NVIDIA DOCA Core

Programming Guide

MLNX-15-060542 _v1.4 | September 2022

Table of Contents

Chapter 1. Introduction........ 1
Chapter 2. Prerequisites........ 2
Chapter 3. Architecture........ 3
3.1. DOCA Device........ 3
3.2. DOCA Buffer........ 4
3.3. DOCA Buffer Inventory........ 4
3.4. DOCA Memory Map........ 4
3.5. DOCA Context........ 5
3.6. DOCA Work Queue........ 5
3.7. DOCA Error........ 5
Chapter 4. API........ 6
4.1. doca_devinfo........ 6
4.1.1. doca_devinfo_list_create........ 7
4.1.2. doca_devinfo_list_destroy........ 7
4.1.3. doca_devinfo_property_get........ 7
4.2. doca_dev........ 7
4.2.1. doca_dev_open........ 8
4.2.2. doca_dev_close........ 8
4.2.3. doca_dev_as_devinfo........ 8
4.3. doca_devinfo_remote........ 8
4.3.1. doca_devinfo_remote_list_create........ 9
4.3.2. doca_devinfo_remote_list_destroy........ 9
4.3.3. doca_devinfo_remote_property_get........ 10
4.4. doca_dev_remote........ 10
4.4.1. doca_dev_remote_open........ 10
4.4.2. doca_dev_remote_close........ 11
4.4.3. doca_dev_as_devinfo........ 11
4.5. doca_buf........ 11
4.5.1. doca_buf_head_get........ 11
4.5.2. doca_buf_len_get........ 11
4.5.3. doca_buf_refcount_rm........ 12
4.5.4. doca_buf_refcount_get........ 12
4.6. doca_buf_inventory........ 12
4.6.1. doca_buf_inventory_create........ 13
4.6.2. doca_buf_inventory_destroy........ 13

4.6.3. doca_buf_inventory_property_set........13
4.6.4. doca_buf_inventory_property_get........13
4.6.5. doca_buf_inventory_start........14
4.6.6. doca_buf_inventory_stop........14
4.6.7. doca_buf_inventory_buf_by_addr........14
4.6.8. doca_buf_inventory_buf_dup........14
4.7. doca_mmap........15
4.7.1. doca_mmap_create........16
4.7.2. doca_mmap_destroy........16
4.7.3. doca_mmap_property_set........16
4.7.4. doca_mmap_property_get........17
4.7.5. doca_mmap_start........17
4.7.6. doca_mmap_stop........17
4.7.7. doca_mmap_dev_add........17
4.7.8. doca_mmap_dev_rm........17
4.7.9. doca_mmap_populate........18
4.7.10. doca_mmap_export........18
4.7.11. doca_mmap_create_from_export........19
4.8. doca_ctx........19
4.8.1. doca_ctx_dev_add........20
4.8.2. doca_ctx_dev_rm........20
4.8.3. doca_ctx_start........21
4.8.4. doca_ctx_stop........21
4.8.5. doca_ctx_workq_add........21
4.8.6. doca_ctx_workq_rm........22
4.9. doca_workq........22
4.9.1. doca_workq_create........22
4.9.2. doca_workq_destroy........23
4.9.3. doca_workq_property_set........23
4.9.4. doca_workq_property_get........23
4.9.5. doca_workq_submit........23
4.9.6. doca_workq_progress_retrieve........24
Chapter 5. Object Life Cycle........25
5.1. Device Life Cycle........25
5.2. Buffer Life Cycle........25
5.3. Buffer Inventory Life Cycle........26
5.4. Memory Map Life Cycle........26
5.5. Context Map Life Cycle........27

5.6. Work Queue Life Cycle.......... 27

Chapter 6. DOCA Workflow.......... 29

6.1. Exporting and Recreating Memory Map Object from Export.......... 29

6.2. Region-based Memory Access.......... 29

6.3. Randomly Accessed Memory.......... 30

6.3.1. Single mmap Option.......... 30

6.3.2. Mmap per Buffer Option.......... 31

Chapter 1. Introduction

The DOCA core objects provide a unified interface to DOCA libraries, allowing for a standardized flow for applications to build upon. They consist of the following:

- DOCA Device (`doca_dev`)
- DOCA Buffer (`doca_buf`)
- DOCA Buffer Inventory (`doca_buf_inventory`)
- DOCA Memory Map (`doca_mmap`)
- DOCA Context (`doca_ctx`)
- DOCA Work Queue (`doca_workq`)

Chapter 2. Prerequisites

DOCA Core objects are supported on the NVIDIA® BlueField® DPU target and the host machine. Both must meet the following prerequisites:

- DOCA version 1.3 (BlueField software 3.9.0)
- FW version 24.33.0100 or greater
- OFED version 5.5 or greater

Chapter 3. Architecture

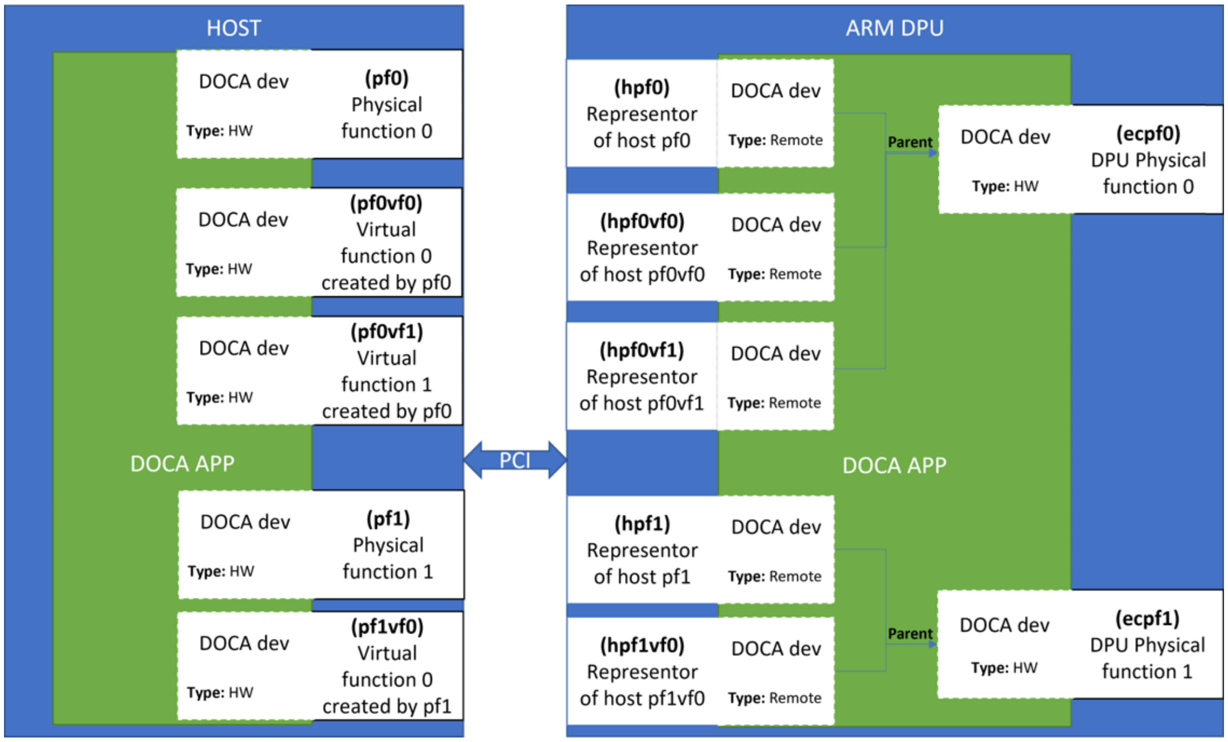
Each of the following DOCA core building blocks has its own structure and API. All DOCA core objects interact with one another to offer easy access to and management of BlueField capabilities.

