



# NVIDIA DOCA Core

## Programming Guide

# Table of Contents

Chapter 1. Introduction.....	1
Chapter 2. Prerequisites.....	3
Chapter 3. Architecture.....	4
3.1. General.....	4
3.1.1. doca_error_t.....	4
3.1.2. Generic Structures/Enum.....	4
3.2. DOCA Device.....	5
3.2.1. Local Device and Representor.....	5
3.2.1.1. Local Device and Representor Matching.....	7
3.2.2. Expected Flow.....	7
3.3. DOCA Memory Subsystem.....	8
3.3.1. Requirements and Considerations.....	9
3.3.2. doca_buf/doca_buf_inventory.....	10
3.3.3. Example Flow.....	10
3.3.4. doca_mmap.....	12
3.4. DOCA Execution Model.....	13
3.4.1. Requirements and Considerations.....	14
3.4.2. DOCA Context.....	15
3.4.3. DOCA WorkQ.....	16
3.4.4. Polling Mode.....	16
3.4.5. Event-driven Mode.....	17
3.4.6. Job Error Handling.....	19
3.5. Object Life Cycle.....	19
Chapter 4. Compatibility.....	21


---

# Chapter 1. Introduction

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 exposes C-language API to applications writes 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:

DOCA Core Module	Description
General	<ul style="list-style-type: none"><li>▶ DOCA core enumerations and basic structures</li><li>▶ Header files – <code>doca_error.h</code>, <code>doca_types.h</code></li></ul>
Device Handling	<ul style="list-style-type: none"><li>▶ Queries device information (host-side and DPU) and device capabilities (e.g., device's PCIe BDF address)<ul style="list-style-type: none"><li>▶ On DPU<ul style="list-style-type: none"><li>▶ Gets local DPU devices</li><li>▶ Gets representors list (representing host local devices)</li></ul></li><li>▶ On host<ul style="list-style-type: none"><li>▶ Gets local devices</li><li>▶ Queries device capabilities and library capabilities</li></ul></li></ul></li><li>▶ Opens and uses the selected device representor</li><li>▶ Relevant entities – <code>doca_devinfo</code>, <code>doca_devinfo_rep</code>, <code>doca_dev</code>, <code>doca_dev_rep</code></li><li>▶ Header files – <code>doca_dev.h</code></li></ul> <div data-bbox="836 1829 1427 1892"> <b>Note:</b> There is a symmetry between device entities on host and its</div>

DOCA Core Module	Description
<p>Memory Management</p> <p>Progress Engine and Job Execution</p>	<p>representor (on the DPU). The convention of adding <code>rep</code> to the API or the object hints that it is representor-specific.</p> <ul style="list-style-type: none"> <li>▶ Handles optimized memory pools to be used by applications and enables sharing resources between DOCA libraries (while hiding HW related technicalities)</li> <li>▶ Data buffer services (e.g., linked list of buffers to support scatter-gather list)</li> <li>▶ Maps host memory to the DPU for direct access</li> <li>▶ Relevant entities – <code>doca_buf</code>, <code>doca_mmap</code>, <code>doca_inventory</code></li> <li>▶ Header files – <code>doca_buf.h</code>, <code>doca_buf_inventory.h</code>, <code>doca_mmap.h</code></li> </ul> <ul style="list-style-type: none"> <li>▶ Enables submitting jobs to DOCA libraries and track job progress (supports both polling mode and event-driven mode)</li> <li>▶ Relevant entities – <code>doca_ctx</code>, <code>doca_job</code>, <code>doca_event</code>, <code>doca_event_handle_t</code>, <code>doca_workq</code></li> <li>▶ Header files – <code>doca_ctx.h</code></li> </ul>

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

---

## Chapter 2. Prerequisites

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

- ▶ DOCA version 1.5 or greater
- ▶ BlueField software 3.9.3 or greater
- ▶ Firmware version 24.35.1012 or greater

---

# Chapter 3. Architecture

The following sections describe the architecture for the various DOCA core software modules.

## 3.1. General

### 3.1.1. `doca_error_t`

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

```
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_t;
```

### 3.1.2. Generic Structures/Enum

The following types are common across all device types in the DOCA Core API.

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

enum doca_access_flags {
```

```

DOCA_ACCESS_LOCAL_READ    = 0,
DOCA_ACCESS_LOCAL_WRITE  = 1,
DOCA_ACCESS_REMOTE_WRITE = (1 << 1),
DOCA_ACCESS_REMOTE_READ  = (1 << 2),
DOCA_ACCESS_REMOTE_ATOMIC = (1 << 3),
};

