Starting

This state cannot be reached.

### Running

In this state, it is expected that application:

- Allocates and submit tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocate previously configured task
- Submit a task
- Call stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

### Stopping

In this state, it is expected that application:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all tasks have been freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all tasks are completed and freed</td>
</tr>
</tbody>
</table>
- Calls progress to complete all inflight tasks (tasks will complete with failure)
- Frees any completed tasks

Allowed operations:

- Call progress

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Call progress and fatal error occurs</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop without freeing all tasks</td>
</tr>
</tbody>
</table>

**Alternative Datapath Options**

DOCA Compress only supports datapath on CPU, see [Execution Phase](#).

**DOCA Compress Samples**

The following samples illustrate how to use the DOCA Compress API to compress and decompress files.

**Note**

DOCA Compress handles payload only unless the `zc` flag is used (available only for deflate samples). In that case, a zlib header and trailer are added in compression and it is considered as part of the input when decompressing.
Running the Sample

1. Refer to the following documents:

   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

   cd /opt/mellanox/doca/samples/doca_compress/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build

   Info
   The binary doca_<sample_name> is created under /tmp/build/.

3. Sample (e.g., doca_compress_deflate) usage:

   Common arguments

   Usage: doca_<sample_name> [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 device PCI device address
-f, --file Input file to compress/decompress
-o, --output Output file
-c, --output-checksum Output checksum

- Sample-specific arguments

<table>
<thead>
<tr>
<th>Sample</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress/Decompress Deflate</td>
<td>-wf, -with-frame</td>
<td>Write/read a file with a frame, compatible with default zlib settings</td>
</tr>
<tr>
<td>Decompress LZ4 Stream</td>
<td>-bc, --has-block-checksum</td>
<td>Flag to indicate if blocks have a checksum</td>
</tr>
<tr>
<td></td>
<td>-bi, --are-blocks-independent</td>
<td>Flag to indicate if blocks are independent</td>
</tr>
<tr>
<td></td>
<td>-wf, -with-frame</td>
<td>Read a file compatible with an LZ4 frame</td>
</tr>
</tbody>
</table>

4. For additional information per sample, use the -h option:

	/tmp/build/doca_<sample_name> -h

**Samples**

**Compress/Decompress Deflate**

This sample illustrates how to use DOCA Compress library to compress or decompress a file.
The sample logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with two relevant buffers; one for the source data and one for the result.
4. Allocating elements in DOCA buffer inventory for each buffer.
5. Allocating and initializing a DOCA Compress deflate task or a DOCA Decompress deflate task.
6. Submitting the task.
7. Running the progress engine until the task is completed.
8. Writing the result into an output file, `out.txt`.
9. Destroying all DOCA Compress and DOCA Core structures.

References:

- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/compress_deflate_sample.c`
- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/compress_deflate_main.c`
- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/meson.build`
- `/opt/mellanox/doca/samples/doca_compress/decompress_deflate/decompress_deflate_sample.c`
- `/opt/mellanox/doca/samples/doca_compress/decompress_deflate/decompress_deflate_main.c`
- `/opt/mellanox/doca/samples/doca_compress/decompress_deflate/meson.build`
- `/opt/mellanox/doca/samples/doca_compress/compress_common.h`
- `/opt/mellanox/doca/samples/doca_compress/compress_common.c`

**Decompress LZ4 Stream**
This sample illustrates how to use DOCA Compress library to decompress a file using the LZ4 stream decompress task.

The sample logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with two relevant buffers; one for the source data and one for the result.
4. Allocating elements in DOCA buffer inventory for each buffer.
5. Allocating and initializing an DOCA Decompress LZ4 stream task.
6. Submitting the task.
7. Running the progress engine until the task is completed.
8. Writing the result into an output file, out.txt.
9. Destroying all DOCA Compress and DOCA Core structures.

References:

- /opt/mellanox/doca/samples/doca_compress/decompress_lz4_stream/decompress_lz4_stream_main.c
- /opt/mellanox/doca/samples/doca_compress/decompress_lz4_stream/meson.build
- /opt/mellanox/doca/samples/doca_compress/compress_common.h
- /opt/mellanox/doca/samples/doca_compress/compress_common.c

**Backward Compatibility**

**Decompress LZ4 Task**
The decompress LZ4 task has been removed. To facilitate decompressing memory with the LZ4 algorithm, use the decompress LZ4 stream task or the decompress LZ4 block task instead.
DOCA SHA

This guide provides instructions on building and developing applications that calculate message digest using the SHA1, SHA2-256 or SHA2-512 algorithms.

Introduction

Note

The DOCA SHA library is currently supported at alpha level.

The library provides an API for executing SHA operations on DOCA buffers, where the buffers reside in either local memory (i.e., within the same host) or host memory accessible by the NVIDIA® BlueField®-2 device (remote memory). Using DOCA SHA, complex cryptographic hash operations can be easily executed in an optimized, hardware-accelerated manner.

Note

NVIDIA® BlueField®-3 does not support this library because it has no SHA acceleration engine.

This document is intended for software developers wishing to accelerate their applications' SHA calculations typically used in digital signature schemes or hash-based message authentication code calculations.

Prerequisites
This library follows the architecture of a DOCA Core context, it is recommended to read the following sections before:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

**Environment**

DOCA SHA-based applications can run either on the host machine or on the BlueField-2 DPU target.

DOCA SHA calculations from the host to BlueField and vice versa can only be run when the DPU is configured in **DPU mode**.

**Architecture**

DOCA SHA is a DOCA Core Context. This library leverages the DOCA Core architecture to expose asynchronous tasks/events offloaded to hardware.

SHA can be used to calculate message digest as illustrated in the following diagrams:

- SHA from local memory to local memory:

  ![Diagram of SHA from local memory to local memory]

- Using the DPU to do SHA using the memory between the host and the DPU:
Using the host to do SHA calculation using memory between the host and the DPU:

Objects

Device and Representor

The library requires a DOCA device to operate. The device is used to access memory and perform the actual SHA calculation. See DOCA Core Device Discovery.

For the same BlueField DPU, it does not matter which device is used (i.e., PF/VF/SF) as these devices utilize the same hardware component. If there are multiple DPUs, then it is possible to create a SHA instance per DPU, providing each instance with a device from a different DPU.
To access non-local memory (i.e., from the host to DPU or vice versa), the DPU side of the application must choose a device with an appropriate representor (see DOCA Core Device Representor Discovery). The device must stay valid for as long as the SHA instance is not destroyed.

**Memory Buffers**

The SHA task requires at least two DOCA buffers containing the destination and the source. Depending on the allocation pattern of the buffers, refer to the DOCA Core Inventory Types table.

Buffers must not be modified or read during the SHA operation. For information on what kind of memory is supported, refer to the table in section "Buffer Support".

**Configuration Phase**

To start using the library, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow the execution of tasks and retrieval of events.

**Configurations**

The context can be configured to match the application use case.

To find if a configuration is supported or its min/max value, refer to section "Device Support".

**Mandatory Configurations**

These configurations must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of tasks and/or events in sections "Tasks" and "Events" respectively for information.
A device with appropriate support must be provided upon creation

### Device Support

DOCA SHA requires a device to operate. For information on choosing a device, see DOCA Core Device Discovery.

As device capabilities may change in the future (see DOCA Core Device Support) it is recommended to select your device using the following methods:

- `doca_sha_cap_task_hash_get_supported`
- `doca_sha_cap_task_partial_hash_get_supported`

Some devices can allow different capabilities such as:

- The maximum number of tasks
- The maximum source buffer size
- The minimum destination buffer size
- The maximum supported number of elements in DOCA linked-list buffer
- Check whether SHA1, SHA2-256 or SHA2-512 is supported

### Buffer Support

Tasks support buffers with the following features:

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Source Buffer</th>
<th>Destination Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local mmap buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from PCIe export buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from RDMA export buffer</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Buffer Type | Source Buffer | Destination Buffer
---|---|---
Linked list buffer | Yes | No

### Execution Phase

This section describes execution on the CPU using DOCA Core Progress Engine.

### Tasks

DOCA SHA exposes asynchronous tasks that leverage DPU hardware according to DOCA Core architecture.

#### SHA Task

The SHA task `doca_sha_task_hash` allows one-shot SHA calculation using buffers as described in section "Buffer Support". One-shot means that the source buffer is used as a whole input, therefore, the SHA operation is completed after this task completion event arrives.

![A single doca_sha_task_hash](source buffer → destination buffer)

### Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_sha_task_hash_set_conf</code></td>
<td><code>doca_sha_cap_task_hash_get_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_sha_task_hash_set_conf</code></td>
<td></td>
</tr>
<tr>
<td>Maximal source buffer size</td>
<td>-</td>
<td><code>doca_sha_cap_get_max_src_buf_size</code></td>
</tr>
<tr>
<td>Maximum source buffer list size</td>
<td>-</td>
<td><code>doca_sha_cap_get_max_list_buf_num_elem</code></td>
</tr>
<tr>
<td>Description</td>
<td>API to Set Configuration</td>
<td>API to Query Support</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>--------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Minimum destination buffer size</td>
<td></td>
<td>doca_sha_cap_get_min_dst_buf_size</td>
</tr>
</tbody>
</table>

**Task Input**

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source buffer</td>
<td>Buffer pointing to the memory used for SHA calculation</td>
<td>Only the data residing in the data segment is to be used</td>
</tr>
<tr>
<td>Destination buffer</td>
<td>Buffer pointing to the memory used for writing the SHA calculation result</td>
<td>The SHA result is appended to the tail segment</td>
</tr>
<tr>
<td>SHA algorithm</td>
<td>SHA algorithm to be used in SHA calculation</td>
<td>Must be one of DOCA_SHA_ALGORITHM_SHA1, DOCA_SHA_ALGORITHM_SHA256, DOCA_SHA_ALGORITHM_SHA512</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in DOCA Core Task.

**Task Completion Success**

After the task completes successfully, the following happens:

- The SHA calculation of data from the source buffer is successfully completed and the result is written to the destination buffer
- The destination buffer data segment is extended to include the SHA result data
Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in DOCA Core Task

Partial-SHA Task

The partial-SHA task `doca_sha_task_partial_hash` allows stateful SHA calculation for a collection of messages. Using buffers as described in section "Buffer Support".

Stateful means that the input data is composed of many segments (may be spatial or timely non-consecutive), therefore, its SHA calculation requires more than one one-shot SHA operation to finish. During any stateful operation, other independent SHA tasks can also be executed.
Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_sha_task_partial_hash_set_conf</code></td>
<td><code>doca_sha_cap_task_partial_hash_get_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_sha_task_partial_hash_set_conf</code></td>
<td>-</td>
</tr>
<tr>
<td>Maximal source buffer size</td>
<td>-</td>
<td><code>doca_sha_cap_get_max_src_buf_size</code></td>
</tr>
<tr>
<td>Maximum source buffer list size</td>
<td>-</td>
<td><code>doca_sha_cap_get_max_list_buf_num_elem</code></td>
</tr>
<tr>
<td>Minimum destination buffer size</td>
<td>-</td>
<td><code>doca_sha_cap_get_min_dst_buf_size</code></td>
</tr>
<tr>
<td>SHA block size</td>
<td></td>
<td><code>doca_sha_cap_get_partial_hash_block_size</code></td>
</tr>
</tbody>
</table>
Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source buffer</td>
<td>Buffer pointing to the memory to be used for SHA calculation</td>
<td>Only the data residing in the data segment is to be used. And the data length for the non-last data segment must be multiple of the SHA block size queried by doca_sha_cap_get_partial_hash_block_size</td>
</tr>
<tr>
<td>Destination buffer</td>
<td>Buffer pointing to the memory is used for writing the SHA calculation result</td>
<td>The SHA result is appended to the tail segment. During the whole calculation process, this buffer cannot be modified.</td>
</tr>
<tr>
<td>SHA algorithm type</td>
<td>SHA algorithm to be used in SHA calculation</td>
<td>Must be one of DOCA_SHA_ALGORITHM_SHA1, DOCA_SHA_ALGORITHM_SHA256, DOCA_SHA_ALGORITHM_SHA512</td>
</tr>
<tr>
<td>Whether the current source buffer is the last segment</td>
<td>Indicate whether the current source Buffer is the last segment data to be used for partial-SHA calculation</td>
<td>Use doca_sha_task_partial_hash_set_is_final_buf to set this property</td>
</tr>
<tr>
<td>Set source buffer</td>
<td>Use to set the subsequent source segment buffer after the initial doca_sha_task_partial_hash_task is allocated</td>
<td>doca_sha_task_partial_hash_set_src</td>
</tr>
</tbody>
</table>

Task Output

Common output as described in DOCA Core Task.

Task Completion Success
After the task completes successfully, the following happens:

- The SHA calculation of data from the source buffer is successfully completed and the result is written to the destination buffer
- The destination buffer data segment is extended to include the SHA result data

**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects is not modified
- The destination buffer contents may be modified

**Task Limitations**

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in [DOCA Core Task](#)

**Events**

DOCA SHA exposes asynchronous events to notify about changes that happen unexpectedly according to the DOCA Core architecture.

The only events SHA exposes are common events as described in [DOCA Core Event](#).
State Machine

The DOCA SHA library follows the context state machine as described in DOCA Core Context State Machine.

The following section describes moving states and what is allowed in each state.

Idle

In this state, it is expected that the application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all tasks have been freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all tasks are completed and freed</td>
</tr>
</tbody>
</table>

Starting

This state cannot be reached.

Running
In this state, it is expected that the application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

**Stopping**

In this state, it is expected that the application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Calling progress

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Call progress and fatal error occurs</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop without freeing all tasks</td>
</tr>
</tbody>
</table>
Alternative Datapath Options

DOCA SHA only supports datapath on the CPU. See section "Execution Phase".

DOCA SHA Samples

This section describes DOCA SHA samples based on the DOCA SHA library.

The samples in this section illustrate how to use the DOCA SHA API to do the following:

- Do SHA calculation of contents of a buffer, and write result to another buffer
- Chop the contents of a buffer into a collection of segments, and do partial-SHA calculation of this collection of segments, and write result to another

Info

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Running the Samples

1. Refer to the following documents:

- NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
- NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

    cd /opt/mellanox/doca/samples/doca_sha/<sample_name>
    meson/tmp/build
ninja -C/tmp/build

3. Sample (e.g., doca_sha_create) usage:

Usage: doca_sha_create [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:
- `-d, --data` user data

4. For additional information per sample, use the `-h` option:

/tmp/build/doca_<sample_name>-h

Samples

SHA Create

This sample illustrates how to perform SHA calculation with DOCA SHA.
The sample logic includes:

1. Locating DOCA device.

2. Initializing required DOCA Core structures.

3. Setting the task_pool configuration for doca_sha_task_hash.

4. Populating DOCA memory map with two relevant buffers.

5. Allocating element in DOCA buffer inventory for each buffer.

6. Allocating and initializing a doca_sha_task_hash.

7. Submitting the task.

8. Retrieving task result once it is done.

Reference:

- /opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_sample.c
- /opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_main.c
- /opt/mellanox/doca/samples/doca_sha/sha_create/meson.build

**SHA-Partial Create**

This sample illustrates how to perform partial-SHA calculation for a collection of data segments with DOCA SHA.

The sample logic includes:

1. Locating DOCA device.

2. Initializing the required DOCA Core structures.

3. Setting the task_pool configuration for doca_sha_task_partial_hash.

4. Chopping the source data into a collection of data segments according to the selected SHA algorithm's block size
5. Populating DOCA memory map with needed buffers for all source data segments and destination buffer.

6. Allocating element in DOCA buffer inventory for the first source buffer and destination buffer.

7. Allocating and initializing a `doca_sha_task_partial_hash` with the first source buffer and the destination buffer.

8. Iteratively repeating the following sub-steps until all data segments are consumed:
   
   1. Submitting the `doca_sha_task_partial_hash`.
   
   2. Waiting for the submitted task to finish.
   
   3. Allocating a `doca_buf` for the next source segment and use `doca_sha_task_partial_hash_set_src` to set it as source buffer of the above allocated task.

   4. If it is the final segment, use `doca_sha_task_partial_hash_set_is_final_buf` to mark it in the allocate task.

9. Retrieving the result of the final iteration in the destination buffer as the full partial-SHA calculation result.

10. Destroying all SHA and DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partial_create_sample.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partial_create_main.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/meson.build`
DOCA Erasure Coding

This guide provides instructions on how to use the DOCA Erasure Coding API.

Introduction

Note

This library is currently supported at alpha version.

The DOCA Erasure Coding (known also as forward error correction or FEC) library provides an API to encode and decode data using hardware acceleration, supporting both host and NVIDIA® BlueField®-3 (and higher) DPU memory regions.

DOCA Erasure Coding recovers lost data fragments by creating generic redundancy fragments (backup). Each redundancy block that the library creates can help recover any block in the original data should a total loss of fragment occur. This increases data redundancy and reduces data overhead.

The library provides an API for executing erasure coding (EC) operations on DOCA buffers residing in either the DPU or host memory.

This document is intended for software developers wishing to accelerate their application's EC memory operations.

Glossary

Familiarize yourself with the following terms to better understand the information in this document:
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Original data, original blocks, blocks of original data to be protected/preserved</td>
</tr>
<tr>
<td>Coding matrix</td>
<td>Coefficients, the matrix used to generate the redundancy blocks and recovery</td>
</tr>
<tr>
<td>Redundancy blocks</td>
<td>Codes; encoded data; the extra blocks that help recover data loss</td>
</tr>
<tr>
<td>Encoding</td>
<td>The process of creating the redundancy blocks. Encoded data is referred to as the original blocks or redundancy blocks.</td>
</tr>
<tr>
<td>Decoding</td>
<td>The process of recovering the data. Decoded data is referred to as the original blocks alone.</td>
</tr>
</tbody>
</table>

**Prerequisites**

DOCA Erasure Coding library follows the architecture of a DOCA Core Context, it is recommended read the following sections before:

- [DOCA Core Execution Model](#)
- [DOCA Core Device](#)
- [DOCA Core Memory Subsystem](#)

**Environment**

DOCA Erasure Coding-based applications can run either on the host machine or on the DPU target (NVIDIA® BlueField®-3 and above).

Erasure Coding can only be run with DPU configured in DPU mode as described in [NVIDIA BlueField Modes of Operation](#).

**Architecture**

DOCA Erasure Coding is a DOCA Context as defined by DOCA Core. This library leverages the DOCA Core architecture to expose asynchronous tasks/events that are offloaded to
The following diagram presents a high-level view of the EC transmission flow:

1. M packets are sent from the source (8 in this case).

2. Before the source send them, the source encode the data by adding to it T redundancy packets (4 in this case).

3. The packets are transmitted to the destination in UDP protocol. Some packets are lost and N' packets are received (in this case 4 packets are lost and 8 are received).

4. The destination decodes the data using all the packets available (both original data in green and redundancy data in red) and gets back the M original data packets.

**Flows**

Regular EC flow consists of the following elements:

1. Creating redundancy blocks from data (EC create).

2. Updating redundancy blocks from updated data (EC update).
3. Recovering data blocks from redundancy blocks (EC recover).

The following sections examine an M:K (where M is the original data and K is redundancy) EC.

**Create Redundancy Blocks**

The user must perform the following:

1. Input M data blocks via `doca_buf` (filled with data, each block size B).
2. Output K empty blocks via `doca_buf` (each block size B).
3. Use DOCA Erasure Coding to create a coding matrix of M by K via `doca_buf`.
4. Use DOCA Erasure Coding Create task to get the K output redundancy blocks.