3.1. DOCA Device

DOCA Device represents an available processing unit backed by hardware or software implementation.

DOCA Device supports two forms: The hardware device and the remote device.

Using DOCA Device, you may easily locate all the available local devices accessible by your system and all the remote devices accessible by a given local device. Each device found can be easily initiated using DOCA Device.


HOST
DOCA dev Type: HW (pf0) Physical function 0
DOCA dev Type: HW (pf0vf0) Virtual function 0 created by pf0
DOCA dev Type: HW (pf0vf1) Virtual function 1 created by pf0
DOCA APP
DOCA dev Type: HW (pf1) Physical function 1
DOCA dev Type: HW (pf1vf0) Virtual function 0 created by pf1
PCI
ARM DPU
(hpf0) Representor of host pf0 DOCA dev Type: Remote
(hpf0vf0) Representor of host pf0vf0 DOCA dev Type: Remote
(hpf0vf1) Representor of host pf0vf1 DOCA dev Type: Remote
Parent
DOCA dev Type: HW (ecpf0) DPU Physical function 0
DOCA APP
(hpf1) Representor of host pf1 DOCA dev Type: Remote
(hpf1vf0) Representor of host pf1vf0 DOCA dev Type: Remote
Parent
DOCA dev Type: HW (ecpf1) DPU Physical function 1

3.2. DOCA Buffer

DOCA Buffer is used for reference data. It holds the information on a memory region that belongs to a DOCA memory map, and its descriptor is allocated from DOCA Buffer Inventory. Among other functions, it can be used to perform DMA operations.

3.3. DOCA Buffer Inventory

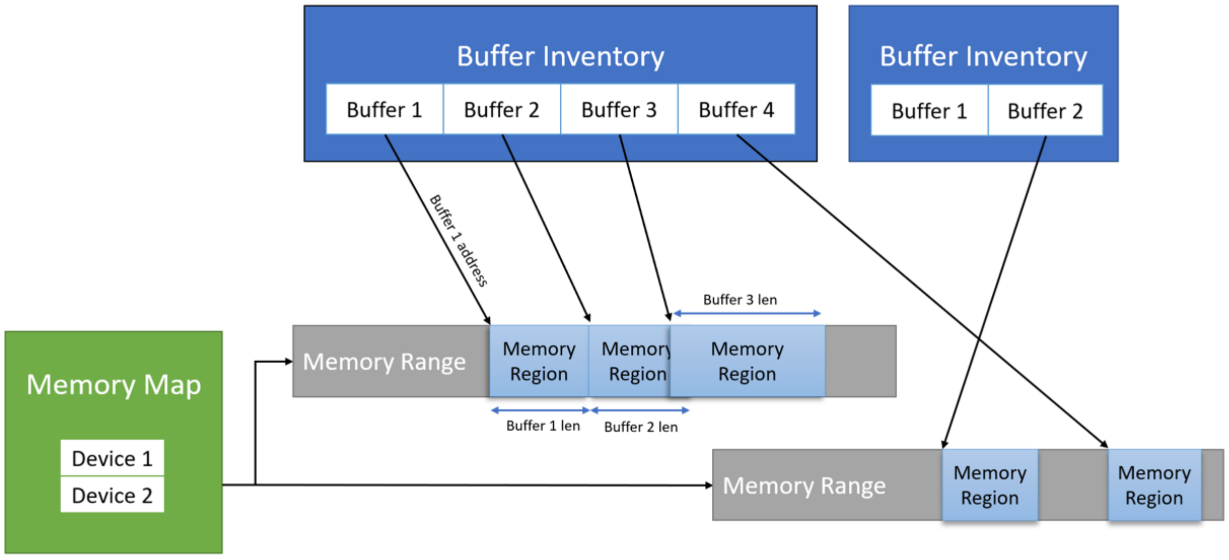
DOCA Buffer Inventory manages a pool of `doca_buf` objects. Each buffer obtained from an inventory is a descriptor that points to a memory region from a doca_mmap memory range of the user's choice.

3.4. DOCA Memory Map

DOCA Memory Map provides a centralized repository and orchestration of registration of several memory ranges for each device attached to the memory map.

Each memory map can represent memory from a single address space (e.g., host memory, DPU memory, etc).

DOCA Memory Map can be exported to allow accessibility of other applications to the memory ranges registered with the memory map. This allows selective sharing of specific memory ranges between applications on a single domain or across domains (e.g., between the DPU and the host).



3.5. DOCA Context

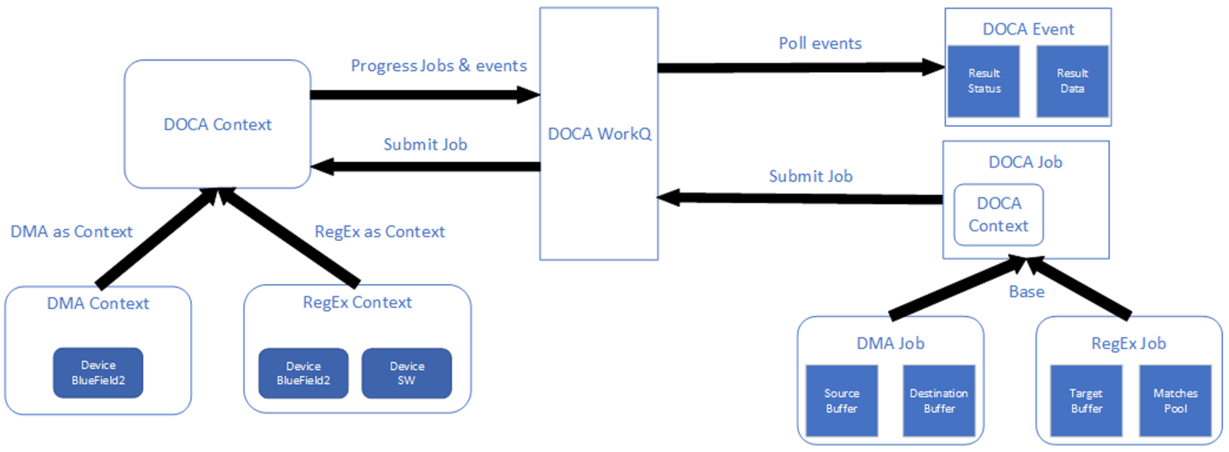
DOCA CTX is the base class of every data-path library in DOCA. It is a specific library/SDK instance object providing abstract data processing functionality. The library exposes events and/or jobs that manipulate data.

3.6. DOCA Work Queue

DOCA WorkQ provides progress engine service for DOCA CTXs. Once a DOCA WorkQ is added to a DOCA CTX, it can be used to submit jobs defined by the CTX and to progress and retrieve events from the CTX. The WorkQ is a thread-unsafe object and is considered the per-thread interface for communicating with CTXs.

3.7. DOCA Error

Provides information regarding different errors caused while using the DOCA core libraries.