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

```

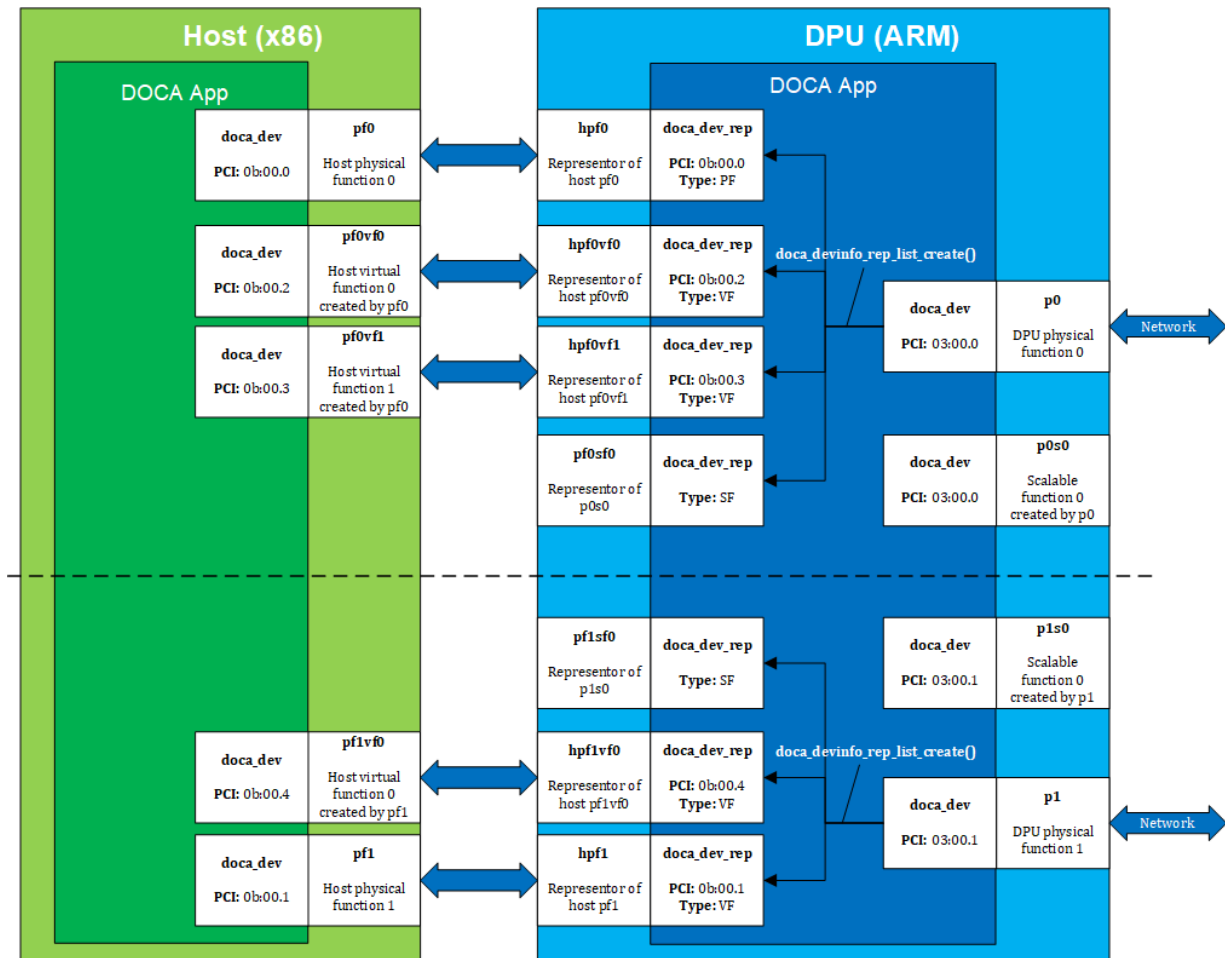
## 3.2. DOCA Device

### 3.2.1. Local Device and Representer

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 (DPU or host) and can perform DOCA library processing jobs (can be a hardware device or device emulation)
- ▶ Representer device – this is a representation of a local device. The local device is usually on the host (except for SFs) and the representer is always on the DPU side (a proxy on the DPU for the host-side device).

The following figure provides an example topology:



The diagram shows a DPU (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 DPU 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 the DPU.

If the user queries local devices on the DPU side (not representor devices), they get the two (in this example) DPU PFs, `p0` and `p1`. These two DPU 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 2 parts (see dotted line), each part is represented by a DPU physical device, `p0` and `p1`, each of which is responsible for creating all other local devices (host PFs, host VFs, and DPU SFs). As such, the DPU 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`).

### 3.2.1.1. Local Device and Representor Matching

Based on the diagram in section [Local Device and Representor](#), the mmap export APIs can be used as follows:

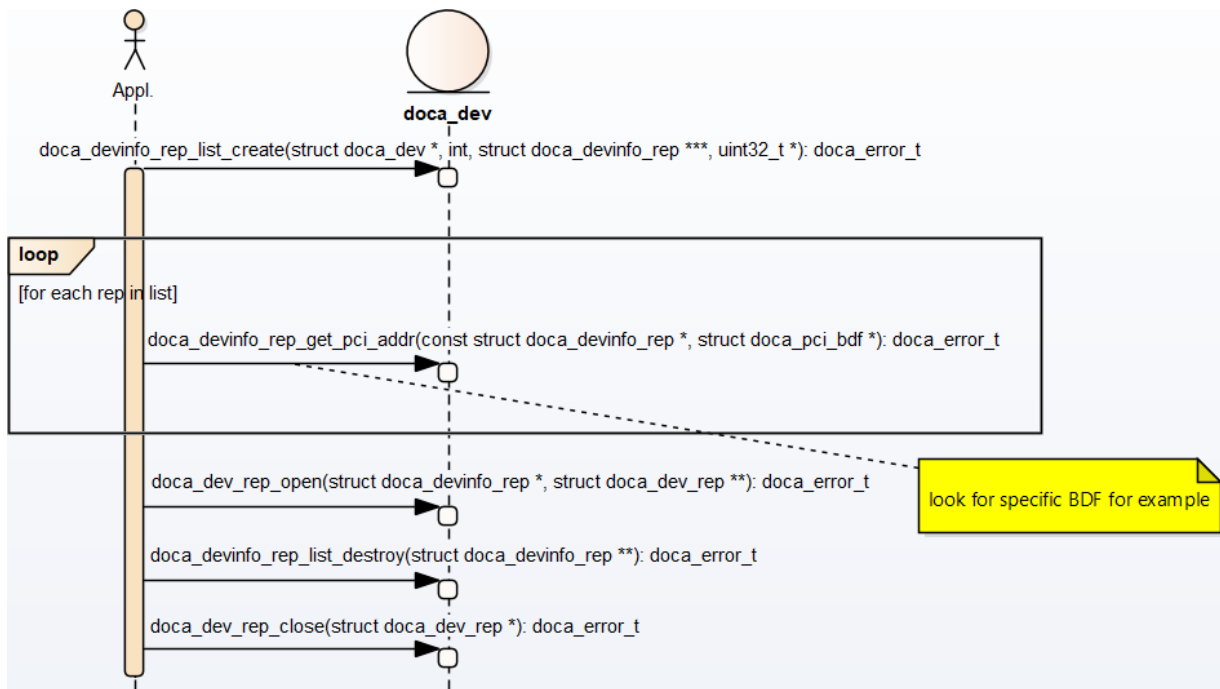
Device to Select on Host When Using <code>doca_mmap_export()</code>	DPU Matching Representor	Device to Select on DPU When Using <code>doca_mmap_from_export()</code>
<code>pf0 - 0b:00.0</code>	<code>hpf0 - 0b:00.0</code>	<code>p0 - 03:00.0</code>
<code>pf0vf0 - 0b:00.2</code>	<code>hpf0vf0 - 0b:00.2</code>	
<code>pf0vf1 - 0b:00.3</code>	<code>hpf0vf1 - 0b:00.3</code>	
<code>pf1 - 0b:00.1</code>	<code>hpf1 - 0b:00.1</code>	<code>p1 - 03:00.1</code>
<code>pf1vf0 - 0b:00.4</code>	<code>hpf1vf0 - 0b:00.4</code>	

### 3.2.2. Expected Flow

To work with DOCA libraries or DOCA Core objects, the application must open and use a representor device on the DPU. Before it can open the representor device and use it, the application needs tools to allow it 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 DPU representor). The flow is as follows:

1. List all representor devices on DPU.
2. Select one with the required capabilities.
3. Open this representor and use it.

As mentioned previously, the DOCA Core API is able to identify devices and their representors that have a unique property (e.g., the BDF address, the same BDF for the device and its DPU representor).



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 DPU side, the application gets a list of device representors for a specific DPU 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 suites 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 has to close `doca_dev` too.



