NVIDIA DOCA Core

Programming Guide

Table of Contents

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

DOCA Core exposes C-language API to application writers and users must include the right header file to use according to the DOCA Core facilities needed for their application.

DOCA Core can be divided into the following software modules:

DOCA Core Module	Description
General	▶ DOCA Core enumerations and basic structures ▶ Header files – `doca_error.h`, `doca_types.h`

Device handling	▶ Queries device information (host-side and DPU) and device capabilities (e.g., device's PCIe BDF address)
	    ▶ On DPU
	        ▶ Gets local DPU devices
	        ▶ Gets representors list (representing host local devices)
	    ▶ On host
	        ▶ Gets local devices
	    ▶ Queries device capabilities and library capabilities
	▶ Opens and uses the selected device representor

DOCA Core Module	Description
	▶ Relevant entities – `doca_devinfo`, `doca_devinfo_rep`, `doca_dev`, `doca_dev_rep` ▶ Header files – `doca_dev.h` 📄 Note: There is a symmetry between device entities on host and its representor (on the DPU). The convention of adding `rep` to the API or the object hints that it is representor-specific.
Memory management	▶ Handles optimized memory pools to be used by applications and enables sharing resources between DOCA libraries (while hiding hardware-related technicalities) ▶ Data buffer services (e.g., linked list of buffers to support scatter-gather list) ▶ Maps host memory to the DPU for direct access ▶ Relevant entities – `doca_buf`, `doca_mmap`, `doca_buf_inventory`, `doca_buf_array`, `doca_bufpool` ▶ Header files – `doca_buf.h`, `doca_buf_inventory.h`, `doca_mmap.h`, `doca_buf_array.h`, `doca_bufpool`
Progress engine and job execution	▶ Enables submitting jobs to DOCA libraries and track job progress (supports both polling mode and event-driven mode) ▶ Relevant entities – `doca_ctx`, `doca_job`, `doca_event`, `doca_event_handle_t`, `doca_workq` ▶ Header files – `doca_ctx.h`
Sync events	▶ Sync events are used to synchronize threads on the same processor/process or on different processors (e.g., synchronize DPA, DPU and host) ▶ Header files – `doca_dpa_sync_event.h`, `doca_sync_event.h`

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 2.0.2 or greater
- ▶ BlueField software 4.0.2 or greater
- ▶ BlueField-3 firmware version 32.37.1000 and higher
- ▶ BlueField-2 firmware version 24.37.1000 and higher

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_ONLY      = 0,
 DOCA_ACCESS_LOCAL_READ_WRITE     = (1 << 0),
 DOCA_ACCESS_RDMA_READ            = (1 << 1),
 DOCA_ACCESS_RDMA_WRITE           = (1 << 2),
 DOCA_ACCESS_RDMA_ATOMIC          = (1 << 3),
 DOCA_ACCESS_DPU_READ_ONLY        = (1 << 4),
 DOCA_ACCESS_DPU_READ_WRITE       = (1 << 5),
};

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

3.2.   DOCA Device

3.2.1.   Local Device and Representor

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)

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

The following figure provides an example topology:
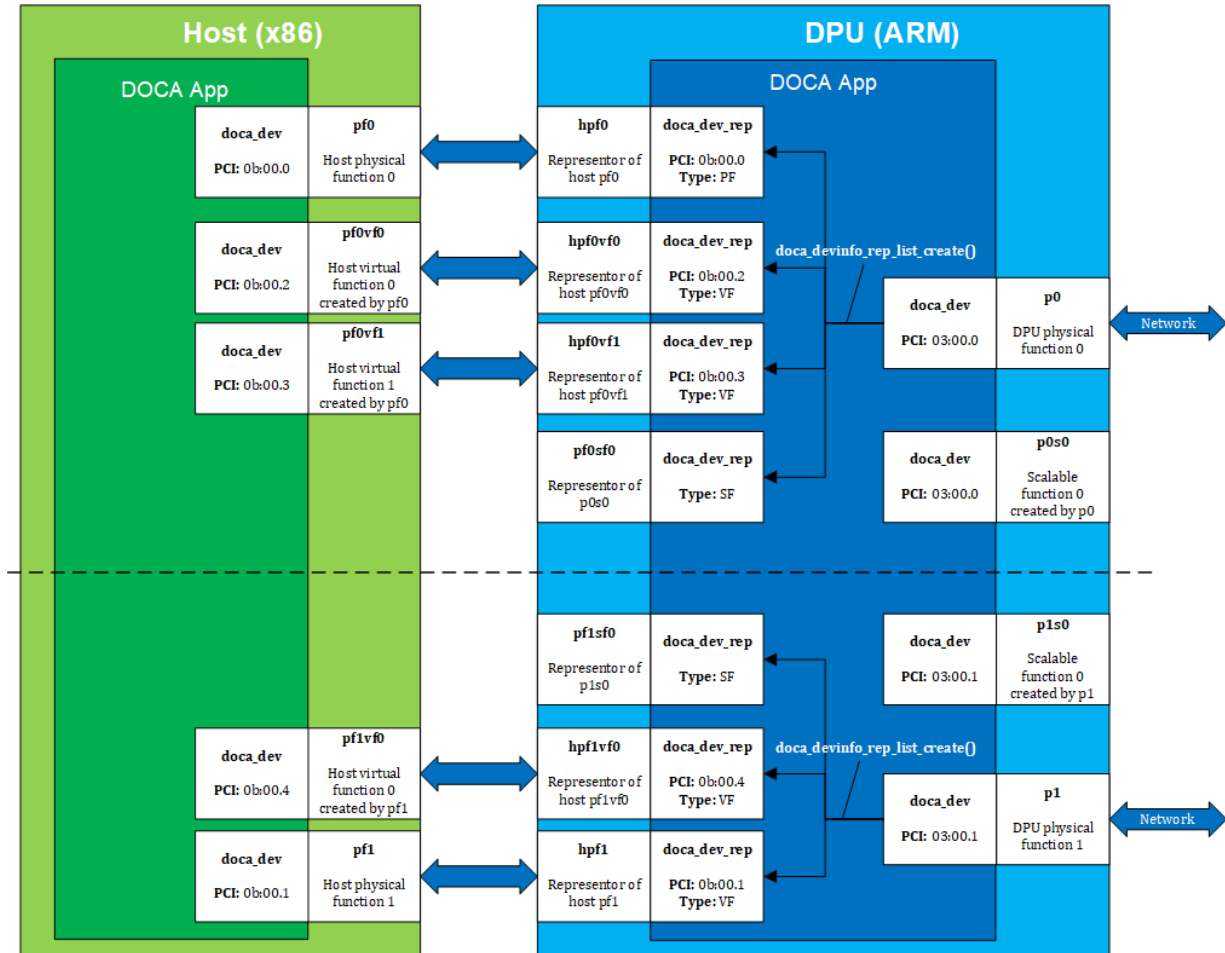
The diagram shows a DPU (on the right side of the figure) connected to a host (on the left side of the figure). The host topology consists of two physical functions (PF0 and PF1). Furthermore, PF0 has two child virtual functions, VF0 and VF1. PF1 has only one VF associated with it, VF0. Using the DOCA 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 doca_mmap_export()	DPU Matching Representor	Device to Select on DPU When Using doca_mmap_from_export()
pf0 – 0b:00.0	hpf0 – 0b:00.0	p0 – 03:00.0
pf0vf0 – 0b:00.2	hpf0vf0 – 0b:00.2	
pf0vf1 – 0b:00.3	hpf0vf1 – 0b:00.3	
pf1 – 0b:00.1	hpf1 – 0b:00.1	p1 – 03:00.1
pf1vf0 – 0b:00.4	hpf1vf0 – 0b:00.4	