Chapter 4. API

Refer to NVIDIA DOCA Libraries API Reference Manual, for more detailed information on the DOCA Core API.

The following sections provide additional details about the library API.

4.1. doca_devinfo

Note: All `doca_devinfo` operations are considered thread-safe.

The device information structure holds information about a device.

Devices are distinguished by their properties and type. Each device exposes a devinfo structure for querying properties and capabilities.

Available device properties:

- Vendor unique identifier (VUID) – a unique string representation of a PCIe function. It is stable across reboots and is constant per device. Read-only property.
- PCIe function address – returned as Bus Device Function format. Read-only property.
- IPv4 address of the underlying network interface. Read-only property.
- IPv6 address of the underlying network interface. Read-only property.
- Network interface name. Read-only property.
- IB device name. Read-only property.

A device can be one of the following types:

- HW device – a PCIe function providing hardware capabilities
- SW device – a virtual device providing software capabilities

To obtain a device information structure, you can use the `doca_devinfo_list_create()` function. Each device information structure can then be queried using `doca_devinfo_property_get()` while providing the devinfo.

Once a `doca_devinfo` is chosen, it can be used to obtain a `doca_dev` by opening one using `doca_dev_open()`.

Once a device has been opened, it can be passed to DOCA Contexts for resource management.

4.1.1. doca_devinfo_list_create

This operation creates a list of available devices on the machine such that each device is represented by a device information structure.

The list must be destroyed after opening the desired devices using doca_devinfo_list_destroy(). This only releases unopened devices.

```
doca_error_t doca_devinfo_list_create(struct doca_devinfo ***devinfo_list);
```

Where:

- `devinfo_list` – represents a pointer to the list of available device information. It is enough to allocate a `struct doca_devinfo **<variable>` and provide a pointer to it. The variable can then be used to iterate over the list as an array of pointers where the variable `[index]` can be used as the device information structure.

4.1.2. doca_devinfo_list_destroy

This operation destroys the list of device information obtained from `doca_devinfo_list_create()`.

The device information list must be destroyed after opening the desired devices such that open devices are not destroyed.

```
doca_error_t doca_devinfo_list_destroy(struct doca_devinfo **devinfo_list);
```

Where:

- `devinfo_list` – a list of device information structures obtained from `doca_devinfo_list_create()`.

4.1.3. doca_devinfo_property_get

This operation gets the up-to-date value of a DOCA Device property.

This can be used to query DOCA Device properties.

```
doca_error_t doca_devinfo_property_get(const struct doca_devinfo *devinfo, enum
 doca_devinfo_property property, void *value, uint32_t size);
```

Where:

- `devinfo` – DOCA Device information structure
- `property` – the requested property to get. See `enum doca_devinfo_property`.
- `value` – the value of the property (output parameter)
- `devinfo` – the size of the property in bytes

4.2. doca_dev

Note: All `doca_dev` operations are considered thread-safe.

The device structure holds active resources for use by different DOCA Contexts.

Each device can be represented as a doca_devinfo structure. To obtain a device information structure out of an open device, use `doca_dev_as_devinfo()`.

The device manages resources that can be shared by different DOCA Contexts, while the devinfo structure is used for querying device properties.

The device can be passed to multiple contexts using `doca_ctx_dev_add()` while providing an open device.

4.2.1. doca_dev_open

This operation creates a device based on the device information.

Devices are a fundamental resource for each DOCA Context. After a device is opened, it can be passed to different DOCA Contexts.

Devices should be closed once they are no longer needed.

```
doca_error_t doca_dev_open(struct doca_devinfo *devinfo, struct doca_dev **dev);
```

Where:

- `devinfo` – DOCA Device information structure with desired properties
- `dev` – holds a DOCA device based on provided devinfo (output parameter)

4.2.2. doca_dev_close

This operation closes a device.

Devices should be closed once they are no longer needed.

```
doca_error_t doca_dev_close(struct doca_dev *dev);
```

Where:

- `dev` – the DOCA Device to close

4.2.3. doca_dev_as_devinfo

This operation returns the DOCA Device information structure.

The returned information holds the same properties used to open the device.

```
doca_error_t doca_dev_as_devinfo(struct doca_dev *dev, struct doca_devinfo
 **devinfo);
```

Where:

- `dev` – the DOCA Device
- `devinfo` – holds returned DOCA Device information (output parameter)

4.3. doca_devinfo_remote

Note: All `doca_devinfo_remote` operations are considered thread-safe.

The remote device information structure holds information about remote devices.

Remote devices are distinguished by their properties and type. Each device exposes a remote devinfo structure for querying properties and capabilities.

Remote devices can be obtained while running on the DPU only.

A remote device can only be a network device. That is, a representor of a network function managed by the host.

To obtain a remote device's information structure, use the `doca_devinfo_remote_list_create()` while providing a local device. Each remote device information structure can then be queried using `doca_devinfo_remote_property_get()` while providing the remote devinfo.

Once a `doca_devinfo_remote` is chosen, it can be used to obtain a `doca_dev_remote` by opening one using `doca_dev_remote_open()`.

Once a remote device is opened, it can be used to refer to a device on the host machine.

4.3.1. doca_devinfo_remote_list_create

This operation creates a list of available devices on the machine such that each device is represented by a remote device information structure.

After opening the desired devices, the list should be destroyed using `doca_devinfo_remote_list_destroy()`. This only releases unopened devices.

This method can only be used on the DPU, where not all `doca_dev`s have access to the devices on the host.

```
doca_error_t doca_devinfo_list_create(struct doca_dev *dev, struct
 doca_devinfo_remote ***devinfo_remote_list);
```

Where:

- `dev` – represents a local device. The returned list holds information about devices on the host that are visible from this local device.
- `devinfo_remote_list` – represents a pointer to the list of available remote device information. It is enough to allocate a `struct doca_devinfo_remote **variable` and provide a pointer to it. The variable can then be used to iterate over the list as an array of pointers, where the variable `[index]` can be used as the device information structure.

Note: `dev` can be closed after the remote list is created.

4.3.2. doca_devinfo_remote_list_destroy

This operation destroys the list of remote device information obtained from `doca_devinfo_remote_list_create()`.

The list of remote device information should be destroyed after opening the desired remote devices such that remote devices that are open are not destroyed.

```
doca_error_t doca_devinfo_remote_list_destroy(struct doca_devinfo_remote
 **devinfo_remote_list);
```

Where:

- `devinfo_remote_list` – a list of remote device information structures obtained from `doca_devinfo_remote_list_create()`

4.3.3. doca_devinfo_remote_property_get

This operation gets the up-to-date value of a DOCA Device property.

This can be used to query the remote DOCA Device's properties.

```
doca_error_t doca_devinfo_remote_property_get(const struct doca_devinfo_remote
 *devinfo_remote, enum doca_devinfo_remote_property property, void *value, uint32_t
 size);
```

Where:

- `devinfo_remote` – the DOCA remote device information structure
- `property` – the property to get. See `enum doca_devinfo_remote_property`.
- `value` – the value of the property (output parameter)
- `size` – the size of the property in bytes

4.4. doca_dev_remote

Note: All `doca_devinfo_remote` operations are considered thread-safe.

The remote device structure holds active resources for use by different DOCA libraries.

Each remote device can be represented as a `doca_devinfo_remote` structure.
To obtain a remote device information structure from an open remote device, use `doca_dev_remote_as_devinfo()`.