**Note:** Regarding device property caching, the functions `doca_devinfo_list_create` and `doca_devinfo_rep_list_create` provide a snapshot of the DOCA device properties when they are called. If any device's properties are changed dynamically (e.g., BDF address may change on bus reset), the device properties that those functions return would not reflect this change. One should call them again to get the updated properties of the devices.

### 3.3. DOCA Memory Subsystem

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

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

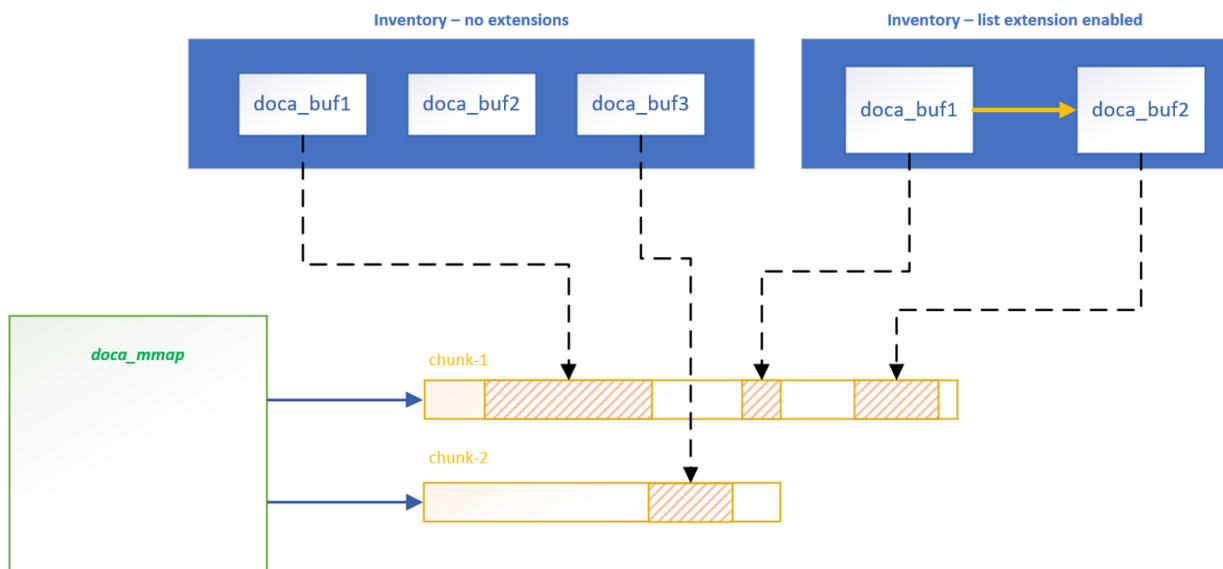
- ▶ `doca_mmap` – this is the data buffers pool (chunks) which are pointed at by `doca_buf`. The application populates this memory pool with buffers/chunks and maps them to devices that must access the data.

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 section [doca\\_buf/doca\\_buf\\_inventory](#)). As all DOCA entities, memory subsystem objects are opaque and can be instantiated by DOCA SDK only.

One of the critical requirements from `doca_buf` is to minimize its size so programs would not run into a lack of memory or scalability issues. For that purpose, DOCA features an extension support for `doca_buf_inventory` which means that the application can assign specific extensions to each `doca_buf_inventory` it creates (can also bitwise OR extensions). By default, the minimal `doca_buf` structure is used without any extensions.

An example for extension is `LINKED_LIST` (see the `doca_buf_extension` enumeration in `doca_buf.h`) which allows the application to chain several `doca_bufs` and create a linked list of `doca_buf` (which can be used for scatter/gather scenarios). All `doca_bufs` originating from the same inventory have the same characteristics (i.e., extensions).

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



In the diagram, you may see two `doca_buf_inventory`s. The left one has no extensions while right has a linked list extension enabled which enables chaining `doca_buf1` and `doca_buf2`. Each `doca_buf` points to a portion of the memory buffer which is part of a `doca_mmap`. The MMAP is populated with two memory buffers, `chunk-1` and `chunk-2`.

### 3.3.1. 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 buffers in the mmap can be mapped to one device or more
- ▶ `doca_buf` points to a specific memory buffer (or part of it) and holds metadata for that buffer (e.g., `1key`)
- ▶ The internals of mapping and working with the device (e.g., memory registrations) is hidden from the application
- ▶ The host-mapped memory chunk can be accessed by DPU

### 3.3.2. `doca_buf/doca_buf_inventory`

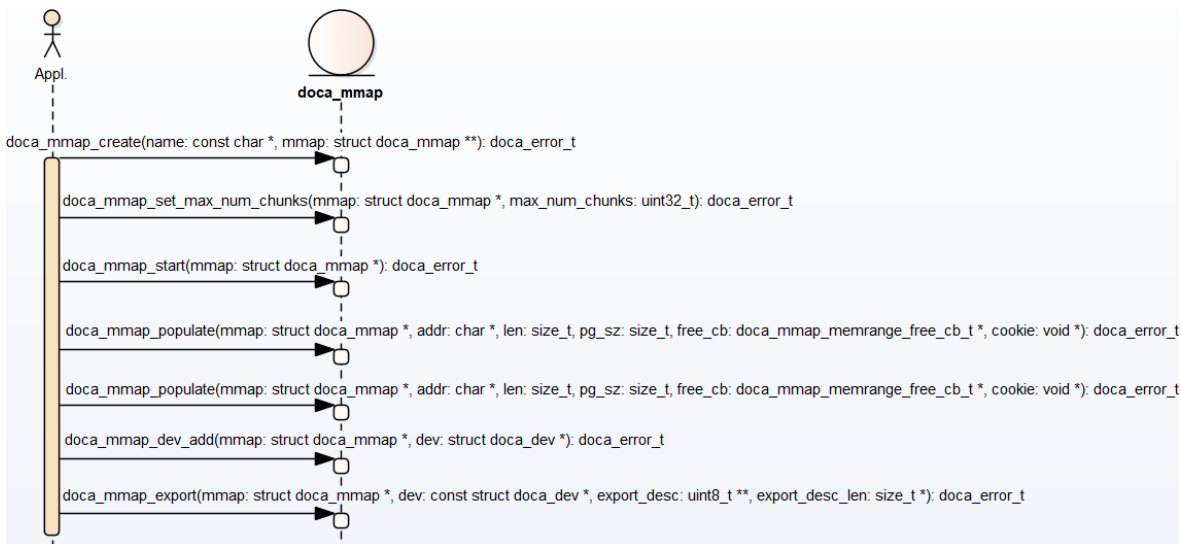
`doca_buf` is opaque and can only be allocated using DOCA API. As previously mentioned, it is the descriptor that points to a specific (portion or entire) mmap buffer (chunk). `doca_buf_inventory` is a pool of `doca_bufs` that the application creates. Still, the `doca_bufs` in such an inventory are placeholders and do not point to the data. When the application desires to assign a `doca_buf` to a specific data buffer, it calls the `doca_buf_inventory_buf_by_addr` API.