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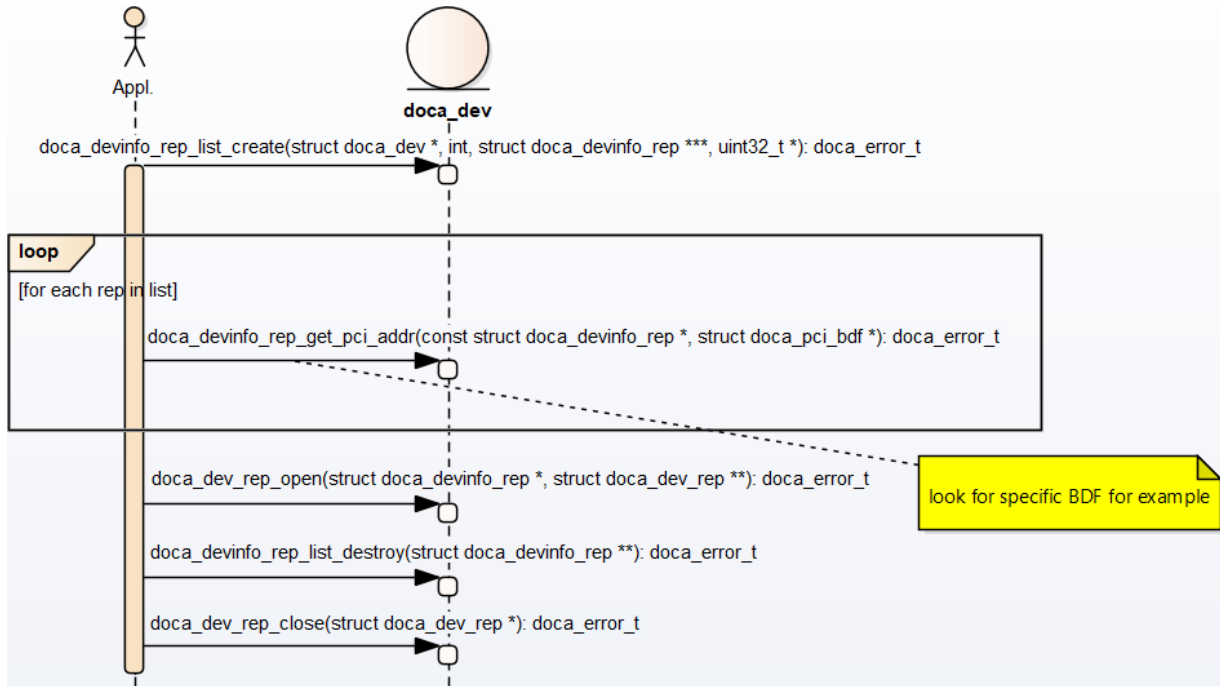
1. The application "knows" which device it wants to use (e.g., by its PCIe BDF address). On the host, it can be done using DOCA Core API or OS services.

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` 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 the following 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 which `doca_buf` points at. The application provides the memory as a single memory region as well as permissions for certain devices to access it.

As the `doca_mmap` serves as the memory pool for data buffers, there is also an entity called `doca_buf_inventory`which serves as a pool of `doca_buf` with same characteristics (see more under [doca_buf and 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. Without any extensions, the `doca_buf` can be used as a linked list which allows the application to chain several `doca_buf`s and create a linked list of `doca_buf` (which can be used for scatter/gather scenarios).

The following diagram shows the various modules within the DOCA memory subsystem:
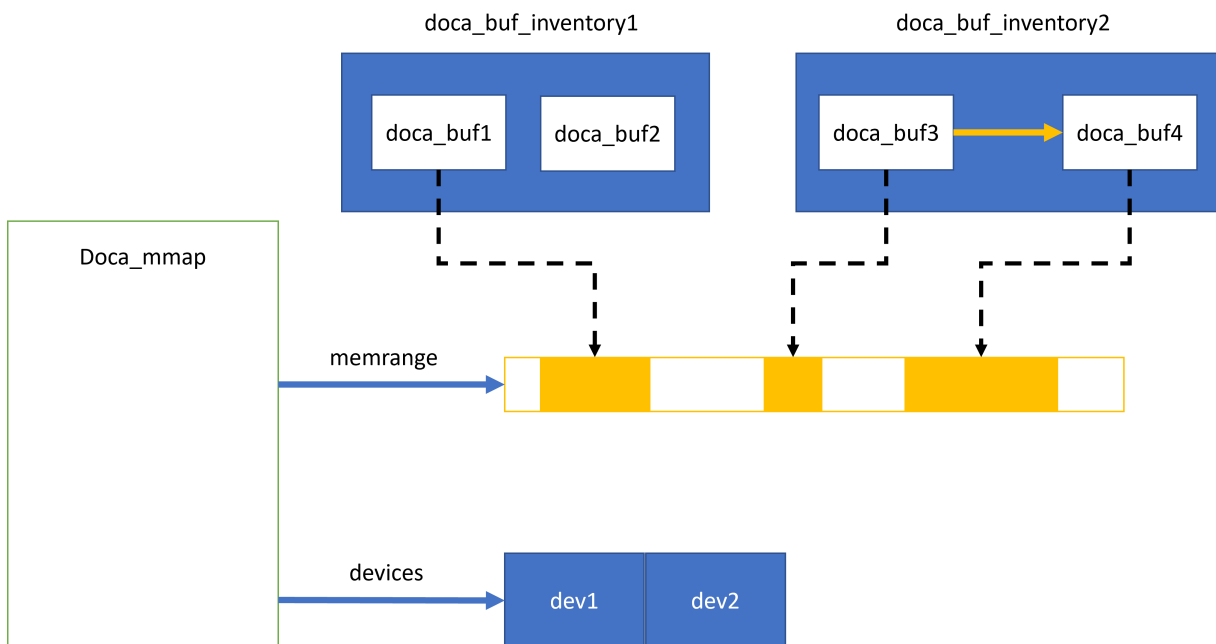
In the diagram, you may see two `doca_buf_inventory`s. Each `doca_buf` points to a portion of the memory buffer which is part of a `doca_mmap`. The mmap is populated with a 1 continuous memory buffer `memrange` and is mapped to 2 devices, `dev1` and `dev2`.

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 buffer in the mmap can be mapped to one device or more

▶ Devices in the mmap are restricted by access permissions defining how they can access the memory buffer

▶ `doca_buf` points to a specific memory buffer (or part of it) and holds the metadata for that buffer

▶ The internals of mapping and working with the device (e.g., memory registrations) is hidden from the application

▶ The host-mapped memory buffer can be accessed by DPU

3.3.2.   doca_mmap

`doca_mmap` is more than just a data buffer as it hides a lot of details (e.g., RDMA technicalities, device handling, etc.) from the application developer while giving the right level of abstraction to the software using it. `doca_mmap`is the best way to share memory between the host and 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 and mmap from export.