The remote device manages resources that can be shared by different DOCA libraries, while the `devinfo` remote structure is used for querying remote device properties.

4.4.1. doca_dev_remote_open

This operation creates a remote device based on the remote device's information.

Opening a remote device activates its resources and prepares it for use by different libraries in DOCA.

Devices should be closed once they are no longer needed.

```
doca_error_t doca_dev_remote_open(struct doca_devinfo_remote *devinfo_remote, struct
 doca_dev_remote **dev_remote);
```

Where:

- `devinfo_remote` – the DOCA remote device information structure with desired properties
- `dev_remote` – holds a DOCA remote device based on provided `devinfo` (output parameter)

4.4.2. doca_dev_remote_close

This operation closes a remote device.

Devices should be closed once they are no longer needed.

```
doca_error_t doca_dev_remote_close(struct doca_dev_remote *dev_remote);
```

Where:

- `dev_remote` – the DOCA remote device to close

4.4.3. doca_dev_as_devinfo

This operation returns the remote device information structure of the device.

The returned remote device information holds the same properties as the one used to open this device.

```
doca_error_t doca_dev_remote_as_devinfo(struct doca_dev_remote *dev_remote, struct
 doca_devinfo_remote **devinfo_remote);
```

Where:

- `dev_remote` – the DOCA remote device
- `dev_remote` – holds the returned DOCA remote device information (output parameter)

4.5. doca_buf

Note: All `doca_buf` operations are considered thread-unsafe.

The DOCA Buffer structure is a descriptor that holds information on a memory region, defined by the address in which the memory region begins, and the length of the memory region in bytes. The address and length of the memory region are set when the buffer is created. They then become read-only.

4.5.1. doca_buf_head_get

This operation returns the address of the memory region that the buffer points to.

```
doca_error_t doca_buf_head_get(struct doca_buf *buf, void **head);
```

Where:

- `buf` – the DOCA Buffer structure
- `head` – the address of the memory region that the buffer points to (output parameter)

4.5.2. doca_buf_len_get

This operation returns the size of the memory region pointed by the buffer in bytes.

```
doca_error_t doca_buf_len_get(struct doca_buf *buf, size_t *len);
```

Where:

- `buf` – the DOCA Buffer structure
- `len` – the length of the memory region pointed by the buffer (output parameter)

4.5.3. doca_buf_refcount_rm

This operation decreases `doca_buf` reference count. Once it reaches zero, the function destroys the `doca_buf` object.

```
doca_error_t doca_buf_refcount_rm(struct doca_buf *buf, uint16_t *refcount);
```

Where:

- `buf` – the DOCA Buffer structure
- `refcount` – the `doca_buf` reference count value before this operation took place (output parameter)

4.5.4. doca_buf_refcount_get

This operation returns `doca_buf` reference count value.

```
doca_error_t doca_buf_refcount_get(struct doca_buf *buf, uint16_t *refcount);
```

Where:

- `buf` – the DOCA Buffer structure
- `refcount` – the `doca_buf` reference count value (output parameter)

4.6. doca_buf_inventory

Note: All `doca_buf_inventory` operations are considered thread-unsafe.

The DOCA buffer inventory structure manages a pool of `doca_buf` objects. After creating `doca_buf_inventory`, the user must start the buffer inventory using `doca_buf_inventory_start()` function.

When creating `doca_buf_inventory`, the returned object can be manipulated with `doca_buf_inventory_property_set()` API. Once all required attributes are set, it should be reconfigured and adjusted to match the settings in `doca_buf_inventory_start()`.

The following operations become possible only after start:

- Retrieval of free elements from the inventory using `doca_buf_inventory_buf_by_addr()`
- Duplicating the content of a buffer into a buffer allocated from the inventory using `doca_buf_inventory_buf_dup()`

Setting the properties of the inventory using `doca_buf_inventory_property_set()` is not possible after first starting the buffer inventory object.

Each buffer allocated by `doca_buf_inventory_buf_by_addr()` points to a memory region of the users' choice.

Un-started/stopped buffer inventory rejects all attempts to allocate new DOCA Buffers, regardless of the number of free elements.

4.6.1. doca_buf_inventory_create

This operation allocates DOCA buffer inventory with the given attributes.

```
doca_error_t doca_buf_inventory_create(const char *name, size_t num_elements,
 uint32_t extensions, struct doca_buf_inventory **buf_inventory);
```

Where:

- `name` – name of created DOCA buffer inventory
- `num_elements` – number of elements in the inventory
- `num_elements` – bitmap of extensions enabled for the inventory described in `doca_buf.h`
- `num_elements` – DOCA buffer inventory on success (output parameter)

4.6.2. doca_buf_inventory_destroy

This operation destroys the DOCA buffer inventory structure and frees its memory. Before calling `doca_buf_inventory_destroy` all allocated buffers must be returned to the inventory.

```
doca_error_t doca_buf_inventory_destroy(struct doca_buf_inventory *inventory);
```

Where:

- `inventory` – DOCA buffer inventory to destroy

4.6.3. doca_buf_inventory_property_set

This operation sets the value of a DOCA buffer inventory property.

```
doca_error_t doca_buf_inventory_property_set(struct doca_buf_inventory
 *inventory, enum doca_buf_inventory_property property, const void *value, uint32_t
 size);
```

Where:

- `inventory` – DOCA buffer inventory structure
- `property` – the requested property to set. See `enum doca_buf_inventory_property`.
- `value` – the new value of the property
- `size` – the size of the property in bytes

Note: All DOCA buffer inventory properties available at this stage are read-only properties.

4.6.4. doca_buf_inventory_property_get

This operation gets the up-to-date value of a DOCA buffer inventory property.

```
doca_error_t doca_buf_inventory_property_get(struct doca_buf_inventory
 *inventory, enum doca_buf_inventory_property property, void *value, uint32_t size);
```

Where:

- `inventory` – DOCA buffer inventory structure
- `property` – the requested property to get. See `enum doca_buf_inventory_property`.
- `value` – the value of the property (output parameter)
- `size` – the size of the property in bytes

4.6.5. doca_buf_inventory_start

This operation starts DOCA buffer inventory.

```
doca_error_t doca_buf_inventory_start(struct doca_buf_inventory *inventory);
```

Where:

- `inventory` – DOCA buffer inventory to start

4.6.6. doca_buf_inventory_stop

This operation stops DOCA buffer inventory.

```
doca_error_t doca_buf_inventory_stop(struct doca_buf_inventory *inventory);
```

Where:

- `inventory` – DOCA buffer inventory to stop

4.6.7. doca_buf_inventory_buf_by_addr

This operation takes a single DOCA buffer from the DOCA buffer inventory and points it to a memory region from the DOCA memory map using the given `addr` and `len` arguments.