When enabling specific extensions for an inventory, the application must check to make sure that the relevant contexts indeed support the relevant extension. If the context does not support the requested extensions, the application must not pass `doca_buf` with these extensions to the context. For example, if the application wishes to use the linked list extension and concatenate several `doca_buf` to a scatter-gather list, it is expected to make sure the application indeed supports a linked list extension by calling `doca_dma_get_max_list_buf_num_elem` (this example checks linked-list support for DMA).

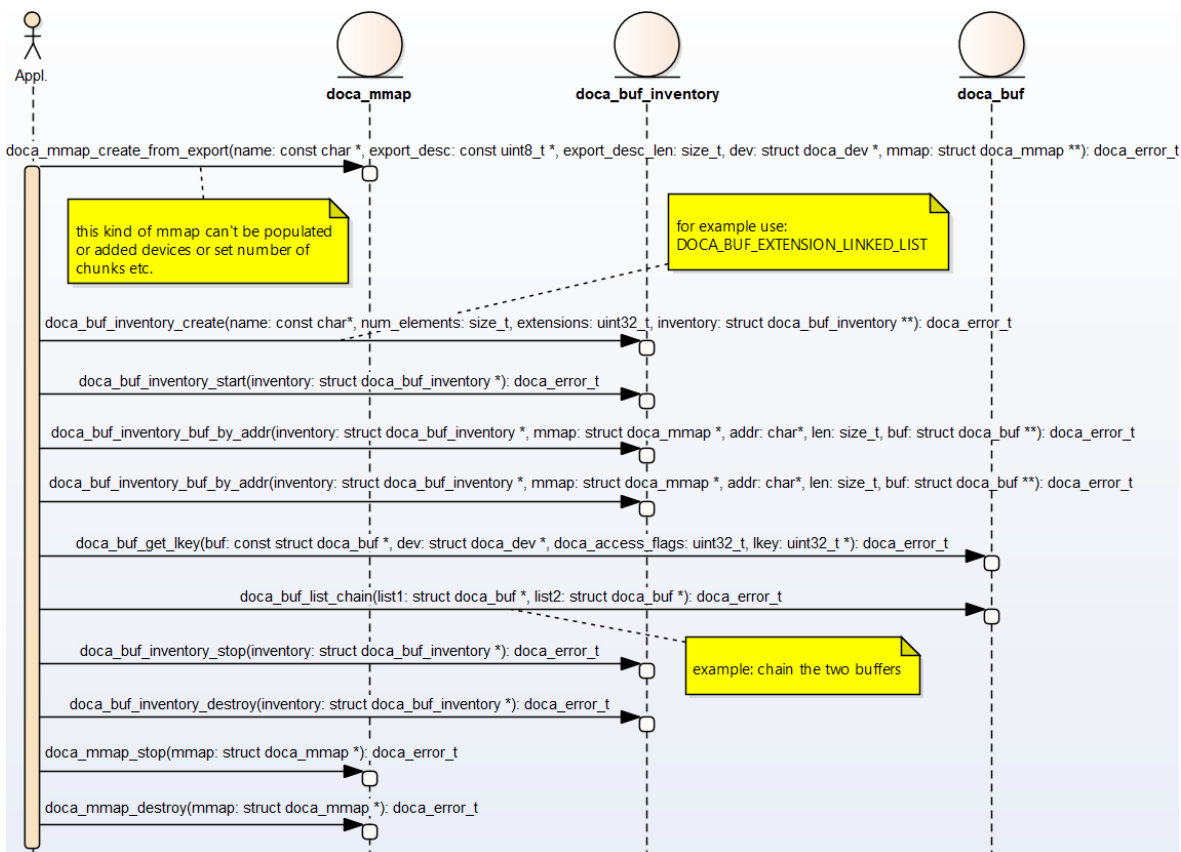
### 3.3.3. Example Flow

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

1. Create mmap on the host (see section [Local Device and Representor Matching](#) for information on how to choose the `doca_dev` to add to mmap). This example populates 2 data buffers and adds a single `doca_dev` to the mmap and exports it so DPU can use it.




2. Import to the DPU (e.g., use the mmap descriptor output parameter as input to doca\_mmap\_create\_from\_export).

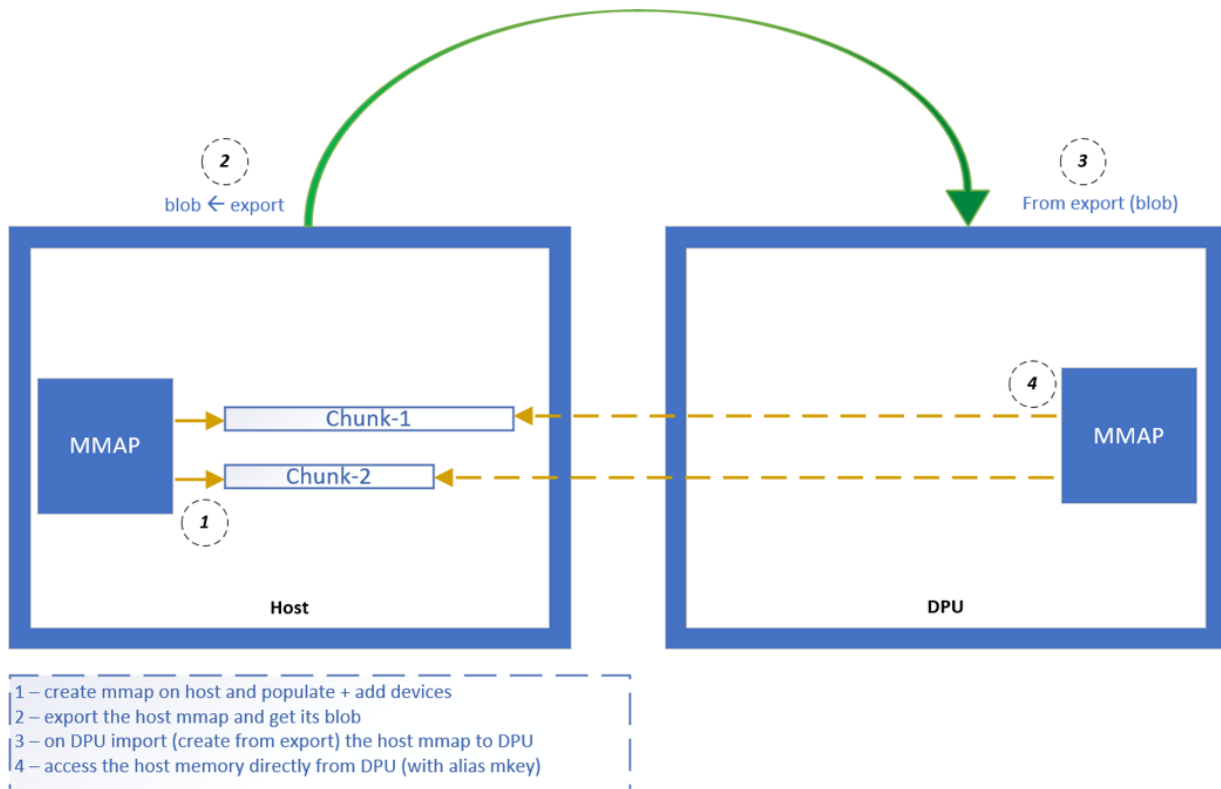


### 3.3.4. `doca_mmap`

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

DOCA SDK supports several types of mmap that help with different use cases:

- ▶ Local mmap – this is the basic type of mmap which maps local buffers to the local device(s)
    1. The application creates and starts `doca_mmap`.
    2. The application adds devices and populates buffers to the mmap granting the devices access to the buffers.
-  **Note:** Populating and adding devices can be done at any order and the mmap will maintain the mapping of every buffer to every device.
- 3. If the application desires the DPU to have access (zero-copy) to mmap buffers, it must call `doca_mmap_export()` and pass the resulting blob to the DPU that could import this mmap and have direct access to the host memory.
  - ▶ From the export mmap (imported) – this mmap can be created on the DPU only
    1. The application calls `doca_mmap_create_from_export` and gives the blob it got by calling `doca_mmap_export` on the host-side mmap.
    2. Now the application can create `doca_buf` to point to this imported mmap and have direct access to the host's memory.



## 3.4. DOCA Execution Model

In DOCA, the workload involves transforming source data to destination data. The basic transformation is a DMA operation on the data which simply moves data from one memory location to another. Other operations involve calculating the SHA value of the source data and writing it to the destination.

The workload can be broken into 3 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 job (`doca_job`).

Jobs describe operations that an application would like to submit to DOCA (hardware or DPU). To do so, the application requires a means of communicating with the hardware/DPU. This is where the `doca_workq` comes into play. The WorkQ is a per-thread object used to queue jobs to offload to DOCA and eventually receive their completion status.

`doca_workq` introduces three main operations:

1. Submission of jobs.
2. Checking progress/status of submitted jobs.
3. Querying job completion status.

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