**Note**
Recover Block

The user must perform the following:

1. Input M-L original blocks via \texttt{doca_buf} (blocks that were not impaired).

2. Input L≤K (any) redundancy blocks via \texttt{doca_buf} (redundancy blocks originating from create/update tasks).

3. Input bitmask or array, indicating which blocks to recover.

4. Output L empty blocks via \texttt{doca_buf} (same size of data block).

5. Use DOCA Erasure Coding to create a recover coding matrix of M by L via \texttt{doca_buf} (unique per bitmask).

6. Use DOCA Erasure Coding Recover task to get the L output recovered data blocks.
Objects

Device and Device Representor

The DOCA Erasure Coding library requires a DOCA device to operate. The device is used to access memory and perform the encoding and decoding operations. See DOCA Core Device Discovery.

For same Bluefield card, it does not matter which device is used (PF/VF/SF), as all these devices utilize the same HW component. If there are multiple DPUs, then it is possible to create an EC instance per DPU, providing each instance with a device from a different DPU. To access memory that is not local (from the host to the DPU and vice versa), the DPU side of the application must pick a device with an appropriate representor. See DOCA Core Representor Device Discovery.

The device must stay valid until the EC instance is destroyed.

Memory Buffers

Executing any DOCA EC task requires two DOCA buffers, a source buffer and a destination buffer.

Depending on the allocation pattern of the buffers, refer to the Inventory Types table.
Buffers must not be modified or read during the execution of any task.

**Configuration Phase**

To start using the library, first, you need to go through a configuration phase as described in [DOCA Core Context Configuration Phase](#).

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

**Configurations**

The context can be configured to match the application use case.

To find if a configuration is supported, or what the min/max value, please refer to [Device Support](#).

**Mandatory Configurations**

These configurations are mandatory and must be set by the application before attempting to start the context:

- At least 1 task/event type needs to be configured. See configuration of Tasks.
- A device with appropriate support must be provided on creation.

**Device Support**

DOCA Erasure Coding needs a device to operate. For picking a device, see [DOCA Core Device Discovery](#).

Erasure Coding can be used in BlueField-3 with some limitations (see [architecture](#)). Any device can be used PF/VF/SF.
As device capabilities may change in the future, it is recommended to choose your device using the following methods:

- `doca_ec_cap_task_galois_mul_is_supported`
- `doca_ec_cap_task_create_is_supported`
- `doca_ec_cap_task_update_is_supported`
- `doca_ec_cap_task_recover_is_supported`

Some devices can allow different capabilities as follows:

- The maximum buffer list length
- The maximum block size

**Note**

Current BlueField-3 limitations:

- Data block count range: 1-128
- Redundancy block count: 1-32
- Block size: 64B-128MB

**Buffer Support**

Tasks support buffers with the following features:

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Source Buffer</th>
<th>Destination Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked list buffer</td>
<td>Depends on the device; check the max_buf_list_len capability</td>
<td>No</td>
</tr>
<tr>
<td>Buffer Type</td>
<td>Source Buffer</td>
<td>Destination Buffer</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Local mmap buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from PCIe export buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from RDMA export buffer</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Execution Phase**

This section describes execution on CPU or DPU using the DOCA Core Progress Engine.

**Matrix Generate**

All tasks require a coding matrix.

**Matrix Type**

DOCA EC provides 2 matrix types which are elaborated on in the following subsections.

**Cauchy**

Cauchy encoding matrix is constructed so that  \( a_{ij} = \frac{1}{(x_i+y_j)} \).

Where:

- \( 0 \leq i < \) number of data blocks
- \( 0 \leq j < \) number of redundancy blocks
- \( x_i = i \)
- \( y_j = j + \) number of data blocks

**Vandermonde**
Vandermonde encoding matrix is constructed so that $a_{ij} = (i + 1)^j$.

Where:

- $0 \leq i < \text{number of data blocks}$
- $0 \leq j < \text{number of redundancy blocks}$

⚠️ Warning

Vandermonde matrix does not guarantee that every submatrix is invertible (i.e., the decode task may fail in some settings).

Matrix Functionality

Create

An encoding matrix is necessary for executing the create task, to create redundancy blocks.

The matrices used for updates and recovery are based on an encoding matrix.

The following subsections describe the available options for creating matrices.

Generic

Generic creation, with the `doca_ec_matrix_create()` function, is used for simple setup using one of matrix types provided by the library.

Input:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>One of matrix types provided by the library</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Data block count</td>
<td>The number of original data blocks</td>
</tr>
<tr>
<td>Redundancy block count</td>
<td>The number of redundancy blocks</td>
</tr>
</tbody>
</table>

**Custom**

Custom creation, with the `doca_ec_matrix_create_from_raw()` function, is used if the desired type of matrix is not provided by the library.

**Input:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>The data of a coding matrix</td>
<td>The size of the data should be</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>data_block_count*rdnc_block_count</code></td>
</tr>
<tr>
<td>Data block count</td>
<td>The number of original data blocks</td>
<td>-</td>
</tr>
<tr>
<td>Redundancy block count</td>
<td>The number of redundancy blocks</td>
<td>-</td>
</tr>
</tbody>
</table>

**Update**

This matrix is necessary for executing the update task, to update the redundancy blocks after a change in the data blocks.

The matrix is created using the `doca_ec_matrix_create_update()` function.

**Input:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding matrix</td>
<td>A coding matrix created by <code>doca_ec_matrix_create()</code> or <code>doca_ec_matrix_create_from_raw()</code></td>
<td>-</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Update indices</td>
<td>An array specifying the indices of the updated data blocks</td>
<td>• The indices must be in ascending order</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The indices should match the order of the data blocks in the matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>creation function</td>
</tr>
<tr>
<td>Number of updates</td>
<td>The number of updated blocks.</td>
<td>The length of the update indices array.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
</tbody>
</table>

**Recover**

This matrix is necessary for executing the recover task, to recover original data blocks.

The matrix is created using the `doca_ec_matrix_create_recover()` function.

**Input:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding matrix</td>
<td>A coding matrix created by <code>doca_ec_matrix_create()</code> or <code>doca_ec_matrix_create_from_raw()</code></td>
<td>–</td>
</tr>
<tr>
<td>Missing indices</td>
<td>An array specifying the indices of the missing data blocks</td>
<td>• The indices must be in ascending order</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The indices should match the order of the data blocks in the matrix</td>
</tr>
<tr>
<td>Number of missing</td>
<td>The number of updated blocks.</td>
<td>The length of the update indices array.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
</tbody>
</table>
Tasks

Task Batching

DOCA Erasure Coding supports task batching mode, which is a task submit mode of work that allows aggregating multiple DOCA tasks of the same type and handling them as a single unit.

Info

For more information on task batching, refer to DOCA Core Task.

DOCA Erasure Coding supports the flags DOCA_TASK_SUBMIT_FLAG_FLUSH and DOCA_TASK_SUBMIT_FLAG_OPTIMIZE_REPORTS.

Galois Mul Task

This task executes Galois multiplication between the original blocks and the coding matrix.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_ec_task_galois_mul_set_config</td>
<td>doca_ec_cap_task_galois_mul_is_supported</td>
</tr>
<tr>
<td>Maximum block size</td>
<td>–</td>
<td>doca_ec_cap_get_max_block_size</td>
</tr>
<tr>
<td>Maximum buffer list length</td>
<td>–</td>
<td>doca_ec_cap_get_max_buf_list_len</td>
</tr>
</tbody>
</table>

Task Input
Common input as described in **DOCA Core Task**.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>coding matrix</td>
<td>A coding matrix as created by doca_ec_matrix_create() or doca_ec_matrix_create_from_raw()</td>
<td>• The data length of src_buf should be a multiplication of the block size&lt;br&gt;• The data length should also be aligned to 64B and with a minimum size of 64B</td>
</tr>
<tr>
<td>source buffer</td>
<td>Source original data buffer, holding a sequence containing all original blocks (e.g., block_1, block_2, etc.); the order matters</td>
<td>• The data is written to the tail segment extending the data segment&lt;br&gt;• The minimal available memory in dst_buf should be the number of redundancy blocks * the block size, aligned to 64B and, in any case, at least 64B</td>
</tr>
<tr>
<td>destination buffer</td>
<td>A destination buffer for the multiplication outcome blocks. The sequence containing all multiplication outcome blocks (dst_block_1, dst_block_2, etc.) is written to it upon successful completion of the task.</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

If a Galois multiplication task matrix is 10x4 (i.e., 10 original blocks, 4 multiplication outcome blocks), and the block size is 64KB:

- src_buf data length should be 10x64KB = 640KB
- The available memory for writing in dst_buf should be at least 4x64KB = 256KB
Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing all multiplication outcome blocks (e.g., dst_block_1, dst_block_2, etc.)
- The destination buffer data segment is extended to include the outcome blocks

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination doca_buf objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination buffer should not be read from/written to
- Source and destination buffers must not overlap
- Other limitations are described in DOCA Core Task
Create Task

This task creates redundancy blocks for the given original data blocks using a given coding matrix.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_ec_task_create_set_conf</td>
<td>doca_ec_cap_task_create_is_supported</td>
</tr>
<tr>
<td>Maximum block size</td>
<td>–</td>
<td>doca_ec_cap_get_max_block_size</td>
</tr>
<tr>
<td>Maximum buffer list length</td>
<td>–</td>
<td>doca_ec_cap_get_max_buf_list_len</td>
</tr>
</tbody>
</table>

Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>coding matrix</td>
<td>A coding matrix created by doca_ec_matrix_create() or doca_ec_matrix_create_from_raw()</td>
<td>-</td>
</tr>
<tr>
<td>original data blocks</td>
<td>Source original data buffer, holding a sequence containing all original blocks (block_1, block_2, etc.); the order matters</td>
<td>- The data length of original_data_blocks should be a multiplication of the block size. - The data length should also be aligned to 64B and with a minimum size of 64B</td>
</tr>
<tr>
<td>redundancy</td>
<td>A destination buffer for the redundancy blocks. The sequence containing all redundancy blocks (rdnc_block_1,</td>
<td>- The data will be written to the tail segment extending the data segment</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>block</td>
<td>rdnc_block_2, etc.) is written to upon successful completion of the task.</td>
<td>• The minimal available memory in rdnc_blocks should be the number of redundancy blocks * the block size, aligned to 64B and, in any case, at least 64B</td>
</tr>
</tbody>
</table>

⚠️ **Note**

If a create task matrix is 10x4 (i.e., 10 original blocks, 4 redundancy blocks), and the block size is 64KB:

- original_data_blocks data length should be 10x64KB = 640KB
- The available memory for writing in redundancy_blocks should be at least 4x64KB = 256KB

**Task Output**

Common output as described in [DOCA Core Task](#).

**Task Completion Success**

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing all redundancy blocks (rdnc_block_1, rdnc_block_2, etc.)
- The destination buffer data segment is extended to include the redundancy blocks
Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination buffers should not be read from/written to
- Source and destination buffers must not overlap
- Other limitations are described in DOCA Core Task

Update Task

This task executes updates the redundancy blocks for the given original data blocks, using an update coding matrix.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_ec_task_update_set_conf</code></td>
<td><code>doca_ec_cap_task_update_is_supported</code></td>
</tr>
<tr>
<td>Maximum block size</td>
<td>–</td>
<td><code>doca_ec_cap_get_max_block_size</code></td>
</tr>
<tr>
<td>Maximum buffer list length</td>
<td>–</td>
<td><code>doca_ec_cap_get_max_buf_list_len</code></td>
</tr>
</tbody>
</table>
Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>update matrix</td>
<td>An update coding matrix created by doca_ec_matrix_create_update() or doca_ec_matrix_create_from_raw()</td>
<td>-</td>
</tr>
</tbody>
</table>
| original updated and RDNC blocks | A source buffer with data, holding a sequence containing the original data block and its updated data block, for each block that was updated, followed by the old redundancy blocks (old_data_block_i, updated_data_block_i, old_data_block_j, updated_data_block_j, ..., rdnc_block_1, rdnc_block_2, etc.) | • The data length of original_updated_and_rdnc_blocks should be a multiplication of the block size  
• The data length should also be aligned to 64B and with a minimum size of 64B                                                                                           |
| updated RDNC blocks         | A destination buffer for the updated redundancy blocks. The sequence containing the updated redundancy blocks (rdnc_block_1, rdnc_block_2, etc.) is written to it upon successful completion of the task | • The data is written to the tail segment extending the data segment  
• The minimal available memory in updated_rdnc_blocks should be the number of redundancy blocks \* the block size, aligned to 64B and, in any case, at least 64B |

Note

using an update task matrix, in which 3 data block were updated and there are 4 redundancy blocks, and the block size is 64KB:
Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing the updated redundancy blocks (rdnc_block_1, rdnc_block_2, etc.)
- The destination buffer data segment is extended to include the updated redundancy blocks

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination doca_buf objects is not modified
- The destination buffer contents may be modified

Task Limitations

- original_updated_and_rdnc_blocks data length should be 
  \((3+3+4=10)\times 64\text{KB} = 640\text{KB}\)
- The available memory for writing in updated_rdnc_blocks should be at least \(4\times 64\text{KB} = 256\text{KB}\)
- The operation is not atomic
- Once the task has been submitted, the source and destination buffers should not be read from/written to
- Source and destination buffers must not overlap
- Other limitations described in DOCA Core Task

**Recover Task**

This task executes recovers data blocks for, using given available original data blocks and redundancy blocks and a given coding matrix.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_ec_task_recover_set_conf</td>
<td>doca_ec_cap_task_recover_is_supported</td>
</tr>
<tr>
<td>Maximum block size</td>
<td>-</td>
<td>doca_ec_cap_get_max_block_size</td>
</tr>
<tr>
<td>Maximum buffer list length</td>
<td>-</td>
<td>doca_ec_cap_get_max_buf_list_len</td>
</tr>
</tbody>
</table>

**Task Input**

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>recover_matrix</td>
<td>A coding matrix create by doca_ec_matrix_create() or</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>doca_ec_matrix_create_from_raw()</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| available blocks         | A source buffer with data, holding a sequence containing available data blocks and redundancy blocks \((data\_block\_a, data\_block\_b, data\_block\_c, ..., rdnc\_block\_x, rdnc\_block\_y, etc.)\)                                                                                                 | • The total number of blocks given should be equal to the number of original data blocks  
• The data length of available_blocks should be a multiplication of the block size  
• The data length should also be aligned to 64B and with a minimum size of 64B                                                                                                                                 |
| recovered data blocks    | A destination buffer for the recovered data blocks. The sequence containing the recovered data blocks \((data\_block\_i, data\_block\_j, etc.)\) is written to it upon successful completion of the task                                                                                                                    | • The data is written to the tail segment extending the data segment  
• The minimal available memory in recovered_data_blocks should be the number of missing data blocks * the block size, aligned to 64B and, in any case, at least 64B.                                                                 |

**Note**

Using a recover task matrix, based on an original 10x4 coding matrix (i.e., 10 original blocks, 4 redundancy blocks), and a block size of 64KB:

- 10 available blocks should be given in total (e.g., 7 data blocks and 3 redundancy blocks)

- available_blocks data length should be 10x64KB = 640KB

- The available memory for writing in recovered_data_blocks should be at least 3x64KB = 192KB
**Task Output**

Common output as described in DOCA Core Task.

**Task Completion Success**

After the task is completed successfully the data is transformed to destination.

**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination doca_buf objects are not modified
- The destination buffer contents may be modified

**Task Limitations**

- The operation is not atomic
- Once the task is submitted, the source and destination buffers should not be read from/written to
- Source and destination must not overlap
- The amount of blocks that can be recovered are limited to the number of redundancy blocks created
- Other limitations are described in DOCA Core Task
DOCA Erasure Coding Samples

This section provides DOCA Erasure Coding sample implementation on top of the BlueField-3 DPU (and higher).

Info

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Sample Prerequisites

N/A

Running the Sample

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

   cd /opt/mellanox/doca/samples/doca_erasure_coding/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample (e.g., `doca_erasure_coding_recover`) usage:

```
Usage: doca_erasure_coding_recover [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 device PCI device address - default: 03:00.0
-i, --input                       Input file/folder to ec - default: self
-o, --output                      Output file/folder to ec - default: /tmp
-b, --both                        Do both (encode & decode) - default: false
-x, --matrix                      Matrix - {cauchy, vandermonde} - default: cauchy
-t, --data                        Data block count - default: 2
-r, --rdnc                        Redundancy block count - default: 2
-d, --delete-index                Indices of data blocks to delete; comma-separated (i.e., 0,3,4) - default: 0
```

Note

Current BlueField-3 limitations:

- Data block count range – 1-128
- Redundancy block count – 1-32
- Block size – 64B-128MB
4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

## Samples

### Erasure Coding Recover

This sample illustrates how to use DOCA Erasure Coding (EC) library to encode and decode a file block (and entire file).

The sample logic includes 3 steps:

1. Encoding – create redundancy.
2. Deleting – simulating disaster.
3. Decoding – recovering data.

The encode logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures, such as the progress engine (PE), memory maps, and buffer inventory.
3. Reading source original data file and splitting it to a specified number of blocks, `<data block count>`, specified for the sample to the output directory.
4. Populating two DOCA memory maps with a memory range, one for the source data and one for the result.
5. Allocating buffers from DOCA buffer inventory for each memory range.
6. Creating an EC object.
7. Connecting the EC context to the PE.
8. Setting a state change callback function for the PE, with the following logic:

- Printing a log with every state change
- Indicating that the user may stop progress the PE once it is back in idle state

9. Setting the configuration to the EC create task, including setting callback functions as follows:

- Successful completion callback:
  1. Writing the resulting redundancy blocks to the output directory (count is specified by `<redundancy block count>`).
  2. Freeing the task.
  3. Saving the result of the task and the callback. If there was an error in step a., the relevant error value is saved.
  4. Stopping the context.

- Failed completion callback:
  1. Saving the result of the task and the callback.
  2. Freeing the task.
  3. Stopping the context.

10. Creating EC encoding matrix by the matrix type specified to the sample.

11. Allocating and submitting an EC create task.

12. Progressing the PE until the context returns to idle state, either as a result of a successful run in which all tasks have been successfully completed, or as a result of a fatal error.

13. Destroying all EC and DOCA Core structures.

The delete logic includes:

1. Deleting the block files specified with `<indices of data blocks to delete>`.
The decode logic includes:

1. Locating a DOCA device.

2. Initializing the required DOCA Core structures, such as the PE, memory maps, and buffer inventory.

3. Reading the output directory (source remaining data) and determining the block size and which blocks are missing (needing recovery).

4. Populating two DOCA memory maps with a memory range, one for the source data and one for the result.

5. Allocating buffers from DOCA buffer inventory for each memory range.

6. Creating an EC object.

7. Connecting the EC context to the PE.

8. Setting a state change callback function for the PE, with the following logic:
   - Printing a log with every state change
   - Indicating that the user may stop progress the PE once it is back in idle state

9. Setting the configuration to the EC recover task, including setting callback functions as following:
   - Successful completion callback:
     1. Writing the resulting recovered blocks to the output directory.
     2. Writing the recovered file to the output path.
     3. Freeing the task.
     4. Saving the result of the task and the callback. If there was an error in step a., the relevant error value is saved.
     5. Stopping the context.
   - Failed completion callback:
1. Saving the result of the task and the callback.

2. Freeing the task.

3. Stopping the context.

10. Creating EC encoding matrix by the matrix type specified to the sample.

11. Creating EC decoding matrix, with `doca_ec_matrix_create_recover()`, using the encoding matrix.

12. Allocating and submitting an EC recover task.

13. Progressing the PE until the context returns to idle state, either as a result of a successful run in which all tasks have been successfully completed, or as a result of a fatal error.

14. Destroying all DOCA EC and DOCA Core structures.

References:

- `/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_coding_recover/erasure_coding_recover`
- `/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_coding_recover/erasure_coding_recover`
- `/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_coding_recover/meson.build`
This guide provides instructions on building and developing applications that require data encryption and decryption using the AES-GCM algorithm.

**Introduction**

The DOCA AES-GCM library is supported at alpha level.