The given address and length must be contained in one of the memory ranges of the DOCA memory map. The operation fails if there is no such suitable memory range.

```
doca_error_t doca_buf_inventory_buf_by_addr(struct doca_buf_inventory
 *inventory, struct doca_mmap *mmap, void *addr, size_t len, struct doca_buf **buf);
```

Where:

- `inventory` – DOCA buffer inventory structure
- `mmap` – DOCA memory map structure
- `addr` – the address of the memory region for the new `doca_buf` to point to
- `len` – the length of the memory region for the new `doca_buf` to point to in bytes
- `buf` – DOCA buffer object (output parameter)

4.6.8. doca_buf_inventory_buf_dup

This operation duplicates the source buffer by taking a new destination buffer from the given DOCA buffer inventory and copying the source buffer content to the destination buffer.

```
doca_error_t doca_buf_inventory_buf_dup(struct doca_buf_inventory
 *inventory, const struct doca_buf *src_buf, struct doca_buf **dst_buf);
```

Where:

- `inventory` – DOCA buffer inventory structure
- `src_buf` – the buffer to duplicate
- `dst_buf` – the duplicated DOCA buffer (output parameter)

4.7. doca_mmap

Note: All `doca_mmap` operations are considered thread-unsafe.

The DOCA memory map (mmap) structure points to a collection of memory ranges from a single address space; either the host or DPU memory.

Definitions:

- Memory range – virtually contiguous fracture of memory space defined by address and length
- Chunk – local system memory range
- Remote chunk – remote system memory range

Each DOCA memory map has defined properties:

- DOCA memory map name set on creation. Read-only property.
- The maximum number of chunks that can be populated. The default value is 1 if not set by the user before `doca_mmap_start`. Once this limit is reached, further chunk populations fail. If `mmap` is from an export, then the number of remote chunks is returned.
- The maximum number of devices that can be added to the `doca_mmap`. The default value is 1 if not set by the user before `doca_mmap_start`. Once this limit is reached, further device additions fail.
- The number of DOCA buffers pointing to memory ranges in the `doca_mmap`. Read-only property.
- Exported flag. Set to true if the `doca_mmap` is exported, false otherwise. Read-only property.
- From export flag. Set to true if the `doca_mmap` has been created from an export, false otherwise. Read-only property.
- Access flags. The `doca_mmap` access flags. See `enum doca_mmap_access_flags` for more.

The DOCA memory map object can be manipulated with `doca_mmap_property_set()` API. Once all required DOCA memory map attributes are set, it should be reconfigured and adjusted to match the settings in `doca_mmap_start()`.

The following operations become possible only after start:

- Adding a device to `doca_mmap` using `doca_mmap_dev_add()`
- Removing a device from `doca_mmap` using `doca_mmap_dev_rm()`
- Adding a memory range to the `doca_mmap` using `doca_mmap_populate()`

- Exporting the `doca_mmap` using `doca_mmap_export()`
- Mapping `doca_buf` structures to the memory ranges using `doca_buf_inventory_buf_by_addr()` or `doca_buf_inventory_buf_dup()`

Setting the properties of `doca_mmap` using `doca_mmap_property_set()` is no longer possible after starting the memory map (mmap) object.

The following are not possible on exported `doca_mmap` or on `doca_mmap` that has been created from export:

- Setting the properties of the `doca_mmap` using `doca_mmap_property_set()`
- Adding a device to the `doca_mmap` using `doca_mmap_dev_add()`
- Removing a device to the `doca_mmap` using `doca_mmap_dev_rm()`
- Adding a memory range to the `doca_mmap` using `doca_mmap_populate()`
- Exporting the `doca_mmap` using `doca_mmap_export()`

4.7.1. doca_mmap_create

This operation allocates zero-size memory map object with default/unset attributes.

```
doca_error_t doca_mmap_create(const char *name, struct doca_mmap **mmap);
```

Where:

- `name` – name of created DOCA memory map
- `mmap` – DOCA memory map structure with default/unset attributes on success

4.7.2. doca_mmap_destroy

This operation destroys DOCA memory map structure and frees its memory. Before calling `doca_mmap_destroy`, all allocated buffers must be returned to the `doca_mmap`.

```
doca_error_t doca_mmap_destroy(struct doca_mmap *mmap);
```

Where:

- `mmap` – DOCA memory map to destroy

4.7.3. doca_mmap_property_set

This operation sets the value of a DOCA memory map property.

```
doca_error_t doca_mmap_property_set(struct doca_mmap *mmap, enum doca_mmap_property
 property, const void *value, uint32_t size);
```

Where:

- `mmap` – DOCA memory map structure
- `property` – the requested property to set. See `enum doca_mmap_property`.
- `value` – the new value of the property
- `size` – the size of the property in bytes

4.7.4. doca_mmap_property_get

This operation gets the up-to-date value of a DOCA memory map property.

```
doca_error_t doca_mmap_property_get(struct doca_mmap *mmap, enum doca_mmap_property
 property, void *value, uint32_t size);
```

Where:

- `mmap` – DOCA memory map structure
- `property` – the requested property to get. See `enum doca_mmap_property`.
- `value` – the value of the property (output parameter)
- `size` – the size of the property in bytes

4.7.5. doca_mmap_start

This operation starts the DOCA memory map.

```
doca_error_t doca_mmap_start(struct doca_mmap *mmap);
```

Where:

- `mmap` – DOCA memory map to start

4.7.6. doca_mmap_stop

This operation stops the DOCA memory map.

```
doca_error_t doca_mmap_stop(struct doca_mmap *mmap);
```

Where:

- `mmap` – DOCA memory map to stop

4.7.7. doca_mmap_dev_add

This operation registers DOCA memory map on the given device.

If the `doca_mmap` has been populated before this operation took place, the function performs memory registration of the new device added with each of the memory ranges attached to the `doca_mmap`.

```
doca_error_t doca_mmap_dev_add(struct doca_mmap *mmap, struct doca_dev *dev);
```

Where:

- `mmap` – DOCA memory map structure
- `dev` – the DOCA device object that should be registered to the `doca_mmap`

4.7.8. doca_mmap_dev_rm

This operation deregisters the given device from the DOCA memory map.

If the doca_mmap contains memory ranges, the function performs memory deregistration of the given device to each of them.

```
doca_error_t doca_mmap_dev_rm(struct doca_mmap *mmap, struct doca_dev *dev);
```

Where:

- mmap – DOCA memory map structure
- dev – the DOCA device object that should be deregistered from the doca_mmap. The device must be a device previously added to the memory map via doca_mmap_dev_add().

4.7.9. doca_mmap_populate

This operation adds memory range to a local DOCA memory map.

If there are devices attached to the doca_mmap, the function performs memory registration of the new memory range added with each of those devices.

```
doca_error_t doca_mmap_populate(struct doca_mmap *mmap, void *addr, size_t len,
 size_t pg_sz, doca_mmap_memrange_free_cb_t *free_cb, void *opaque);
```

Where:

- mmap – DOCA memory map structure
- addr – the address in which the memory range begins
- len – the length of the memory range in bytes
- pg_sz – page size alignment of the provided memory range. Must be ≥4096 and a power of 2.
- free_cb – callback function to free the populated memory range on doca_mmap_destroy()
- opaque – opaque value to be passed to free_cb once called

4.7.10. doca_mmap_export

This operation composes a DOCA memory map representation as a memory export descriptor for later import with doca_mmap_create_from_export() for one of the previously added devices to the memory map.

This function allows access to specific host memory ranges registered to the mmap from the DPU.

Once this function is called on an object, it is considered exported.

Freeing the memory buffer marked with *export is the caller's responsibility.