A context can be obtained from some libraries within the DOCA SDK. For example, to submit DMA jobs, 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 job types.

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

### 3.4.1. Requirements and Considerations

In DOCA, the workload involves transforming source data to destination data. The basic transformation is a DMA operation on the data which simply moves data from one memory location to another. Other operations involve calculating the SHA value of the source data and writing it to the destination.

The workload can be broken into 3 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 job (`doca_job`).

Jobs describe operations that an application would like to submit to DOCA (hardware or DPU). To do so, the application requires a means of communicating with the hardware/DPU. This is where the `doca_workq` comes into play. The WorkQ is a per-thread object used to queue jobs to offload to DOCA and eventually receive their completion status.

`doca_workq` introduces three main operations:

1. Submission of jobs.
2. Checking progress/status of submitted jobs.
3. Querying job completion status.

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

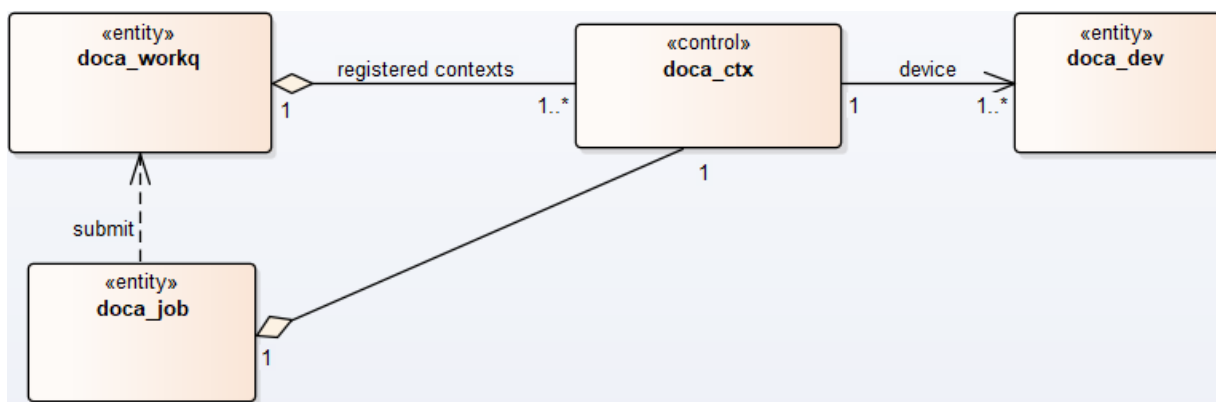
A context can be obtained from some libraries within the DOCA SDK. For example, to submit DMA jobs, a DMA context can be acquired from `doca_dma.h`, whereas SHA context can be obtained using `doca_sha.h`. Each such context may allow submission of several job types.

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



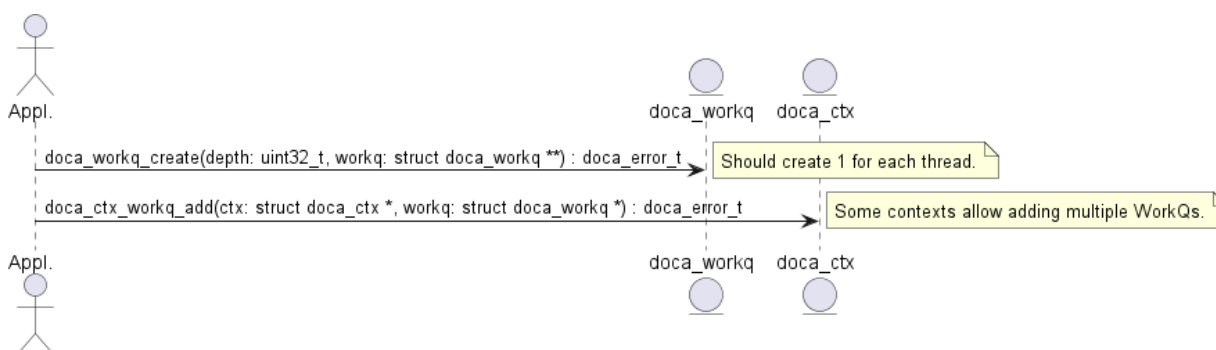
### 3.4.2. DOCA Context

`doca_ctx` represents an instance of a specific DOCA library (e.g., DMA, SHA). Before submitting jobs to the WorkQ for execution, the job must be associated to a specific context that executes the job. The application is expected to associate (i.e., add) WorkQ with that context. Adding a WorkQ to a context allows submitting a job to the WorkQ using that context. Context represents a set of configurations including the job type and the device that runs it such that each job submitted to the WorkQ is associated with a context that has already been added. The following diagram shows the high-level (domain model) relations between various DOCA core entities.



1. `doca_job` is associated to a relevant `doca_ctx` that executes the job (with the help of the relevant `doca_dev`).
2. `doca_job`, after it is initialized, is submitted to `doca_workq` for execution.
3. `doca_ctxs` are added to the `doca_workq`. once a `doca_job` is queued to `doca_workq`, it is submitted to the `doca_ctx` that is associated with that job type in this WorkQ.

The following diagram describes the initialization sequence of a context:



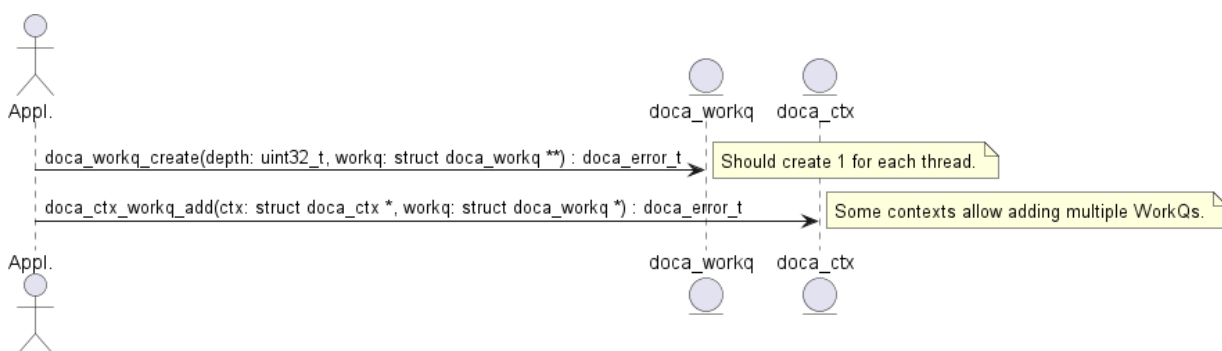
After the context is started, it can be used to enable the submission of jobs to a WorkQ based on the types of jobs that the context supports. See section TBD for more information.