The library provides an API for executing AES-GCM operations on DOCA buffers, where the buffers reside in either local memory (i.e., within the same host) or host memory accessible by the DPU (remote memory). Using DOCA AES-GCM, complex encrypt/decrypt operations can be easily executed in an optimized, hardware-accelerated manner.

This document is intended for software developers wishing to accelerate their application's encrypt/decrypt operations.

**Prerequisites**

This library follows the architecture of a DOCA Core context, it is recommended to read the following sections before:

- [DOCA Core Execution Model](#)
- [DOCA Core Device](#)
- [DOCA Core Memory Subsystem](#)

**Environment**
DOCA AES-GCM-based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target.

Encrypting/decrypting from the host to DPU and vice versa can only be run when the DPU is configured in DPU mode.

**Architecture**

DOCA AES-GCM is a DOCA Core Context. This library leverages the DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

AES-GCM can be used to encrypt/decrypt data as illustrated in the following diagrams:

- Encrypt/decrypt from local memory to local memory:

![Local Memory Diagram](image)

- Using the DPU to copy memory between the host and the DPU:

![Host-DPU Diagram](image)
Objects

Device and Representor

The library requires a DOCA device to operate. The device is used to access memory and perform the actual encrypt/decrypt. See DOCA Core Device Discovery.

For the same BlueField DPU, it does not matter which device is used (i.e., PF/VF/SF) as all these devices utilize the same hardware component. If there are multiple DPUs, then it is possible to create a AES-GCM instance per DPU, providing each instance with a device from a different DPU.

To access memory that is not local (i.e., from the host to DPU or vice versa), the DPU side of the application must pick a device with an appropriate representor (see DOCA Core Device Representor Discovery). The device must stay valid as long as AES-GCM instance is not destroyed.

Memory Buffers

The encrypt/decrypt task, requires two DOCA buffers containing the destination and the source.

Depending on the allocation pattern of the buffers, consider the DOCA Core Inventory Types table.

To find what kind of memory is supported, refer to the following table.
Buffers must not be modified or read during the encrypt/decrypt operation.

**Configuration Phase**

To start using the library users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

**Configurations**

The context can be configured to match the application use case.

To find if a configuration is supported or its min/max value, refer to Device Support.

**Mandatory Configurations**

These configurations must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of Tasks and/or Events.
  
  - A device with appropriate support must be provided upon creation

**Device Support**

DOCA AES-GCM requires a device to operate. For picking a device, see DOCA Core Device Discovery.

As device capabilities may change in the future (see DOCA Core Device Support) it is recommended to select your device using the following method:

  - doca_aes_gcm_cap_task_encrypt_is_supported
Some devices can allow different capabilities as follows:

- The maximum number of tasks
- The maximum buffer size
- The maximum supported number of elements in DOCA linked-list buffer
- The maximum initialization vector length
- Check if authentication tag of size 96-bit is supported
- Check if authentication tag of size 128-bit is supported
- Check if a given AES-GCM key type is supported

**Buffer Support**

Tasks support buffers with the following features:

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Source Buffer</th>
<th>Destination Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local mmap buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from PCIe export buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from RDMA export buffer</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Linked list buffer</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Execution Phase**

This section describes execution on the CPU using DOCA Core Progress Engine.

**Tasks**
DOCA AES-GCM exposes asynchronous tasks that leverage DPU hardware according to the DOCA Core architecture.

**Encrypt Task**

The encrypt task allows data encryption using buffers as described in Buffer Support.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_aes_gcm_task_encrypt_set_conf</td>
<td>doca_aes_gcm_cap_task_encrypt_is_supported</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_aes_gcm_task_encrypt_set_conf</td>
<td>doca_aes_gcm_cap_get_max_num_tasks</td>
</tr>
<tr>
<td>Maximal buffer size</td>
<td>-</td>
<td>doca_aes_gcm_cap_task_encrypt_get_max_buf_size</td>
</tr>
<tr>
<td>Maximum buffer list size</td>
<td>-</td>
<td>doca_aes_gcm_cap_task_encrypt_get_max_list_buf_num_elem</td>
</tr>
<tr>
<td>Maximum initialization vector length</td>
<td>-</td>
<td>doca_aes_gcm_cap_task_encrypt_get_max_iv_length</td>
</tr>
<tr>
<td>Enable authentication tag size</td>
<td>-</td>
<td>doca_aes_gcm_cap_task_encrypt_is_tag_96_supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>doca_aes_gcm_cap_task_encrypt_is_tag_128_supported</td>
</tr>
<tr>
<td>Enable key type</td>
<td>-</td>
<td>doca_aes_gcm_cap_task_encrypt_is_key_type_supported</td>
</tr>
</tbody>
</table>

**Task Input**

Common input as described in DOCA Core Task

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>source buffer</td>
<td>Buffer pointing to the memory to be encrypted</td>
<td>Only the data residing in the data segment is encrypted</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>destinatio buffer</td>
<td>Buffer pointing to where memory is encrypted to</td>
<td>The encrypted data is appended to the tail segment</td>
</tr>
<tr>
<td>key</td>
<td>Key to encrypt the data</td>
<td>Created by the function doca_aes_gcm_key_create Users should use the same key to encrypt and decrypt the data</td>
</tr>
<tr>
<td>initialization vector (IV)</td>
<td>Initialization vector to be used by the AES-GCM algorithm</td>
<td>Users should use the same IV to encrypt and decrypt the data</td>
</tr>
<tr>
<td>initialization vector length</td>
<td>Initialization vector length that must be supplied for the AES-GCM algorithm</td>
<td>Represented in bytes, 0B-12B values are supported</td>
</tr>
<tr>
<td>authentication tag size</td>
<td>Authentication tag size to be supplied for the AES-GCM algorithm</td>
<td>Represented in bytes, only 12B and 16B values are supported</td>
</tr>
<tr>
<td>additional authenticated data size</td>
<td>Additional authenticated data size to be supplied for the AES-GCM algorithm. This data, which should be present at the beginning of the source buffer, is will not encrypted but is authenticated.</td>
<td>Represented in bytes</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in [DOCA Core Task](#).

**Task Completion Success**
After the task completes successfully, the following happens:

- The data from the source buffer is encrypted and written to the destination buffer
- The destination buffer data segment is extended to include the encrypted data

**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

**Task Limitations**

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in [DOCA Core Task](#)

**Decrypt Task**

The decrypt task allows data decryption. Using buffers as described in [Buffer Support](#).

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_aes_gcm_task_decrypt</code></td>
<td><code>doca_aes_gcm_cap_task_decrypt_is_support</code></td>
</tr>
<tr>
<td>Description</td>
<td>API to Set Configuration</td>
<td>API to Query Support</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>t_set_conf</code></td>
<td></td>
</tr>
<tr>
<td>Maximal buffer size</td>
<td><code>doca_aes_gcm_task_decrypt_set_conf</code></td>
<td><code>doca_aes_gcm_cap_task_decrypt_get_max_buf_size</code></td>
</tr>
<tr>
<td>Maximum buffer list size</td>
<td>-</td>
<td><code>doca_aes_gcm_cap_task_decrypt_get_max_list_buf_num_elem</code></td>
</tr>
<tr>
<td>Maximum initialization vector length</td>
<td>-</td>
<td><code>doca_aes_gcm_cap_task_decrypt_get_max_iv_length</code></td>
</tr>
<tr>
<td>Enable authentication tag size</td>
<td>-</td>
<td><code>doca_aes_gcm_cap_task_decrypt_is_tag_96_supported</code> <code>doca_aes_gcm_cap_task_decrypt_is_tag_128_supported</code></td>
</tr>
<tr>
<td>Enable key type</td>
<td>-</td>
<td><code>doca_aes_gcm_cap_task_decrypt_is_key_type_supported</code></td>
</tr>
</tbody>
</table>

### Task Input

**Common input as described in DOCA Core Task.**

<p>| Name             | Description                                           | Notes                                                                 |
|------------------|-------------------------------------------------------|                                                                     |
| Source buffer    | Buffer pointing to the memory to be decrypted         | Only the data residing in the data segment is decrypted             |
| Destination buffer | Buffer pointing to where memory is decrypted to       | The decrypted data is appended to the tail segment extending the data segment |
| Key              | Key to decrypt the data                               | Created by the function <code>doca_aes_gcm_key_create</code> The user should use the same key to encrypt and decrypt the data |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization vector (IV)</td>
<td>Initialization vector to be used by the AES-GCM algorithm</td>
<td>The user should use the same IV to encrypt and decrypt the data</td>
</tr>
<tr>
<td>Initialization vector length</td>
<td>Initialization vector length that must be supplied for the AES-GCM algorithm</td>
<td>Represented in bytes, 0B-12B values are supported</td>
</tr>
<tr>
<td>Authentication tag size</td>
<td>Authentication tag size to be supplied for the AES-GCM algorithm. The tag,</td>
<td>Represented in bytes, only 12B and 16B values are supported</td>
</tr>
<tr>
<td></td>
<td>present at the end of the source buffer, is verified and is not present in</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the destination buffer.</td>
<td></td>
</tr>
<tr>
<td>Additional authenticated data size</td>
<td>Additional authenticated data size to be supplied for the AES-GCM algorithm.</td>
<td>Represented in bytes</td>
</tr>
<tr>
<td></td>
<td>This data, present at the beginning of the source buffer, is not encrypted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>but is authenticated.</td>
<td></td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in [DOCA Core Task](#).

**Task Completion Success**

After the task completes successfully, the following happens:

- The data from the source buffer is decrypted and written to the destination buffer
- The destination buffer data segment is extended to include the decrypted data

**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
• The source and destination doca_buf objects is not modified

• The destination buffer contents may be modified

**Task Limitations**

• The operation is not atomic

• Once the task is submitted, the source and destination should not be read/written to

• Other limitations are described in DOCA Core Task

**Events**

DOCA AES-GCM exposes asynchronous events to notify about changes that happen unexpectedly according to the DOCA Core architecture.

The only events AES-GCM exposes are common events as described in DOCA Core Event.

**State Machine**

The DOCA AES-GCM library follows the Context state machine as described in DOCA Core Context State Machine.

The following section describes moving states and what is allowed in each state.

**Idle**

In this state, it is expected that the application either:

• Destroys the context
Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all tasks have been freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all tasks are completed and freed</td>
</tr>
</tbody>
</table>

Starting

This state cannot be reached.

Running

In this state, it is expected that the application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating previously configured task
- Submitting a task
- Calling stop
It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

### Stopping

In this state, it is expected that the application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Calling progress

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Call progress and fatal error occurs</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop without freeing all tasks</td>
</tr>
</tbody>
</table>

### Alternative Datapath Options

DOCA AES-GCM only supports datapath on the CPU. See Execution Phase.

### DOCA AES-GCM Samples

This section describes DOCA AES-GCM samples based on the DOCA AES-GCM library.

The samples in this section illustrate how to use the DOCA AES-GCM API to do the following:
• Encrypt contents of a buffer to another buffer
• Decrypt contents of a buffer to another buffer

Info

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Running the Samples

1. Refer to the following documents:
   • NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   • NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_aes_gcm/<sample_name>
   meson/tmp/build
   ninja -C/tmp/build
   ```

   Info

   The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample (e.g., `doca_aes_gcm_encrypt`) usage:
4. For additional information per sample, use the -h option:

```
/tmp/build/doca_<sample_name>-h
```

**Samples**

**AES-GCM Encrypt**

This sample illustrates how to encrypt data with AES-GCM.

The sample logic includes:

1. Locating DOCA device.
2. Initializing required DOCA Core structures.

3. Setting the AES-GCM encrypt tasks configuration.

4. Populating DOCA memory map with two relevant buffers.

5. Allocating element in DOCA buffer inventory for each buffer.

6. Creating DOCA AES-GCM key.

7. Allocating and initializing AES-GCM encrypt task.

8. Submitting AES-GCM encrypt task.

9. Retrieving AES-GCM encrypt task once it is done.

10. Checking task result.

11. Destroying all AES-GCM and DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_encrypt/aes_gcm_encrypt_sample.c
- /opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_encrypt/aes_gcm_encrypt_main.c
- /opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_encrypt/meson.build

AES-GCM Decrypt

This sample illustrates how to decrypt data with AES-GCM.

The sample logic includes:

1. Locating DOCA device.

2. Initializing needed DOCA Core structures.

3. Setting the AES-GCM decrypt tasks configuration.

4. Populating DOCA memory map with two relevant buffers.
5. Allocating element in DOCA buffer inventory for each buffer.

6. Creating DOCA AES-GCM key.

7. Allocating and initializing AES-GCM decrypt task.

8. Submitting AES-GCM decrypt task.

9. Retrieving AES-GCM decrypt task once it is done.

10. Checking task result.

11. Destroying all AES-GCM and DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_decrypt/aes_gcm_decrypt_sample.c
- /opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_decrypt/aes_gcm_decrypt_main.c
- /opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_decrypt/meson.build
DOCA Rivermax

This guide provides instructions on building and developing applications that require media/data streaming.

Introduction

DOCA Rivermax (RMAX) is a DOCA API for NVIDIA® Rivermax®, an optimized networking SDK for media and data streaming applications. Rivermax leverages NVIDIA® BlueField® DPU hardware streaming acceleration technology which enables direct data transfers to and from the GPU, delivering best-in-class throughput and latency with minimal CPU utilization for streaming workloads.

This document is intended for software developers wishing to accelerate their networking operations.

Prerequisites

This library follows the architecture of DOCA Core Context. it is recommended read the following content before proceeding:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

Environment

Info

DOCA Rivermax-based applications can run on the target DPU only.
DOCA Rivermax-based application must be run with root privileges.

- The Rivermax library must compile and run and Rivermax license to run applications. Refer to NVIDIA Rivermax SDK page to obtain that license.
- An IP address to the device being used must be set up.
- It is recommended to have at least 800 huge pages enabled to achieve maximum performance:

```bash
dpu> echo 1000000000 > /proc/sys/kernel/shmmax
dpu> echo 800 > /proc/sys/vm/nr_hugepages
```

**Architecture**

- DOCA Rivermax Input Stream is a DOCA Context as defined by DOCA Core
- DOCA Rivermax leverages DOCA Core architecture to expose asynchronous events that are offloaded to hardware
- DOCA Rivermax can be used to define input streams that allow packet acquisition on an IP port. Furthermore, the input stream can be split to TCP/UDP 5-tuples to allow separate handling of flows.

**Objects**

- `doca_rmax_flow` – is a flow object that represents an IP/port tuple
- `doca_rmax_in_stream` – is a `doca_ctx` that represents the input stream and can be thought of as a receive queue which scatters the received data into memory. Each stream can receive one or more flows.
Configuration Phase

To start using the library users must first go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or its min/max value, refer to section "Device Support".

Mandatory Configurations

These configurations must be set by the application before attempting to start the context:

- An event type must be configured. See configuration of Events.
- CPU affinity and then Rivermax library global initialization in this order. The following APIs can be used to achieve this doca_rmax_set_cpu_affinity_mask() and doca_rmax_init()
- The memory block that holds packet memory
- The number of stream elements
- Minimal packet segment size(s)
- Maximal packet segment size(s)

Optional Configurations

If the following configurations are not set, then a default value is used:

- The input stream type – defaults to generic
- The input stream packet's data scatter type – defaults to raw
- The input stream timestamp format – defaults to raw counter

**Device Support**

DOCA Rivermax Input Stream requires a device to operate. For picking a device see DOCA Core Device Discovery.

The device must be from within the DPU: Either a PF or SF.

It is recommended to choose your device using the following method:

- `doca_devinfo_get_ipv4_addr()`

Some devices can allow different capabilities as follows:

- PTP clock support.

**Buffer Support**

Memory block support buffers with the following features:

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Memory Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local mmap buffer</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from PCIe export buffer</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from RDMA export buffer</td>
<td>No</td>
</tr>
<tr>
<td>Linked list buffer</td>
<td>Yes (header split mode)</td>
</tr>
</tbody>
</table>

**Execution Phase**
This section describes execution on CPU using **DOCA Core Progress Engine**.

## Events

DOCA Rivermax exposes asynchronous events to notify about changes that happen unexpectedly according to the DOCA Core architecture.

Common events are described in [DOCA Core Event](#).

### Rx Data

The Rx Data event is used by the stream to notify application that data has been received from the network.

### Event Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event</td>
<td><code>doca_rmax_in_stream_event_rx_data_register</code></td>
<td>-</td>
</tr>
</tbody>
</table>

### Event Trigger Condition

The event is triggered anytime packet(s) arrive.

### Event Output

Common output as described in [DOCA Core Event](#).

In case of success, the following is provided:

- Number of packets received
- Time of arrival of the first packet
- Time of arrival of the last packet
- Sequence number of the first packet
- Array of memory blocks as configured by input stream

In case of error, the following is provided:

- An error code
- A human readable message

Note

The parameters are valid only inside the event callback.

Event Handling

Once an event is triggered, the application may decide to process the received data.

Runtime Configurations

These configurations can be made after the context has been started:

- The minimal number of packets that the input stream must return in Rx event.
- The maximal number of packets that the input stream must return in Rx event.
- The receive timeout. The number of μsecs that library would do busy wait (polling) for reception of at least \( \text{min\_packets} \) number of packets.

State Machine
The DOCA RMAX library follows the Context state machine as described in DOCA Core Context State Machine

The following section describes how to move to the state and what is allowed in each state.

**Idle**

In this state, it is expected that application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop</td>
</tr>
</tbody>
</table>

**Starting**

This state is not expected to be reached.

**Running**

In this state, it is expected that application:

- Calls progress to receive events
Allowed operations:

- Calling stop
- Changing runtime configurations as described in Runtime Configurations

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

**Stopping**

This state is not expected to be reached.

**DOCA Rivermax Samples**

The samples illustrate how to use the DOCA Rivermax API to:

- List available devices, including their IP and supported capabilities
- Set CPU affinity for the internal Rivermax thread to achieve better performance
- Set the PTP clock device to be used internally in DOCA Rivermax
- Create a stream, create a flow and attach it to the created stream, and finally to start receiving data buffers (based on the attached flow)
- Create a stream in header-data split mode when packet headers and payload are split to different RX buffers
Running the Samples

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples

2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_rmax/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   **Info**
   
   The binary `doca_<sample_name>` is created under `/tmp/build/`.

3. Sample (e.g., `doca_rmax_create_stream`) usage:

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

   **DOCA Flags:**
   - `-h, --help` to print a help synopsis
   - `-v, --version` to print program version information
   - `-l, --log-level` to set the (numeric) log level for the program (10=DISABLE, 20=Critical, 30=Error, 40=Warning, 50=Info, 60=Debug, 70=Trace)
For additional information per sample, use the `--h` option:

```
/tmp/build/<sample_name> -h
```

### Note

When running DOCA Rivermax samples, the IPv4 address 192.168.105.2 must be configured to an available uplink prior to running it for the samples to run as expected:

```
ifconfig p0 192.168.105.2
```

### Samples

#### List Devices

This sample illustrates how to list all available devices, dump their IPv4 addresses, and tell whether or not the PTP clock is supported.

The sample logic includes:

1. Initializing DOCA Rivermax library.
2. Iterating over the available devices.
3. Dumping their IPv4 addresses
4. Dumping whether a PTP clock is supported for each device.

5. Releasing DOCA Rivermax library.

References:

- /opt/mellanox/doca/samples/doca_rmax/rmax_list_devices/rmax_list_devices_sample.c
- /opt/mellanox/doca/samples/doca_rmax/rmax_list_devices/rmax_list_devices_main.c
- /opt/mellanox/doca/samples/doca_rmax/rmax_list_devices/meson.build
- /opt/mellanox/doca/samples/doca_rmax/rmax_common.h;
  /opt/mellanox/doca/samples/doca_rmax/rmax_common.c

**Set CPU Affinity**

This sample illustrates how to set the CPU affinity mask for Rivermax internal thread to achieve better performance. This parameter must be set before library initialization otherwise it will not be applied.

The sample logic includes:

1. Setting CPU affinity using the DOCA Rivermax API.
2. Initializing DOCA Rivermax library.
3. Releasing DOCA Rivermax library.

References:

- /opt/mellanox/doca/samples/doca_rmax/rmax_set_affinity/rmax_set_affinity_sample.c
- /opt/mellanox/doca/samples/doca_rmax/rmax_set_affinity/rmax_set_affinity_main.c
- /opt/mellanox/doca/samples/doca_rmax/rmax_set_affinity/meson.build
- /opt/mellanox/doca/samples/doca_rmax/rmax_common.h;
  /opt/mellanox/doca/samples/doca_rmax/rmax_common.c
**Set Clock**

This sample illustrates how to set the PTP clock device to be used internally in DOCA Rivermax.

The sample logic includes:

1. Opening a DOCA device with a given PCIe address.
2. Initializing the DOCA Rivermax library.
3. Setting the device to use for obtaining PTP time.
4. Releasing the DOCA Rivermax library.

References:

- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_clock/rmax_set_clock_sample.c`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_clock/rmax_set_clock_main.c`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_set_clock/meson.build`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_common.h`
- `/opt/mellanox/doca/samples/doca_rmax/rmax_common.c`

**Create Stream**

This sample illustrates how to create a stream, create a flow and attach it to the created stream, and finally to start receiving data buffers (based on the attached flow).

The sample logic includes:

1. Opening a DOCA device with a given PCIe address.
2. Initializing the DOCA Rivermax library.
3. Creating an input stream.
4. Creating the context from the created stream.
5. Initializing DOCA Core related objects.

6. Setting the attributes of the created stream.

7. Creating a flow and attaching it to the created stream.

8. Starting to receive data buffers.

9. Clean up—detaches flow and destroys it, destroys created stream and DOCA Core related objects.

References:

- /opt/mellanox/doca/samples/doca_rmax/rmax_create_stream/rmax_create_stream_sample.c
- /opt/mellanox/doca/samples/doca_rmax/rmax_create_stream/rmax_create_stream_main.c
- /opt/mellanox/doca/samples/doca_rmax/rmax_create_stream/meson.build
- /opt/mellanox/doca/samples/doca_rmax/rmax_common.h
- /opt/mellanox/doca/samples/doca_rmax/rmax_common.c

**Create Stream – Header-data Split Mode**

This sample illustrates how to create a stream in header-data split mode when packet headers and payload are split to different RX buffers.

The sample logic includes:

1. Opening a DOCA device with a given PCIe address.

2. Initialize the DOCA Rivermax library.

3. Creating an input stream.

4. Creating a context from the created stream.

5. Initializing DOCA Core related objects.

6. Setting attributes of the created stream. Chaining buffers and setting header size to non-zero is essential to create a stream with header-data split mode.
7. Creating a flow and attaching it to the created stream.

8. Starting to receive data to split buffers.

9. Clean up—detaches flow and destroys it, destroys created stream and DOCA Core related objects.

References:

- /opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hds/rmax_create_stream_hds_sample.c
- /opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hds/rmax_create_stream_hds_main.c
- /opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hds/meson.build
- /opt/mellanox/doca/samples/doca_rmax/rmax_common.h;
  /opt/mellanox/doca/samples/doca_rmax/rmax_common.c
DOCA Telemetry Exporter

This guide provides an overview and configuration instructions for DOCA Telemetry Exporter API.

Introduction

DOCA Telemetry Exporter API offers a fast and convenient way to transfer user-defined data to DOCA Telemetry Service (DTS). In addition, the API provides several built-in outputs for user convenience, including saving data directly to storage, NetFlow, Fluent Bit forwarding, and Prometheus endpoint.

The following figure shows an overview of the telemetry exporter API. The telemetry exporter client side, based on the telemetry exporter API, collects user-defined telemetry and sends it to the DTS which runs as a container on BlueField. DTS does further data routing, including export with filtering. DTS can process several user-defined telemetry exporter clients and can collect pre-defined counters by itself. Additionally, telemetry exporter API has built-in data outputs that can be used from telemetry exporter client applications.

The following scenarios are available:

- Send data via IPC transport to DTS. For IPC, refer to Inter-process Communication.
- Write data as binary files to storage (for debugging data format).

- Export data directly from DOCA Telemetry Exporter API application using the following options:
  - Fluent Bit exports data through forwarding
  - NetFlow exports data from NetFlow API. Available from both API and DTS. See details in Data Outputs.
  - Prometheus creates Prometheus endpoint and keeps the most recent data to be scraped by Prometheus.

Users can either enable or disable any of the data outputs mentioned above. See Data Outputs to see how to enable each output.

The library stores data in an internal buffer and flushes it to DTS/exporters in the following scenarios:

- Once the buffer is full. Buffer size is configurable with different attributes.
- When `doca_telemetry_exporter_source_flush(void *doca_source)` function is invoked.
- When the telemetry exporter client terminates. If the buffer has data, it is processed before the library's context cleanup.

**Architecture**

DOCA Telemetry Exporter API is fundamentally built around four major parts:

- DOCA schema – defines a reusable structure (see `doca_telemetry_exporter_type`) of telemetry data which can be used by multiple sources
DOCA Telemetry Exporter API Walkthrough

The NVIDIA DOCA Telemetry Exporter API's definitions can be found in the `doca_telemetry_exporter.h` file.

The following is a basic walkthrough of the needed steps for using the DOCA Telemetry Exporter API.

1. Create `doca_schema`. 

- Source – the unique identifier of the telemetry exporter source that periodically reports telemetry data.
- Report – exports the information to the DTS
- Finalize – releases all the resources
1. Initialize an empty schema with default attributes:

```
struct doca_telemetry_exporter_schema *doca_schema;
doca_telemetry_exporter_schema_init("example_doca_schema_name", &doca_schema);
```

2. Set the following attributes if needed:

- `doca_telemetry_exporter_schema_set_buffer_attr_*(...)`
- `doca_telemetry_exporter_schema_set_file_write_*(...)`
- `doca_telemetry_exporter_schema_set_ipc_*(...)`

3. Add user event types:

Event type (struct `doca_telemetry_exporter_type`) is the user-defined data structure that describes event fields. The user is allowed to add multiple fields to the event type. Each field has its own attributes that can be set (see example). Each event type is allocated an index (doca_telemetry_exporter_type_index_t) which can be used to refer to the event type in future API calls.

```
struct doca_telemetry_exporter_type *doca_type;
struct doca_telemetry_exporter_field *field1;

doca_telemetry_exporter_type_create(&doca_type);
doca_telemetry_exporter_field_create(&field1);

doca_telemetry_exporter_field_set_name(field1, "sport");
doca_telemetry_exporter_field_set_description(field1, "Source port")
doca_telemetry_exporter_field_set_type_name(field1, DOCA_TELEMETRY_EXPORTER_FIELD_TYPE_UINT16);
doca_telemetry_exporter_field_set_array_length(field1, 1);

/* The user loses ownership on field1 after a successful invocation of the function */
doca_telemetry_exporter_type_add_field(type, field1);

/* Add more fields if needed */

/* The user loses ownership on doca_type after a successful invocation of the function */
```
4. Apply attributes and types to start using the schema:

```c
doca_telemetry_exporter_schema_add_type(doca_schema, "example_event", doca_type, &type_index);
```

2. Create `doca_source`:

1. Initialize:

```c
struct doca_telemetry_exporter_source *doca_source;
doca_telemetry_exporter_source_create(doca_schema, &doca_source);
```

2. Set source ID and tag:

```c
doca_telemetry_exporter_source_set_id(doca_source, "example id");
doca_telemetry_exporter_source_set_tag(doca_source, "example tag");
```

3. Apply attributes to start using the source:

```c
doca_telemetry_exporter_source_start(doca_source)
```

You may optionally add more `doca_sources` if needed.

3. Collect the data per source and use:

```c
doca_telemetry_exporter_source_report(source, type_index, &my_app_test_ev1, num_events)
```

4. Finalize:
1. For every source:

\[
\text{doca\_telemetry\_exporter\_source\_destroy}(\text{source})
\]

2. Destroy:

\[
\text{doca\_telemetry\_exporter\_schema\_destroy}(\text{doca\_schema})
\]

Example implementation may be found in the `telemetry\_export` DOCA sample (telemetry\_export\_sample.c).

**DOCA Telemetry Exporter NetFlow API Walkthrough**

The DOCA telemetry exporter API also supports NetFlow using DOCA Telemetry Exporter NetFlow API. This API is designed to allow customers to easily support the NetFlow protocol at the endpoint side. Once an endpoint produces NetFlow data using the API, the corresponding exporter can be used to send the data to a NetFlow collector.

The NVIDIA DOCA Telemetry Exporter Netflow API's definitions can be found in the `doca\_telemetry\_exporter\_netflow\_h` file.

The following are the steps to use the NetFlow API:

1. Initiate the API with an appropriate source ID:

\[
\text{doca\_telemetry\_exporter\_netflow\_init}(\text{source\_id})
\]

2. Set the relevant attributes:

- \[
\text{doca\_telemetry\_exporter\_netflow\_set\_buffer\_\_*}(\text{\_\_\_\_})
\]

- \[
\text{doca\_telemetry\_exporter\_netflow\_set\_file\_write\_\_*}(\text{\_\_\_\_})
\]
3. Start the API to use the configured attribute:

```c
doca_telemetry_exporter_netflow_start();
```

4. Form a desired NetFlow template and the corresponding NetFlow records.

5. Collect the NetFlow data.

```c
doca_telemetry_exporter_netflow_send(...)
```

6. (Optional) Flush the NetFlow data to send data immediately instead of waiting for the buffer to fill:

```c
doca_telemetry_exporter_netflow_flush()
```

7. Clean up the API:

```c
doca_telemetry_exporter_netflow_destroy()
```

Example implementation may be found in the `telemetry_export_netflow` DOCA sample (`telemetry_export_netflow_sample.c`).

**API**

Refer to [NVIDIA DOCA Library APIs](#), for more detailed information on DOCA Telemetry Exporter API.
The following sections provide additional details about the library API.

Some attributes are optional as they are initialized with default values. Refer to the documentation of the setter functions of respective attributes for more information.

**DOCA Telemetry Exporter Buffer Attributes**

Buffer attributes are used to set the internal buffer size and data root used by all DOCA sources in the schema.

Configuring the attributes is optional as they are initialized with default values.

```c
doca_telemetry_exporter_schema_set_buffer_size(doca_schema, 16 * 1024); /* 16KB - arbitrary value */
doca_telemetry_exporter_schema_set_buffer_data_root(doca_schema, "/opt/mellanox/doca/services/telemetry/data/");
```

- `buffer_size [in]` – the size of the internal buffer which accumulates the data before sending it to the outputs. Data is sent automatically once the internal buffer is full. Larger buffers mean fewer data transmissions and vice versa.

- `data_root [in]` – the path to where data is stored (if `file_write_enabled` is set to true). See section "DOCA Telemetry Exporter File Write Attributes".

**DOCA Telemetry Exporter File Write Attributes**

File write attributes are used to enable and configure data storage to the file system in binary format.

**Note**

The pkg-config (*.pc file) for the DOCA Telemetry Exporter library is `doca-telemetry-exporter`.

```c
doca_telemetry_exporter_schema_set_buffer_size(doca_schema, 16 * 1024); /* 16KB - arbitrary value */
doca_telemetry_exporter_schema_set_buffer_data_root(doca_schema, "/opt/mellanox/doca/services/telemetry/data/");
```
Configuring the attributes is optional as they are initialized with default values.

```c

file_write_enable [in] – use this function to enable storage. Storage/FileWrite is disabled by default.

file_write_max_size [in] – maximum file size (in bytes) before a new file is created.

file_write_max_age [in] – maximum file age (in microseconds) before a new file is created.

DOCA Telemetry Exporter IPC Attributes

IPC attributes are used to enable and configure IPC transport. IPC is disabled by default.

Configuring the attributes is optional as they are initialized with default values.

Note

It is important to make sure that the IPC location matches the IPC location used by DTS, otherwise IPC communication will fail.

```
• `ipc_enabled [in]` – use this function to enable communication. IPC is disabled by default.

• `ipc_sockets_dir [in]` – a directory that contains UDS for IPC messages. Both the telemetry exporter program and DTS must use the same folder. DTS that runs on BlueField as a container has the default folder `/opt/mellanox/doa/services/telemetry/ipc_sockets`.

• `ipc_reconnect_time [in]` – maximum reconnection time in milliseconds after which the client is considered disconnected.

• `ipc_reconnect_tries [in]` – maximum reconnection attempts.

• `ipc_socket_timeout [in]` – timeout for the IPC socket.

### DOCA Telemetry Exporter Source Attributes

Source attributes are used to create proper folder structure. All the data collected from the same host is written to the `source_id` folder under data root.

**Note**

Sources attributes are mandatory and must be configured before invoking `doca_telemetry_exporter_source_start()`.

```c
doca_telemetry_exporter_source_set_id(doca_source, "example_source");
doca_telemetry_exporter_source_set_tag(doca_source, "example_tag");
```

• `source_id [in]` – describes the data’s origin. It is recommended to set it to the hostname. In later dataflow steps, data is aggregated from multiple hosts/DPUs and `source_id` helps navigate in it.
• source_tag [in] – a unique data identifier. It is recommended to set it to describe the data collected in the application. Several telemetry exporter apps can be deployed on a single node (host/DPU). In that case, each telemetry data would have a unique tag and all of them would share a single source_id.

**DOCA Telemetry Exporter Netflow Collector Attributes**

DOCA Telemetry Exporter NetFlow API attributes are optional and should only be used for debugging purposes. They represent the NetFlow collector’s address while working locally, effectively enabling the local NetFlow exporter.

```c
#include <docatelemetry.h>

// Setting the collector's address
doca_telemetry_exporter_netflow_set_collector_addr("127.0.0.1");
// Setting the collector's port
doca_telemetry_exporter_netflow_set_collector_port(6343);
```

• collector_addr [in] – NetFlow collector's address (IP or name). Default value is NULL.

• collector_port [in] – NetFlow collector's port. Default value is `DOCA_NETFLOW_DEFAULT_PORT` (2055).

**doca_telemetry_exporter_source_report**

The source report function is the heart of communication with the DTS. The report operation causes event data to be allocated to the internal buffer. Once the buffer is full, data is forwarded onward according to the set configuration.

```c
#include <docatelemetry.h>

doca_error_t doca_telemetry_exporter_source_report(struct doca_telemetry_exporter_source *doca_source,
                                                doca_telemetry_exporter_type_index_t index,
                                                void *data,
                                                int count);
```

• doca_source [in] – a pointer to the `doca_telemetry_exporter_source` which reports the event
index [in] – the event type index received when the schema was created

data [in] – a pointer to the data buffer that needs to be sent

count [in] – numbers of events to be written to the internal buffer

The function returns DOCA_SUCCESS if successful, or a doca_error_t if an error occurs. If a memory-related error occurs, try a larger buffer size that matches the event's size.

doca_telemetry_exporter_schema_add_type

This function allows adding a reusable telemetry data struct, also known as a schema. The schema allows sending a predefined data structure to the telemetry service. Note that it is mandatory to define a schema for proper functionality of the library. After adding the schemas, one needs to invoke the schema start function.

```c
    doca_error_t doca_telemetry_exporter_schema_add_type(struct doca_telemetry_exporter_schema *doca_schema,
                        const char *new_type_name,
                        struct doca_telemetry_exporter_type *type,
                        doca_telemetry_exporter_type_index_t *type_index);
```

- doca_schema [in] – a pointer to the schema to which the type is added
- new_type_name [in] – name of the new type
- fields [in] – user-defined fields to be used for the schema. Multiple fields can (and should) be added.
- type_index [out] – type index for the created type is written to this output variable

The function returns DOCA_SUCCESS if successful, or doca_error_t if an error occurs.

Telemetry Data Format
The internal data format consists of 2 parts: A schema containing metadata, and the actual binary data. When data is written to storage, the data schema is written in JSON format, and the data is written as binary files. In the case of IPC transport, both schema and binary data are sent to DTS. In the case of export, data is converted to the formats required by exporter.

Adding custom event types to the schema can be done using `doca_telemetry_exporter_schema_add_type` API call.

**Note**

See available `DOCA_TELEMETRY_EXPORTER_FIELD_TYPES` in `doca_telemetry_exporter.h`. See example of usage in `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/telemetry_export_sample.c`.

**Note**

It is highly recommended to have the timestamp field as the first field since it is required by most databases. To get the current timestamp in the correct format use:

```c
#include <doa_error.h>

doca_error_t
doca_telemetry_exporter_get_timestamp(doca_telemetry_exporter_timestamp_t *timestamp);
```

**Data Outputs**

This section describes available exporters:

- IPC
- NetFlow
Fluent Bit and Prometheus exporters are presented in both API and DTS. Even though DTS export is preferable, the API has the same possibilities for development flexibility.

**Inter-process Communication**

IPC transport automatically transfers the data from the telemetry-exporter-based program to DTS service.

It is implemented as a UNIX domain socket (UDS) sockets for short messages and shared memory for data. DTS and the telemetry-exporter-based program must share the same `ipc_sockets` directory.

When IPC transport is enabled, the data is sent from the DOCA-telemetry-exporter-based application to the DTS process via shared memory.

To enable IPC, use the `doca_telemetry_exporter_schema_set_ipc_enabled` API function.

> **Note**

IPC transport relies on system folders. For the host's usage, run the DOCA-telemetry-exporter-API-based application with `sudo` to be able to use IPC with system folders.

To check the IPC status for the current context, use:

```c
doca_error_t doca_telemetry_exporter_check_ipc_status(struct doca_telemetry_exporter_source *doca_source,
                                                    doca_telemetry_exporter_ipc_status_t *status)
```

If IPC is enabled and for some reason connection is lost, it would try to automatically reconnect on every report's function call.
Using IPC with Non-container Application

When developing and testing a non-container DOCA Telemetry-Exporter-based program and its IPC interaction with DTS, some modifications are necessary in DTS's deployment for the program to interact with DTS over IPC:

- Shared memory mapping should be removed: telemetry-ipc-shm
- Host IPC should be enabled: hostIPC

File before the change:

```yaml
spec:
  hostNetwork: true
  volumes:
  - name: telemetry-service-config
    hostPath:
      path: /opt/mellanox/doca/services/telemetry/config
      type: DirectoryOrCreate
  ...
  - name: telemetry-ipc-shm
    hostPath:
      path: /dev/shm/telemetry
      type: DirectoryOrCreate
  containers:
    ...
    volumeMounts:
    - name: telemetry-service-config
      mountPath: /config
    ...
  - name: telemetry-ipc-shm
    mountPath: /dev/shm
```

File after the change:

```yaml
spec:
  hostNetwork: true
  hostIPC: true
  volumes:
  - name: telemetry-service-config
```
These changes ensure that a DOCA-based program running outside of a container is able to communicate with DTS over IPC.

**NetFlow**

When the NetFlow exporter is enabled (NetFlow Collector Attributes are set), it sends the NetFlow data to the NetFlow collector specified by the attributes: Address and port. This exporter must be used when using DOCA Telemetry Exporter NetFlow API.

**Fluent Bit**

Fluent Bit export is based on fluent_bit_configs with .exp files for each destination. Every export file corresponds to one of Fluent Bit's destinations. All found and enabled .exp files are used as separate export destinations. Examples can be found after running DTS container under its configuration folder (/opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/).

All .exp files are documented in-place.
Fluent Bit .exp files have 2-level data routing:

- source_tags in .exp files (documented in-place)
- Token-based filtering governed by .fset files (documented in ufm_enterprise.fset)

To run with Fluent Bit exporter, set `enable=1` in required .exp files and set the environment variables before running the application:

```bash
export FLUENT_BIT_EXPORT_ENABLE=1
export FLUENT_BIT_CONFIG_DIR=/path/to/fluent_bit_configs
export LD_LIBRARY_PATH=/opt/mellanox/collectx/lib
```

**Prometheus**

Prometheus exporter sets up endpoint (HTTP server) which keeps the most recent events data as text records.

The Prometheus server can scrape the data from the endpoint while the DOCA-Telemetry-Exporter-API-based application stays active.

Check the generic example of Prometheus records:

```plaintext
event_name_1{label_1="label_1_val", label_2="label_2_val", label_3="label_3_val", label_4="label_4_val"} counter_value_1 timestamp_1
event_name_2{label_1="label_1_val", label_2="label_2_val", label_3="label_3_val", label_4="label_4_val"} counter_value_2 timestamp_2
...
```
Labels are customizable metadata which can be set from data file. Events names could be filtered by token-based name-match according to .fset files.

Set the following environment variables before running.