This operation is not permitted for:

- Un-started/stopped memory map object
- Memory map object that has been exported or created from export

```
doca_error_t doca_mmap_export(struct doca_mmap *mmap, const struct doca_dev
 *dev, void **export_desc, size_t *export_desc_len);
```

Where:

- mmap – DOCA memory map structure

- `dev` – DOCA device previously added to the memory map via `doca_mmap_dev_add()`
- `export_desc` – on success, return should have a pointer to the allocated export descriptor representing the memory map object for the device provided as `dev` (output parameter)
- `export_desc_len` – length in bytes of the `export_desc` parameter (output parameter)

Note: Only a `doca_mmap` consisting of a single chunk is supported.

4.7.11. doca_mmap_create_from_export

This operation creates a DOCA memory map object representing memory ranges in remote system memory space.

Note: The created object is not backed by local memory.

This function allows access to specific host memory ranges registered to the `mmap` from the DPU.

Once this function is called on an object, it is considered as `from_export`.

```
doca_error_t doca_mmap_create_from_export(const char *name, const void *export_desc,
 size_t export_desc_len, struct doca_dev *dev, struct doca_mmap **mmap);
```

Where:

- `name` – name of newly created DOCA memory map
- `export_desc` – the export descriptor generated by doca_mmap_export
- `export_desc_len` – length in bytes of the `export_desc` parameter
- `dev` – a local device connected to the device that resides in the exported mmap
- `mmap` – DOCA memory map granting access to remote memory (output parameter)

Note: Only a `doca_mmap` consisting of a single chunk is supported.

Note: The name given by the user does not play a role, implementation-wise.

4.8. doca_ctx

Note: All `doca_ctx` operations are considered thread-safe.

The DOCA Context structure holds configurations for the execution of data-processing jobs. DOCA libraries that want to expose data-path jobs can implement the DOCA Context interface.

DOCA CTX provides a common interface for configuring and starting said libraries.

Devices can be provided to libraries using `doca_ctx_dev_add()`. Other configurations can be provided using a library-specific API. Once all configurations are provided, the library can be started using `doca_ctx_start()`. Once a library has been created, it starts accepting one or more DOCA WorkQ through `doca_ctx_workq_add()`. The WorkQ can then be used for submitting jobs, progressing jobs, and polling events.

The following operations become possible only after starting the DOCA Context:

- Adding WorkQ to CTX using `doca_ctx_workq_add()`
- Removing WorkQ from CTX using `doca_ctx_workq_rm()`
- Submitting a job using `doca_workq_submit()`

The following are not possible after start and become possible again after calling `doca_ctx_stop`:

- Adding device to CTX using `doca_ctx_dev_add()`
- Removing device from CTX using `doca_ctx_dev_rm()`

Note: A `doca_ctx` cannot be created by itself. Instead, you need to create a library instance (e.g., `doca_dma_create()`). Then you can acquire a CTX by converting that to a `doca_ctx` (e.g., `doca_dma_as_ctx()`).

4.8.1. doca_ctx_dev_add

This operation is a common interface for providing a device to a library.

Each library expects different capabilities and properties to exist for the device. If the provided device does not meet the requirements, it is rejected, causing it to fail.

Each successfully added device must be removed before destroying the CTX.

```
doca_error_t doca_ctx_dev_add(struct doca_ctx *ctx, struct doca_dev *dev);
```

Where:

- `ctx` – a DOCA Context, representing a data-path library instance
- `dev` – a DOCA Device with required capabilities

Note: This function cannot be used after CTX is started (`doca_ctx_start()`). CTX must be stopped (`doca_ctx_stop()`) to use it again.

4.8.2. doca_ctx_dev_rm

This operation is a common interface for removing a device from a library.

Devices must be removed from a context after stopping it and before destroying it.

```
doca_error_t doca_ctx_dev_rm(struct doca_ctx *ctx, struct doca_dev *dev);
```

Where:

- `ctx` – a DOCA Context, representing a data-path library instance

- `dev` – same DOCA Device that was previously provided through `doca_ctx_dev_add()`

> **Note:** This function cannot be used after CTX is started (`doca_ctx_start()`). CTX must be stopped (`doca_ctx_stop()`) to use it again.

4.8.3. doca_ctx_start

This operation is a common interface for starting a library instance. First, provide all configurations to the library and then call start.

Once a CTX has been started the following operations become possible:

- Adding WorkQ to CTX using `doca_ctx_workq_add()`
- Removing WorkQ from CTX using `doca_ctx_workq_rm()`
- Submitting a job related to this CTX using `doca_workq_submit()`

Whereas the following operations no longer stay possible:

- Adding device to CTX using `doca_ctx_dev_add()`
- Removing a device from CTX using `doca_ctx_dev_rm()`

To re-enable them, call `doca_ctx_stop()`.

```
doca_error_t doca_ctx_start(struct doca_ctx *ctx);
```

Where:

- `ctx` – a DOCA Context, representing a data-path library instance

4.8.4. doca_ctx_stop

This operation is a common interface for stopping a library instance. This can be done to provide further configurations to the library after starting it.

For a list of allowed operations after stop/start, refer to doca_ctx_start.

```
doca_error_t doca_ctx_stop(struct doca_ctx *ctx);
```

Where:

- `ctx` – a DOCA Context, representing a data-path library instance

4.8.5. doca_ctx_workq_add

This operation is a common interface for providing a WorkQ to a library.

For a library to start accepting and processing jobs, it must first have WorkQ attached to it so that jobs can be submitted to this WorkQ, and so the WorkQ can be polled to retrieve job completions and/or events.

Each successfully added WorkQ must be removed before stopping the CTX.

```
doca_error_t doca_ctx_workq_add(struct doca_ctx *ctx, struct doca_workq *workq);
```

Where:

- `ctx` – a DOCA Context, representing a data-path library instance
- `workq` – DOCA WorkQ for job handling

Note: This function can only be used after CTX is started (`doca_ctx_start()`). The user must remove all WorkQs before stopping CTX (`doca_ctx_stop()`).

4.8.6. doca_ctx_workq_rm

This operation is a common interface for removing a WorkQ from a library.

Added WorkQs must be removed before stopping a CTX using this API.

To remove a WorkQ, the user's responsibility is to ensure that no inflight jobs are pending completion.

```
doca_error_t doca_ctx_workq_rm(struct doca_ctx *ctx, struct doca_workq *workq);
```

Where:

- `ctx` – a DOCA Context, representing a data-path library instance
- `workq` – same DOCA WorkQ previously provided to `doca_ctx_workq_add()`

Note: This function can only be used after CTX is started (`doca_ctx_start()`). The user must remove all WorkQs before stopping CTX (`doca_ctx_stop()`).

4.9. doca_workq

Note: All `doca_workq` operations are considered thread-unsafe.

The DOCA WorkQ structure holds execution resources that different DOCA contexts can utilize, where the WorkQ provides a common interface for submitting a job, progressing a job until completion, and polling events.

The same WorkQ can be provided to multiple different CTXs, creating a single place for submitting jobs to any CTX and eventually receiving their results.

Jobs can be submitted using `doca_workq_submit()`. They can then be progressed using `doca_workq_progress_retrieve()`. Once the job is completed, a job completion event is received from the method.

In addition to jobs, some contexts expose events that can be polled using `doca_workq_progress_retrieve()`.

4.9.1. doca_workq_create

This operation creates a DOCA WorkQ instance, with a set depth. The depth defines the maximum number of in-flight jobs allowed on this WorkQ.

Once a WorkQ is created, it can be passed to multiple CTXs using `doca_ctx_workq_add()`, allowing job submission and retrieval of events.