Context is a thread-safe object. Some contexts can be used across multiple WorkQs while others can only be used on 1. Please refer to documentation of the specific context for specific information per context (e.g., `doca_dma`).

### 3.4.3. DOCA WorkQ

`doca_workq` is a logical representation of DOCA thread of execution (non-thread-safe). WorkQ is used to submit jobs to the relevant context/library (hardware offload most of the time) and query the job's completion status. To start submitting jobs, however, the WorkQ must be configured to accept that type of job. Each WorkQ can be configured to accept any number of job types depending on how it initialized.

The following diagram describes the initialization flow of the WorkQ:



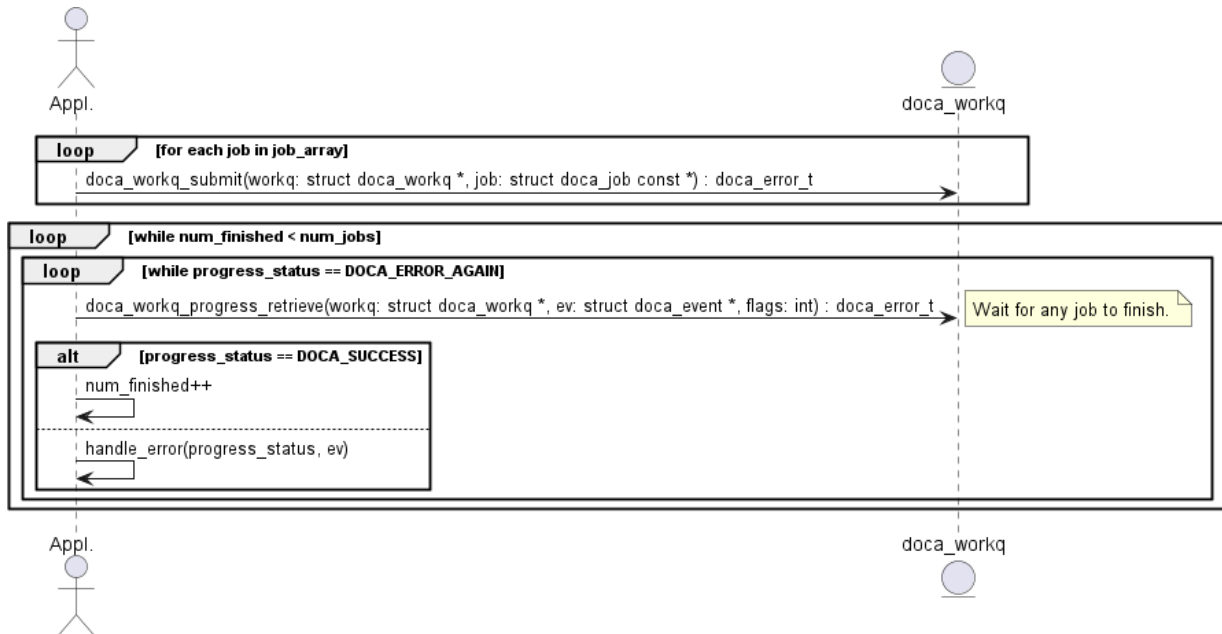
After the WorkQ has been created and added to a context, it can start accepting jobs that the context defines. Refer to the context documentation to find details such as whether the context supports adding multiple `doca_workqs` to the same context and what jobs can be submitted using the context.

Please note that the WorkQ can be added to multiple contexts. Such contexts can be of the same type or of different types. This allows submitting different job types to the same WorkQ and waiting for any of them to finish from the same place/thread.

### 3.4.4. Polling Mode

In this mode, the application submits a job and then does busy-wait to find out when the job has completed. Polling mode is enabled by default.

The following diagram demonstrates this sequence:



1. The application submits all jobs (one or more) and tracks the number of completed jobs to know if all jobs are done.
2. The application waits for a job to finish.
  - a). If `doca_workq_progress_retrieve()` returns `DOCA_ERROR_AGAIN`, it means that jobs are still running (i.e. no result).
  - b). Once a job is done, `DOCA_SUCCESS` is returned from `doca_workq_progress_retrieve()`.
  - c). If another status is returned, that means an error has occurred (see section [Job Error Handling](#)).
3. Once a job has finished, the counter for tracking the number of finished jobs is updated.

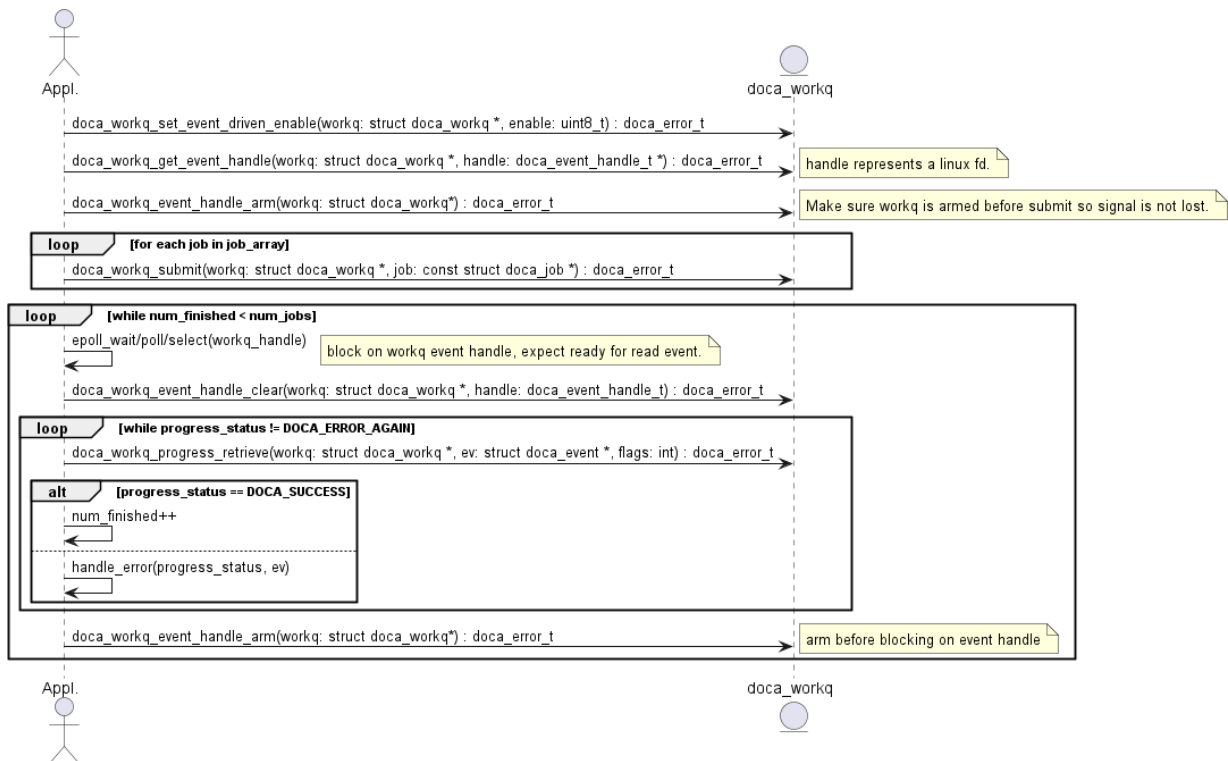


**Note:** In this mode the application is always using the CPU even when it is doing nothing (during busy-wait).

### 3.4.5. Event-driven Mode

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

The following diagram shows this sequence:



1. The application enables event-driven mode of the WorkQ. If this step fails (`DOCA_ERROR_NOT_SUPPORTED`), it means that one or more of the contexts associated with the WorkQ (via `doca_ctx_workq_add`) do not support this mode. To find out if a context supports this event-driven mode, refer to the context documentation. Alternatively, the API `doca_ctx_get_event_driven_supported()` can be called during runtime.
2. The application gets an event handle from the `doca_workq` representing a Linux file descriptor which is used to signal the application that some work has finished.
3. The application then arms the WorkQ.



**Note:** This must be done every time an application is interested in receiving a signal from the WorkQ.

4. The application submits a job to the WorkQ.