```bash
# Set the endpoint host and port to enable export.
export PROMETHEUS_ENDPOINT=http://0.0.0.0:9101

# Set indexes as a comma-separated list to keep data for every index field. In this example most recent data will be kept for every record with unique `port_num`. If not set, only one data per source will be kept as the most recent.
export PROMETHEUS_INDEXES=Port_num

# Set path to a file with Prometheus custom labels. Use labels to store information about data source and indexes. If not set, the default labels will be used.
export CLX_METADATA_FILE=/path/to/labels.txt

# Set the folder which contains fset-files. If set, Prometheus will scrape only filtered data according to fieldsets.
export PROMETHEUS_CSET_DIR=/path/to/prometheus_cset
```

**Note**

To scrape the data without the Prometheus server, use:

```bash
curl -s http://0.0.0.0:9101/metrics
```

Or:

```bash
curl -s http://0.0.0.0:9101/{fset_name}
```
DOCA Telemetry Exporter Samples

This section provides DOCA Telemetry Exporter sample implementations on top of the BlueField DPU.

The telemetry exporter samples in this document demonstrate an initial recommended configuration that covers two use cases:

- Standard DOCA Telemetry Exporter data
- DOCA Telemetry Exporter for NetFlow data

The telemetry exporter samples run on the BlueField. If write-to-file is enabled, telemetry data is stored to BlueField's storage. If inter-process communication (IPC) is enabled, data is sent to the DOCA Telemetry Service (DTS) running on the same BlueField.

For information on initializing and configuring DTS, refer to NVIDIA DOCA Telemetry Service Guide.

Info

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Running the Sample

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:
3. Sample (e.g., telemetry_export) usage:

Usage: doca_telemetry_export [DOCA 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

4. For additional information per sample, use the -h option:

/tmp/build/doca_<sample_name> -h

Samples

Telemetry Export
This sample illustrates how to use the telemetry exporter API. The sample uses a custom schema for telemetry exporter.

The sample logic includes:

2. Initializing schema.
3. Creating telemetry exporter source.
4. Creating example events.
5. Reporting example events via DOCA Telemetry Exporter.
6. Destroying source and schema.

Reference:

- /opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/telemetry_export_sample.c
- /opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/telemetry_export_main.c
- /opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/meson.build

**Telemetry Export NetFlow**

This sample illustrates how to use the NetFlow functionality of the telemetry exporter API.

The sample logic includes:

2. Initializing NetFlow.
3. Creating telemetry exporter source.
5. Creating example events.
6. Reporting example events via DOCA Telemetry Exporter.


Reference:

- /opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_netflow/telemetry_export_netfl
- /opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_netflow/telemetry_export_netfl
- /opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_netflowt/meson.build
DOCA Device Emulation

Introduction

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) provide the ability to emulate a PCIe device. The DOCA Device Emulation subsystem provides a low-level software API for users to develop PCIe devices and their controllers. These APIs include discovery, configuration, hot plugging/unplugging, management, and IO path handling. In simpler terms, the libraries enable the user to implement a hardware PCIe function using software, such that the host is not aware that the PCIe function is emulated, and all interactions from the host are routed to software on the BlueField instead of actual hardware.

The diagram shows the potential for device emulation to replace a regular PCIe function of some PCIe device.

- On the left is a conventional setup where the host is connected to a PCIe device (e.g., NVMe SSD). On the host, user applications interact with the kernel driver of that device, using some software interface, and the driver communicates with the hardware/firmware of the device.
- On the right is a setup where the PCIe device is replaced with a BlueField with an application using DOCA Device Emulation. The application can use the DOCA
DevEmu PCI library to control the device, and intercept any IOs written by the host to the PCIe device. Additionally, the application can use other DOCA libraries to perform IO processing (e.g., copying data from host memory using DMA, sending RDMA/Ethernet traffic) and other acceleration libraries for encryption, compression, etc.

**Known Limitations**

- This library is supported at alpha level; backward compatibility is not guaranteed
- VFs are not currently supported
- Some limitations apply when creating a generic emulated function, for more details refer to DOCA DevEmu PCI Generic Limitations.
- Consult your NVIDIA representative for limitations on the emulated device's behavior

**DOCA DevEmu PCI**

**Note**

This library is supported at alpha level; backward compatibility is not guaranteed.

**Introduction**

DOCA DevEmu PCI is part of the DOCA Device Emulation subsystem. It provides low-level software APIs that allow management of an emulated PCIe device using the emulation capability of NVIDIA® BlueField® networking platforms.

It is a common layer for all PCIe emulation modules, such as DOCA DevEmu PCIe Generic Emulation, and DOCA DevEmu Virtio subsystem emulation.

**Prerequisites**
This library follows the architecture of a DOCA Core Context. It is recommended read the following sections beforehand:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

Generic device emulation is part of DOCA device emulation. It is recommended to read the following guides beforehand:

- DOCA Device Emulation

**Environment**

DOCA DevEmu PCI Emulation is supported only on the BlueField target. The BlueField must meet the following requirements

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

.INFO

Please refer to the DOCA Backward Compatibility Policy.

The library must be run with root privileges.

Perform the following:

1. Configure the BlueField to work in DPU mode as described in *NVIDIA BlueField Modes of Operation*.

2. Enable the PCIe switch emulation capability needed for hot plugging emulated PCIe devices. This can be done by running the following command on the host or BlueField:
3. Perform a BlueField system-level reset for the mlxconfig settings to take effect.

To support hot-plug feature, the host must have the following boot parameters:

- Intel CPU:

  ```
  intel_iommu=on iommu=pt pci=realloc
  ```

- AMD CPU:

  ```
  iommu=pt pci=realloc
  ```

This can be done using the following steps:

1. Add the boot parameters:

   ```
   host> sudo nano /etc/default/grub
   Find the variable
   GRUB_CMDLINE_LINUX_DEFAULT="<existing-params>"
   Add the params at the end
   GRUB_CMDLINE_LINUX_DEFAULT="<existing-params> intel_iommu=on iommu=pt pci=realloc"
   ```

2. Update configuration.

**Info**

This process may vary depending on the host OS. Users can find multiple guides online describing this process.
3. Perform warm boot.

4. Confirm that the parameters are in effect:

```bash
host> update-grub
host> grub2-mkconfig -o /boot/grub2/grub.cfg
```

```bash
host> cat /proc/cmdline
<existing-params> intel_iommu=on iommu=pt pci=realloc
```

### Architecture

The DOCA DevEmu PCI library provides 2 main software abstractions, the PCIe type, and the PCIe device. The PCIe type represents the configurations of the emulated device, while the PCIe device represents an instance of an emulated device. Furthermore, any PCIe device instance must be associated with a single PCIe type, while PCIe type can be associated with many PCIe devices.

### Pre Defined PCI Type vs. Generic PCI Type

A PCIe type object can be acquired in 2 different ways:

- Acquire a pre-defined type, using emulation libraries of existing protocols such as DOCA DevEmu Virtio FS library
- Create from scratch using the DOCA DevEmu Generic library

In case of pre-defined type, the configurability of the type is limited.
PCIe Type Name

As part of the DOCA PCIe emulation, every type has a name assigned to it. This property is not part of the PCIe specification, but rather it is a mechanism in DOCA that uniquely identifies the PCIe type.

There cannot be 2 different PCIe types with the same name, even across different processes, unless the type in the second process is configured in identical manner to the first one. Furthermore, attempting to configure the second type with same name but with slight configuration difference will fail.

Create Emulated Device

After configuring the desired DOCA Devemu PCIe type, it is possible to create an emulated device based on the configured type using `doca_devemu_pci_dev_create_rep`. This sequential process ensures that the DOCA DevEmu PCIe device is created with the specified parameters and configuration defined by the PCIe type object. Furthermore, it is possible to destroy the emulated device using `doca_devemu_pci_dev_destroy_rep`.

The created device representor starts in "power_off" state and is not visible to the host until hot-plug sequence is issued by the user, see Hot-plug Emulated Device. The device can then be destroyed only while in "power_off" state.

ℹ️ Info

The created emulated device may outlive the application that created it, see Objects Lifecycle and Persistency.

Hot-plug Emulated Device

Hot-plugging refers to the process of emulating the physical attachment of a PCIe device to the host PCIe subsystem after the system has been powered on and initialized. Note
that some operating systems require additional settings to enable the process of hot-plugging a PCIe device. For supported systems, this feature proves particularly advantageous for systems that need to remain operational at all times while expanding their hardware resources, such as additional storage and networking capabilities. DOCA DevEmu PCI provides software APIs that allow users to emulate this process in an asynchronous manner.

![Diagram](image-url)
When creating a PCIe device object, if it starts in "power off" state, then the device is not yet visible to the host. It is possible then, from the BlueField, to hot-plug the device. This starts an async process of the device getting hot-plugged towards the host. Once the process completes, the emulated device transitions to "power on" and becomes visible to the host. Usually at this stage, the emulated device receives its BDF address. The hot-unplug process works in similar async manner.

Using DOCA API, the BlueField Arm can register to any changes to the hot-plug state of each emulated device using `doca_devemu_pci_dev_event_hotplug_state_change_register`.

**Emulated Device Discovery**

The emulated device is represented as a `doca_devinfo_rep`. It is possible to iterate through all the emulated devices as explained in DOCA Core Representor Discovery.

There are 2 ways of filtering the list of emulated devices:

- Get all emulated devices – use `DOCA_DEVINFO_REP_FILTER_EMULATED` as the filter argument in `doca_devinfo_rep_create_list`

- Get all emulated devices that belong to a certain type – `doca_devemu_pci_type_create_rep_list`

**Objects Lifecycle and Persistency**

This section creates distinction between firmware resources and software resources:

- Firmware resources persist until the next power cycle, and can be accessible from different processes on the BlueField Arm. Such resources are not cleared once the application exits.

- Software resources are representations of firmware resources, and are only relevant for the same thread

Using this terminology, it is possible to describe the objects as follows:
- The PCIe type object `doca_devemu_pci_type` represents a PCIe type firmware resource. The resource persists if any of the following apply:
  - There is at least 1 process holding reference to the PCIe type
  - There is at least 1 PCIe device firmware resource belonging to this type
- The emulated device representor, `doca_devinfo_rep`, represents an emulated PCIe function firmware resource:
  - `doca_devemu_pci_dev_create_rep` can be used to create such firmware resource
  - To destroy the firmware resource, `doca_devemu_pci_dev_destroy_rep` can be used
  - For static functions, the representor resource persists until configured otherwise in NVCONFIG
  - To find existing PCIe device firmware resources, use `doca_devemu_pci_type_create_rep_list`

**Function Level Reset**

The created emulated devices support PCIe function level reset (FLR).

Using DOCA API, the BlueField Arm can register to FLR event using `doca_devemu_pci_dev_event_flr_register`. Once the driver requests FLR, this event is triggered, calling the user provided callback.

Once FLR is detected, it is expected for the BlueField Arm to do the following:

- Destroy all resources related to the PCIe device. For information on such resources, refer to the guide of concrete PCIe type (generic/virtiofs).
- Stop the PCIe device
- Start the PCIe device again
Device Support

DOCA PCIe Device emulation requires a device to operate. For picking a device, see DOCA Core Device Discovery.

The device emulation library is only supported for BlueField-3.

As device capabilities may change in the future (see Capability Checking), it is recommended that users choose a device using the following method:

- `doca_devemu_pci_cap_type_is_hotplug_supported` – for create and hot-plug support
- `doca_devemu_pci_cap_type_is_mgmt_supported` – for device discovery only

PCle Device

Configuration Phase

To start using the DOCA DevEmu PCI Device, users must first go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow retrieval of events.

Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

Mandatory Configurations

All mandatory configurations are provided during the creation of the PCIe device.

These configurations are as follows:

- A DOCA DevEmu PCIe type object
- A DOCA Device Representor, representing an emulated function with the same type as the provided PCIe object type
• A DOCA Progress Engine object

Optional Configurations

These configurations are optional. If not set, then a default value is used:

• Registering to events as described in the "Events" section. By default, the user does not receive events.

Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine.

Events

The DOCA DevEmu PCI device exposes asynchronous events to notify about sudden changes according to DOCA Core architecture.

Common events are described in DOCA Core Event.

Hotplug State Change

The hotplug state change event allows users to receive notifications whenever the hotplug state of the emulated device changes. See section "Hot-plug Emulated Device".

Event Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the</td>
<td>doca_devemu_pci_dev_event_hotplug_state_change_register</td>
<td>doca_devemu_pci_cap_type_is_hotplug_supported</td>
</tr>
<tr>
<td>event</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Event Trigger Condition
The event is triggered anytime an asynchronous transition happens as follows:

- DOCA_DEVEMU_PCI_HP_STATE_PLUG_IN_PROGRESS  DOCA_DEVEMU_PCI_HP_STATE_POWER_ON
- DOCA_DEVEMU_PCI_HP_STATE_UNPLUG_IN_PROGRESS  DOCA_DEVEMU_PCI_HP_STATE_POWER_OFF
- DOCA_DEVEMU_PCI_HP_STATE_POWER_ON  DOCA_DEVEMU_PCI_HP_STATE_UNPLUG_IN_PROGR... (when initiated by the host)

Any transition initiated by user is not triggered (e.g., calling hotplug to transition from POWER_OFF to PLUG_IN_PROGRESS).

The following APIs can be used to initiate hotplug or hot-unplug transition processes:

- `doca_devemu_pci_dev_hotplug`
- `doca_devemu_pci_dev_hotunplug`

**Event Output**

Common output as described in DOCA Core Event.

Additionally, the internal cached hotplug state is updated and can be fetched using `doca_devemu_pci_dev_get_hotplug_state`.

**Event Handling**

Once the event is triggered, it means that the hotplug state has changed. The application is expected to do the following:

- Retrieve the new hotplug state using `doca_devemu_pci_dev_get_hotplug_state`

**Function Level Reset**
The FLR event allows users to receive notifications whenever the host initiates an FLR flow. See section "Function Level Reset".

**Event Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event</td>
<td>doca_devemu_pci_dev_event_flr_register</td>
</tr>
</tbody>
</table>

**Event Trigger Condition**

The event is triggered anytime the host driver initiates an FLR flow. See section "Function Level Reset".

**Event Output**

Common output as described in DOCA Core Event.

Additionally, the internal cached FLR indicator is updated and can be fetched using doca_devemu_pci_dev_is_flr.

**Event Handling**

Once the event is triggered, it means that the host driver has initiated the FLR flow.

The user must handle the FLR flow by doing the following:

1. Flush all the outstanding requests back to the associated resource
2. Release all the PCIe device resources dynamically created after device start
3. Stop the PCIe device – doca_ctx_stop
4. Start the PCIe device again – doca_ctx_start
Call doca_pe_progress repeatedly until the PCIe device transitions to "running" state.

For more information on starting the PCIe device again, refer to section "State Machine".

State Machine

The DOCA DevEmu PCI device object follows the context state machine as described in DOCA Core Context State Machine.

The following section describes how to transition to any state and what is allowed in each state.

Idle

In this state, it is expected that application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all resources have been destroyed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all resources have been destroyed</td>
</tr>
</tbody>
</table>
**Starting**

In this state, it is expected that application:

- Calls progress to allow transition to next state
- Keeps context in this state until FLR flow is complete

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after receiving FLR event (i.e., while FLR is in progress)</td>
</tr>
</tbody>
</table>

**Running**

In this state, it is expected that application:

- Calls progress to receive events
- Creates/destroys PCIe device resources

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
<tr>
<td>Starting</td>
<td>Call progress until FLR flow is completed</td>
</tr>
</tbody>
</table>

**Stopping**

In this state, it is expected that application:

- Destroys all emulated device resources as described in section "Function Level Reset".
Allowed operations:

- Destroying PCIe device resources

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Call stop without freeing emulated device resources</td>
</tr>
</tbody>
</table>

## DOCA DevEmu PCI Generic

**Note**

This library is supported at alpha level; backward compatibility is not guaranteed.

This guide provides instructions on building and developing applications that require emulation of a generic PCIe device.

### Introduction

DOCA DevEmu PCI Generic is part of the DOCA Device Emulation subsystem. It provides low-level software APIs that allow creation of a custom PCIe device using the emulation capability of NVIDIA® BlueField®.

For example, it enables emulating an NVMe device by creating a generic emulated device, configuring its capabilities and BAR to be compliant with the NVMe spec, and operating it from the DPU as necessary.

### Prerequisites
This library follows the architecture of a DOCA Core Context. It is recommended to read the following sections beforehand:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

Generic device emulation is part of DOCA PCIe device emulation. It is recommended to read the following guides beforehand:

- DOCA Device Emulation
- DOCA DevEmu PCI

**Environment**

DOCA DevEmu PCI Generic Emulation is supported only on the BlueField target. The BlueField must meet the following requirements:

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

ℹ️ **Info**

Please refer to the DOCA Backward Compatibility Policy.

Library must be run with root privileges.

Please refer to DOCA DevEmu PCI Environment, for further necessary configurations.

**Architecture**

DOCA DevEmu PCI Generic allows the creation of a generic PCI type. The PCI Type is part of the DOCA DevEmu PCI library. It is the component responsible for configuring the
capabilities and bar layout of emulated devices.

The PCI Type can be considered as the template for creating emulated devices. Such that the user first configures a type, and then they can use it to create multiple emulated devices that have the same configuration.

For a more concrete example, consider that you would like to emulate an NVMe device, then you would create a type and configure its capabilities and BAR to be compliant with the NVMe spec, after that you can use the same type, to generate multiple NVMe emulated devices.

**PCIe Configuration Space**

The PCIe configuration space is 256 bytes long and has a header that is 64 bytes long. Each field can be referred to as a register (e.g., device ID).

Every PCIe device is required to implement the PCIe configuration space as defined in the PCIe specification.

The host can then read and/or write to registers in the PCIe configuration space. This allows the PCIe driver and the BIOS to interact with the device and perform the required setup.

It is possible to configure registers in the PCIe configuration space header as shown in the following diagram:
The following registers are read-only, and they are used to identify the device:

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Code</td>
<td>Defines the functionality of the device</td>
<td>0x020000&lt;br&gt;Class: 0x02 (Network Controller)&lt;br&gt;Subclass: 0x00 (Ethernet Controller)&lt;br&gt;Prog IF: 0x00 (N/A)</td>
</tr>
<tr>
<td>Revision ID</td>
<td>Unique identifier of the device revision&lt;br&gt;Vendor allocates ID by itself</td>
<td>0x01&lt;br&gt;(Rev 01)</td>
</tr>
<tr>
<td>Vendor ID</td>
<td>Unique identifier of the chipset vendor&lt;br&gt;Vendor allocates ID from the PCI-SIG</td>
<td>0x15b3&lt;br&gt;Nvidia</td>
</tr>
<tr>
<td>Device ID</td>
<td>Unique identifier of the chipset&lt;br&gt;Vendor allocates ID by itself</td>
<td>0xa2dc&lt;br&gt;BlueField-3 integrated ConnectX-7 network controller</td>
</tr>
<tr>
<td>Subsystem Vendor ID</td>
<td>Unique identifier of the card vendor&lt;br&gt;Vendor allocates ID from the PCI-SIG</td>
<td>0x15b3&lt;br&gt;Nvidia</td>
</tr>
<tr>
<td>Subsystem ID</td>
<td>Unique identifier of the card&lt;br&gt;Vendor allocates ID by itself</td>
<td>0x0051</td>
</tr>
</tbody>
</table>
While the PCIe configuration space can be used to interact with the PCIe device, it is not enough to implement the functionality that is targeted by the device. Rather, it is only relevant for the PCIe layer.

To enable protocol-specific functionality, the device configures additional memory regions referred to as base address registers (BARs) that can be used by the host to interact with the device. Different from the PCIe configuration space, BARs are defined by the device and interactions with them is device-specific. For example, the PCIe driver interacts with an NVMe device’s PCIe configuration space according to the PCIe spec, while the NVMe driver interacts with the BAR regions according to the NVMe spec.

Any read/write requests on the BAR are typically routed to the hardware, but in case of an emulated device, the requests are routed to the software.

The DOCA DevEmu PCI type library provides APIs that allow software to pick the mechanism used for routing the requests to software, while taking into consideration common design patterns utilized in existing devices.

Each PCIe device can have up to 6 BARs with varying properties. During the PCIe bus enumeration process, the PCIe device must be able to advertise information about the layout of each BAR. Based on the advertised information, the BIOS/OS then allocates a memory region for each BAR and assigns the address to the relevant BAR in the PCIe configuration space header. The driver can then use the assigned memory address to perform reads/writes to the BAR.

**BAR Layout**

The PCIe device must be able to provide information with regards to each BAR's layout.
The layout can be split into 2 types, each with their own properties as detailed in the following subsections.

**I/O Mapped**

According to the PCIe specification, the following represents the I/O mapped BAR:

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

Additionally, the BAR register is responsible for advertising the requested size during enumeration.

⚠️ **Info**

The size must be a power of 2.

Users can use the following API to set a BAR as I/O mapped:

```c
doca_devemu_pci_type_set_io_bar_conf(struct doca_devemu_pci_type *pci_type, uint8_t id, uint8_t log_sz)
```

- `id` – the BAR ID
- `log_sz` – the log of the BAR size

**Memory Mapped**

According to the PCIe specification, the following represents the memory mapped BAR:
Additionally, the BAR register is responsible for advertising the requested size during enumeration.

![Diagram of BAR register]

**Info**

The size must be a power of 2.

The memory mapped BAR allows a 64-bit address to be assigned. To achieve this, users must specify the bar Memory Type as 64-bit, and then set the next BAR's (BAR ID + 1) size to be 0.

Setting the pre-fetchable bit indicates that reads to the BAR have no side-effects.

Users can use the following API to set a BAR as memory mapped:

```c
docha_devemu_pci_type_set_memory_bar_conf(struct docha_devemu_pci_type *pci_type, uint8_t id, uint8_t log_sz, enum docha_devemu_pci_bar_mem_type memory_type, uint8_t prefetchable)
```

- **id** – the BAR ID
- **log_sz** – the log of the BAR size. If set to 0, then the size is considered as 0 (instead of 1).
- **memory_type** – specifies the memory type of the BAR. If set to 64-bit, then the next BAR must have `log_sz` set to 0.
- **prefetchable** – indicates whether the BAR memory is pre-fetchable or not (a value of 1 or 0 respectively)
BAR Regions

BAR regions refer to memory regions that make up a BAR layout. This is not something that is part of the PCIe specification, rather it is a DOCA concept that allows the user to customize behavior of the BAR when interacted with by the host.

The BAR region defines the behavior when the host performs a read/write to an address within the BAR, such that every address falls in some memory region as defined by the user.

Common Configuration

All BAR regions have these configurations in common:

- **id** – the BAR ID that the region is part of
- **start_addr** – the start address of the region within the BAR layout relative to the BAR. 0 indicates the start of the BAR layout.
- **size** – the size of the BAR region

Currently, there are 4 BAR region types, defining different behavior:

- Stateful
- DB by offset
- DB by data
Generic Control Path (Stateful BAR Region)

Stateful region can be used as a shared memory, such that the contents are maintained in firmware. A read from the driver returns the latest value, while a write updates the value and triggers an event to software running on the DPU.

This can be useful for communication between the driver and the device, during the control path (e.g., exposing capabilities, initialization).

Info

Some limitations apply, please see Limitations section

Driver Read

A read from the driver returns the latest value written to the region, whether written by the host or by the driver itself.
**Driver Write**

A write from the driver updates the value at the written address and notifies software running on the Arm that a write has occurred. The notification on the Arm arrives as an asynchronous event (see `doca_devemu_pci_dev_event_bar_stateful_region_driver_write`).

**Info**

The event that arrives to Arm software is asynchronous such that it may arrive after the driver has completed the write.

**DPU Read**

The DPU can read the values of the stateful region using `doca_devemu_pci_dev_query_bar_stateful_region_values`. This returns the latest snapshot of the stateful region values. It can be particularly useful to find what was written by the driver after the "stateful region driver write event" occurs.

**DPU Write**

The DPU can write the values of the stateful region using `doca_devemu_pci_dev_modify_bar_stateful_region_values`. This updates the values such that
subsequent reads from the driver or the DPU returns these values.

**Default Values**

The DPU is able to set default values to the stateful region. Default values come in 2 layers:

- **Type default values** – these values are set for all devices that have the same type. This can be set only if no device currently exists.

- **Device default values** – these values are set for a specific device and take affect on the next FLR cycle or the next hotplug of the device

A read of the stateful region follows the following hierarchy:

1. Return the latest value as written by the host or driver (whichever was done last).
2. Return the device default values.
3. Return the type default values.
4. Return 0.
No Defaults

- Driver Write
- DPU Write

Zeroes

Type Default

- Driver Write
- DPU Write

Type Defaults

Zeroes

Device Default

- Driver Write
- DPU Write

Device Defaults

Type Defaults
Generic Data Path (DB BAR Region)

Doorbell (DB) regions can be used to implement a consumer-producer queue between the driver and the DPU, such that a write from the driver would trigger an event on the DPU through DPA, allowing it to fetch the written value. This can be useful for communication between the driver and the device, during the data path allowing IO processing.

While DBs are not part of the PCIe specification, it is a widely used mechanism by vendors (e.g., RDMA QP, NVMe SQ, virtio VQ, etc).

The same DB region can be used to manage multiple DBs, such that each DB can be used to implement a queue.

The DPU software can utilize DB resources individually:

- Each DB resource has a unique zero-based index referred to as DB ID
- DB resource can be managed (create/destroy/modify/query) individually
- Each DB resource has a separate notification mechanism. That is, the notification on DPU is triggered for each DB separately.

Driver Write
The DB usually consists of a numeric value (e.g., \texttt{uint32\_t}) representing the consumer/producer index of the queue.

When the driver writes to the DB region, the related DB resource gets updated with the written value, and a notification is sent to the DPU.

When driver writes to the DB BAR region it must adhere to the following:

- The size of the write must match the size of the DB value (e.g., \texttt{uint32\_t})
- The offset within the region must be aligned to the DB stride size or the DB size

The flow would look something as the following:

- Driver performs a write of the DB value at some offset within the DB BAR region
- DPU calculates the DB ID that the write is intended for. Depending on the region type:
  - \texttt{DB by offset} – DPU calculates the DB ID based on the write offset relative to the DB BAR region
  - \texttt{DB by data} – DPU parses the written DB value and extracts the DB ID from it
- DPU updates the DB resource with the matching DB ID to the value written by the driver
- DPU sends a notification to the DPA application, informing it that the value of DB with DB ID has been updated by the driver

**Driver Read**

The driver should not attempt to read from the DB region. Doing so results in anomalous behavior.

**BlueField Write**
The BlueField can update the value of each DB resource individually using `doca_devemu_pci_db_modify_value`. This produces similar side effects as though the driver updated the value using a write to the DB region.

**BlueField Read**

The BlueField can read the value of each DB resource individually using one of the following methods:

- Read the value from the BlueField Arm using `doca_devemu_pci_db_query_value`
- Read the value from the DPA using `doca_dpa_dev_devemu_pci_db_get_value`

The first option is a time consuming operation and is only recommended for the control path. In the data path, it is recommended to use the second option only.

**DB by Offset**

The API `doca_devemu_pci_type_set_bar_db_region_by_offset_conf` can be used to set up DB by offset region. When the driver writes a DB value using this region, the DPU receives a notification for the relevant DB resource, based on the write offset, such that the DB ID is calculated as follows: \( \text{db_id} = \frac{\text{write_offset}}{\text{db_stride_size}} \).

⚠️ **Warning**
The area that is part of the stride but not part of the doorbell, should not be used for any read/write operation, doing so will result in undefined anomalous.

**DB by Data**

The API `doса_devemu_pci_type_set_bar_db_region_by_data_conf` can be used to set up DB by data region. When the driver writes a DB value using this region, the DPU receives a notification for the relevant DB resource based on the written DB value, such that there is no relation between the write offset and the DB triggered. This DB region assumes that the DB ID is embedded within the DB value written by the driver. When setting up this region, the user must specify where the Most Significant Byte (MSB) and Least Significant Byte (LSB) of the DB ID are embedded in the DB value.

The DPU follows these steps to extract the DB ID from the DB value:

- **Driver writes the DB value**
- **BlueField extracts the bytes between MSB and LSB**
- **DPU compares MSB index with LSB index**
  - If MSB index greater than LSB index: The extracted value is interpreted as Little Endian
- If LSB index greater than MSB index: The extracted value is interpreted as Big Endian

Example:

DB size is 4 bytes, LSB is 1, and MSB is 3.

- Driver writes value \(0xCCDDEEFF\) to DB region at index 0 in Little Endian
  - The value is written to memory as follows: \([0]=FF\ [1]=EE\ [2]=DD\ [3]=CC\)
  - The relevant bytes, are the following: \([1]=EE\ [2]=DD\ [3]=CC\)
  - Since MSB (3) is greater than LSB (1), the value is interpreted as Little Endian: \(db_id = 0xCCDDEE\)

**MSI-X Capability (MSI-X BAR Region)**

Message signaled interrupts extended (MSI-X) is commonly used by PCIe devices to send interrupts over the PCIe bus to the host driver. DOCA APIs allow users to expose the MSI-X capability as per the PCIe specification, and to later use it to send interrupts to the host driver.

To configure it, users must provide the following:

- The number of MSI-X vectors which can be done using `doca_devemu_pci_type_set_num_msix`
- Define an MSI-X table
- Define an MSI-X PBA

**MSI-X Table BAR Region**

As per the PCIe specification, to expose the MSI-X capability, the device must designate a memory region within its BAR as an MSI-X table region. In DOCA, this can be done using `doca_devemu_pci_type_set_bar_msix_table_region_conf`. 
**MSI-X PBA BAR Region**

As per the PCIe specification, to expose the MSI-X capability, the device must designate a memory region within its BAR as an MSI-X pending bit array (PBA) region. In DOCA, this can be done using `doca_devemu_pci_type_set_bar_msix_pba_region_conf`.

**Raising MSI-X From DPU**

It is possible to raise an MSI-X for each vector individually. This can be done only using the DPA API `doca_dpa_dev_devemu_pci_msix_raise`.

**DMA Memory**

Some operations require accessing memory which is set up by the host driver. DOCA's device emulation APIs allow users to access such I/O memory using the DOCA mmap (see [DOCA Core Memory Subsystem](#)).

After starting the PCIe device, it is possible to acquire an mmap that references the host memory using `doca_devemu_pci_mmap_create`. After creating this mmap, it is possible to configure it by providing:

- Access permissions
- Host memory range
- DOCA devices that can access the memory

The mmap can then be used to create buffers that reference memory on the host. The buffers' addresses would not be locally accessible (i.e., CPU cannot dereference the address), instead the addresses would be I/O addresses as defined by the host driver.

The buffers created from the mmap can then be used with other DOCA libraries and accept a `doca_buf` as an input. This includes:

- **DOCA DMA**
• **DOCA RDMA**

• **DOCA Ethernet**

• **DOCA AES-GCM**

## Function Level Reset

FLR can be handled as described in [DOCA DevEmu PCI FLR](#). Additionally, users must ensure that the following resources are destroyed before stopping the PCIe device:

- Doorbells created using `doa_devemu_pci_db_create_on_dpa`
- MSI-X vectors created using `doa_devemu_pci_msix_create_on_dpa`
- Memory maps created using `doa_devemu_pci_mmap_create`

## Limitations

Based on explanation in "Driver Write", user can assume that DOCA DevEmu PCI Generic supports creating emulated PCI devices with the limitation that when a driver writes to a register, the value is immediately available for subsequent reads from the same register. However, this immediate availability does not ensure that any required internal actions triggered by the write have been completed. It is recommended to rely on specific different register values to confirm completion of the write action. For instance, when implementing a write-to-clear operation, e.g. writing 1 to register A to clear register B, it is advisable to poll register B until it indicates the desired state. This approach ensures that the write action has been successfully executed. If a device specification requires certain actions to be completed before exposing written values for subsequent reads, such a device cannot be emulated using the DOCA DevEmu PCI generic framework.

## Device Support

DOCA PCI Device emulation requires a device to operate. For information on picking a device, see [DOCA DevEmu PCI Device Support](#).
Some devices can allow different capabilities as follows:

- The maximum number of emulated devices
- The maximum number of different PCIe types
- The maximum number of BARs
- The maximum BAR size
- The maximum number of doorbells
- The maximum number of MSI-X vectors

For each BAR region type there are capabilities for:

- Whether the region is supported
- The maximum number of regions with this type
- The start address alignment of the region
- The size alignment of the region
- The min/max size of the region

**Tip**

As the list of capabilities can be long, it is recommended to use the NVIDIA DOCA Capabilities Print Tool to get an overview of all the available capabilities.

Run the tool as root user as follows:

```
$ sudo /opt/mellanox/doca/tools/doca_caps -p <pci-address> -b devemu_pci
```

Example output:
```
devemu_pci
  max_hotplug_devices  15
  max_pci_types        2
  type_log_min_bar_size 12
```
type_log_max_bar_size 30
type_max_num_msix 11
type_max_num_db 64
type_log_min_db_size 1
type_log_max_db_size 2
type_log_min_db_stride_size 2
type_log_max_db_stride_size 12
type_max_bars 2
bar_max_bar_regions 12
type_max_bar_regions 12
bar_db_region_identify_by_offset supported
bar_db_region_identify_by_data supported
bar_db_region_block_size 4096
bar_db_region_max_num_region_blocks 16
type_max_bar_db_regions 2
bar_max_bar_db_regions 2
bar_db_region_start_addr_alignment 4096
bar_stateful_region_block_size 64
bar_stateful_region_max_num_region_blocks 4
type_max_bar_stateful_regions 1
bar_max_bar_stateful_regions 1
bar_stateful_region_start_addr_alignment 64
bar_msix_table_region_block_size 4096
bar_msix_table_region_max_num_region_blocks 1
type_max_bar_msix_table_regions 1
bar_max_bar_msix_table_regions 1
bar_msix_table_region_start_addr_alignment 4096
bar_msix_pba_region_block_size 4096
bar_msix_pba_region_max_num_region_blocks 1
type_max_bar_msix_pba_regions 1
bar_max_bar_msix_pba_regions 1
bar_msix_pba_region_start_addr_alignment 4096
bar_is_32_bit_supported unsupported
bar_is_1_mb_supported unsupported
bar_is_64_bit_supported supported
pci_type_hotplug supported
pci_type_mgmt supported
**Configurations**

This section describes the configurations of the DOCA DevEmu PCI Type object, that can be provided before start.

To find if a configuration is supported or what its min/max value is, refer to [Device Support](#).

**Mandatory Configurations**

The following are mandatory configurations and must be provided before starting the PCI type:

- A DOCA device that is an emulation manager or hotplug manager. See [Device Support](#).

**Optional Configurations**

The following configurations are optional:

- The PCIe device ID
- The PCIe vendor ID
- The PCIe subsystem ID
- The PCIe subsystem vendor ID
- The PCIe revision ID
- The PCIe class code
- The number of MSI-X vectors for MSI-X capability
- One or more memory mapped BARs
- One or more I/O mapped BARs
- One or more DB region
- An MSI-X table and PBA regions
- One or more stateful regions

**Info**

If these configurations are not set then a default value is used.

## PCI Device

### Configuration Phase

This section describes additional configuration options, on top of the ones already described in [DOCA DevEmu PCI Device Configuration Phase](#).

### Configurations

The context can be configured to match the application's use case.

To find if a configuration is supported or what its min/max value is, refer to [Device Support](#).

### Optional Configurations

The following configurations are optional:

- Setting the stateful regions' default values – If not set, then the type default values are used. See [stateful region default values](#) for more.
Execution Phase

This section describes additional events, on top of the ones already described in DOCA DevEmu PCI Device Events.

Events

DOCA DevEmu PCI Device exposes asynchronous events to notify about changes that happen suddenly according to the DOCA Core architecture.

Common events are described in DOCA Core Event.

BAR Stateful Region Driver Write

The stateful region driver write event allows you to receive notifications whenever the host driver writes to the stateful BAR region. See section "Driver Write" for more information.

Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event</td>
<td>doca_devemu_pci_dev_event_bar_stateful_region_driver_write_register</td>
<td>doca_devemu_pci_cap_type_get_max_bar_stateful_regions</td>
</tr>
</tbody>
</table>

If there are multiple stateful regions for the same device, then registration is done separately for each region. The details provided on registration (i.e., bar_id and start address) must match a region previously configured for PCIe type.

Trigger Condition

The event is triggered anytime the host driver writes to the stateful region. See section "Driver Write" for more information.

Output

Common output as described in DOCA Core Event.
Additionally, the event callback receives an event object of type `struct
doca_devemu_pci_dev_event_bar_stateful_region_driver_write` which can be used to retrieve:

- The DOCA DevEmu PCI Device representing the emulated device that triggered the event – `doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_pci_dev`

- The ID of the BAR containing the stateful region – `doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_bar_id`

- The start address of the stateful region – `doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_bar_region_start_addr`

### Event Handling

Once the event is triggered, it means that the host driver has written to someplace in the region.

The user must perform either of the following:

- Query the new values of the stateful region – `doca_devemu_pci_dev_query_bar_stateful_region_values`

- Modify the values of the stateful region – `doca_devemu_pci_dev_modify_bar_stateful_region_values`

It is possible also to do both. However, it is important that the memory areas that the host wrote to are either queried or overwritten with a modify operation.

### Note

Failure to do so results in a recurring event. For example, if the host wrote to the first half of the region, but BlueField Arm only queries the second half of the region after receiving the event. Then the library retriggers the event, assuming that the user did not handle the event.
PCI Device DB

After the PCIe device has been created, it can be used to create DB objects, each DB object represents a DB resource identified by a DB ID. See Generic Data Path (DB BAR Region).

When creating the DB, the DB ID must be provided, this can hold different meaning for DB by offset and DB by data. The DB object can then be used to get a notification to the DPA once a driver write occurs, and to fetch the latest value using the DPA.

Configuration

The flow for creating and configuring a DB should be as follows:

1. Create the DB object:

   arm> doca_devemu_pci_db_create_on_dpa

2. (Optional) Query the DB value:

   arm> doca_devemu_pci_db_query_value

3. (Optional) Modify the DB value:

   arm> doca_devemu_pci_db_modify_value

4. Get the DB DPA handle for referencing the DB from the DPA:
5. Bind the DB to the DB completion context using the handle from the previous step:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_bind_db
```

⚠️ **Warning**

It is important to perform this step before the next one. Otherwise, the DB completion context will start receiving completions for an unbound DB.