Each successfully created WorkQ must be destroyed using `doca_workq_destroy()`.

```
doca_error_t doca_workq_create(uint32_t depth, struct doca_workq **workq);
```

Where:

- `depth` – the depth of the WorkQ. This can also be set using the property setter.
- `workq` – holds the newly created WorkQ (output parameter)

4.9.2. doca_workq_destroy

This operation destroys a DOCA WorkQ instance.

Before destroying a WorkQ, it must be removed from all the CTXs that are using it via `doca_ctx_workq_rm()`.

```
doca_error_t doca_workq_destroy(struct doca_workq *workq);
```

Where:

- `workq` – a DOCA WorkQ created via `doca_workq_create()`

4.9.3. doca_workq_property_set

This operation sets the value of a DOCA WorkQ property. It returns an error if the operation is not successful.

```
doca_error_t doca_workq_property_set(struct doca_workq *workq, enum
 doca_workq_property property, const void *value, uint32_t size);
```

Where:

- `workq` – DOCA WorkQ structure whose property to change
- `property` – the property to set. See `enum doca_workq_property`.
- `value` – the new value of the property
- `size` – the size of the property in bytes

4.9.4. doca_workq_property_get

This operation gets the up-to-date value of a DOCA WorkQ property. It returns an error if the operation is not successful.

```
doca_error_t doca_workq_property_get(const struct doca_workq *workq, enum
 doca_workq_property property, void *value, uint32_t size);
```

Where:

- `workq` – DOCA WorkQ structure whose property to return
- `property` – the property to get. See `enum doca_workq_property`.
- `value` – the value of the property (output parameter)
- `size` – the size of the property in bytes

4.9.5. doca_workq_submit

This is a common interface for submitting a job to a library.

To submit a job, a library job must be built first. Each library that implements a DOCA Context exposes several jobs in its header.

These jobs can be submitted to the library using the CTX of that library.

The WorkQ must have been previously added to a CTX that can handle the job.

For the job to complete, `doca_workq_progress_retrieve()` must be called until it succeeds and returns the job result.

```
doca_error_t doca_workq_submit(struct doca_workq *workq, struct doca_job const
 *job);
```

Where:

- `workq` – DOCA WorkQ previously added to a CTX that can handle the provided job
- `job` – base of a job defined by a library implementing DOCA CTX. The job also holds a pointer to the CTX that handles the job. Both must be compatible.

4.9.6. doca_workq_progress_retrieve

This is a common interface for polling of events.

This is a polling method that can be used to wait for events.

Events can be categorized to two types:

- Job completion events – can only be received as a result of a submitted job
- External events – can always be received based on events exposed by associated CTXs

Each CTX exposes jobs and events. Submitting a job using `doca_workq_submit()` eventually returns a job completion event using `doca_workq_progress_retrieve()`.

Other events can always be received based on the documentation of the library implementing CTX.

Calling `doca_workq_progress_retrieve()` progresses all jobs but in no particular order. Events are returned as soon as an they occur, regardless of the order in which they are submitted.

```
doca_error_t doca_workq_progress_retrieve(struct doca_workq *workq, struct
 doca_event *ev, int flags);
```

Where:

- `workq` – DOCA WorkQ previously added to a CTX that can handle the provided job
- `ev` – holds the retrieved event (output parameter valid only if the method returns `DOCA_SUCCESS`)
- `flags` – a combination of enum `doca_workq_retrieve_flags`

Chapter 5. Object Life Cycle

5.1. Device Life Cycle

1. Locate all local devices accessible by your system with `doca_devinfo_list_create()`.
2. Locate all remote devices accessible by a given local device with `doca_devinfo_remote_list_create()`.
3. At all times you can find out the properties of the device using `doca_devinfo_property_get()` or `doca_devinfo_remote_property_get()`.
4. Initialize the devices with `doca_dev_open()` or `doca_dev_remote_open()`, accordingly. Once opened, a `doca_dev` or `doca_dev_remote` is returned by the operation representing a running device. This structure serves as a handle to the underlying hardware and allocates and manages the device resources. Note that if a local device has already been opened, the same `doca_dev` is returned. At all times, the `doca_devinfo` structure can be accessed from `doca_dev` or `doca_dev_remote` with `doca_dev_as_devinfo()` or `doca_dev_remote_as_devinfo()` , respectively.
5. It is the user's responsibility to free the list of device/remote device information using `doca_devinfo_list_destroy()` or `doca_devinfo_remote_list_destroy()` at the end. At any point, the user can free an open device by using `doca_dev_close()` or `doca_dev_remote_close()`.
6. A remote device information list stays valid even after closing the `doca_dev` instance used to acquire it.

5.2. Buffer Life Cycle

1. Create a buffer with the buffer inventory API by following these instructions:
    - Retrieve a buffer from the inventory pointing to a certain memory region on a given `mmap` with `doca_buf_inventory_buf_by_addr()`
    - You may duplicate a buffer using `doca_buf_inventory_buf_dup()`. The duplicated buffer is retrieved from the inventory provided.

    Note: The buffer extensions are set upon the creation of the inventory by the user.

2. To retrieve the `doca_buf` address and `len`, use `doca_buf_head_get()` and `doca_buf_len_get()`, respectively.
3. To return the buffer back to the buffer inventory, use `doca_buf_refcount_rm()`

5.3. Buffer Inventory Life Cycle

1. Create a buffer inventory with `doca_buf_inventory_create()`.
2. View the default/current properties of the buffer inventory using `doca_buf_inventory_property_get()`. You may change the inventory properties to suit your needs with `doca_buf_inventory_property_set()`.
3. Enable the retrieval of buffers from the buffer by calling `doca_buf_inventory_start()`. Notice that on the first call to `doca_buf_inventory_start()`, the inventory properties become read-only and can no longer be changed.
4. At any point, `doca_buf_inventory_stop()` can be called to prevent new buffers from being retrieved from the inventory until `doca_buf_inventory_start()` is called once again.
5. It is the user's responsibility to free the buffer inventory using `doca_buf_inventory_destroy()` at the end. Notice that all allocated buffers must be returned to the inventory before calling this operation for it to succeed.

5.4. Memory Map Life Cycle

1. Create an mmap to hold relevant memory ranges and details regarding the devices associated with those ranges using `doca_mmap_create()`.
2. View the default/current properties of mmap using `doca_mmap_property_get()`. You may change the properties to suit your needs via `doca_mmap_property_set()`.
3. Enable mapping of buffers to the memory regions, add/remove devices, and populate the mmap by calling `doca_mmap_start()`. Note that on the first call to `doca_mmap_start()`, the mmap properties become read-only and can no longer be changed.
4. The mmap is initialized without any memory ranges. The user can add a memory range to the mmap using `doca_mmap_populate()`.
5. Associate different devices with the memory ranges held by the mmap by calling `doca_mmap_dev_add()` and disassociate a device added previously using `doca_mmap_dev_rm()`. Note that the order in which `doca_mmap_populate()` and `doca_mmap_dev_add()` are called is interchangeable and does not affect the process of associating the memory regions and the devices to one another.
6. At any point, `doca_mmap_stop()` can be called to prevent the mmap from changing until `doca_mmap_start()` is called once again. When the mmap is stopped, no memory range or device can be added to the mmap, and the mmap cannot be exported. Moreover, when the mmap stops the creation of buffers, pointing to a memory region in said mmap is disabled.
7. The mmap can be exported from the host to the BlueField by calling `doca_mmap_export()` from the host and recreating the mmap by calling `doca_mmap_create_from_export()`

from the BlueField. For more information on executing these operations, see section Exporting and Recreating Memory Map Object from Export.