5. The application waits (e.g., Linux `epoll/select`) for a signal to be received on the `workq-fd`.
6. The application clears the received events, notifying the WorkQ that a signal has been received and allowing it to do some event handling.
7. The application attempts to retrieve a result from the WorkQ.



**Note:** There is no guarantee that the call to `doca_workq_progress_retrieve` would return a job completion event but the WorkQ can continue the job.

8. Increment the number of finished jobs if successful or handle error.
9. Arm the WorkQ to receive the next signal.
10. Repeat steps 5-9 until all jobs are finished.

## 3.4.6. Job Error Handling

After a job is submitted successfully, consequent calls to `doca_workq_progress_retrieve` may fail (i.e., return different status from `DOCA_SUCCESS` or `DOCA_ERROR_AGAIN`). In this case, the error is split into 2 main categories:

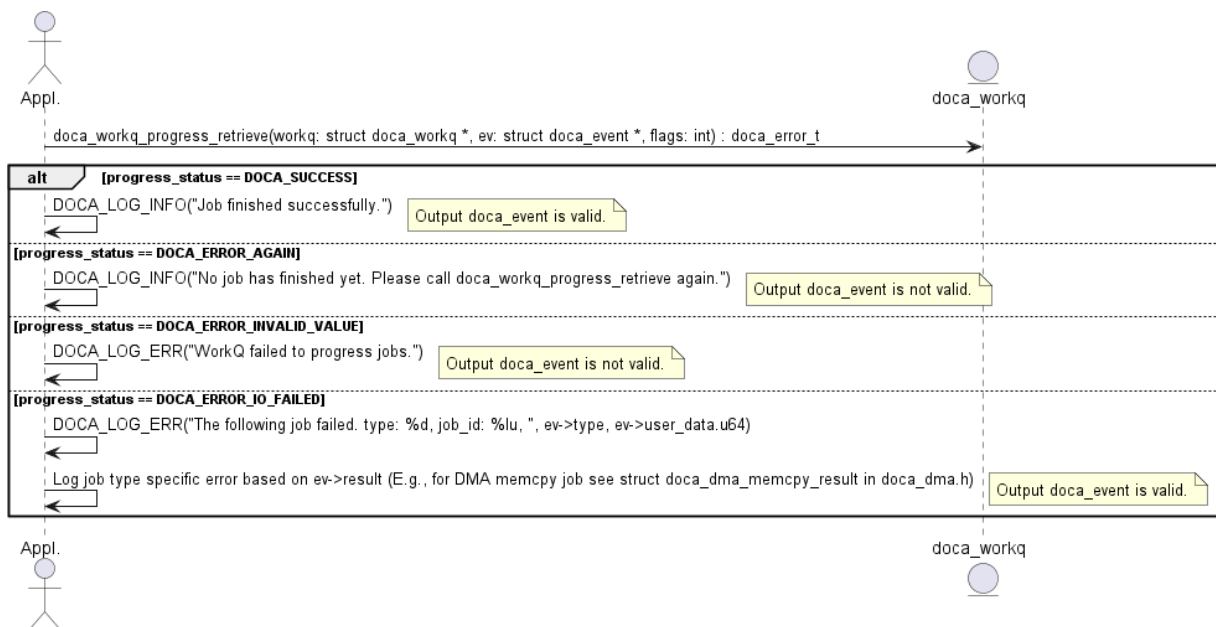
1. `DOCA_ERROR_INVALID_VALUE`

This means that some error has occurred within the WorkQ that is not related to any submitted job. This can happen due to the application passing invalid arguments or to some objects that have been previously provided (e.g., a `doca_ctx` that was associated using `doca_ctx_workq_add`) getting corrupted. In this scenario, the output parameter of type `doca_event` is not valid and no more information is given about the error.

2. `DOCA_ERROR_IO_FAILED`

This means that a specific job has failed where the output variable of type `doca_event` is valid and can be used to trace the exact job that failed. Additional error code explaining the exact failure reason is given. To find the exact error, refer to the documentation of the context that provides the job type (e.g., if the job is DMA memcopy, then refer to `doca_dma.h`).

The following diagram shows how an application is expected to handle error from `doca_workq_progress_retrieve`:



## 3.5. Object Life Cycle

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

1. The object is allocated by DOCA as 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_max_num_chunks`).
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`).

---

# Chapter 4. Compatibility

An application that uses the hardware relies on a subset of features to be present for it to be able to function. As such, it is customary to check if the subset of features exists. The application may also need to identify the specific hardware resource to work with based on specific properties. The same applies for an application that uses a DOCA library.

It is up to the application to:

- ▶ Check which library's APIs are supported for a given `doca_devinfo`.
- ▶ Configure the library context through the dedicated API according to the library's limitations.
- ▶ Check library's configuration limitations.

DOCA capabilities is a set of APIs (DOCA library level) with a common look and feel to achieve this.

For example:

- ▶ A hotplug (of emulated PCIe functions) oriented application can check if a specific DOCA device information structure enables hotplugging emulated devices, by calling:

```
doca_error_t doca_devinfo_get_is_hotplug_manager_supported(const struct
doca_devinfo *devinfo, uint8_t *is_hotplug_manager);
```
- ▶ An application that works with DOCA memory map to be shared between the host and DPU must export the `doca_mmap` from the host and import it from the DPU. Before starting the workflow, the application can check if those operations are supported for a given `doca_devinfo` using the following APIs:

```
doca_error_t doca_devinfo_get_is_mmap_export_supported(const struct doca_devinfo
*devinfo, uint8_t *mmap_export);
doca_error_t doca_devinfo_get_is_mmap_from_export_supported(const struct
doca_devinfo *devinfo, uint8_t *from_export);
```
- ▶ When working with DOCA inventory extensions (e.g., a `LINKED_LIST` extension which allows chaining several `doca_bufs` in a linked-list manner) an application can check if the list extension is supported by `doca_buf_inventory` with:

```
doca_error_t doca_buf_inventory_get_list_supported(struct doca_buf_inventory
*inventory, uint8_t *list_supported);
```

## Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation nor any of its direct or indirect subsidiaries and affiliates (collectively: "NVIDIA") make no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assume no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

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

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

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

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

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

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

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

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

## Trademarks

NVIDIA, the NVIDIA logo, and Mellanox are trademarks and/or registered trademarks of Mellanox Technologies Ltd. and/or NVIDIA Corporation in the U.S. and in other countries. The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis. Other company and product names may be trademarks of the respective companies with which they are associated.

## Copyright

© 2022 NVIDIA Corporation & affiliates. All rights reserved.