3.3.2.1.   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 sets a memory region to the mmap, granting the devices access to that memory region.
3. The application can specify the access permission for the devices to that memory range using `doca_mmap_set_permissions`.

   - ▶ If the mmap is used only locally, then `DOCA_ACCESS_LOCAL_*` must be specified

   - ▶ If the mmap is shared with the DPU (see step 4), then `DOCA_ACCESS_DPU_*` must be specified

   - ▶ If the mmap is shared with a remote RDMA target, then `DOCA_ACCESS_DPU_*` must be specified

4. To share the mmap with the DPU or the RDMA remote target, call `doca_mmap_export_dpu` or `doca_mmap_export_rdma` respectively. If appropriate access has not been provided, the export fails.
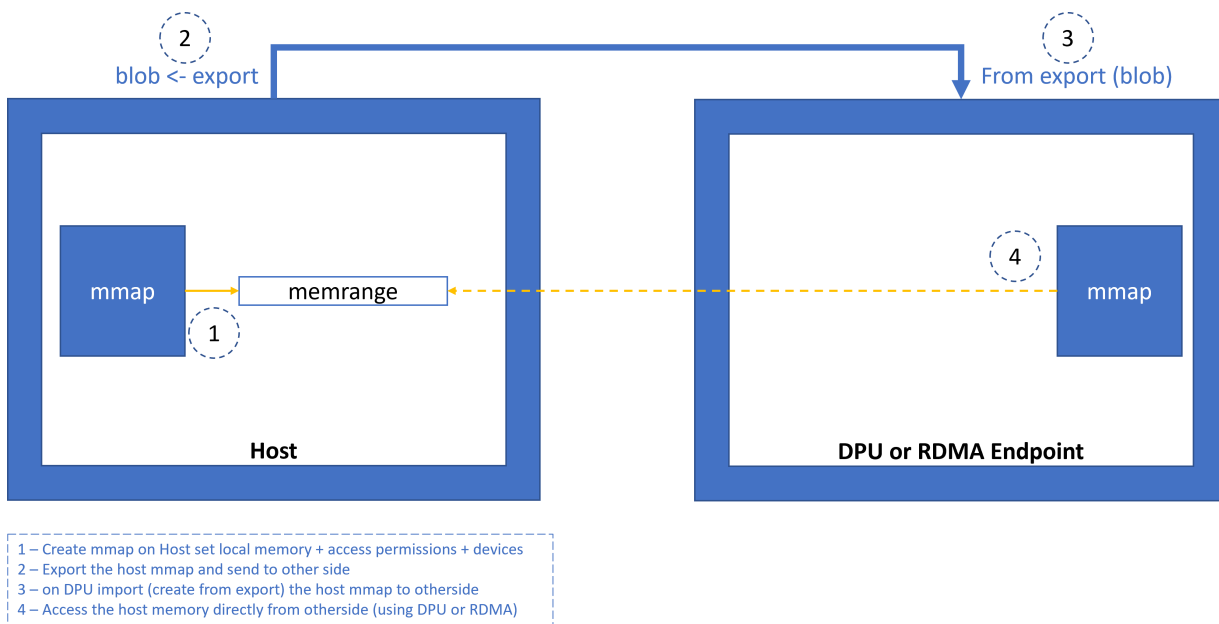
5. The generated blob from previous step can be shared out of band using a socket. If sharing with a DPU, it is recommended to use the DOCA Comm Channel. See the DMA Copy application for the exact flow.

> Note: The DPU can share its memory only through RDMA using `doca_mmap_export_rdma`.

3.3.2.2.   mmap from Export

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

1. The application receives a blob from the other side. The blob contains data returned from step 4 in the former bullet.
2. The application calls `doca_mmap_create_from_export` and receives a new mmap that represents memory defined by the other side.

1 – Create mmap on Host set local memory + access permissions + devices
2 – Export the host mmap and send to other side
3 – on DPU import (create from export) the host mmap to otherside
4 – Access the host memory directly from otherside (using DPU or RDMA)

Now the application can create `doca_buf` to point to this imported mmap and have direct access to the other machine's memory.

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

> Note: The host can only access memory exported through RDMA. This can be memory on a remote host, remote DPU, or DPU on same machine.

3.3.3. doca_buf and 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_buf`s that the application creates. Still, the `doca_buf`s 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.

> Note: If the application wishes to use a linked list buffer and concatenate several `doca_buf` to a scatter-gather list, the application is expected to make sure the library indeed supports a linked list buffer by calling `doca_dma_get_max_list_buf_num_elem` (this example checks linked-list support for DMA).

3.3.3.1. Inventory Types

| Inventory Type	Characteristics	When to Use	Notes
`doca_buf_inventory`	Multiple mmaps, flexible address, flexible buffer size.	When multiple sizes or mmaps are used.	Most common use case.
`doca_buf_array`	Single mmap, fixed buffer size. User receives an array of pointers to DOCA buffers.	Use for creating DOCA buffers on GPU.	`doca_buf_arr` is configured on the CPU and created on the GPU.
`doca_bufpool`	Single mmap, fixed buffer size, address not controlled by the user.	Use as a pool of buffers of the same characteristics when buffer address is not important.	Slightly faster than `doca_buf_inventory`.

3.3.4. 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 <u>Local Device and Representor Matching</u> for information on how to choose the `doca_dev` to add to mmap if exporting to DPU). This example adds a single `doca_dev` to the mmap and exports it so the DPU/RDMA endpoint can use it.