8. It is the user's responsibility to free the mmap using `doca_mmap_destroy()` at the end. The mmap cannot be released as long as there are buffers pointing to memory regions in the mmap. Hence those buffers must be freed first.

5.5. Context Map Life Cycle

1. Create library context by using the library's create method `doca_T_create()` (e.g., `doca_dma_create()`).
2. Obtain a DOCA CTX by converting the library context into a DOCA CTX using the library's convert method, `doca_T_as_ctx()` (e.g., `doca_dma_as_ctx()`). Now the library context and the DOCA CTX are the same, any modification that is done automatically applies to both instances.
3. Add a required device to the CTX using `doca_ctx_dev_add()`. The device must be compatible with the library. This can be verified using the library's query capabilities API, `doca_T_devinfo_caps_get()` (e.g., `doca_dma_devinfo_caps_get()`).
4. After adding the required device, the CTX can be started using `doca_ctx_start()`.
5. Only After the CTX has been started, WorkQs can be added to it using `doca_ctx_workq_add()`. The WorkQ represents a single thread's handle to the CTX for submission and polling of jobs.
6. After adding a WorkQ to a CTX, it can start accepting jobs defined in the library's header file (e.g., `struct doca_dma_job_memcpy`).
7. Refer to the library's documentation to find if multiple WorkQs can be attached to the same CTX, or if each WorkQ requires an exclusive CTX.
8. After all jobs submitted to the CTX through a WorkQ are done, the WorkQ can be removed using `doca_ctx_workq_rm()`.
9. The CTX can be stopped using `doca_ctx_stop()` only after removing all WorkQs from it.
10. After stopping the CTX, any previously added devices must be removed using `doca_ctx_dev_rm()`.
11. After removing all WorkQs and devices, it becomes possible to destroy the CTX using the library destroy method `doca_T_destroy()` (e.g., `doca_dma_destroy()`) while using the library context as reference.

Note: The CTX must not be destroyed if any resources (i.e., device, WorkQ, job…) are still attached to it.

5.6. Work Queue Life Cycle

1. Create a WorkQ using `doca_workq_create()`, providing the depth property.
2. A WorkQ can be used to submit and progress jobs from a single thread only. As such, it is only appropriate to create one WorkQ per thread in a multi-threaded application.

3. Add a WorkQ to a started CTX using `doca_ctx_workq_add()`. Multiple WorkQs can be attached to the same CTX based on the API library's CTX documentation.
4. The same WorkQ can be added to multiple CTXs, of any type. This allows out of order progression of jobs from multiple CTXs.
5. After adding a WorkQ to a CTX, use `doca_workq_submit()` to submit jobs to the library. The job should reference the same CTX that holds the WorkQ.
6. Refer to the library CTX's header (e.g., `doca_dma.h`) to find jobs that can be submitted to the CTX through the WorkQ.
7. Submitted jobs should be progressed until completion using the method `doca_workq_progress_retrieve()`.
8. A submitted job should remain valid until a matching job completion event has been received from `doca_workq_progress_retrieve()`.
9. Once there are no more pending jobs in the WorkQ intended for a CTX, it can be removed from that CTX using `doca_ctx_workq_rm()`.
10. After removing the WorkQ from all CTXs, it can be destroyed using `doca_workq_destroy()`.
11. Only if a WorkQ is not currently added to any CTX then the property setter `doca_workq_property_set()` can be called.
12. The properties of the WorkQ can be queried at all times using `doca_workq_property_get()`.

Chapter 6. DOCA Workflow

6.1. Exporting and Recreating Memory Map Object from Export

From the host side:

1. Create a local mmap object.
2. Populate it with a single memory range.
3. Add the relevant device to the mmap.
4. Call `doca_mmap_export()`, providing the device previously added. The exported mmap is saved as an export descriptor object passed along to the BlueField.

From the BlueField side, create an mmap from an export using the function `doca_mmap_create_from_export()`, giving it the export descriptor object generated by `doca_mmap_export()` on the host side.

Note: The user is responsible for matching the devices used between the host and the BlueField.

An example sample can be found in the NVIDIA DOCA DMA Sample Guide.

6.2. Region-based Memory Access

Some applications have a set of memory ranges known at initialization time, such that all accessed memory during the applications' runtime is within these memory ranges.

If that is the case, the flow is split into two paths:

- Control path – all memory ranges are assumed to be known
- Data path – it is assumed that the application receives sets of addresses that must be accessed using a `doca_buf`, and that these sets of addresses fall within the previously mentioned memory ranges

Control path flow for each thread:

1. Create an empty memory map object.
2. Set the max chunks property to be equal to or greater than the number of known memory ranges.
3. Populate the mmap with all the memory ranges.
4. Add relevant device to mmap.
5. Create a buffer inventory with enough buffers.

Data path flow for each thread:

1. A set of (address, length) pairs ready to be accessed is received.
2. For each (address, length) pair, use `doca_buf_inventory_buf_by_addr()`.
3. Use the produced buffers to perform an operation (e.g., DMA memory copy job).
4. Free the buffers using `doca_buf_refcount_rm()` on each buffer.
5. Wait for the next set of (address, length) pairs.

6.3. Randomly Accessed Memory

Some applications may require random access to memory, such that addresses are received during runtime, but they do not reside in the same memory region. In this case, the flow is split into two paths:

- Control path – the assumption is that nothing is known about the accessed memory
- Data path – the assumption is that the application receives sets of addresses that must be accessed using a `doca_buf`

6.3.1. Single mmap Option

Control path flow for each thread:

1. Create an empty memory map object.
2. Set the max chunks property to be equal to the maximum number of expected buffers to be created during the entirety of the program's runtime.
3. Add relevant device to mmap.
4. Create a buffer inventory with enough buffers.

Data path flow for each thread:

1. Set of (address, length) pairs ready to be accessed is received.
2. For each (address, length) pair, use `doca_mmap_populate()` to add the pair to the mmap.
3. Use same pair with `doca_buf_inventory_buf_by_addr()` to receive the `doca_buf`.
4. Use the produced buffers to perform an operation (e.g., DMA memory copy job).
5. Free the buffers using `doca_buf_refcount_rm()` on each buffer.
6. Wait for the next set of (address, length) pairs.

Limitation: This option can create a maximum of 4 billion (2^32) `doca_bufs` during the application's lifetime.

6.3.2. Mmap per Buffer Option

Control path flow for each thread:

1. Create a buffer inventory with enough buffers.

Data path flow for each thread:

1. Set of (address, length) pairs ready to be accessed is received.
2. Create an empty mmap.
3. Add the relevant device to mmap.
4. Set the max chunks property to be equal to the number of pairs.
5. For each pair, use `doca_mmap_populate()` to add the pair to the mmap.
6. Use same pair with `doca_buf_inventory_buf_by_addr()` to receive the `doca_buf`.
7. Use the produced buffers to perform an operation (e.g., DMA memory copy job).
8. Free the buffers using `doca_buf_refcount_rm()` on each buffer.
9. Remove the device from the mmap.
10. Destroy the mmap.
11. Wait for the next set of (address, length) pairs.

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com