6. Start the DB to start receiving completions on DPA:

```
arm> doca_devemu_pci_db_start
```

ℹ️ **Info**

Once DB is started, a completion is immediately generated on the DPA.

Similarly the flow for destroying a DB would look as follows:

1. Stop the DB to stop receiving completions:

```
arm> doca_devemu_pci_db_stop
```
2. Acknowledge all completions related to this DB:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_ack
```

3. Unbind the DB from the DB completion context:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_unbind_db
```

⚠️ Warning

Make sure to not perform this step more than once.

4. Destroy the DB object:
Fetching DBs on DPA

To fetch DBs on DPA, a DB completion context can be used. The DB completion context serves the following purposes:

- Notifying a DPA thread that a DB value has been updated (wakes up thread)
- Providing information about which DB has been updated

The following flow shows how to use the same DB completion context to get notified whenever any of the DBs are updated, and to find which DBs were actually updated, and finally to get the DBs' values:

1. Get DB completion element:
   
   ```
   arm> doca_devemu_pci_db_destroy
   ```

2. Get DB from completion:

   ```
   doca_dpa_dev_devemu_pci_get_db_completion
   ```

3. Store the DB (e.g., in an array).

4. Repeat steps 1-3 until there are no more completions.

5. Acknowledge the number of received completions:

   ```
   doca_dpa_dev_devemu_pci_db_completion_ack
   ```
6. Request notification on DPA for the next completion:

   `doca_dpa_dev_devemu_pci_db_completion_request_notification`

7. Go over the DBs stored in step 3 and for each DB:

   1. Request a notification for the next time the host driver writes to this DB:

      `doca_dpa_dev_devemu_pci_db_request_notification`

   2. Get the most recent value of the DB:

      `doca_dpa_dev_devemu_pci_db_get_value`

**Query/Modify DB from Arm**

It is possible to query the DB value of a particular DB using `doca_devmu_pci_db_query_value` on the Arm. Similarly, it is possible to modify the DB value using `doca_devmu_pci_db_modify_value`. When modifying the DB value, the side effects of such modification is the same as if the host driver updated the DB value.

**Tip**

Querying and modifying operations from the Arm are time consuming and should be used in the control path only. Fetching DBs on DPA is the recommended approach for retrieval of DB values in the data path.
**PCle Device MSI-X Vector**

After the PCle device has been created, it can be used to create MSI-X objects. Each MSI-X object represents an MSI-X vector identified by the vector index.

The MSI-X object can be used to send a notification to the host driver from the DPA.

**Configuration**

The MSI-X object can be created using `doca_devemu_pci_msix_create_on_dpa`. An MSI-X vector index must be provided during creation, this is a value in the range $[0, \text{num}_{\text{msix}})$, such that $\text{num}_{\text{msix}}$ is the value previously set using `doca_devemu_pci_type_set_num_msix`.

Once the MSI-X object is created, `doca_devemu_pci_msix_get_dpa_handle` can be used to get a DPA handle for use within the DPA.

**Raising MSI-X**

The MSI-X object can be used on the DPA to raise an MSI-X vector using `doca_dpa_dev_devemu_pci_msix_raise`.

**DOCA DevEmu Generic Samples**

This section describes DOCA DevEmu Generic samples.

The samples illustrate how to use the DOCA DevEmu Generic API to do the following:

- List details about emulated devices with same generic type
- Create and hot-plug/hot-unplug an emulated device with a generic type
- Handle Host driver write using stateful region
- Handle Host driver write using DB region
- Raise MSI-X to the Host driver
• Perform DMA operation to copy memory buffer between the Host driver and the DPU Arm

Structure

All the samples utilize the same generic PCI type. The configurations of the type reside in /opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h

The structure for some samples is as follows:

• /opt/mellanox/doca/samples/doca_devemu/<sample directory>
  1. dpu
     1. host
     2. device
  2. host

Samples following this structure will have two binaries: dpu (1) and host (2), the former should be run on the BlueField and represents the controller of the emulated device, while the latter should be run on the host and represents the host driver.

For simplicity, the host (2) side is based on the VFIO driver, allowing development of a driver in user-space.

Within the dpu (a) directory, there is a host (a) and device (b) directories. host in this case refers to the BlueField Arm processor, while device refers to the DPA processor. Both directories are compiled into a single binary.

Running the Samples

1. Refer to the following documents:
   • NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   • NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

```bash
cd /opt/mellanox/doca/samples/doca_devemu/<sample_name>[/dpu or /host]
meson /tmp/build
ninja -C /tmp/build
```

**Info**

The binary `doca_<sample_name>[_dpu or _host]` is created under `/tmp/build/`.

3. Sample (e.g., `doca_devemu_pci_device_db`) usage:

1. BlueField side (`doca_devemu_pci_device_db_dpu`):

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

   **DOCA Flags:**
   - `-h, --help ` Print a help synopsis
   - `--version ` Print program version information
   - `--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 ` The DOCA device PCI address. Format: XXXX:XX:XX.X or XX:XX.X
   - `-u, --vuid ` DOCA Devemu emulated device VUID. Sample will use this device to handle Doorbells from Host
   - `-r, --region-index ` The index of the DB region as defined in `devemu_pci_type_config.h`. Integer
   - `-i, --db-id ` The DB ID of the DB. Sample will listen on DBs related to this DB ID. Integer
2. Host side (doca_devemu_pci_device_db_host):

Usage: doca_devemu_pci_device_db_host [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=D 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=D 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` PCI address of the emulated device. Format: XXXX:XX:XX.X
- `g, --vfio-group` VFIO group ID of the device. Integer
- `r, --region-index` The index of the DB region as defined in devemu_pci_type_config.h. Integer
- `d, --db-index` The index of the Doorbell to write to. The sample will write at byte offset (db-index * db-stride)
- `w, --db-value` A 4B value to write to the Doorbell. Will be written in Big Endian

4. For additional information per sample, use the `-h` option:

/tmp/build/<sample_name> -h

Additional sample setup:

- The BlueField samples require the emulated device to be already hot-plugged:
  - Such samples expect the VUID of the hot-plugged device (`-u, --vuid`)
  - The list sample can be used to find if any hot-plugged devices exist and what their VUID is
  - The hot-plug sample can be used to hot plug a device if no such device already exists
The host samples require the emulated device to be already hot-plugged and that the device is bound to the VFIO driver:

- The samples expect 2 parameters `-p (--pci-addr)` and `-g (--vfio-group)` of the emulated device as seen by the host
- The [PCI Device List](https://example.com) sample can be used from the BlueField to find the PCIe address of the emulated device on the host
- Once the PCIe address is found, the host can use the script 
  `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py` to bind the VFIO driver

```bash
$ sudo python3 /opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py
<pcie-address-of-emulated-dev>
```

- The script is a python3 script which expects the PCIe address of the emulated device as a positional argument (e.g., `0000:3e:00.0`)
- The script outputs the VFIO group ID
- The script must be used only once after the device is hot-plugged towards the host for the first time

The hot-unplug sample requires that the device be unbound from the VFIO driver:

- Use the script located at
  `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py` from the host to unbind the VFIO driver as follows:

```bash
$ sudo python3 /opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py
<pcie-address-of-emulated-dev> --unbind
```

- This python3 script expects the PCIe address of the emulated device as a positional argument (e.g., `0000:3e:00.0`) along with the `--unbind` argument
Samples

PCI Device List

This sample illustrates how to list all emulated devices that have the generic type configured in /opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h.

The sample logic includes:

1. Initializing the generic PCIe type based on /opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h.

2. Creating a list of all emulated devices belonging to this type.

3. Iterating over the emulated devices.

4. Dumping their VUID.

5. Dumping their PCIe address as seen by the host.

6. Releasing the resources.

References:

- /opt/mellanox/doca/samples/doca_devemu/
  - devemu_pci_device_list/
    - devemu_pci_device_list_sample.c
    - devemu_pci_device_list_main.c
  - meson.build
    - devemu_pci_common.h; devemu_pci_common.c
    - devemu_pci_type_config.h

PCI Device Hot-Plug
This sample illustrates how to create and hot-plug/hot-unplug an emulated device that has the generic type configured in
/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h.

The sample logic includes:

1. Initializing the generic PCIe type based on
   /opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h.

2. Acquiring the emulated device representor:
   - If the user did not provide VUID as input, then creating and using a new emulated device.
   - If the user provided VUID as an input, then searching for an existing emulated device with a matching VUID and using it.

3. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).

4. Registering to the PCIe device's hot-plug state change event.

5. Initializing hot-plug/hot-unplug of the device:
   1. If the user did not provide VUID as input, then initializing hot-plug flow of the device.
   2. If the user provided VUID as input, then initializing hot-unplug flow of the device.

6. Using the PE to poll for hot-plug state change event.

7. Waiting until hot-plug state transitions to expected state (power on or power off).

8. Cleaning up resources.
   - If hot-unplug was requested, then the emulated device is destroyed as well.
   - Otherwise, the emulated device persists.

References:
PCI Device Stateful Region

This sample illustrates how the host driver can write to a stateful region, and how the BlueField Arm can handle the write operation.

This sample consists of a host sample and BlueField sample. It is necessary to follow the additional sample setup detailed previously.

The BlueField sample logic includes:

1. Initializing the generic PCIe type based on
   /opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h.

2. Acquiring the emulated device representor that matches the provided VUID.

3. Creating a PCIe device context to manage the emulated device, and connecting it to a progress engine (PE).

4. For each stateful region configured in
   /opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h, registering to the PCIe device's stateful region write event.

5. Using the PE to poll for driver write to any of the stateful regions.

   - Every time the host driver writes to the stateful region, the handler is invoked and performs the following:
1. Queries the values of the stateful region that the host wrote to.

2. Logs the values of the stateful region.

   ○ The sample polls indefinitely until the user presses [Ctrl+c] to close the sample.

6. Cleaning up resources.

The host sample logic includes:

1. Initializing the VFIO device with a matching PCIe address and VFIO group.

2. Mapping the stateful memory region from the BAR to the process address space.

3. Writing the values provided as input to the beginning of the stateful region.

References:

- /opt/mellanox/doca/samples/doca_devemu/
  - devemu_pci_device_stateful_region/dpu/
    - devemu_pci_device_stateful_region_dpu_sample.c
    - devemu_pci_device_stateful_region_dpu_main.c
    - meson.build
  - devemu_pci_device_stateful_region/host/
    - devemu_pci_device_stateful_region_host_sample.c
    - devemu_pci_device_stateful_region_host_main.c
    - meson.build
  - devemu_pci_common.h; devemu_pci_common.c
  - devemu_pci_host_common.h; devemu_pci_host_common.c
  - devemu_pci_type_config.h
PCI Device DB

This sample illustrates how the host driver can ring the doorbell and how the BlueField can retrieve the doorbell value. The sample also demonstrates how to handle FLR.

This sample consists of a host sample and BlueField sample. It is necessary to follow the additional sample setup detailed previously.

The BlueField sample logic includes:

- Host (BlueField Arm) logic:
  1. Initializing the generic PCIe type based on
     `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`.
  2. Initializing DPA resources:
     1. Creating DPA instance, and associating it with the DPA application.
     2. Creating DPA thread and associating it with the DPA DB handler.
     3. Creating DB completion context and associating it with the DPA thread.
  3. Acquiring the emulated device representor that matches the provided VUID.
  4. Creating a PCIe device context to manage the emulated device, and connecting it to progress engine (PE).
  5. Registering to the context state changes event.
  6. Registering to the PCIe device FLR event.
  7. Using the PE to poll for any of the following:
     1. Every time the PCIe device context state transitions to running, the handler performs the following:
        1. Creates a DB object.
        2. Makes RPC to DPA, to initialize the DB object.
2. Every time the PCIe device context state transitions to stopping, the handler performs the following:

   1. Makes RPC to DPA, to un-initialize the DB object.
   
   2. Destroys the DB object.

3. Every time the host driver initializes or destroys the VFIO device, an FLR event is triggered. The FLR handler performs the following:

   1. Destroys DB object.
   
   2. Stops the PCIe device context.
   
   3. Starts the PCIe device context again.

4. The sample polls indefinitely until the user presses [Ctrl+c] to close the sample.

   ![Note]

   During this time, the DPA may start receiving DBs from the host.

8. Cleaning up resources.

- Device (BlueField DPA) logic:

  1. Initializing application RPC:

     1. Setting the global context to point to the DB completion context DPA handle.
     
     2. Binding DB to the doorbell completion context.

  2. Un-initializing application RPC:

     1. Unbinding DB from the doorbell completion context.
3. DB handler:
   1. Getting DB completion element from completion context.
   2. Getting DB handle from the DB completion element.
   3. Acknowledging the DB completion element.
   4. Requesting notification from DB completion context.
   5. Requesting notification from DB.
   6. Getting DB value from DB.

The host sample logic includes:
   1. Initializing the VFIO device with its matching PCIe address and VFIO group.
   2. Mapping the DB memory region from the BAR to the process address space.
   3. Writing the value provided as input to the DB region at the given offset.

References:
   - /opt/mellanox/doca/samples/doca_devemu/
     - devemu_pci_device_db/dpu/
       - host/
         - devemu_pci_device_db_dpu_sample.c
       - device/
         - devemu_pci_device_db_dpu_kernels_dev.c
         - devemu_pci_device_db_dpu_main.c
       - meson.build
     - devemu_pci_device_db/host/
       - devemu_pci_device_db_host_sample.c
PCI Device MSI-X

This sample illustrates how BlueField can raise an MSI-X vector, sending a signal towards the host, and shows how the host can retrieve this signal.

This sample consists of a host sample and a BlueField sample. It is necessary to follow the additional sample setup detailed previously.

The BlueField sample logic includes:

- Host (BlueField Arm) logic:
  1. Initializing the generic PCIe type based on /opt/mellanox/doca/samples/doca-devemu/devemu_pci_type_config.h.
  2. Initializing DPA resources:
     1. Creating a DPA instance and associating it with the DPA application.
     2. Creating a DPA thread and associating it with the DPA DB handler.
  3. Acquiring the emulated device representor that matches the provided VUID.
  4. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).
  5. Creating an MSI-X vector and acquiring its DPA handle.
  6. Sending an RPC to the DPA to raise the MSI-X vector.
  7. Cleaning up resources.
Device (BlueField DPA) logic:

1. Raising the MSI-X RPC by using the MSI-X vector handle.

The host sample logic includes:

1. Initializing the VFIO device with the matching PCIe address and VFIO group.

2. Mapping each MSI-X vector to a different FD.

3. Reading events from the FDs in a loop.

   1. Once the DPU raises MSI-X, the FD matching the MSI-X vector returns an event which is then printed to the screen.

   2. The sample polls the FDs indefinitely until the user presses [Ctrl+c] to close the sample.

References:

- /opt/mellanox/doca/samples/doca_devemu/
  - devemu_pci_device_msix/dpu/
    - host/
      - devemu_pci_device_msix_dpu_sample.c
    - device/
      - devemu_pci_device_msix_dpu_kernels_dev.c
    - devemu_pci_device_msix_dpu_main.c
    - meson.build
  - devemu_pci_device_msix/host/
    - devemu_pci_device_msix_host_sample.c
    - devemu_pci_device_msix_host_main.c
    - meson.build
PCI Device DMA

This sample illustrates how the host driver can set up memory for DMA, then the DPU can use that memory to copy a string from the BlueField to the host and from the host to the BlueField.

This sample consists of a host sample and a BlueField sample. It is necessary to follow the additional sample setup detailed previously.

The BlueField sample logic includes:

1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`.
2. Acquiring the emulated device representor that matches the provided VUID.
3. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).
4. Creating a DMA context to use for copying memory across the host and BlueField.
5. Setting up an mmap representing the host driver memory buffer.
6. Setting up an mmap representing a local memory buffer.
7. Use the DMA context to copy memory from host to BlueField.
8. Use the DMA context to copy memory from BlueField to host.
9. Cleaning up resources.

The host sample logic includes:

1. Initializing the VFIO device with the matching PCIe address and VFIO group.
2. Allocating memory buffer.

3. Mapping the memory buffer to I/O memory. The BlueField can now access the memory using the I/O address through DMA.

4. Copying the string provided by user to the memory buffer.

5. Waiting for the BlueField to write to the memory buffer.

6. Un-mapping the memory buffer.

7. Cleaning up resources.

References:

- /opt/mellanox/doca/samples/doca_devemu/
  - devemu_pci_device_dma/dpu/
    - devemu_pci_device_dma_dpu_sample.c
    - devemu_pci_device_dma_dpu_main.c
    - meson.build
  - devemu_pci_device_dma/host/
    - devemu_pci_device_dma_host_sample.c
    - devemu_pci_device_dma_host_main.c
    - meson.build

DOCA DevEmu Virtio

⚠️ Note
Introduction

DOCA DevEmu Virtio, which is part of the DOCA Device Emulation subsystem, introduces low-level software APIs that provide building blocks for developing and manipulating virtio devices using the device emulation capability of NVIDIA® BlueField®. This subsystem incorporates a core library that handles a common logic for various types of virtio devices, such as virtio-FS. One of its key responsibilities is managing the standard "device reset" procedure outlined in the virtio specification. This core library serves as a foundation for implementing shared functionalities across different virtio device types, ensuring consistency and efficiency in device operations and behaviors.

DOCA provides support for emulating virtio devices over the PCIe bus. The PCIe transport is commonly used for virtio devices. Configuration, discovery, and features related to PCIe (such as MSI-X and PCIe device hot plug/unplug) are managed through the DOCA DevEmu PCI APIs. This modular design enables each layer within the DOCA Device Emulation subsystem to manage its own business logic and facilitates seamless integration with the other layers, ensuring independent functionality and operation throughout the system.

This subsystem also includes device-specific libraries for various virtio device types (e.g., a library for a virtio-FS device).

From the host's perspective, there is no difference between para-virtual, DOCA-emulated, and actual hardware devices. The host uses the same virtio device drivers to operate the device under all circumstances.

Prerequisites

Virtio device emulation is part of the DOCA Device Emulation subsystem. It is, therefore, recommended to read the following guides beforehand:

- DOCA Device Emulation
- DOCA DevEmu PCI
Environment

DOCA DevEmu Virtio is supported on the BlueField target only.

The BlueField must meet the following requirements

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

Library must be run with root privileges.

Architecture

The DOCA DevEmu Virtio core library provides the following software abstractions:

- Virtio type – extends the PCIe type, represents common/default virtio configurations of emulated virtio devices
- Virtio device – extends the PCIe device, represents an instance of an emulated virtio device
- Virtio IO context – represents a progress context which is responsible for processing virtio descriptors and their associated virtio queues

DOCA DevEmu Virtio library does not provide APIs to configure the entire BAR layout of the virtio device as this configuration is done internally. However, the library offers APIs to configure some of the registers within the common configuration structure (see Virtio Device).

Virtio Common Configuration
According to the virtio specification, the common PCIe configuration structure layout is as follows:

```c
struct virtio_pci_common_cfg {
    /* About the whole device. */
    le32 device_feature_select; /* read-write */
    le32 device_feature; /* read-only for driver */
    le32 driver_feature_select; /* read-write */
    le32 driver_feature; /* read-write */
    le16 config_msix_vector; /* read-write */
    le16 num_queues; /* read-only for driver */
    u8 device_status; /* read-write */
    u8 config_generation; /* read-only for driver */

    /* About a specific virtqueue. */
    le16 queue_select; /* read-write */
    le16 queue_size; /* read-write */
    le16 queue_msix_vector; /* read-write */
    le16 queue_enable; /* read-write */
    le16 queue_notify_off; /* read-only for driver */
    le64 queue_desc; /* read-write */
    le64 queue_driver; /* read-write */
    le64 queue_device; /* read-write */
    le16 queue_notify_data; /* read-only for driver */
    le16 queue_reset; /* read-write */
};
```

The DOCA DevEmu Virtio core library provides the ability to configure some of the listed registers using the appropriate setters.