2. Import to the DPU/RDMA endpoint (e.g., use the mmap descriptor output parameter as input to `doca_mmap_create_from_export`).
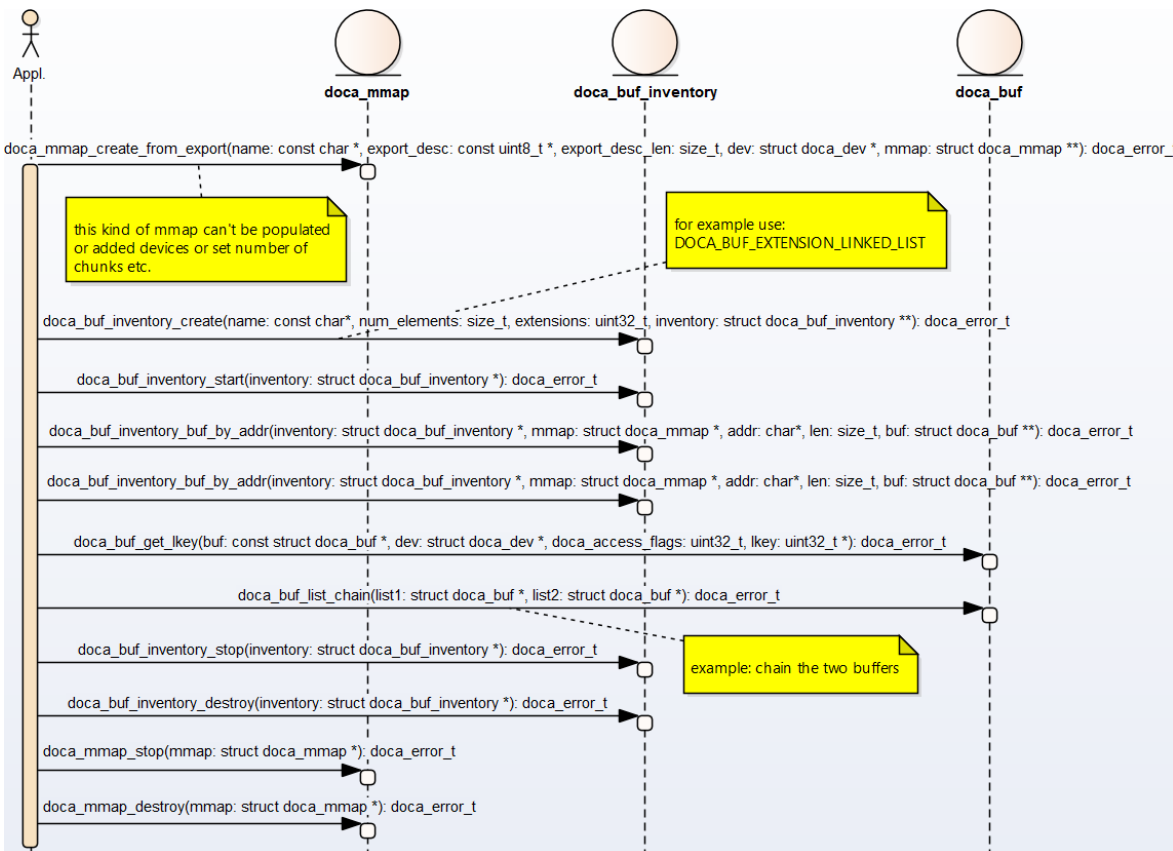
3.4. DOCA Execution Model

In DOCA, the workload involves transforming source data to destination data. The basic transformation is a DMA operation on the data which simply copies 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

▶ The job submission/execution flow/API is optimized for performance (latency)

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

▶ The basic object for executing the task is a `doca_job`. Each job is mapped to a specific DOCA library context.

▶ `doca_workq` represents a logical thread of execution for the application and jobs submitted to workq

> 📝 Note: WorkQ is not thread safe and it is expected that each WorkQ is managed by a single application thread (to submit a job and manage the progress engine).

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

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

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

▶ A WorkQ can either work in polling mode or event-driven mode, but not in both at same time

▶ Not all DOCA contexts support event-driven mode (i.e., can be added to a WorkQ that supports event-driven mode). The following API can query whether a context supports event-driven mode or not:

`doca_ctx_get_event_driven_supported(struct doca_ctx*, uint8)`

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.
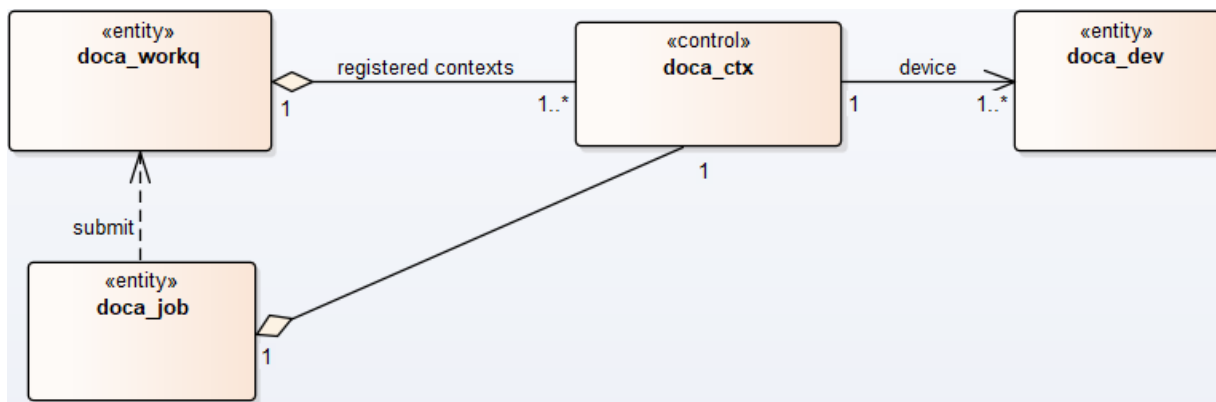
1. `doca_job` is associated to a relevant `doca_ctx` that executes the job (with the help of the relevant `doca_dev`).

2. `doca_job`, after it is initialized, is submitted to `doca_workq` for execution.
3. `doca_ctx`s 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:
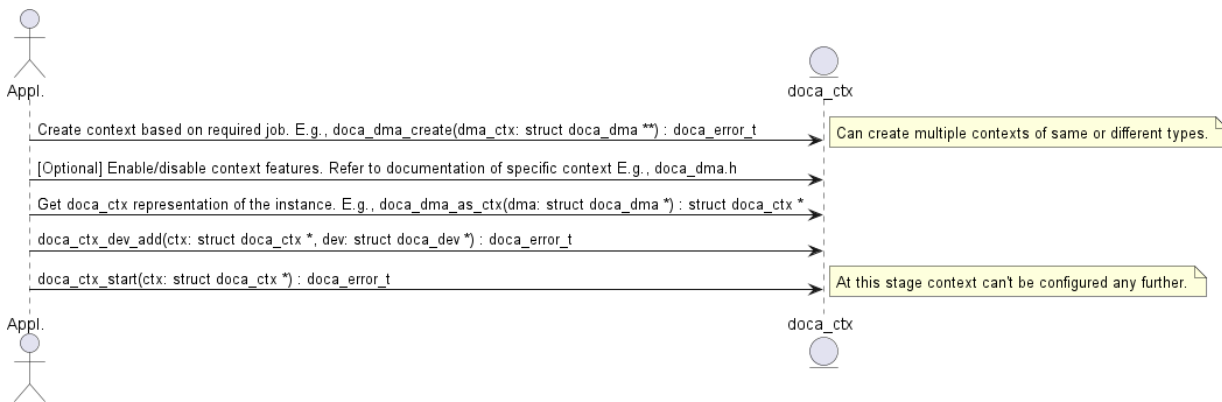
After the context is started, it can be used to enable the submission of jobs to a WorkQ based on the types of jobs that the context supports. See [DOCA WorkQ](#) for more information.

Context is a thread-safe object. Some contexts can be used across multiple WorkQs while others can only be added only to a single WorkQ. 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:
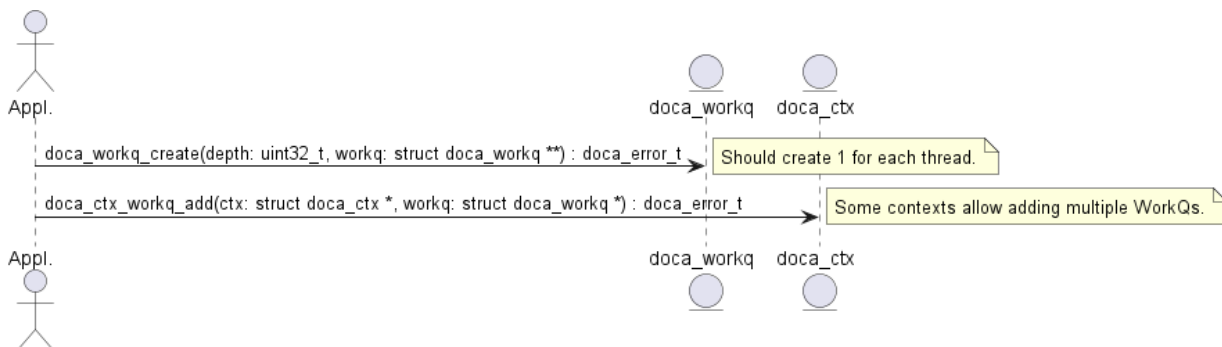
After the WorkQ has been created and added to a context, it can start accepting jobs that the context defines. Refer to the context documentation to find details such as whether the context supports adding multiple `doca_workq`s to the same context and what jobs can be submitted using the context.

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:
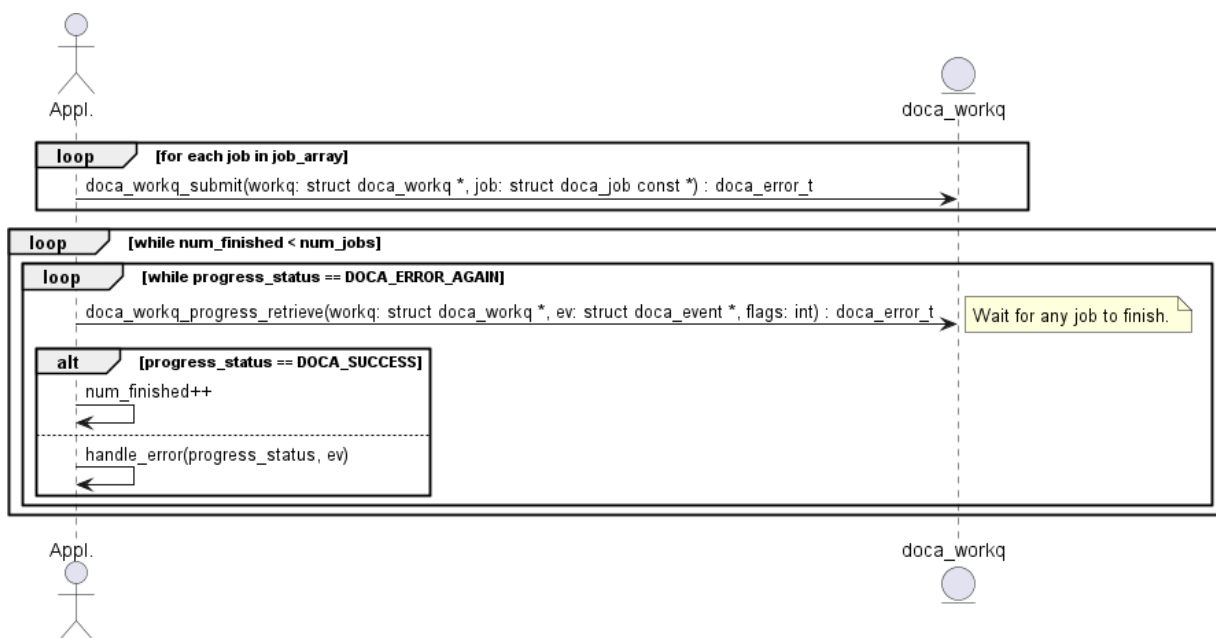
1. The application submits all jobs (one or more) and tracks the number of completed jobs to know if all jobs are done.
2. The application waits for a job to finish.

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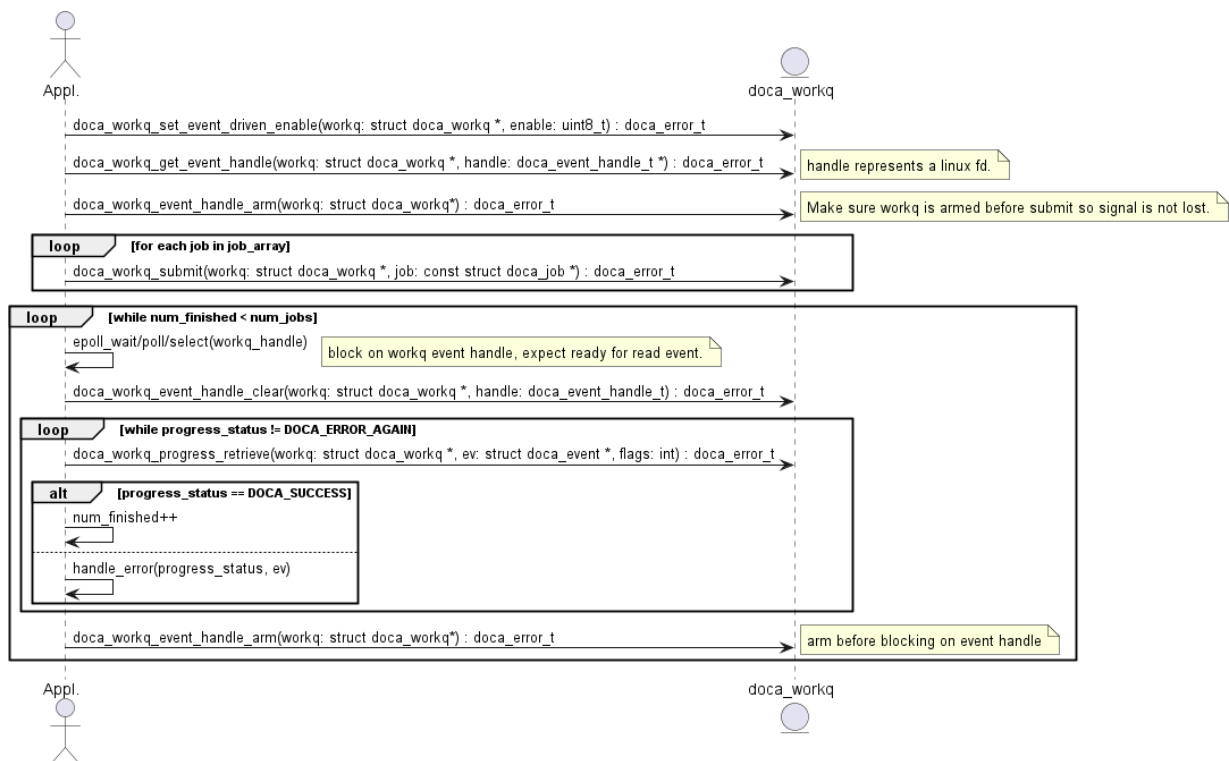
> 📝 Note: In this mode the application is always using the CPU even when it is doing nothing (during busy-wait).

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. DOCA Sync Event

> Important: DOCA Sync Event does not currently support DPA or GPU related features (see DOCA Sync Event Limitations and Disclaimers for more limitations).

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

DOCA Sync Event defines a subscriber and publisher:

▶ Publisher – the entity which updates (sets or increments) the event value

▶ Subscriber – the entity which gets and waits upon the sync event

Each DOCA Sync Event is configured with a single publisher location and a single subscriber location which can be the CPU or DPU.

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

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

3.4.6.1. Creating DOCA Sync Event

> Important: DOCA Sync Event does not currently support DPA or GPU related features (see [DOCA Sync Event Limitations and Disclaimers](#) for more limitations).

DOCA Sync Event exposes different types of handles per execution unit:

- ▶ CPU (host/DPU) handle type, `struct doca_sync_event`
- ▶ DPU handle type, `struct doca_sync_event`
- ▶ DPA handle type, `doca_dpa_dev_sync_event_t`
- ▶ GPU handle type, `doca_gpu_dev_sync_event_t`

Each one of the handle types above has its own dedicated API for creation.

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

DPU, DPA, and GPU handle creation is part of the DOCA Sync Event export flow, as illustrated in the following diagram:
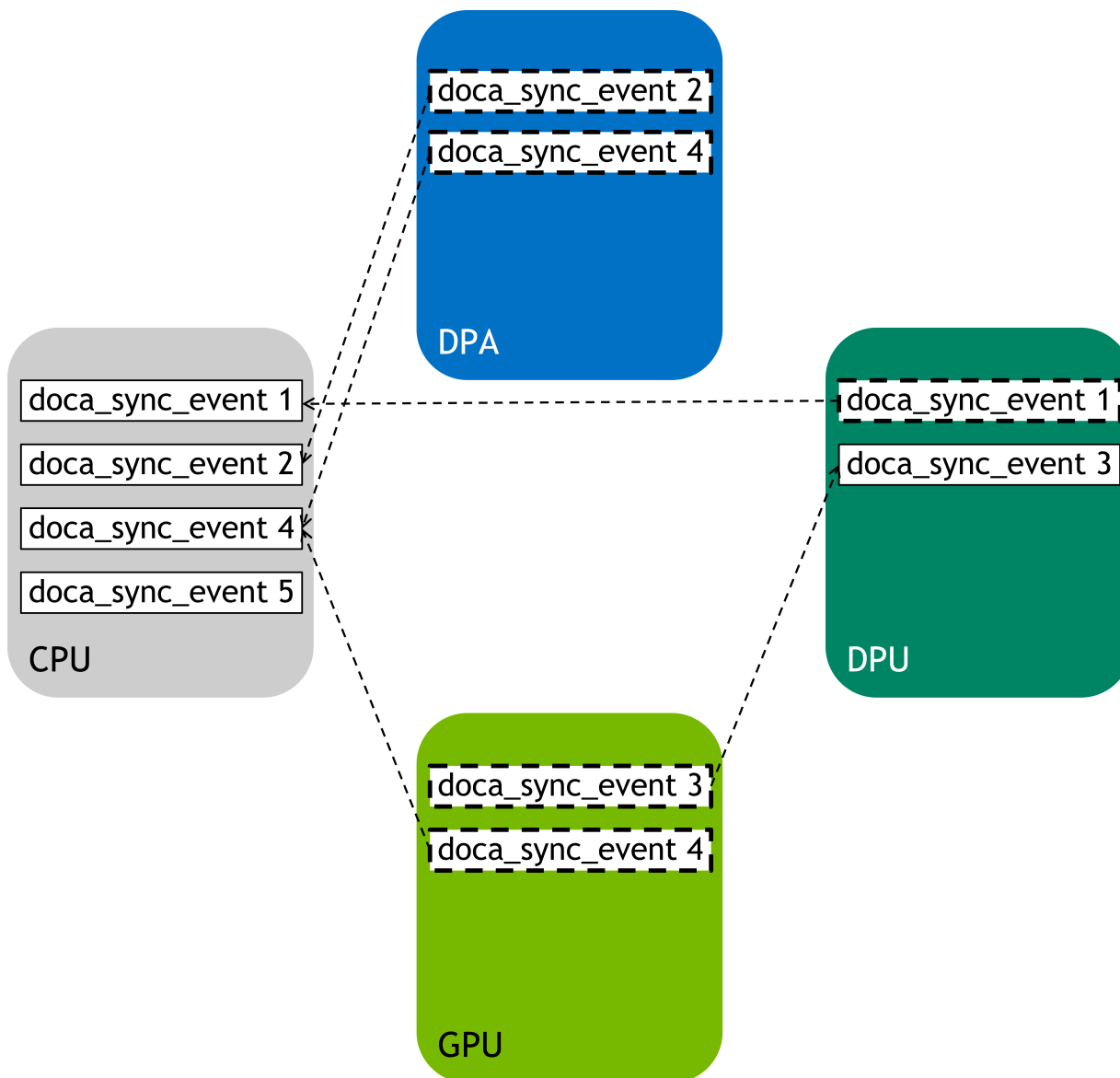
3.4.6.2. Configuring DOCA Sync Event

> **Important:** DOCA Sync Event does not currently support DPA or GPU related features (see DOCA Sync Event Limitations and Disclaimers for more limitations).

Once the DOCA Sync Event (CPU handle) is created, it must be configured by providing the runtime hints on the publisher and subscriber locations.

Both the subscriber and publisher locations must be configured using the following APIs:

▶ `doca_sync_event_publisher_add_location_<cpu|dpa|gpu>`

▶ `doca_sync_event_subscriber_add_location_<cpu|dpa|gpu>`

Another optional configuration, allowed for CPU-DPU sync events only is to provide an 8-byte host buffer to be used as the backing memory of the sync event. If set, it is the user responsibility to handle the memory (i.e., preserve the memory allocated during all DOCA Sync Event lifecycle and free it after DOCA Sync Event destruction).

3.4.6.3.   DOCA Sync Event Operation Modes

After creating a CPU handle and configuring it, the next step is to start the sync event.

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

> 🗎 Note: Once started, sync event operation mode cannot be changed.

3.4.6.3.1. Synchronous Mode

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

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

> 🗎 Note: An operation is considered done if the requested change fails and the exact error can be reported or if the requested change has been taken effect.

3.4.6.3.2. Asynchronous Mode

To start the sync event to operate in asynchronous mode, convert the sync event instance to `doca_ctx` by calling `doca_sync_event_as_ctx`. Then use DOCA CTX API to start the sync event and DOCA WorkQ API to submit operation jobs on the sync event (see DOCA WorkQ for more details).

3.4.6.4.   Exporting DOCA Sync Event to Another Execution Unit

> 🗎 Important: DOCA Sync Event does not currently support DPA or GPU related features (see DOCA Sync Event Limitations and Disclaimers for more limitations).

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

▶ DPA – `doca_sync_event_export_to_dpa` returns DOCA Sync Event DPA handle (`doca_dpa_dev_sync_event_t`) which later can be passed to the DPA Sync Event data path APIs from the DPA kernel.

▶ GPU – `doca_sync_event_export_to_gpu` returns DOCA Sync Event GPU handle (`doca_gpu_dev_sync_event_t`) which later can be passed to the GPU Sync Event data path APIs for the CUDA kernel.

▶ DPU – `doca_sync_event_export_to_dpu` returns opaque which later can be used from the DPU CPU to instantiate a DOCA Sync Event DPU handle (`struct doca_sync_event`) using the `doca_sync_event_create_from_export` function.

> 💬 Note: Users must verify the DPU sync event creation is supported from the DPU by using `doca_sync_event_get_create_from_export_supported`.

> 💬 Note: Once created from an export, the sync event DPU handle `struct doca_sync_event` cannot be configured but must be started before it is used.

> 💬 Note: Prior to calling any export function, users must first verify it is supported by calling the corresponding export capability getter: `doca_sync_event_get_export_to_dpa_supported`, `doca_sync_event_get_export_to_gpu_supported`, or `doca_sync_event_get_export_to_dpu_supported`.

3.4.6.5. DOCA Sync Event Data Path Operations

The DOCA Sync Event synchronization mechanism is achieved by using exposed data path operations. The API exposes a function for "writing" to the sync event and for "reading" the sync event.

The synchronous API is a set of functions which can be called directly by the user, while the asynchronous API is exposed by defining a corresponding `doca_job` type for each synchronous function to be submitted on a DOCA WorkQ in addition to a `doca_sync_event_result` type that can be retrieved from the DOCA WorkQ (see DOCA WorkQ and DOCA Context for more additional information).

> 💬 Note: Prior to asynchronous job submission, users must check if the job is supported using `doca_error_t doca_sync_event_job_get_supported`.

The following subsections describe the DOCA Sync Event data path operation with respect to these two operation modes.

3.4.6.5.1. Publishing on DOCA Sync Event

3.4.6.5.1.1. Setting DOCA Sync Event Value

Users can set DOCA Sync Event to a 64-bit value:

▶ Synchronously by calling `doca_sync_event_update_set`
▶ Asynchronously by submitting `doca_sync_event_job_update_set` job

3.4.6.5.1.2. Adding to DOCA Sync Event Value

Users can atomically increment the value of a DOCA Sync Event:

▶ Synchronously by calling `doca_sync_event_update_add`
▶ Asynchronously by submitting `doca_sync_event_job_update_add` job

3.4.6.5.2. Subscribing on DOCA Sync Event

3.4.6.5.2.1.   Getting DOCA Sync Event Value

Users can get the value of a DOCA Sync Event:

▶ Synchronously by calling `doca_sync_event_get`

▶ Asynchronously by submitting `doca_sync_event_job_get` job

3.4.6.5.2.2.   Waiting on DOCA Sync Event

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

Users can wait until a sync event reaches some specified value in a variety of ways.

Synchronously

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

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

> 🗩 Note: This wait method is supported only from the CPU.

Asynchronously

DOCA Sync Event exposes asynchronous wait method by defining a wait `doca_job`. Submitting `doca_job` on a `doca_workq` is an asynchronous non-blocking API.

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

▶ Blocking – get a `doca_workq` event handle to blocking wait on

▶ Polling – poll the wait job status by calling `doca_workq_progress_retrieve`

Refer to [DOCA WorkQ](#) and [DOCA Context](#) for more information.

3.4.6.6.   DOCA Sync Event Tear Down

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

> 🗩 Note: Users must validate active handles associated with the CPU handle during the teardown flow because DOCA Sync Event does not do that.

3.4.6.6.1. Stopping DOCA Sync Event

To stop a DOCA Sync Event:

▶ Synchronous – call `doca_sync_event_stop` on the CPU handle

▶ Asynchronous – stop the DOCA CTX associated with the DOCA Sync Event instance

> 🗨 Note: Stopping a DOCA Sync Event must be followed by destruction. Refer to Destroying DOCA Sync Event for details.

3.4.6.6.2. Destroying DOCA Sync Event

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

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

3.4.6.7. DOCA Sync Event Sample

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

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

3.4.6.7.1. Running DOCA Sync Event Sample

1. Refer to the following documents:

   ▶ NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

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

2. To build a given sample:
   ```
   cd /opt/mellanox/doca/samples/doca_common/sync_event_<host|dpu>
   meson build
   ninja -C build
   ```

   > 🗨 Note: The binary `doca_sync_event_<host|dpu>` is created under `./build/`.

3. Sample (e.g., `sync_event_dpu`) usage:
   ```
   Usage: doca_sync_event_dpu [DOCA Flags] [Program Flags]

   DOCA Flags:
     -h, --help                      Print a help synopsis
     -v, --version                   Print program version information
     -l, --log-level                 Set the log level for the program
    <CRITICAL=20, ERROR=30, WARNING=40, INFO=50, DEBUG=60>
   ```

```
Program Flags:
  -d, --dev-pci-addr                   Device PCI address
  -r, --rep-pci-addr                   DPU representor PCI address
  --async                              Start DOCA Sync Event in asynchronous
(synchronous by default)
  --qdepth                             DOCA WorkQ depth for asynchronous mode
  --atomic                             Update DOCA Sync Event using Add operation
(Set operation by default)
```

> 📝 Note: The flag `--rep-pci-addr` is relevant only for the DPU.

For additional information per sample, use the `-h` option:

```
./build/doca_sync_event_<host|dpu> -h
```

3.4.6.7.2. Samples

3.4.6.7.2.1.   Sync Event DPU

> 📝 Note: This sample should be run on the DPU before <u>Sync Event Host</u>.

This sample demonstrates creating a sync event from an export on the DPU which is associated with a sync event on the host, and interacting with the sync event to achieve synchronization between the host and DPU. This sample should be run on the DPU.

The sample logic includes:

1. Reading configuration files and saving their content into local buffers.
2. Locating and opening DOCA devices and DOCA representors matching the given PCIe addresses.
3. Initializing DOCA Comm Channel.
4. Receiving sync event blob through Comm Channel.
5. Creating sync event from export.
6. Starting the above sync event in the requested operation mode (synchronous or asynchronous)
7. Interacting with the sync event from the DPU:

   a). Waiting for signal from the host – synchronously or asynchronously (with busy wait polling) according to user input.
   b). Signaling the sync event for the host – synchronously or asynchronously, using set or atomic add, according to user input.
8. Cleaning all resources.

Reference:

▶ `/opt/mellanox/doca/samples/doca_common/sync_event_dpu/`
   `sync_event_dpu_sample.c`

▶ `/opt/mellanox/doca/samples/doca_common/sync_event_dpu/`
   `sync_event_dpu_main.c`

▶ `/opt/mellanox/doca/samples/doca_common/sync_event_dpu/meson.build`

3.4.6.7.2.2. *Sync Event Host*

> 💬 Note: This sample should be run on the DPU before Sync Event DPU.

This sample demonstrates how to initialize a sync event on the host to be shared with the DPU, how to export it to DPU, and how to interact with the sync event to achieve synchronization between the host and DPU. This sample should be run on the host.

The sample logic includes:

1. Reading configuration files and saving their content into local buffers.
2. Locating and opening the DOCA device matching the given PCIe address.
3. Creating and configuring the sync event to be shared with the DPU.
4. Starting the above sync event in the requested operation mode (synchronous or asynchronous).
5. Initializing DOCA Comm Channel.
6. Exporting the sync event and sending it through the Comm Channel.
7. Interacting with the sync event from the host:

   a). Signaling the sync event for the DPU – synchronously or asynchronously, using set or atomic add, according to user input.
   b). Waiting for a signal from the DPU – synchronously or asynchronously, with busy wait polling, according to user input.
8. Cleaning all resources.

Reference:

▶ `/opt/mellanox/doca/samples/doca_common/sync_event_host/`
  `sync_event_host_sample.c`

▶ `/opt/mellanox/doca/samples/doca_common/sync_event_host/`
  `sync_event_host_main.c`

▶ `/opt/mellanox/doca/samples/doca_common/sync_event_host/meson.build`

3.4.6.8. DOCA Sync Event Limitations and Disclaimers

▶ DOCA Sync Event API is considered thread unsafe

▶ GPU and DPA not currently supported

▶ Asynchronous wait (blocking/polling) is supported on NVIDIA® BlueField®-3 and NVIDIA® ConnectX®-7 and newer

3.4.7. DOCA Graph Execution

DOCA Graph facilitates running a set of actions (jobs, user callbacks, graphs) in specific order and dependencies. DOCA Graph runs on a DOCA work queue.

DOCA Graph creates graph instances that are submitted to the work queue (`doca_workq_graph_submit`).

3.4.7.1. Nodes

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

▶ Root nodes – a root node does not have a parent. The graph can have one or more root nodes. All roots begin running when the graph instance is submitted.

▶ Edge nodes – an edge node is a node that does not have child nodes connected to it. The graph instance is completed when all edge nodes are completed.

▶ Intermediate node – a node with parent and child nodes connected to it.

3.4.7.1.1. Context Node

A context node runs a specific DOCA job and uses a specific DOCA context (`doca_ctx`). The context must be added to the work queue before the graph is started.

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

3.4.7.1.2. User Node

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

3.4.7.1.3. Sub-graph Node

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

3.4.7.2. Using DOCA Graph

1. Create the graph using `doca_graph_create`.
2. Create the graph nodes (e.g., `doca_graph_ctx_node_create`).
3. Define dependencies using `doca_graph_add_dependency`.

> 📝 Note: DOCA graph does not support circle dependencies (e.g., A => B => A).

4. Start the graph using `doca_graph_start`.
5. Add the graph to a work queue using `doca_graph_workq_add`.
6. Create the graph instance using `doca_graph_instance_create`.
7. Set the nodes data (e.g., `doca_graph_instance_set_ctx_node_data`).
8. Submit the graph instance to the work queue using `doca_workq_graph_submit`.
9. Call `doca_workq_progress_retrieve` until it returns `DOCA_SUCCESS`:

   ▶ `doca_workq_progress_retrieve` returns `DOCA_ERROR_AGAIN` for every node and returns `DOCA_SUCCESS` when the graph instance is completed

   ▶ `doca_event::type == DOCA_GRAPH_JOB` indicates that a graph instance is completed

- ▶ `doca_event::result::u64` contains the graph instance status (`0` implies `DOCA_SUCCESS`)
- ▶ Work queue can run graph instances and standalone jobs simultaneously

3.4.7.3. DOCA Graph Limitations

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

3.4.7.4. DOCA Graph Sample

The graph sample is based on the DOCA SHA and DOCA DMA libraries. The sample calculates a SHA value and copies a source buffer to a destination buffer in parallel.

The graph ends with a user callback node that prints the SHA value and compares the source with the DMA destination.

3.4.7.4.1. Running DOCA Graph Sample

1. Refer to the following documents:

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

2. To build a given sample:
   ```
   cd /opt/mellanox/doca/samples/doca_common/graph/
   meson build
   ninja -C build
   ```

   > 📄 Note: The binary `doca_sync_event_<host|dpu>` is created under `./build/`.

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

   No parameters required.

3.4.8. 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 memcpy, then refer to `doca_dma.h`).

The following diagram shows how an application is expected to handle error from `doca_workq_progress_retrieve`:
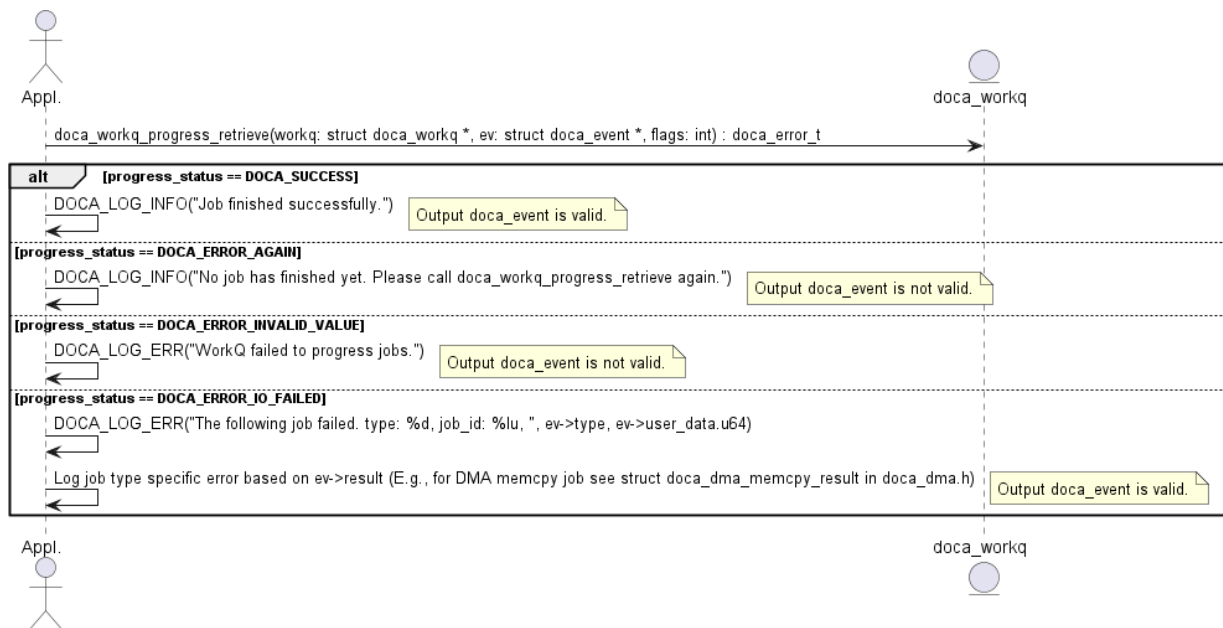
3.5.   Object Life Cycle

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

1. The object is allocated by DOCA so it is opaque for the application (e.g., `doca_buf_inventory_create`, `doca_mmap_create`).
2. The application initializes the object and sets the desired properties (e.g., `doca_mmap_set_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`).

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

1. Memory is allocated on Machine1.
2. Mmap is created and is provided memory from step 1.
3. Mmap is exported to the Machine2 pinning the memory.
4. On the Machine2, an imported mmap is created and holds a reference to actual memory residing on Machine1.
5. Imported mmap can be used by Machine2 to allocate buffers.
6. Imported mmap is destroyed.
7. Exported mmap is destroyed.
8. Original memory is destroyed.

3.6.   RDMA Bridge

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

3.6.1.   Requirements and Considerations

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

3.6.2.   DOCA Core Objects to RDMA Core Objects Mapping

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

RDMA Core Object	DOCA Equivalent	RDMA Object to DOCA Object	DOCA Object to RDMA Object
`ibv_pd`	`doca_dev`	`doca_dev_open_from_pd`	`doca_dev_get_pd`
`ibv_mr`	`doca_buf`	–	`doca_buf_get_mkey`

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_dpu_supported(const struct
 doca_devinfo *devinfo, uint8_t *mmap_export);
        doca_error_t
 doca_devinfo_get_is_mmap_from_export_dpu_supported(const struct doca_devinfo
 *devinfo, uint8_t *from_export);
```

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