**Virtio Type**

The virtio type extends the PCIe type and describes the common/default configuration of emulated virtio devices, including the common virtio configuration space registers (such as num_queues, queue_size, and others).

Virtio type is currently read-only (i.e., only getter APIs are available to retrieve information). The following methods can be used for this purpose:
• `doca_devemu_virtio_type_get_num_queues` – for getting the default initial value of the `num_queues` register for the associated virtio devices

• `doca_devemu_virtio_type_get_queue_size` – for getting the default initial value of the `queue_size` register for the associated virtio devices

• `doca_devemu_virtio_type_get_device_features_63_0` – for getting the default initial values of the `device_feature` bits (0-63) for the associated virtio devices

• `doca_devemu_virtio_type_get_config_generation` – for getting the default initial value of the `config_generation` register for the associated virtio devices

The default virtio type is extended by a virtio device's specific type (e.g., virtio-FS type) and cannot be created on demand.

**Virtio Device**

The virtio device extends the PCIe device. Before using the DOCA DevEmu Virtio device, it is recommended to read the guidelines of [DOCA DevEmu PCI device](#) and [DOCA Core context configuration phase](#).

The virtio device is extended by a virtio-specific device (e.g., virtio FS device) and cannot be created on demand.

**Virtio Device Configurations**

The virtio device context can be configured to match the application use case and optimize the utilization of system resources.

**Mandatory Configurations**

The mandatory configurations are as follows:

• `doca_devemu_virtio_dev_set_num_required_running_virtio_io_ctxs` – to set the number of required running virtio IO contexts to be bound to the virtio device context. The virtio device context does not move to running state (according to the [DOCA Core context state machine](#)) before having this amount of running virtio IO contexts bound to it.
- `doca_devemu_virtio_dev_event_reset_register` – to register to the virtio device reset event. This configuration is mandatory.

**Optional Configurations**

The optional configurations are as follows:

- `doca_devemu_virtio_set_device_features_63_0` – to set the values of the `device_feature` bits (0-63). If not set, the default value is taken from the virtio type configuration.

- `doca_devemu_virtio_dev_set_num_queues` – to set the value of the `num_queues` register. If not set, the default value is taken from the virtio type configuration.

- `doca_devemu_virtio_dev_set_queue_size` – to set the value of the `queue_size` register for all virtio queues. If not set, the default value is taken from the virtio type configuration.

**Events**

DOCA DevEmu Virtio device exposes asynchronous events to notify about sudden changes, according to DOCA Core architecture.

- **Info**
  
  Common events are described in [DOCA DevEmu PCI Device events](#) and in [DOCA Core Event](#).

**Reset Event**

The reset event allows users to receive notifications whenever the device reset flow is initialized by the device driver. Upon receiving this event, it is guaranteed that no further requests are routed to the user via any associated virtio IO context until the reset flow is completed.
To complete the reset flow the user must:

1. Flush all outstanding requests back to the virtio IO context associated with the request.

2. Perform one of the following:
   - Call `doca_devemu_virtio_dev_reset_complete`.
   - Follow FLR flow:
     1. `doca_ctx_stop` – stop the virtio device with its associated virtio IO contexts and wait until the device and its associated virtio IO contexts transition to idle state
     2. `doca_ctx_start` – start the virtio device with its associated virtio IO contexts and wait until the device and its associated virtio IO contexts transition to running state

Now, the device and its associated virtio IO contexts should be fully operational again, the device is allowed to route new requests via any associated virtio IO context.

**Virtio IO**

The virtio IO context extends the DOCA Core context. Before using the DOCA DevEmu Virtio IO, it is recommended to read the guidelines of DOCA Core context configuration phase.

This context is associated with a single DOCA virtio device and is bound to the virtio device context upon start. The virtio IO context is a thread-unsafe object and is progressed by a single DOCA Core progress engine. Usually, users configure a single virtio IO context per BlueField core used by the application service.

The virtio IO context is responsible to route new incoming virtio requests towards the application and to complete handled requests back to the device driver. It can only route requests while in running state and when its associated virtio device is also in running state.
DOCA DevEmu Virtio-FS

**Note**

This library is supported at alpha level; backward compatibility is not guaranteed.

**Introduction**

The DOCA DevEmu Virtio-FS library is part of the DOCA DevEmu Virtio subsystem. It provides low-level software APIs that provide building blocks for developing and manipulating virtio filesystem devices using the device emulation capability of NVIDIA® BlueField® DPUs.

DOCA supports emulating virtio-FS devices over the PCIe bus. The PCIe transport is the common transport used for virtio devices. Configuration, discovery, and features related to PCIe (e.g., MSI-X and PCIe device hot plug/unplug) are managed through the DOCA DevEmu PCI APIs. Configuring common virtio registers and handling generic virtio logic (e.g., virtio device reset flow) is handled by the DOCA Virtio common library. This modular design enables each layer within the DOCA Device Emulation subsystem to manage its own business logic. It facilitates seamless integration with the other layers, ensuring independent functionality and operation throughout the system.

The DOCA Devemu Virtio-FS library efficiently handles virtio descriptors, carrying FUSE requests, sent by the device driver, and translating them into abstract virtio-FS requests which are then routed to the user. This translation process ensures that the underlying device-specific acceleration details are abstracted away, allowing applications to interact with abstracted virtio-FS requests.

Users of this library are responsible for developing a virtio-FS controller, which manages the underlying DOCA Devemu Virtio-FS device alongside an external backend file system which is outside DOCA’s scope. The controller application is designed to receive DOCA
Virtio-FS requests and process them according to virtio-FS and FUSE specifications, translating FUSE-based commands into the appropriate backend filesystem protocol.

**Prerequisites**

Virtio-FS device emulation is part of DOCA DevEmu Virtio subsystem. It is, therefore, recommended to read the following guides before proceeding:

- [DOCA Device Emulation](#)
- [DOCA DevEmu PCI](#)
- [DOCA DevEmu Virtio](#)

**Environment**

DOCA DevEmu Virtio-FS is supported on the BlueField target only. The BlueField must meet the following requirements:

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

ℹ️ **Info**

Please refer to the [DOCA Backward Compatibility Policy](#).

ℹ️ **Note**

Library must be run with root privileges.

Perform the following:
1. Configure BlueField to work in DPU mode as described in NVIDIA BlueField Modes of Operation.

2. Enable emulation by running the following on the host or DPU:

```
host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_FS_EMULATION_ENABLE=1
```

3. Configure the number of static virtio-FS physical functions and the number of MSIX for each physical function to expose. This can be done by running the following command on the DPU:

```
host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_FS_EMULATION_NUM_PF=2
VIRTIO_FS_EMULATION_NUM_MSIX=18
```

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

⚠️ **Note**

DOCA does not support hot plugging virtio-FS PF devices into the host PCIe subsystem or SR-IOV for virtio-FS devices.

**Architecture**

The DOCA DevEmu Virtio-FS library provides the following main software abstractions:

- The virtio-FS type – extends the virtio type; represents common/default virtio-FS configurations of emulated virtio-FS devices

- The virtio-FS device – extends the virtio device; represents an instance of an emulated virtio-FS device

- The virtio-FS IO context – extends the virtio IO context; represents a progress context responsible for processing virtio descriptors, carrying FUSE requests, and
their associated virtio queues (e.g., hiprio, request, admin, and notification queues).

- The virtio-FS request

**Virtio-FS Feature Bits**

According to the virtio specification, a virtio-FS device may report support for `VIRTIO_FS_F_NOTIFICATION` which indicates the ability to handle FUSE notify messages sent via the notification queue.

**Note**

Currently, DOCA does not support reporting the `VIRTIO_FS_F_NOTIFICATION` feature to the driver.

**Virtio-FS Configuration Layout**

According to the virtio specification, the virtio-FS configuration structure layout is as follows:

```c
struct virtio_fs_config {
    char tag[36];
    le32 num_request_queues;
    le32 notify_buf_size;
};
```

The `tag` and `num_request_queues` fields are always available. The `notify_buf_size` field is only available when `VIRTIO_FS_F_NOTIFICATION` is set.

**Note**
Virtio-FS Type

The virtio-FS type extends the virtio type and describes the common/default configuration of emulated virtio-FS devices, including some of the virtio-FS configuration space registers (e.g., `num_request_queues`).

Currently, the virtio-FS type is read-only (i.e., only getter APIs are available to retrieve information). The following method can be used for this purpose:

- `doca_devemu_vfs_type_get_num_request_queues` – to get the default initial value of the `num_request_queues` register for the associated virtio-FS devices

DOCA supports the default virtio-FS type. To retrieve the default virtio-FS type, users use the following method:

- `doca_devemu_vfs_is_default_vfs_type_supported` – check if the default DOCA Virtio-FS type is supported by the device. If supported:
  - `doca_dev_open` – open supported DOCA device
  - `doca_devemu_vfs_find_default_vfs_type_by_dev` – get the default DOCA Virtio-FS type associated with the device

Virtio-FS Device

The virtio-FS device extends the virtio device. Before using the DOCA DevEmu Virtio-FS device, it is recommended to read the guidelines of DOCA DevEmu Virtio device, DOCA DevEmu PCI device, and DOCA Core context configuration phase.

This section describes how to create, configure, and operate the virtio-FS device.
Virtio-FS Device Configurations

The virtio-FS emulated device might be in several different visibility levels from the host point of view:

- Visible/non-visible to the PCIe subsystem – If the device is visible to the PCIe subsystem, the user is not able to configure PCIe-related parameters (e.g., number of MSI-X vector, subsystem_id).

- Visible/non-visible to the virtio subsystem – If the device is visible to the virtio subsystem, the user is not be able to configure virtio-related parameters (e.g., number of queues, queue_size).

The flow for creating and configuring a virtio-FS device is as follows:

1. `doca_devemu_vfs_dev_create` – Create a new DOCA DevEmu Virtio-FS device instance.

2. `doca_devemu_vfs_dev_set_tag` – Set a unique tag for the device according to the virtio specification.

3. `doca_devemu_vfs_dev_set_num_request_queues` – Set the number of request queues for the device.

4. `doca_devemu_vfs_dev_set vfs_req_user_data_size` – Set the user data size of the virtio-FS request. If set, a buffer with this size is allocated for each DOCA DevEmu Virtio-FS on behalf of the user.

5. Configure virtio-related parameters as described in DOCA Virtio configurations.

6. Configure PCIe-related parameters as described in DOCA DevEmu PCI configurations.

**Note**

`doca_devemu_virtio_dev_set_num_queues` should be equal to the number of request queues +1 (for the hiprio queue) since DOCA does not currently support the virtio-FS notification queue.
7. `doca_ctx_start` – Start the virtio-FS device context to finalize the configuration phase.

   - The virtio-FS device object follows the DOCA context state machine as described in [DOCA Core context state machine](#).
   - The virtio-FS device context moves to running state after the initial number of virtio IO contexts is bound to it and turns to running state, as described at [DOCA DevEmu Virtio configurations](#).

At this point, the DOCA Devemu Virtio-FS context is fully operational.

**Mandatory Configurations**

The following are mandatory configurations:

- `doca_devemu_vfs_dev_set_tag` – set a unique tag for the device

**Optional Configurations**

The optional configurations are as follows:

- `doca_devemu_vfs_dev_set_num_request_queues` – set the number of request queues for the device. If not set, the default value is taken from the virtio-FS type configuration.

- `doca_devemu_vfs_dev_set_vfs_req_user_data_size` – set the user data size of the virtio-FS request. If not set, user data size defaults to 0.

**Virtio-FS Device Events**

DOCA DevEmu Virtio-FS device exposes asynchronous events to notify about changes that happen out of the blue, according to the DOCA Core architecture.

Common events are described in [DOCA DevEmu Virtio device events](#), [DOCA DevEmu PCI device events](#) and in [DOCA Core event](#).
Virtio-FS IO

The virtio-FS IO context extends the Virtio IO Context. To start using the DOCA DevEmu Virtio-FS IO it is recommended to read the guidelines of DOCA DevEmu Virtio IO and DOCA Core context configuration phase.

This section describes how to create, configure and operate the virtio-FS IO context.

Virtio-FS IO Configurations

The flow for creating and configuring a virtio-FS IO context should be as follows:

1. `doca_devemu_vfs_io_create` – Create a new DOCA DevEmu Virtio-FS IO instance.
2. `doca_devemu_vfs_io_event_vfs_req_notice_register` – Register event handler for incoming virtio-FS requests.
3. `doca_ctx_start` – Start the virtio-FS IO context to finalize the configuration phase. The virtio-FS IO object follows the DOCA Core context state machine. The virtio-FS device context moves to running state after the initial number of virtio-FS IO contexts is bound to it and moves to running state (as described at DOCA DevEmu Virtio configurations).

Mandatory Configurations

The following are mandatory configurations:

- `doca_devemu_vfs_io_event_vfs_req_notice_register` – Register event handler for incoming virtio-FS requests is mandatory

Virtio-FS Request

The virtio-FS request object serves as an abstraction for handling requests arriving on virtio-FS queues, including high-priority, request, or notification queues. These requests are initially generated by the device driver through created virtio queues and then routed to the user via a registered event handler, which is set up using `doca_devemu_vfs_io_event_vfs_req_notice_register`, on the associated virtio IO context. This event
handler, issued by the DOCA Virtio FS library, ensures that users can receive and process virtio-FS requests effectively within their application. Once the event handler is called, the ownership of the virtio-FS request and the associated request user data move to the user. The request ownership moves back to the associated virtio IO context once it is completed by the user by calling `doca_devemu_vfs_req_complete`.

The following APIs operate a virtio-FS request:

1. `doca_devemu_vfs_req_get_datain` – Get a DOCA buffer representing the data-in of the virtio-FS request. This DOCA buffer represents the host memory for the device-readable part of the request according to the virtio specification.

2. `doca_devemu_vfs_req_get_dataout` – Get a DOCA buffer representing the data-out of the virtio-FS request. This DOCA buffer represents the host memory for the device-writable part of the request according to the virtio specification.

3. `doca_devemu_vfs_req_complete` – Complete the virtio-FS request. The associated virtio-FS IO context completes the request toward the device driver according to the virtio-FS specification.

**Discovery**

Emulated virtio-FS PCIe functions are represented by a `doca_devinfo_rep` . To find the suitable `doca_devinfo_rep` that is used as the input parameter for `doca_devemu_vfs_dev_create`, users should first discover the existing device representors using the below:

1. `doca_devinfo_create_list` – Get a list of all DOCA devices.

2. `doca_devemu_vfs_is_default_vfs_type_supported` – Check whether the device can manage device associated to virtio-FS type.

3. If supported:
   1. `doca_dev_open` – Get an instance of the DOCA device that can be used as virtio-FS emulation manager.

   2. `doca_devemu_vfs_find_default_vfs_type_by_dev` – Get the default virtio-FS device type.

   3. `doca_devemu_vfs_type_as_pci_type` – Cast virtio-FS type to PCIe type.
4. doca_devemu_pci_type_rep_list_create – Create a list of all available representor devices for the virtio-FS type.

4. At this point, the user can choose the preferred representor device, open it using doca_dev_rep_open, and proceed with the flow described in section "Virtio-FS Device Configurations".

Initialization

This section describes the initialization flow of a DOCA DevEmu Virtio-FS device and one or more DOCA DevEmu Virtio-FS IO contexts (4 in this example). In this procedure, the user sets up and prepares the environment before starting to receive control path events (from the virtio-FS device context) and IO requests (from the virtio-FS IO contexts). During initialization, the user should configure various essential components to ensure correct behavior.

The user should perform the following:

1. Choose 4 Arm cores to run the application threads on.

2. Create 4 DOCA Core progress engine (PE) objects (pe1, pe2, pe3, pe4).

3. Find the suitable representor device according to the Discovery flow or any other method.

4. Create, configure, and start a new virtio-FS device according to the virtio-FS device configuration flow. Assume pe1 is associated with the virtio-FS device and doca_devemu_virtio_dev_set_num_required_running_virtio_io_ctxs is set to 4.

5. Create, configure, and start 4 new virtio-FS IO contexts according to the virtio-FS IO configuration flow. Assume pe1, pe2, pe3, and pe4 are associated with each of the 4 virtio-FS IO contexts respectively.

6. At this point, the 4 virtio-FS IO contexts transition to running state, followed by the virtio-FS device context transitioning to running state.

Note

During the initialization flow, it is guaranteed that no virtio/PCIe control path or IO path events are generated until the virtio-FS device
Teardown

This section describes the teardown flow of DOCA DevEmu Virtio-FS device and one or more DOCA DevEmu Virtio-FS IO contexts (4 in this example). In this procedure, the user cleans all the resources allocated in the initialization flow and all the outstanding events and requests.

The user should perform the following:

1. Start the teardown flow by calling `doca_ctx_stop`. This causes the DOCA Virtio-FS device context to transition to `stopping` state. It is guaranteed that no virtio/PCIe control path events is generated during this state.

2. Call `doca_ctx_stop` for any DOCA Virtio-FS IO context. This causes the DOCA Virtio-FS IO context to transition to `stopping` state. It is guaranteed that no IO path events are generated during this state.

3. Flush all outstanding virtio-FS requests to the associated virtio-FS IO contexts by calling `doca_devemu_vfs_req_complete`. Upon completing all the requests associated with a virtio-FS IO context, the DOCA Virtio-FS IO context transitions to `idle` state.

4. At this point, it is safe to destroy the virtio-FS IO context by calling `doca_devemu_vfs_io_destroy`. Destroying a virtio-FS IO context not in `idle` state will fail.

5. Once all 4 virtio-FS IO contexts associated with the virtio-FS device transition to `idle` state, the DOCA Virtio-FS device context transitions to `idle` state as well.

6. At this point, it is safe to destroy the virtio-FS device context by calling `doca_devemu_vfs_dev_destroy`. Destroying a virtio-FS device context not in `idle` state will fail.

Execution Phase

This section describes execution on BlueField Arm cores using several DOCA Core PE objects (one per core):

1. Choose 4 Arm cores to run the application threads on.
2. Create 4 DOCA Core PE objects. The application threads should periodically call `doxa_pe_progress` to advance all DOCA contexts associated with the PE.

3. Create, configure, and start the DOCA Virtio-FS device.

4. Create, configure, and start 4 DOCA Virtio-FS IO contexts.

The progress of DOCA Virtio-FS objects is illustrated by the following diagram:

**Control Path**

The DOCA Virtio-FS device context extends the DOCA Virtio device context (which extends the DOCA PCIe device context). This means that the DOCA Virtio-FS device control path is comprised by all the object it extends (i.e., DOCA Context, DOCA DevEmu PCI device, and DOCA DevEmu Virtio device).

The following events can be triggered by a virtio-FS device context:

- DOCA context state change events as described in DOCA Core context state machine and in DOCA DevEmu PCI state machine
- DOCA DevEmu PCI **FLR flow**
- DOCA DevEmu Virtio **reset flow**
The DOCA Virtio-FS IO context extends the DOCA Virtio IO context (which extends the DOCA core context). This means that the DOCA Virtio-FS IO context control path is comprised by all the object it extends (i.e., DOCA Context and DOCA DevEmu Virtio IO).

The following events can be triggered by a Virtio-FS IO context:

- DOCA context state change events as described in DOCA Core context state machine

In addition to the control path events, the DOCA DevEmu Virtio-FS IO context also produces IO path events as described in IO path.

## IO Path

This section describes the flow for a single virtio-FS request sent by the device driver until its completion.

It is assumed that the user properly configured an event handler for an incoming virtio-FS request as explained in section "Virtio-FS IO Configurations".

It is also assumed that the user is familiar with the virtio-FS specification and has the ability to perform DMA operations to/from the host using DOCA DMA or any other suitable method.

The DOCA virtio-FS flow is illustrated in the following diagram: