DOCA Common

Table of contents

DOCA Common is comprised of the following libraries:

- DOCA Core
- DOCA Log

DOCA Core

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

Introduction

> **Note**
>
> The DOCA Core library is supported at beta level.

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

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

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

DOCA Core can be divided into the following software modules:

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

DOCA Core Module	Description

Device handling	• Queries device information (host-side and BlueField) and device capabilities (e.g., device's PCIe BDF address)
	  ◦ On BlueField
	  ▪ Gets local BlueField devices
	  ▪ Gets representors list (representing host local devices)
	  ◦ On the host
	  ▪ Gets local devices
	  ◦ Queries device capabilities and library capabilities
	• Opens and uses the selected device representor
	• Relevant entities – `doca_devinfo`, `doca_devinfo_rep`, `doca_dev`, `doca_dev_rep`
	• Header files – `doca_dev.h`
	
	Info
	There is a symmetry between device entities on host and its representor (on BlueField). The convention of adding `rep` to the API or the object hints that it is representor-specific.

Memory management	• Handles optimized memory pools to be used by applications and enables sharing resources between DOCA libraries (while hiding hardware-related technicalities) • Data buffer services (e.g., linked list of buffers to support scatter-gather list) • Maps host memory to BlueField for direct access • Relevant entities – `doca_buf`, `doca_mmap`, `doca_buf_inventory`, `doca_buf_array`, `doca_bufpool` • Header files – `doca_buf.h`, `doca_buf_inventory.h`, `doca_mmap.h`, `doca_buf_array.h`, `doca_bufpool`

DOCA Core Module	Description
Progress engine and task execution	• Enables submitting tasks to DOCA libraries and track task progress (supports both polling mode and event-driven mode) • Relevant ent ities – `doca_ctx`, `doca_task`, `doca_event`, `doca_event_handle_t`, `doca_pe` • Header files – `doca_ctx.h`
Sync events	• Sync events are used to synchronize different processors (e.g., synchronize BlueField and host) • header files – `doca_dpa_sync_event.h`, `doca_sync_event.h`

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

Prerequisites

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

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

Changes From Previous Releases

Changes in 2.8.0

Added

- `doca_bitfield.h`

- `doca_error_t doca_buf_inventory_expand(struct doca_buf_inventory *inventory, uint32_t num_elements)`
- `void doca_ctx_flush_tasks(struct doca_ctx *ctx)`
- `doca_error_t doca_devinfo_cap_is_notification_moderation_supported(const struct doca_devinfo *devinfo, uint8_t *is_notification_moderation_supported)`
- New DOCA errors: `DOCA_ERROR_AUTHENTICATION`, `DOCA_ERROR_BAD_CONFIG`, `DOCA_ERROR_SKIPPED`
- `doca_error_t doca_task_submit_ex(struct doca_task *task, uint32_t flags)`
- `doca_error_t doca_pe_set_notification_affinity(struct doca_pe *pe, uint32_t core_id)`
- `doca_error_t doca_pe_is_set_notification_affinity_supported(const struct doca_devinfo *devinfo, uint8_t *is_set_notification_affinity_supported`

Changed

- `doca_error_t doca_devinfo_get_active_rate(const struct doca_devinfo *devinfo, doubleuint64_t *active_rate); // Gb/s -> bits/s`
- `doca_buf_set_data_len` is STABLE API
- Imported mmap can be exported to RDMA

Architecture

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

General

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

The flow is as follows:

1. Create the object instance (e.g., `doca_mmap_create`).
2. Configure the instance (e.g., `doca_mmap_set_memory_range`).
3. Start the instance (e.g., `doca_mmap_start`).

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

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

doca_error_t

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

```
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_ALREADY_EXIST,               /**< Resource already exist */
        DOCA_ERROR_FULL,                        /**< No more space in resource */
        DOCA_ERROR_EMPTY,                       /**< No entry is available in
resource */
        DOCA_ERROR_IN_PROGRESS,                 /**< Operation is in progress */
        DOCA_ERROR_TOO_BIG,                     /**< Requested operation too big to
be contained */
 } doca_error_t;
```

See `doca_error.h` for more.

Generic Structures/Enum

The following types are common across all DOCA APIs.

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

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

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

For more see `doca_types.h`.

DOCA Device

Local Device and Representor

Prerequisites

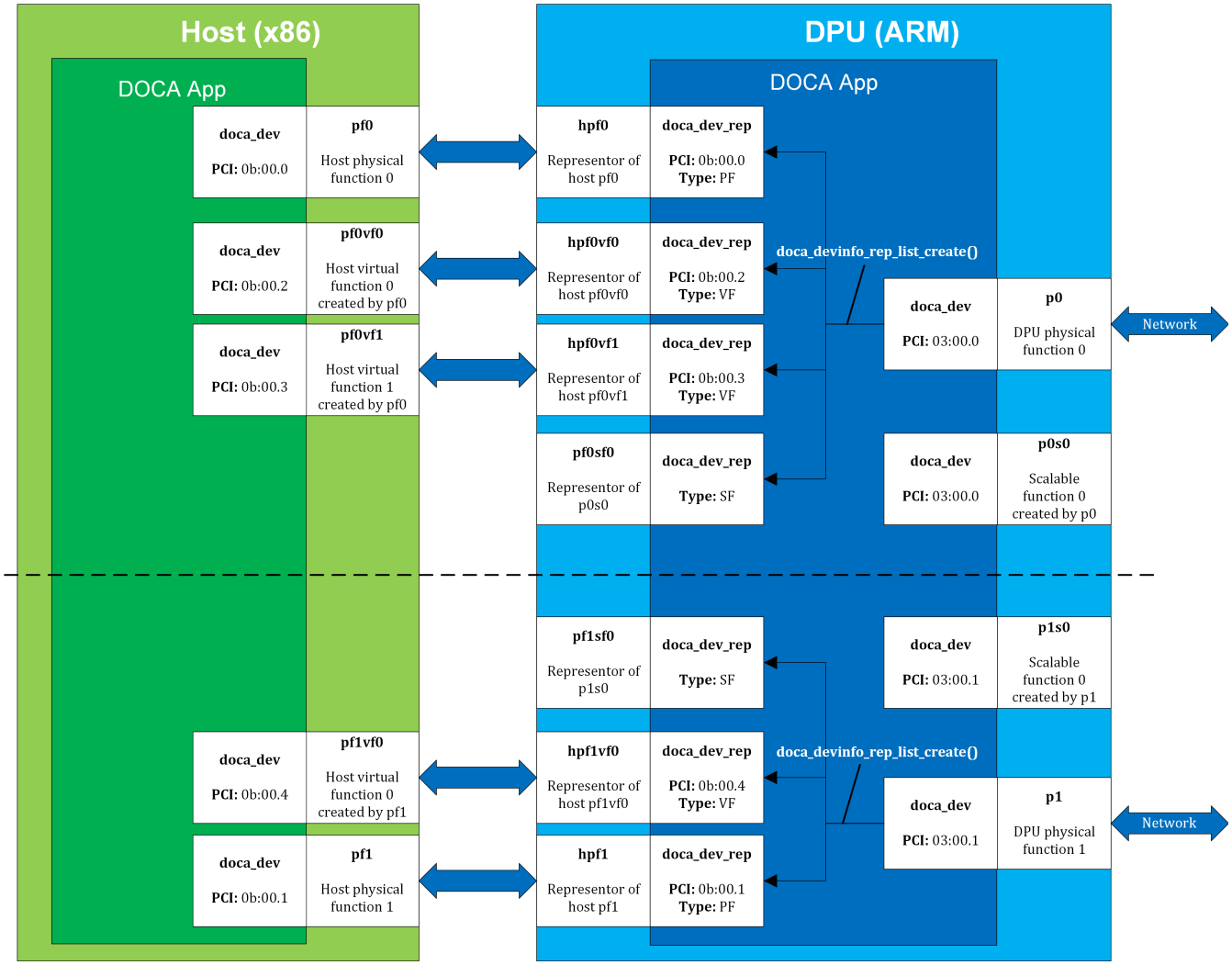
For the representors model, BlueField must be operated in DPU mode. See NVIDIA BlueField Modes of Operation.

Topology

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

- Local device – this is an actual device exposed in the local system (BlueField or host) and can perform DOCA library processing tasks.
- Representor device – this is a representation of a local device. The local device is usually on the host (except for SFs) and the representor is always on BlueField side (a proxy on BlueField for the host-side device).

The following figure provides an example topology:



The diagram shows a BlueField device (on the right side of the figure) connected to a host (on the left side of the figure). The host topology consists of two physical functions (PF0 and PF1). Furthermore, PF0 has two child virtual functions, VF0 and VF1. PF1 has only one

VF associated with it, VF0. Using the DOCA SDK API, the user gets these five devices as local devices on the host.

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

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

- 7 representor devices –
    - 5 representor devices shown as arrows to/from the host (devices with the prefix `hpf*`) in the diagram
    - 2 representor devices for the SF devices, `pf0sf0` and `pf1sf0`
- 2 local SF devices (not the SF representors), `p0s0` and `p1s0`

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

Local Device and Representor Matching

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

Device to Select on Host When Using doca_mmap_export_dpu()	BlueField Matching Representor	Device to Select on BlueField When Using doca_mmap_create_from_export()
pf0 – 0b:00.0	hpf0 – 0b:00.0	p0 – 03:00.0
pf0vf0 – 0b:00.2	hpf0vf0 – 0b:00.2	

Device to Select on Host When Using doca_mmap_export_dpu()	BlueField Matching Representor	Device to Select on BlueField When Using doca_mmap_create_from_export()
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	

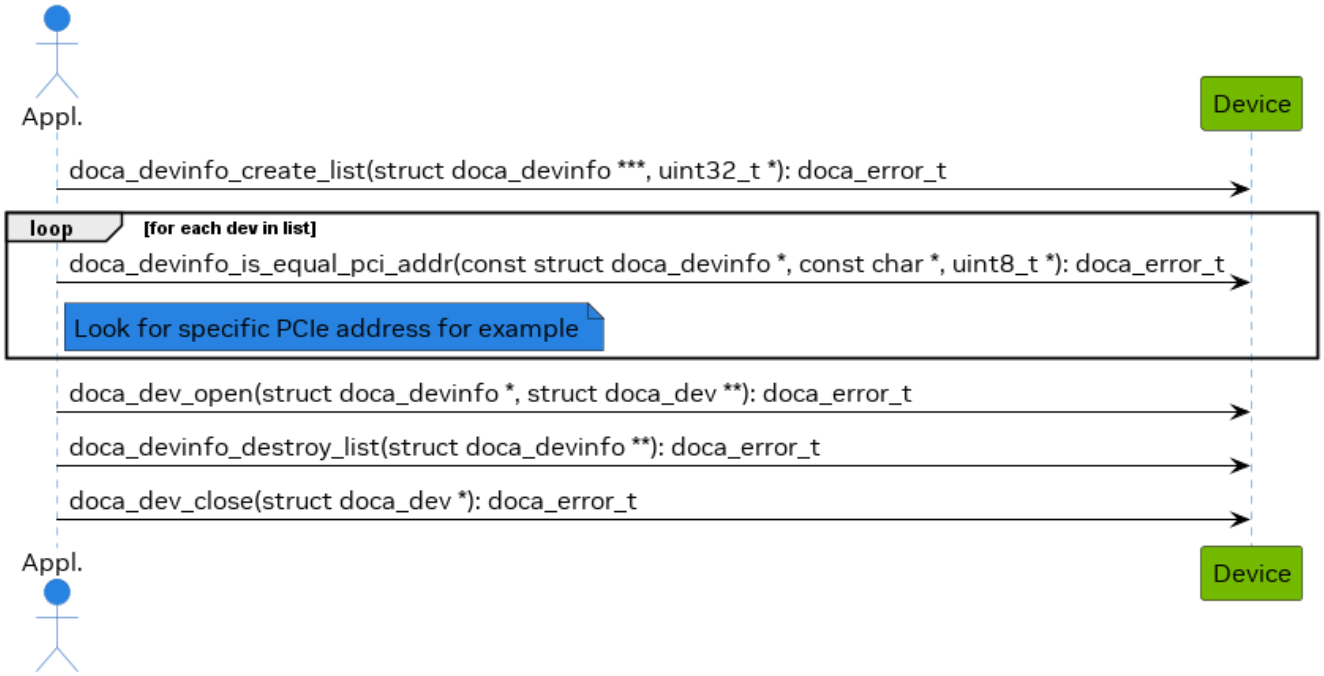
Expected Flow

Device Discovery

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

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

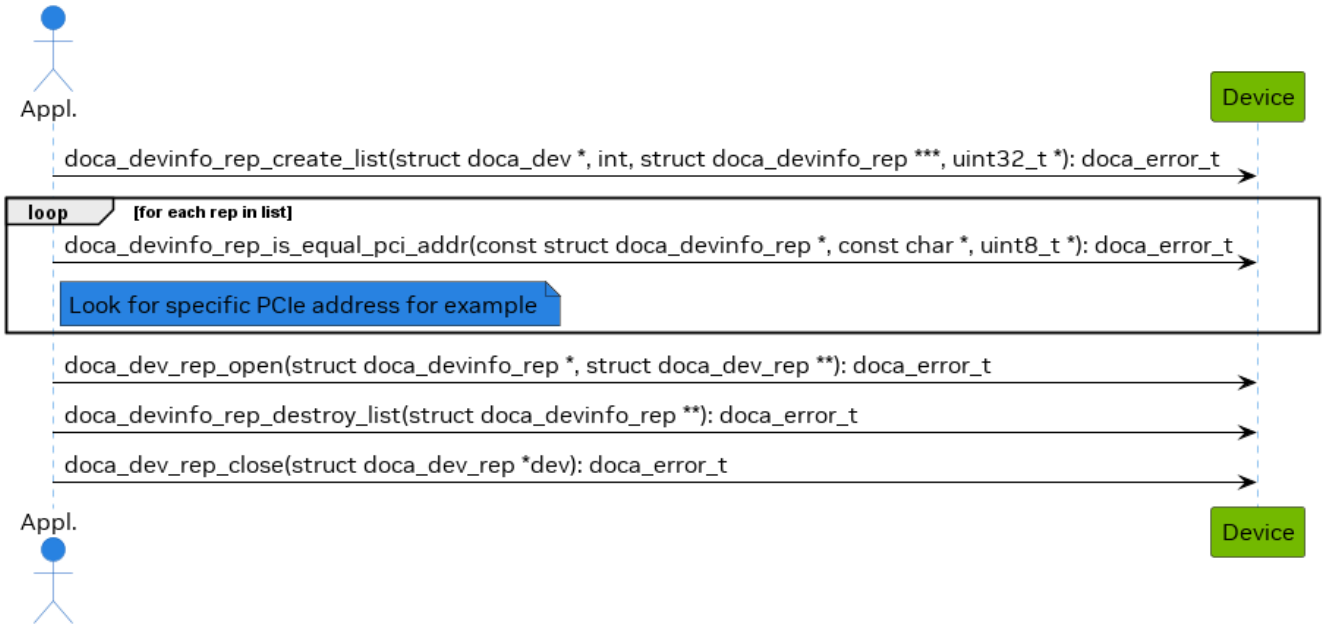
An application can decide which device to select based on capabilities, the DOCA Core API, and every other library which provides a wide range of device capabilities. The flow is as follows:



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

Representor Device Discovery

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



1. The application "knows" which device it wants to use (e.g., by its PCIe BDF address). On the host, it can be done using DOCA Core API or OS services.
2. On the BlueField side, the application gets a list of device representors for a specific BlueField local device.

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

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

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

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

> **Note**
>
> Regarding representor device property caching, the function `doca_devinfo_rep_create_list` provides a snapshot of the DOCA representor device properties when it is called. If any representor's properties are changed dynamically (e.g., BDF address changes after bus reset), t he device properties that the function returns would not reflect this change. One should create the list again to get the updated properties of the representors.

DOCA Memory Subsystem

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

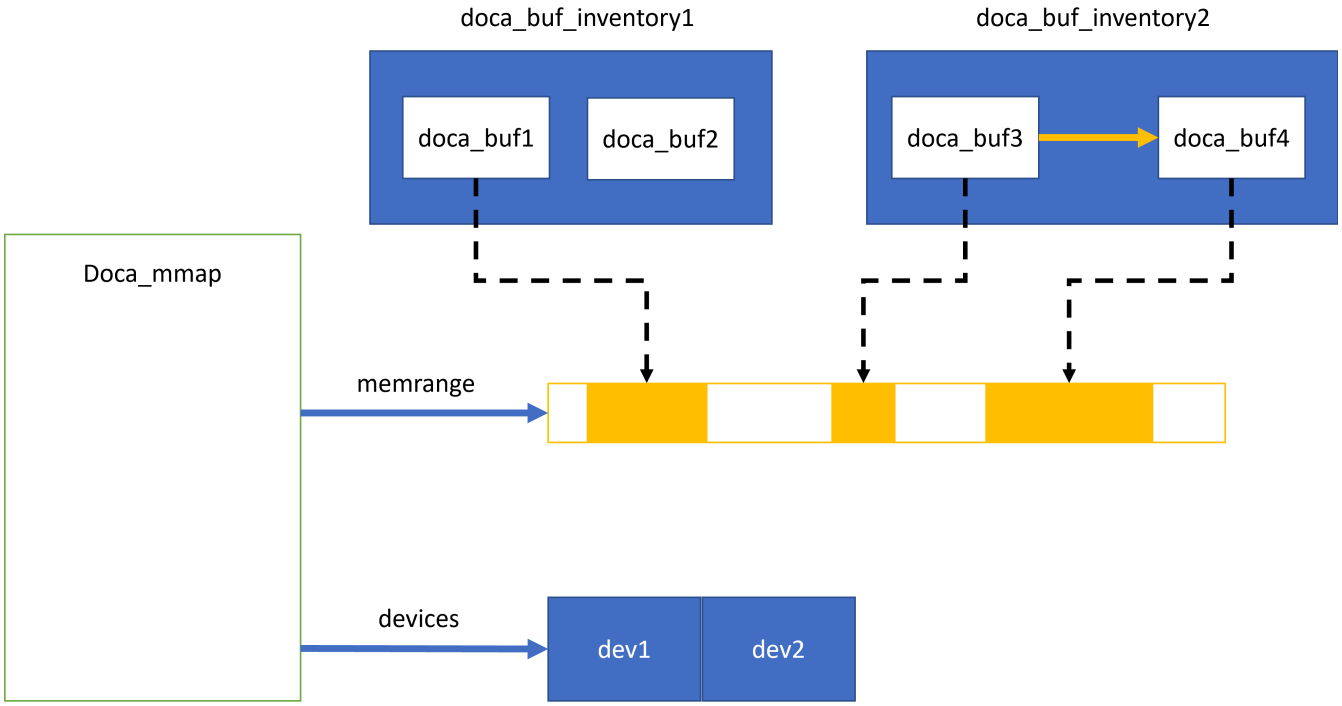
DOCA memory has the following main components:

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

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

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

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



In the diagram, you may see two `doca_buf_inventory`s. Each `doca_buf` points to a portion of the memory buffer which is part of a `doca_mmap`. The mmap is populated with one continuous memory buffer `memrange` and is mapped to two devices, `dev1` and `dev2`.

Requirements and Considerations

- The DOCA memory subsystem mandates the usage of pools as opposed to dynamic allocation
    - Pool for `doca_buf` → `doca_buf_inventory`

    - Pool for data memory → `doca_mmap`

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

doca_mmap

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

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

Local mmap

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

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

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

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

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

5. The application starts the mmap.

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

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

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

7. The generated blob from the previous step can be shared out of band using a socket. If shared with a BlueField, it is recommended to use the DOCA Comm

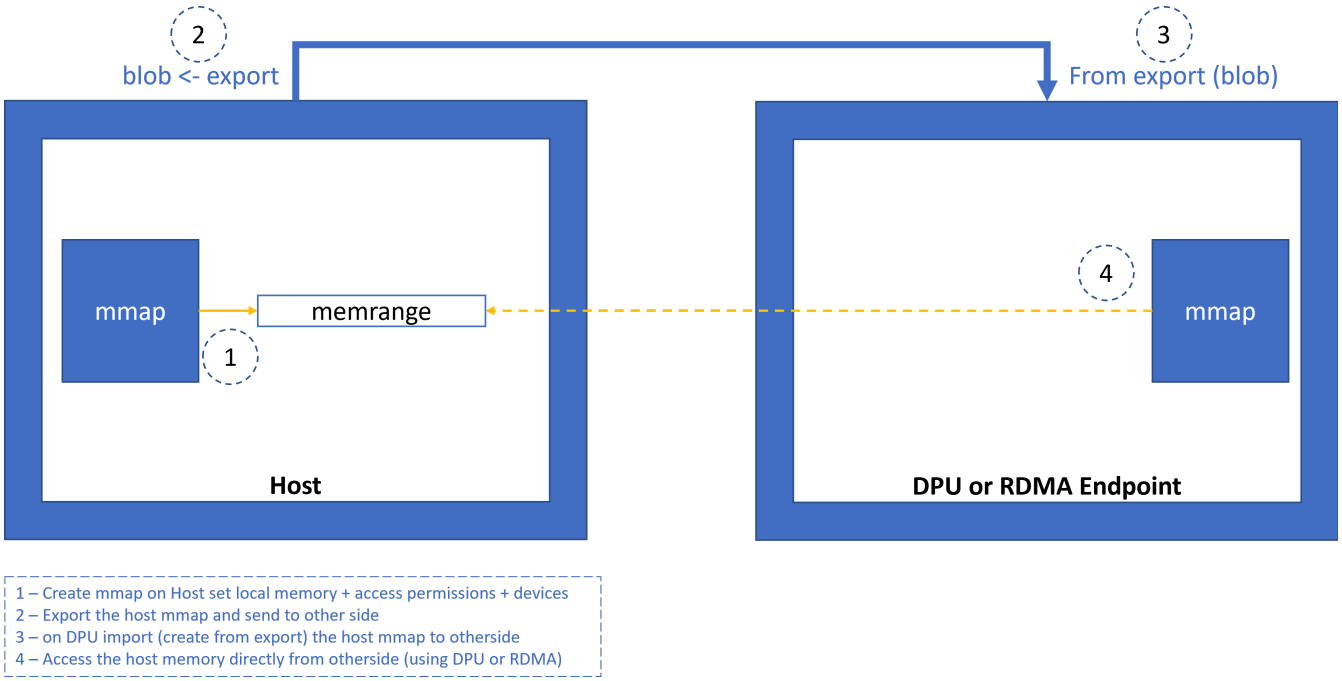
Channel instead. See the DMA Copy application for the exact flow.

mmap from Export

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

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

2. The application calls `doca_mmap_create_from_export` and receives a new mmap that represents memory defined by the other side.



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

> **Note**
>
> BlueField can access memory exported to BlueField if the exporter is a host on the same machine. Or it can access memory exported

through RDMA which can be on the same machine, a remote host, or on a remote BlueField.

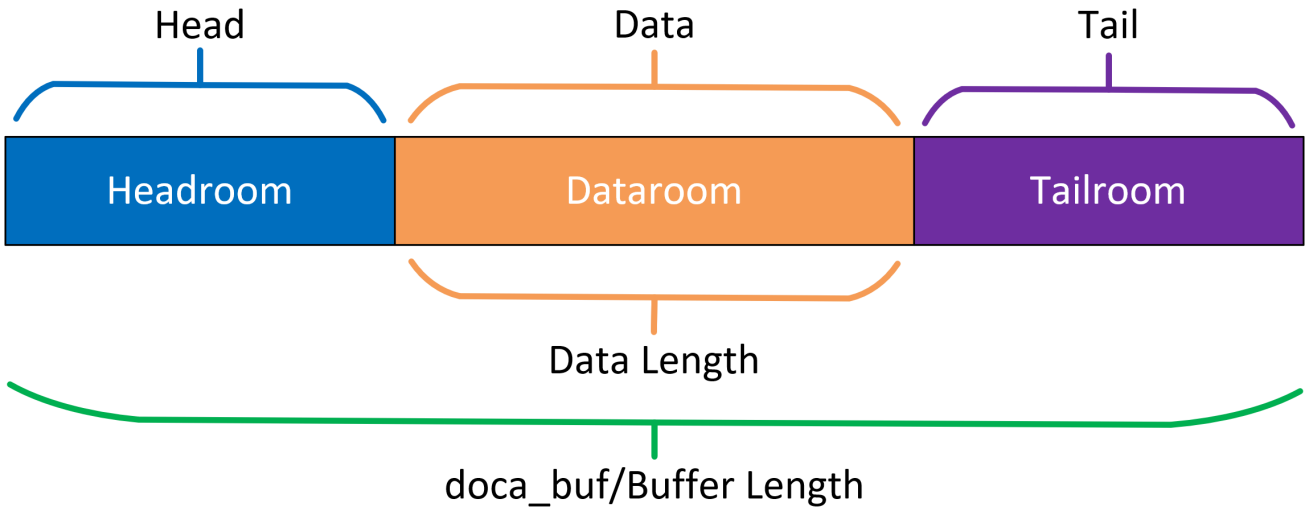
Note

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

Buffers

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

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



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

Buffer Considerations

- There are multiple ways to create the buffer but, once created, it behaves in the same way (see section "Inventories").
- The buffer may reference memory that is not accessible by the CPU (e.g., RDMA memory)
- The buffer is a thread-unsafe object
- The buffer can be used to represent non-continuous memory regions (scatter/gather list)
- The buffer does not own nor manage the data it references. Freeing a buffer does not affect the underlying memory.

Headroom

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

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

Dataroom

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

Tailroom

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

Buffer as Source

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

Buffer as Destination

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

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

Scatter/Gather List

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

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

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

reached.

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

When using the buffer list as a source, the data of each buffer (in the dataroom) is gathered and used as continuous data for the given operation.

When using the buffer list as destination, data is scattered in the tailroom of the buffers in the list until it is all written (some buffers may not be written to).

Buffer Use Cases

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

Buffer use-case considerations:

- If the application wishes to use a linked list buffer and concatenate several `doca_buf`s to a scatter/gather list, the application is expected to ensure the library indeed supports a linked list buffer. For example, to check linked-list support for DMA memcpy task, the application may call `doca_dma_cap_task_memcpy_get_max_buf_list_len()`.
- Operations made on the buffer's data are not atomic unless stated otherwise
- Once a buffer has been passed to the library as part of the task, ownership of the buffer moves to the library until that task is complete

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

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

Inventories

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

Inventory Considerations

- All inventories adhere to zero allocation after start.
- Allocation of a DOCA buffer requires a data source and an inventory.
    - The data source defines where the data resides, what can access it, and with what permissions.
    - The data source must be created by the application. For creation of mmaps, see `doca_mmap`.
- The inventory describes the allocation pattern of the buffers, such as, random access or pool, variable-size or fixed-size buffers, and continuous or non-continuous memory.
- Some inventories require providing the data source, `doca_mmap`, when allocating the buffers, others require it on creation of the inventory.
- All inventory types are thread-unsafe.

Inventory Types

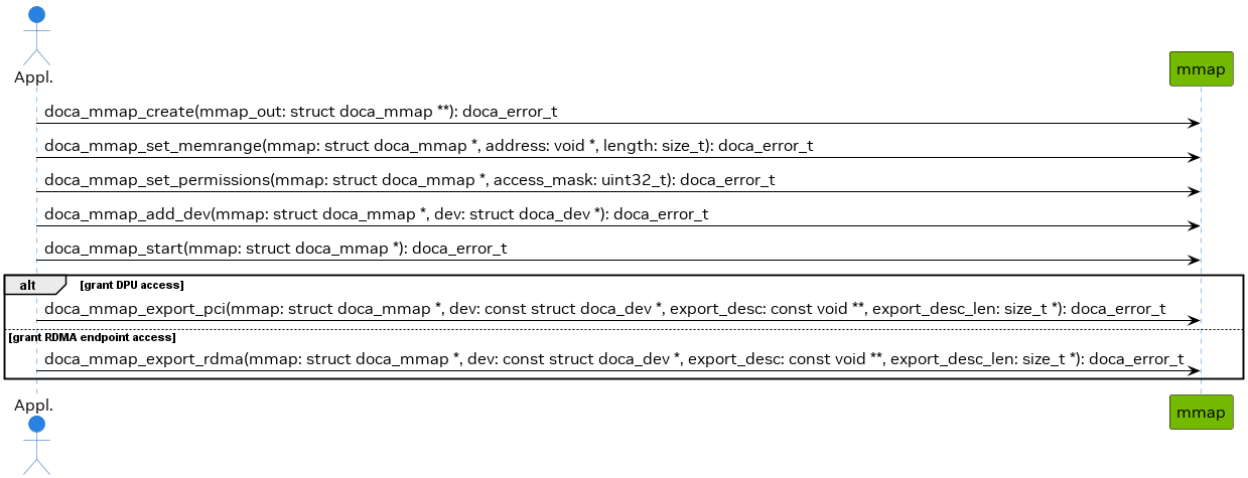
Inventory Type	Characteristics	When to Use	Notes
`doca_buf_inventory`	Multiple mmaps, flexible address, flexible buffer size.	When multiple sizes or mmaps are used.	Most common use case.

Inventory Type	Characteristics	When to Use	Notes
`doca_buf_array`	Single mmap, fixed buffer size. User receives an array of pointers to DOCA buffers. In case of DPA, mmap and buffer size can be unconfigured and later can be set from the DPA.	Use for creating DOCA buffers on GPU or DPA.	`doca_buf_arr` can be configured on the CPU and created on the GPU or DPA
`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` .

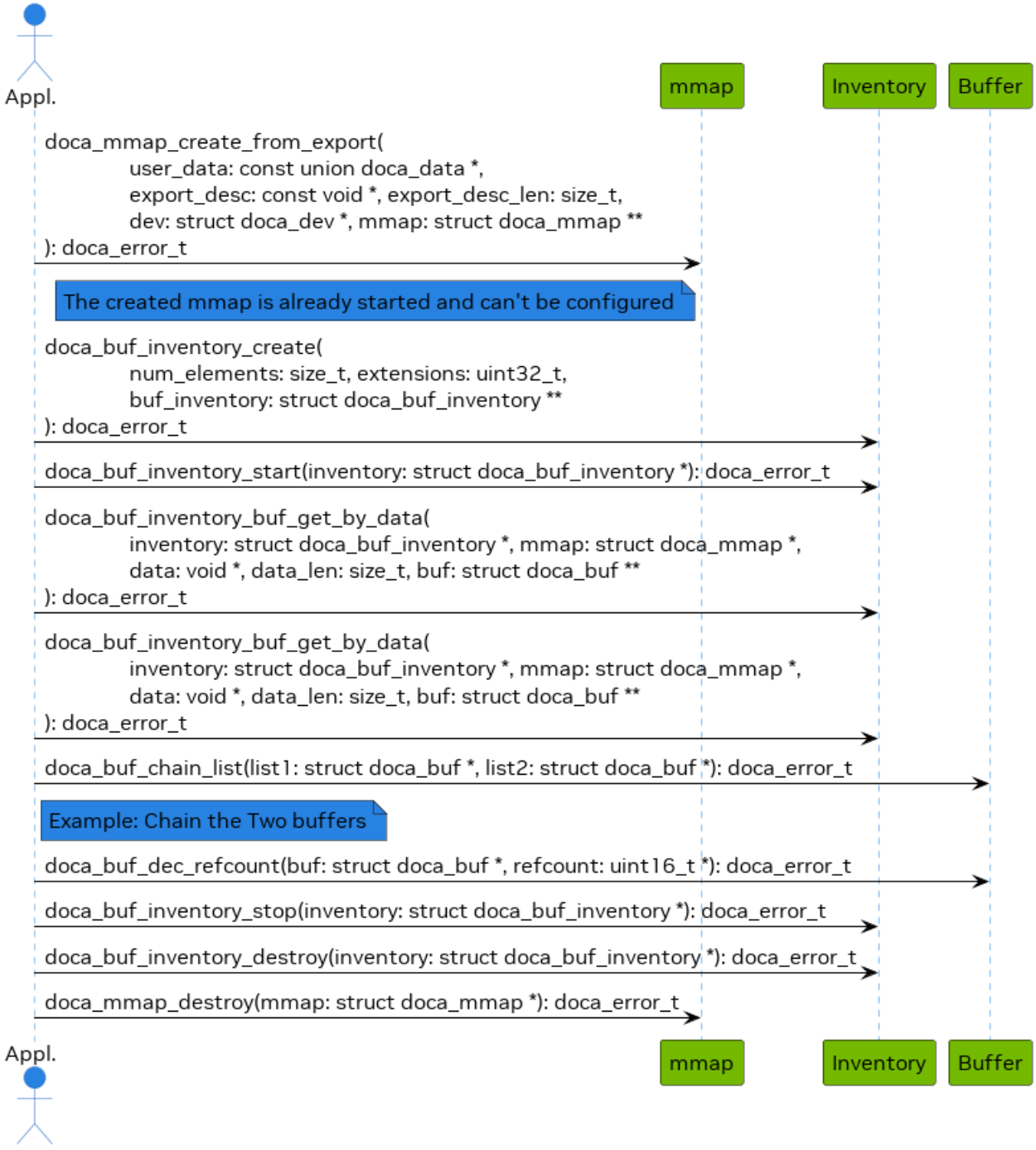
Example Flow

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

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



2. Import to the BlueField/RDMA endpoint (e.g., use the mmap descriptor output parameter as input to `doca_mmap_create_from_export`).



DOCA Execution Model

The execution model is based on hardware processing on data and application threads. DOCA does not create an internal thread for processing data.

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

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

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

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

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

`doca_pe` introduces three main operations:

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

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

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

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

Requirements and Considerations

- The task submission/execution flow/API is optimized for performance (latency)
- DOCA does not manage internal (operating system) threads. Rather, progress is managed by application resources (calling DOCA API in polling mode or waiting on DOCA notification in event-driven mode).
- The basic object for executing the task is a `doca_task`. Each task is allocated from a specific DOCA library context.
- `doca_pe` represents a logical thread of execution for the application and tasks submitted to the progress engine (PE)

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

- Execution-related elements (e.g., `doca_pe`, `doca_ctx`, `doca_task`) are opaque and the application performs minimal initialization/configuration before using these elements
- A task submitted to PE can fail (even after the submission succeeds). In some cases, it is possible to recover from the error. In other cases, the only option is to reinitialize the relevant objects.
- PE does not guarantee order (i.e., tasks submitted in certain order might finish out-of-order). If the application requires order, it must impose it (e.g., submit a dependent task once the previous task is done).
- A PE can either work in polling mode or event-driven mode, but not in both at same time

- All DOCA contexts support polling mode (i.e., can be added to a PE that supports polling mode)

DOCA Context

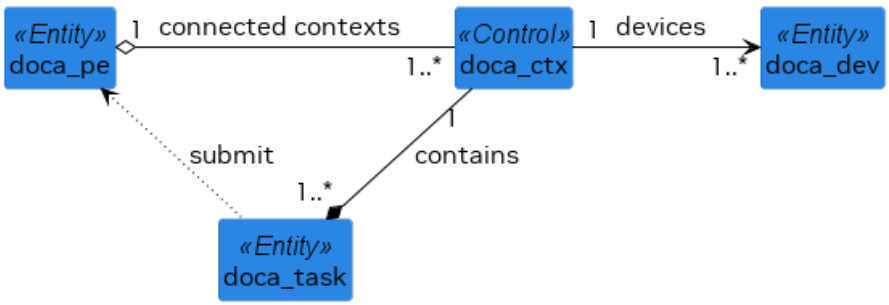
DOCA Context (`struct doca_ctx`) defines and provides (implements) task/event handling. A context is an instance of a specific DOCA library (i.e., when the library provides a DOCA Context, its functionality is defined by the list of tasks/events it can handle). When more than one type of task is supported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

The following list defines the relationship between task contexts:

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

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

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

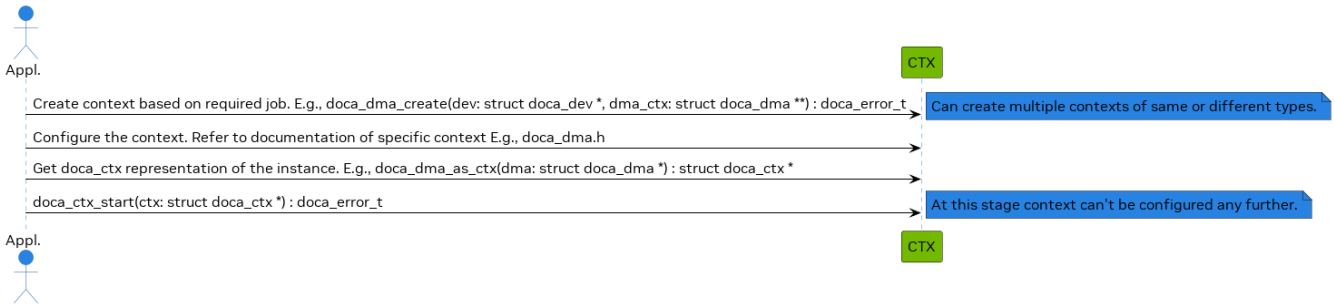


1. `doca_task` is associated to a relevant `doca_ctx` that executes the task (with the help of the relevant `doca_dev`).

2. `doca_task`, after it is initialized, is submitted to `doca_pe` for execution.

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

The following diagram describes the initialization sequence of a context:



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

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

Configuration Phase

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

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

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

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

Examples of common configurations:

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

Execution Phase

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

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

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

- Polling Mode
- Notification-driven Mode

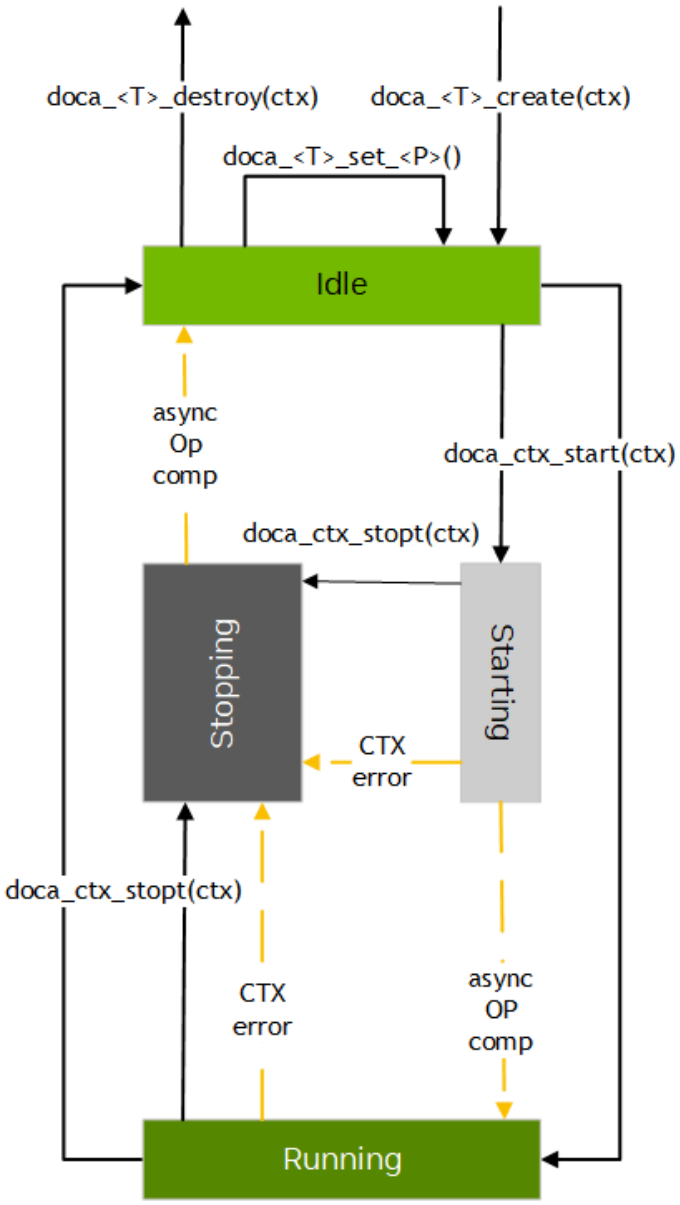
In this phase, the context and all core objects perform zero allocations by utilizing memory pools. It is recommended that the application utilizes same approach for its own logic.

State Machine

State	Description
Idle	• 0 in-flight tasks • On init (right after `doca_<T>_create(ctx)`): All configuration APIs enabled • On reconf (on transition from stopping state): Some configuration APIs enabled

State	Description
Starting	This state is mandatory for CTXs where transition to running state is conditioned by one or more async op completions/external events. For example, when a client connects to comm channel, it enters running state. Waiting for state change can be terminated by a voluntary (user) `doca_ctx_stop()` call or involuntary context state change due to internal error.
Running	• Task allocation/submission enabled (disabled in all other states) • All configuration APIs are disabled
Stopping	• Preparation before stopped state • Clean all in-flight tasks that may not complete in near future • Procedures relying on external entity actions should be terminated by CTX logic

The following diagram describes DOCA Context state transitions:



Internal Error

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

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

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

DOCA Task

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

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

Task Properties

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

The following are generic task properties:

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

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

The following are generic task APIs:

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

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

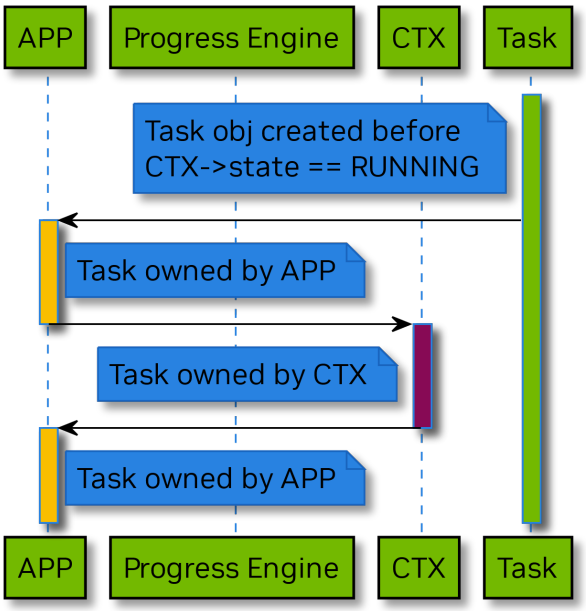
Task Lifecycle

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

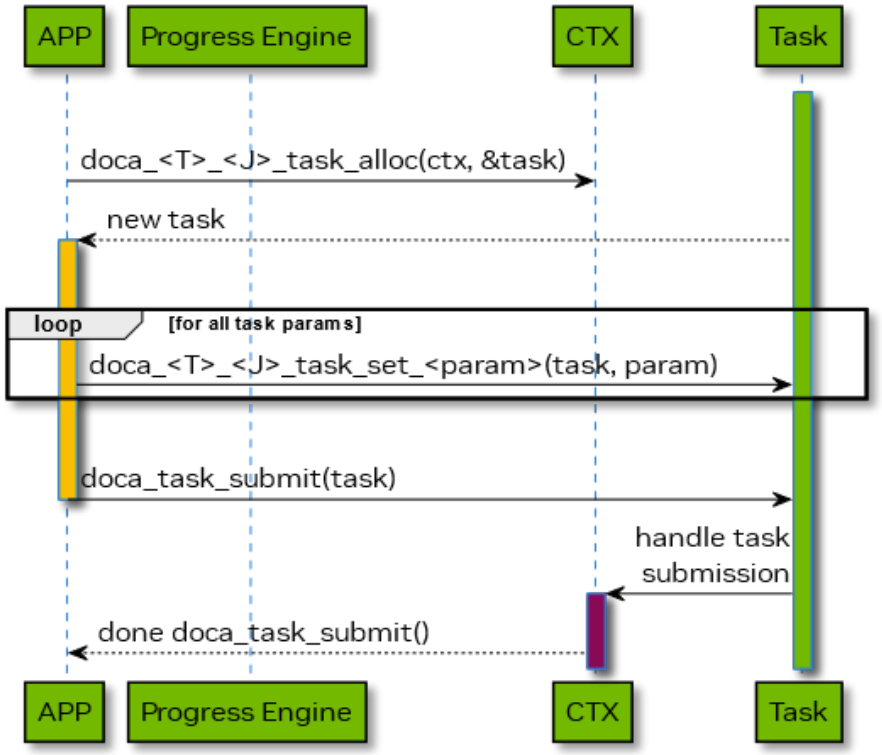
- starts on the event of entering *Running* state by the DOCA Context owning the task i.e., once *Running* state entered application can obtain the task from CTX by calling `doca_<CTX name>_task_<Task name>_alloc_init(ctx, ... &task)`.

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

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



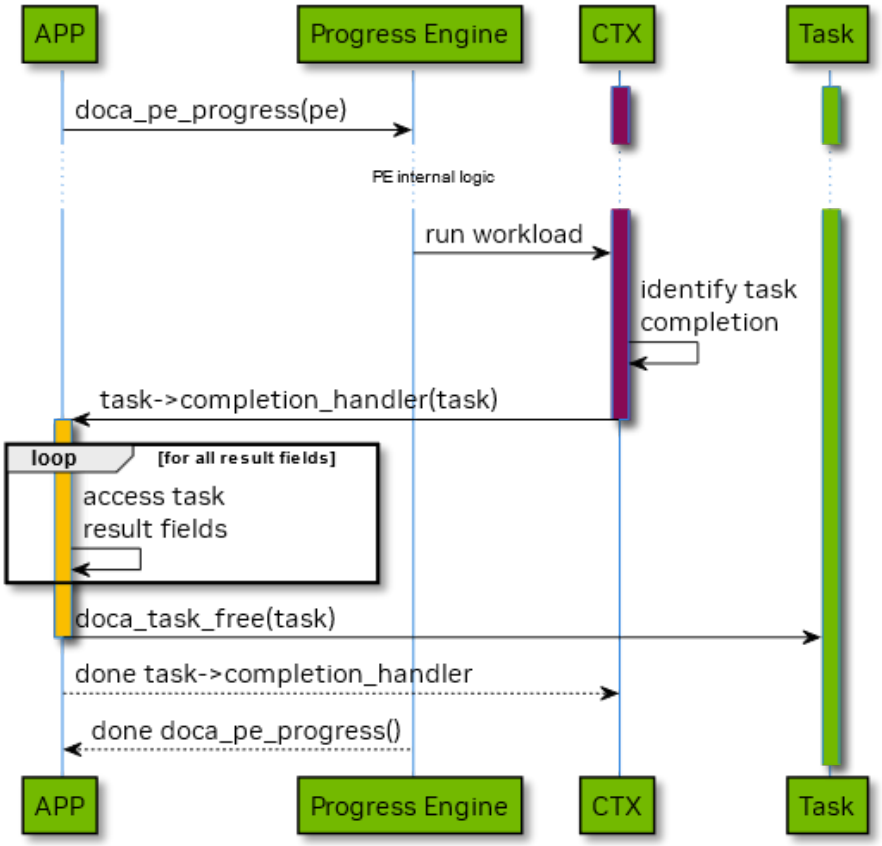
The diagram below shows the lifecycle of DOCA Task staring from its allocation to its submission.



The diagram above displays following ownership transitions during DOCA Task object lifecycle:

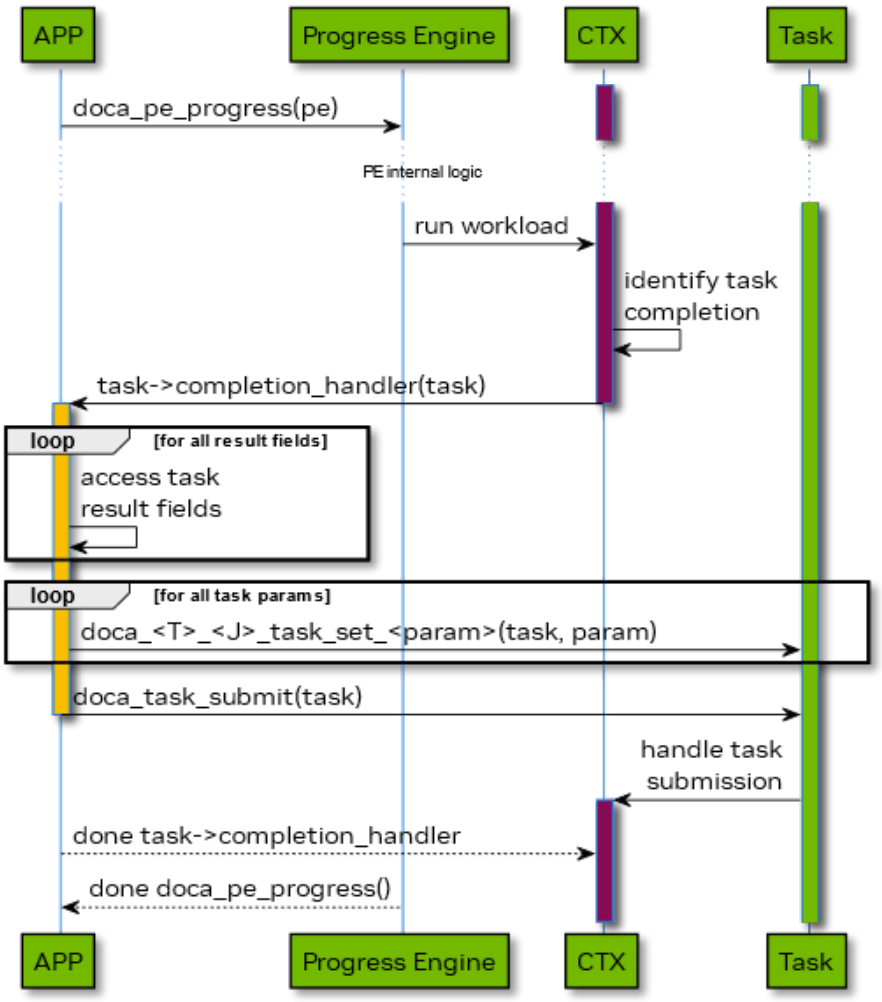
- starting from allocation task ownership passed from context to application
- application may modify task attributes via API templated as `doca_<CTX name>_task_<Task name>_set_<Parameter name>(task, param)`; on return from the task modification call the ownership of the task object returns to application.
- submit the task for processing in the PE, once all required modifications/settings of the task object completed. On task submission the ownership of the object passed to the related context.

The next two diagrams below shows the lifecycle of DOCA Task on its completion.



The diagram above displays following ownership transitions during *DOCA Task* object lifecycle:

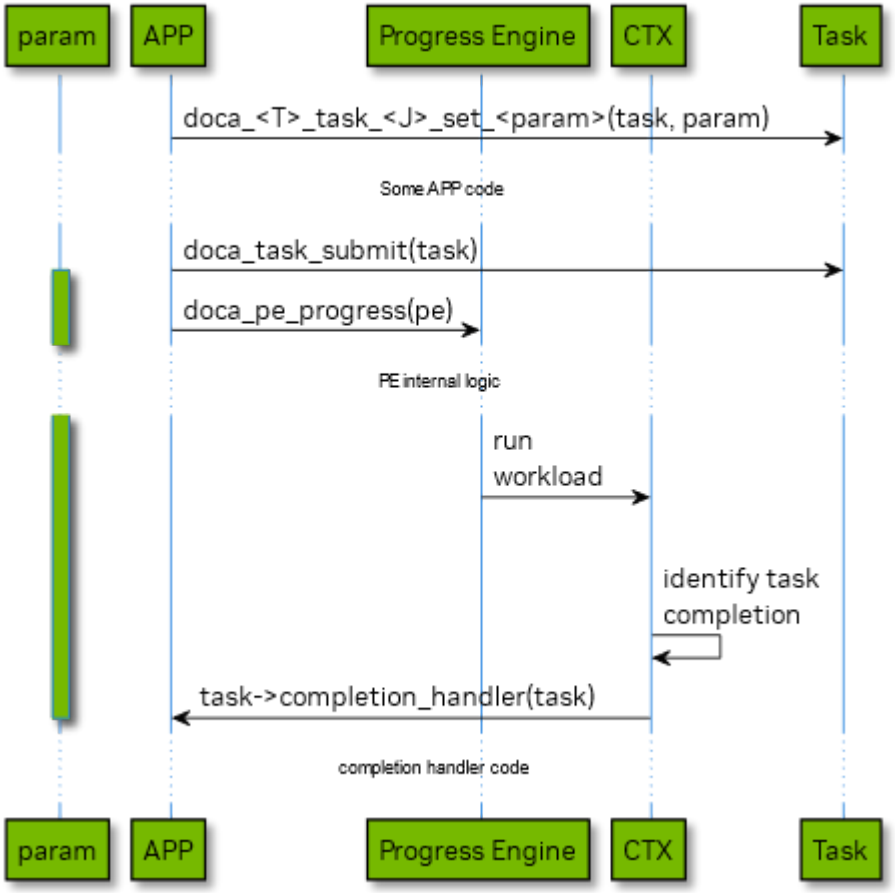
- on *DOCA Task* completion the appropriate handler provided by application invoked; on handler invocation the *DOCA Task* ownership passed to application.
- after *DOCA Task* completion application may access task attributes & result fields utilizing appropriate APIs; application remains owner of the task object.
- application may call `doca_task_free()` when task is no longer needed; on return from the call task ownership passed to *DOCA Context* while task became uninitialized & pre-allocated till the context enters Idle state.



The diagram above displays similar to the previous diagram ownership transitions during *DOCA Task* object lifecycle with the only difference that instead of `doca_task_free(task)` `doca_task_submit(task)` was called:

- *DOCA Task* result (related attributes) can be accessed right after enter successful task completion callback, similar to the previous case
- lifecycle of the *DOCA Task* results ends on exit from the task completion callback scope.
- On `doca_task_free()` or
  `doca_<CTX name>_task_<Task name>_set_<Parameter name>(task, param)`
  call all task results should be considered invalidated regardless of scope.

The diagram below shows the lifecycle of *DOCA Task* set-able parameters while API to set such a parameter templated as `doca_<CTX name>_task_<Task name>_set_<Parameter name>(task, param)` .



Green activation of **param** participant describes the time slice when all *DOCA Task* parameters owned by DOCA library. On `doca_task_submit()` call the ownership on all task arguments passed from application to the DOCA Context the related Task object belongs to. The ownership of task arguments passed back to application on task completion. The application should not modify and/or destroy/free Task argument related objects if it doesn't own the argument.

DOCA Progress Engine

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

PE is designed to be thread unsafe (i.e., it can only be used in one thread at a time) but a single OS thread can use multiple PEs. The user can assign different priorities to different contexts by adding them to different PEs and adjusting the polling frequency for each PE

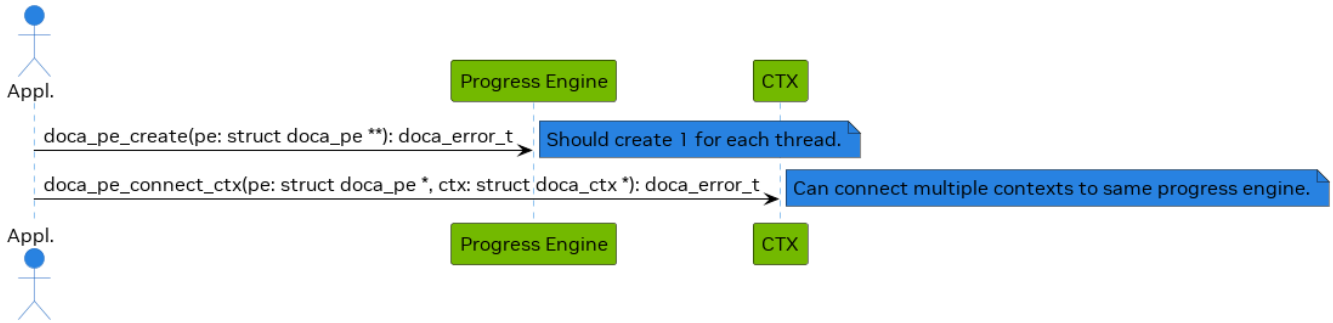
accordingly. Another way to view the PE is as a queue of workload units that are scheduled for execution.

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

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

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

The following diagram describes the initialization flow of the PE:



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

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

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

- Polling mode
- Blocking (notification-driven) mode

PE as Event Loop Mode of Operation

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

Run the selected workload unit. For the following cases:

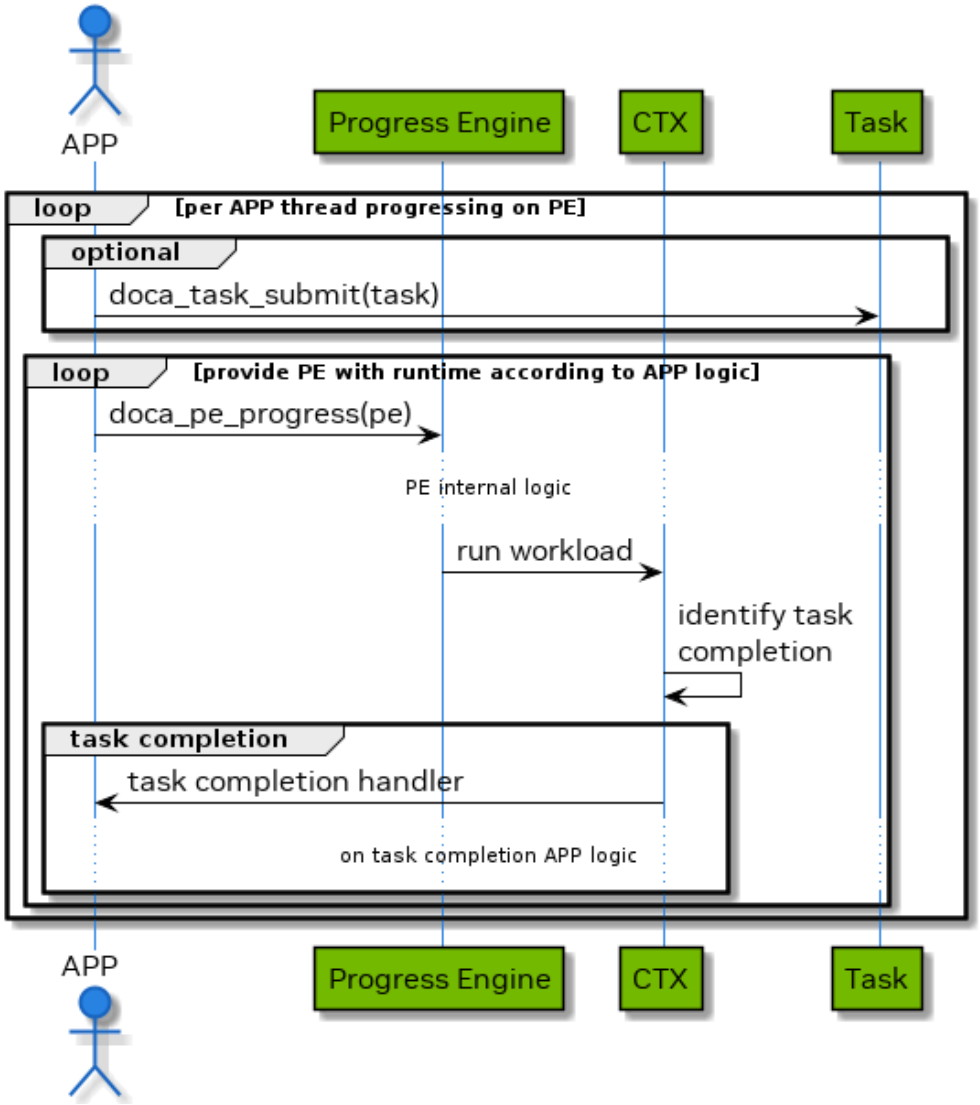
- Task completion, execute associated handler and break the loop and return status `made some progress`
- Positive probe of event, execute associated handler and break the loop and return status `made some progress`
- Considerable progress is made to contribute to future task completion or positive event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

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

Polling Mode

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

The following diagram demonstrates this sequence:



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

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

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

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

    3. If a task is completed with an error, preset task completion with error callback is executed (see section "Error Handling").

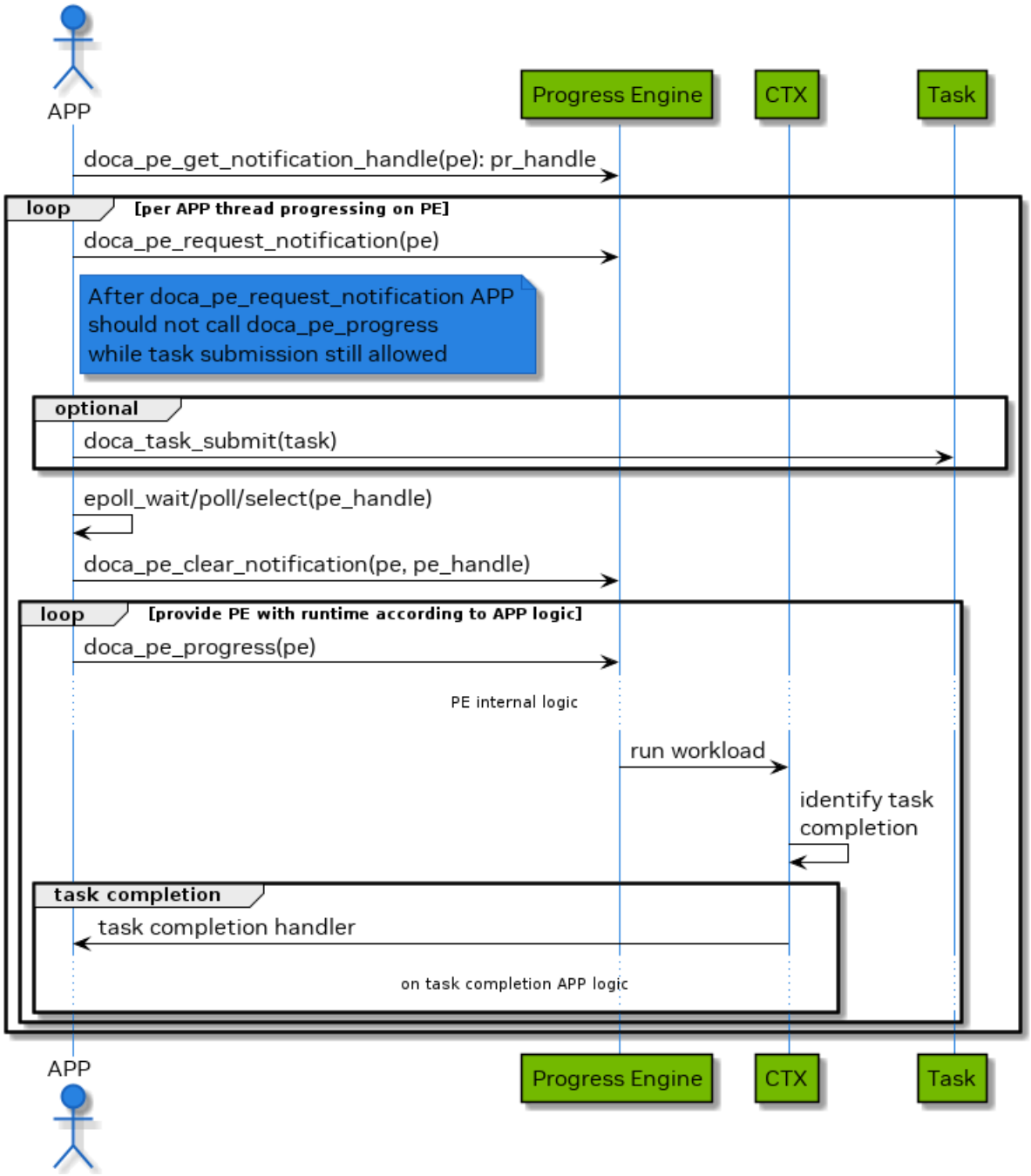
3. The application may add code to completion callbacks or event handlers for tracking the amount of completed and pending workloads.

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

Blocking Mode - Notification Driven

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

The following diagram demonstrates this sequence:



1. The application gets a notification handle from the `doca_pe` representing a Linux file descriptor which is used to signal the application that some work has finished.

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

> **Note**

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

Note

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

3. The application submits a task.
4. The application waits (e.g., Linux epoll/select) for a signal to be received on the `pe-fd`.
5. The application clears the notifications received, notifying the PE that a signal has been received and allowing it to perform notification handling.
6. The application attempts to handle received notifications via (multiple) calls to `doca_pe_progress()`.

Note

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

7. The application handles its internal state changes caused by task completions and event handlers called in the previous step.
8. Repeat steps 2-7 until all tasks are completed and all expected events are handled.

Progress Engine versus Epoll

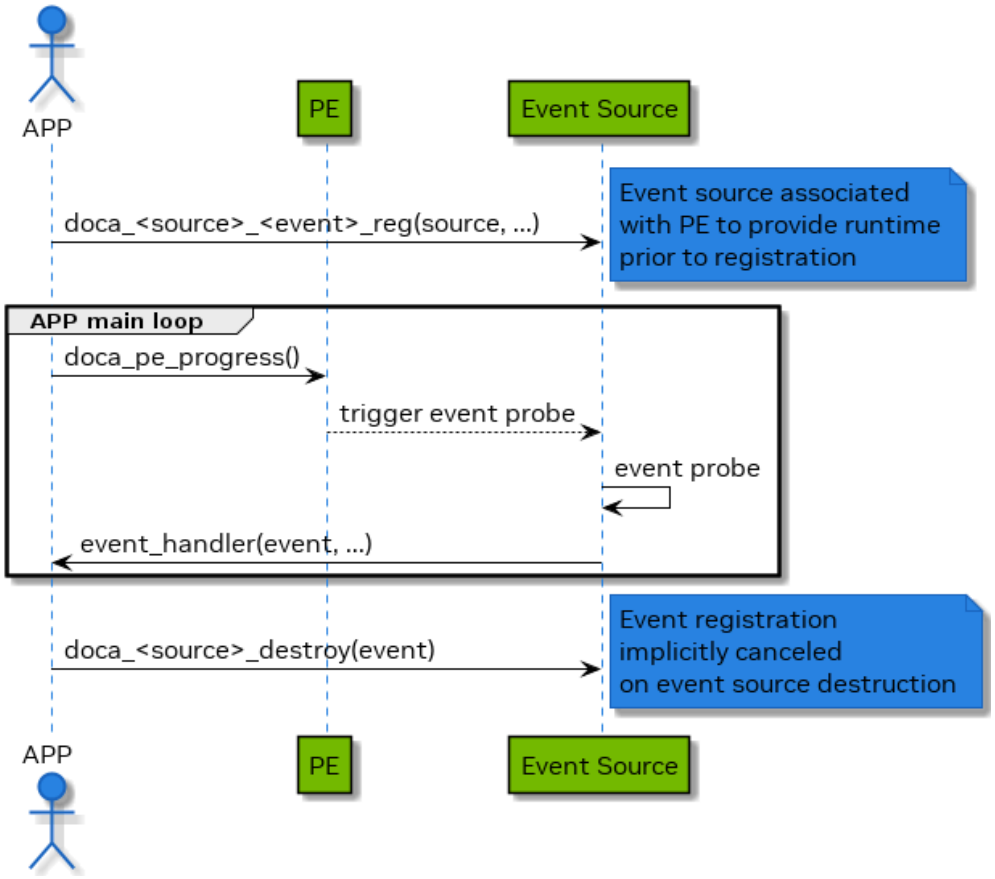
The epoll mechanism in Linux and the DOCA PE handles high concurrency in event-driven architectures. Epoll, like a post office, tracks "mailboxes" (file descriptors) and notifies the "postman" (the `epoll_wait` function) when a "letter" (event) arrives. DOCA PE, like a restaurant, uses a single "waiter"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

DOCA Event

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

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

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

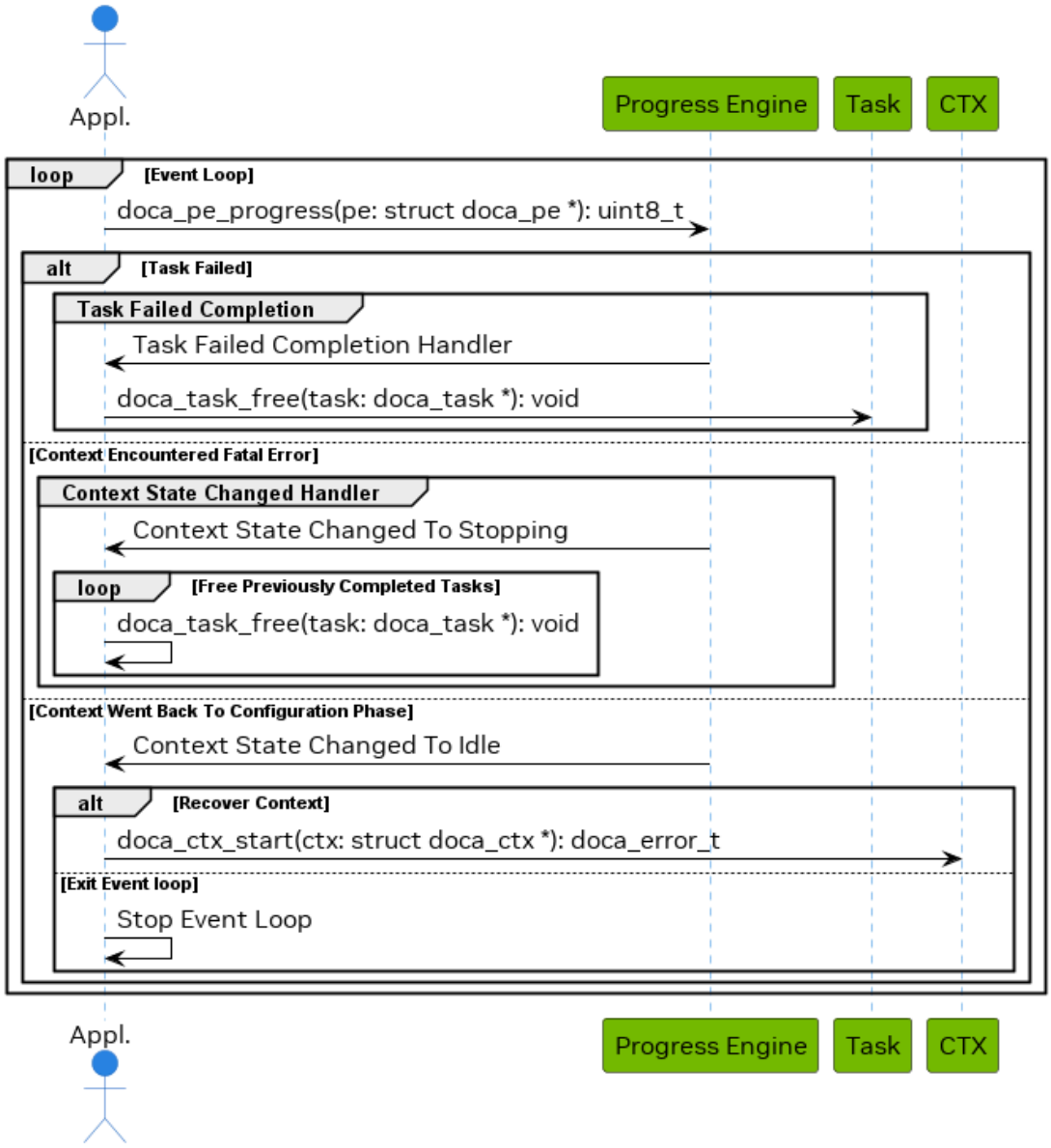


Error Handling

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

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

The following diagram shows how an application may handle an error from `doca_pe_progress()`:



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

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

Task and Event Batching

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

Batch Task/Event

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

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

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

Iterative Batch

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

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

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

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

Submit Flag [1]	**Effect on Current Task**		**Effect on Previous Tasks** [2]		**Default Behavior of doca_task_submit**	**Comments**
	Flag Provided	Flag not Provided	Flag Provided	Flag not Provided		
`DOCA_TASK_SUBMIT_FLAG_FLUSH`	Task is submitted for hardware execution immediately, and is considered "flushed".	Task may not be submitted for hardware execution, and is considered "unflushed".	All previous tasks which are considered unflushed become flushed.	None	Flag is provided	As long as the task is unflushed, it never completes. The flag allows batching such that multiple tasks are flushed at once, instead of individually.

Submit Flag [1]	Effect on Current Task		Effect on Previous Tasks [2]		Default Behavior of doca_task_submit	Comments
`DOCA_TASK_SUBMIT_FLAG_OPTIMIZE_REPORTS`	The user does not receive task completion after hardware has completed execution of the task, and the completion is considered "unreported".	The user receives task completion after hardware has completed execution of the task, and the completion is considered "reported".	None	Once the hardware completes execution of this task, all previous [3] unreported completions become reported.	Flag is not provided	As long as the task is unreported, the user would never know that it has been completed. The completion of a task is reported through a completion callback using the progress engine. The library does not guarantee any order of execution/completion of tasks. The flag allows batching, such that multiple task completions are reported using a single hardware completion, instead of receiving a completion for every task.

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

submit behavior. __

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

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

DOCA Graph Execution

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

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

Nodes

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

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

Context Node

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

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

User Node

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

Sub-graph Node

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

Using DOCA Graph

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

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

4. Start the graph using `doca_graph_start`.
5. Create the graph instance using `doca_graph_instance_create`.
6. Set the nodes data (e.g., `doca_graph_instance_set_ctx_node_data`).
7. Submit the graph instance to the pe using `doca_graph_instance_submit`.
8. Call `doca_pe_progress` until the graph callback is invoked.
    - Progress engine can run graph instances and standalone tasks simultaneously.

DOCA Graph Limitations

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

DOCA Graph Sample

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

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

Running DOCA Graph Sample

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

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

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

   ```
   ./build/doca_graph
   ```

   No parameters required.

Alternative Data Path

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

Considerations:

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

DPA

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

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

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

GPU

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

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

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

Object Life Cycle

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

1. The object is allocated by DOCA so it is opaque for the application (e.g., `doca_buf_inventory_create`, `doca_mmap_create`).
2. The application initializes the object and sets the desired properties (e.g., `doca_mmap_set_memrange`).
3. The object is started, and no configuration or attribute change is allowed (e.g., `doca_buf_inventory_start`, `doca_mmap_start`).
4. The object is used.
5. The object is stopped and deleted (e.g., `doca_buf_inventory_stop` → `doca_buf_inventory_destroy`, `doca_mmap_stop` → `doca_mmap_destroy`).

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

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

6. Imported mmap is destroyed.
7. Exported mmap is destroyed.
8. Original memory is destroyed.

RDMA Bridge

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

Requirements and Considerations

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

DOCA Core Objects to RDMA Core Objects Mapping

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

RDMA Core Object	DOCA Equivalent	RDMA Object to DOCA Object	DOCA Object to RDMA Object
`ibv_pd`	`doca_dev`	`doca_rdma_bridge_open_dev_from_pd`	`doca_rdma_bridge_get_dev_pd`
`ibv_mr`	`doca_buf`	`-`	`doca_rdma_bridge_get_buf_mkey`

DOCA Core Samples

Info

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

Progress Engine Samples

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

pe_common

`pe_common.c` and `pe_common.h` contain code that is used in most or all PE samples.

Users can find core code (e.g., create MMAP) and common code that uses PE (e.g., `poll_for_completion`).

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

pe_polling

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

Info

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

The sample demonstrates the following functions:

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

> **Note**
>
> Pay attention to the order of destruction (e.g., all contexts must be destroyed before the PE).

The sample performs the following:

1. Uses one DMA context.
2. Allocates and submits 16 DMA tasks.

> **Info**
>
> Task completion callback checks that the copied content is valid.

3. Polls until all tasks are completed.

pe_async_stop

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

The sample demonstrates the following functions:

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

The sample performs the following:

1. Submits 16 tasks and stops the context after half of the tasks are completed.
2. Polls until all tasks are complete (half are completed successfully, half are flushed).

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

pe_event

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

The sample demonstrates the following functions:

- How to run the PE in event-driven mode

The sample performs the following:

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

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

pe_multi_context

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

The sample demonstrates the following functions:

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

The sample performs the following:

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

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

pe_reactive

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

The sample demonstrates the following functions:

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

The user must make sure to:

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

The sample performs the following:

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

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

pe_single_task_cb

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

DOCA recommends using 2 callbacks:

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

The sample demonstrates the following functions:

- How to use a single callback instead of two callbacks

The sample performs the following:

1. Runs 16 DMA tasks.
2. Handles completion with a single callback.

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

pe_task_error

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

The sample demonstrates the following functions:

- How to mitigate a task error during runtime

The user must make sure to:

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

The sample performs the following:

1. Submits 255 tasks.
2. Allocates the second task with invalid parameters that cause hardware to fail.
3. Mitigates the failure and polls until all submitted tasks are flushed.

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

pe_task_resubmit

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

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

The sample demonstrates the following functions:

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

The sample performs the following:

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

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

pe_task_try_submit

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

> **Note**
>
> Task validation impacts performance and should not be used in production.

The sample demonstrates the following functions:

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

The sample performs the following:

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

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

Graph Sample

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

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

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

The sample runs 10 instances of the graph in parallel.

Backward Compatibility of DOCA Core doca_buf

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

DOCA Core doca_buf

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

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

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

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

Sync Event

Note

DOCA Sync Event API is considered thread-unsafe

Note

DOCA Sync Event does not currently support GPU related features.

Introduction

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

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

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

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

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

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

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

Prerequisites

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

- DOCA Execution Model
- DOCA Device

- DOCA Memory Subsystem

Environment

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

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

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

Architecture

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

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

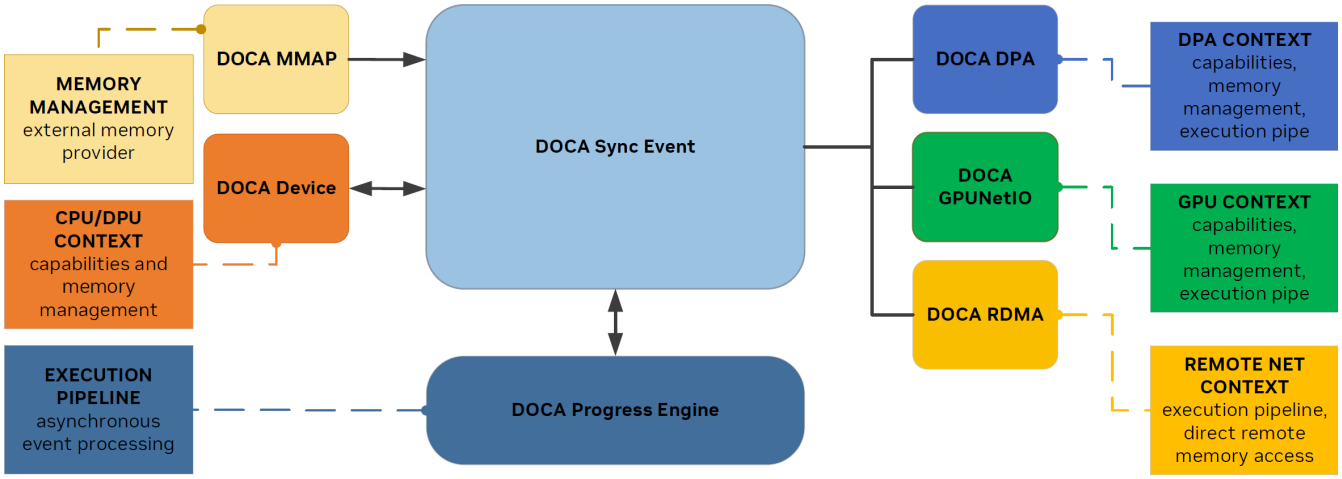
The figure that follows demonstrates components used by DOCA SE. DOCA Device provides information on the capabilities of the configured HW used by SE to control system resources.

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

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

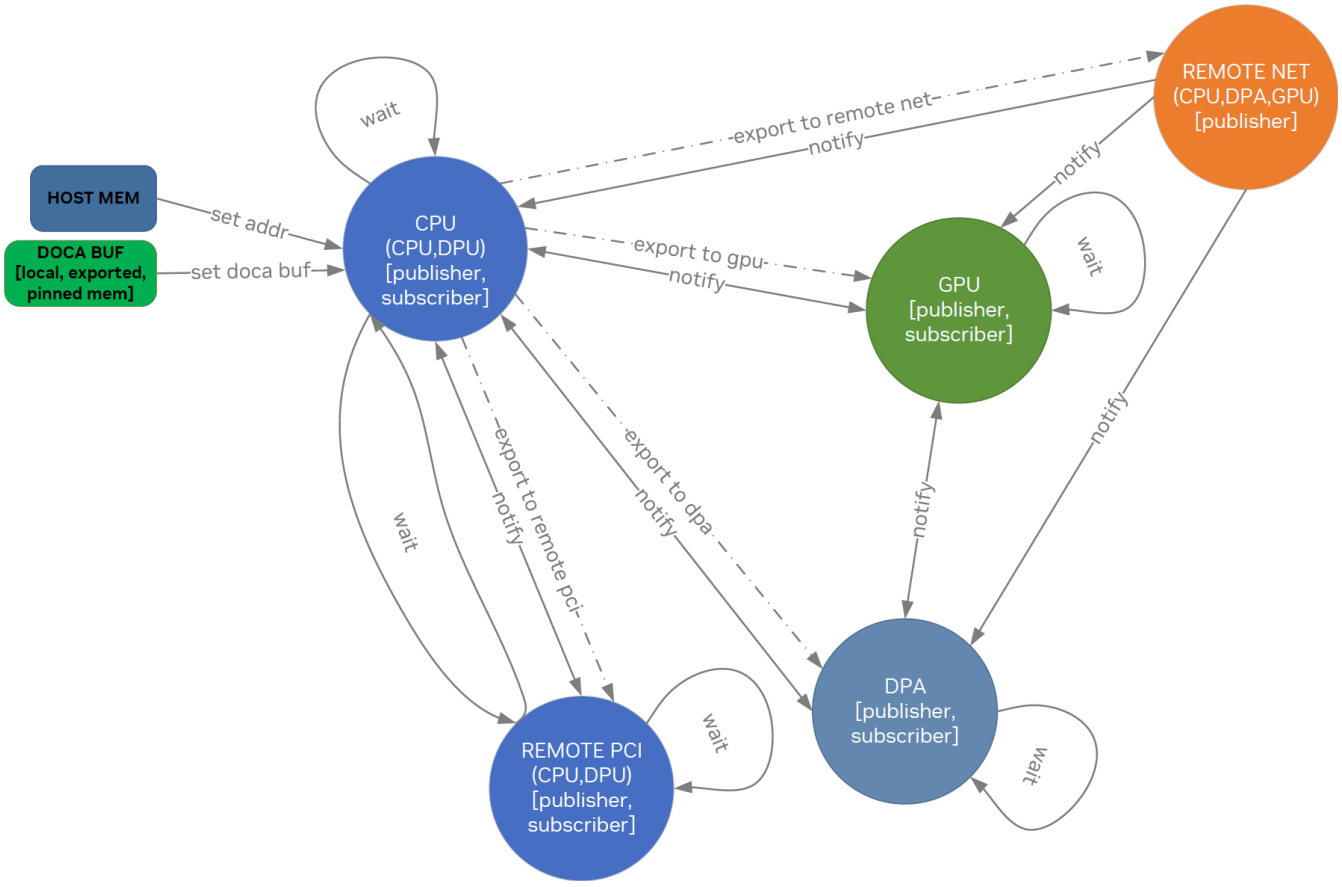
For asynchronous operation scheduling, SE uses the DOCA Progress Engine (PE) module.

DOCA Sync Event Components Diagram



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

DOCA Sync Event Interaction Diagram



DOCA Sync Event Objects

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

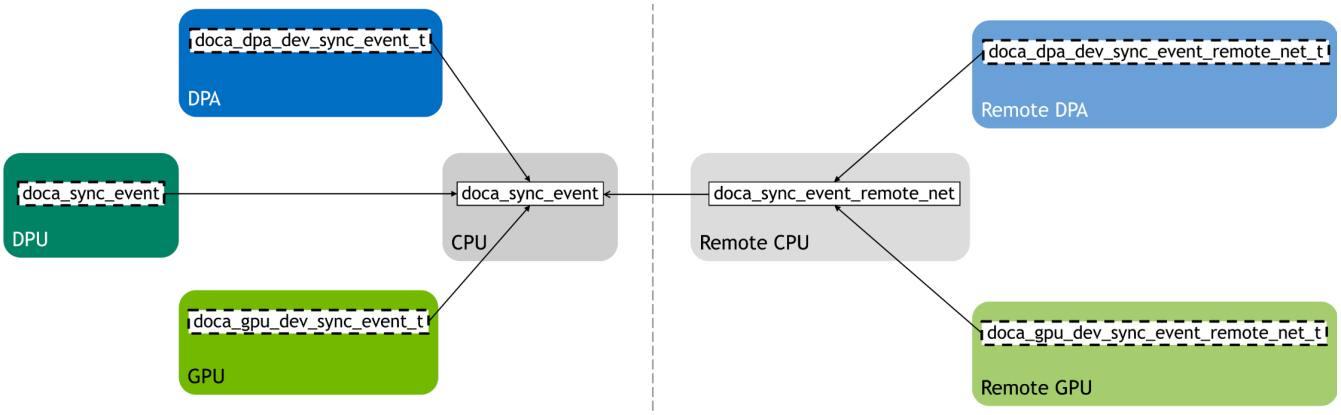
Execution Unit	Type	Description
CPU (host/DPU)	`struct doca_sync_event`	Handle for interacting with the SE from the CPU
DPU	`struct doca_sync_event`	Handle for interacting with the SE from the DPU
DPA	`doca_dpa_dev_sync_event_t`	Handle for interacting with the SE from the DPA
GPU	`doca_gpu_dev_sync_event_t`	Handle for interacting with the SE from the GPU
Remote net CPU	`doca_sync_event_remote_net`	Handle for interacting with the SE from a remote CPU
Remote net DPA	`doca_dpa_dev_sync_event_remote_net_t`	Handle for interacting with the SE from a remote DPA
Remote net GPU	`doca_gpu_dev_sync_event_remote_net_t`	Handle for interacting with the SE from a remote GPU

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

Configuration Phase

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

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



Operation Modes

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

> **Note**
>
> Once started, SE operation mode cannot be changed.

Synchronous Mode

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

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

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

Asynchronous Mode

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

Configurations

Mandatory Configurations

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

- DOCA SE CPU handle must be configured by providing the runtime hints on the publisher and subscriber locations. Both the subscriber and publisher locations must be configured using the following APIs:
  - `doca_sync_event_add_publisher_location_<cpu|dpa|gpu|remote_pc`
  - `doca_sync_event_add_subscriber_location_<cpu|dpa|gpu|remote_p`
- For the asynchronous use case, at least one task/event type must be configured. See configuration of tasks.

Optional Configurations

> **Info**
>
> If these configurations are not set, a default value is used.

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

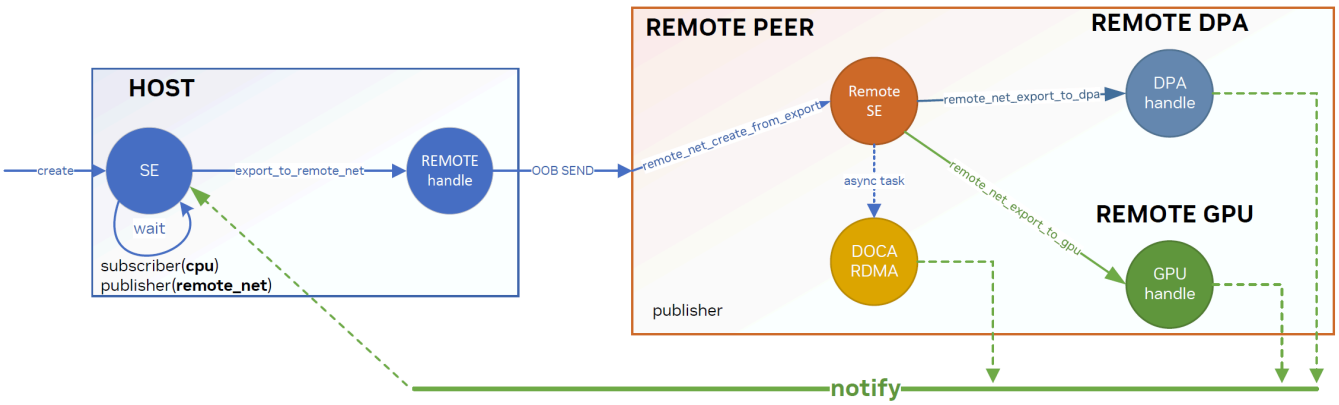
- `doca_sync_event_set_addr`
- `doca_sync_event_set_doca_buf`

Export DOCA Sync Event to Another Execution Unit

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

- DPA – `doca_sync_event_get_dpa_handle` returns a DOCA SE DPA handle (`doca_dpa_dev_sync_event_t`) which can be passed to the DPA SE data path APIs from the DPA kernel
- GPU – `doca_sync_event_get_gpu_handle` returns a DOCA SE GPU handle (`doca_gpu_dev_sync_event_t`) which can be passed to the GPU SE data path APIs for the CUDA kernel
- DPU – `doca_sync_event_export_to_remote_pci` returns a blob which can be used from the DPU CPU to instantiate a DOCA SE DPU handle (`struct doca_sync_event`) using the `doca_sync_event_create_from_export` function

DOCA SE allows notifications from remote peers (remote net) utilizing capabilities of the DOCA RDMA library. The following figure illustrates the remote net export flow:



- Remote net CPU – `doca_sync_event_export_to_remote_net` returns a blob which can be used from the remote net CPU to instantiate a DOCA SE remote net CPU handle (`struct doca_sync_event_remote_net`) using the

`doca_sync_event_remote_net_create_from_export` function. The handle can be used directly for submitting asynchronous tasks through the `doca_rdma` library or exported to the remote DPA/GPU.

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

Note

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

Note

Prior to calling any export function, users must first verify it is supported by calling the corresponding export capability getter: `doca_sync_event_cap_is_export_to_dpa_supported`, `doca_sync_event_cap_is_export_to_gpu_supported`, `doca_sync_event_cap_is_export_to_remote_pci_supported`, or `doca_sync_event_cap_is_export_to_remote_net_supported`.

Note

Prior to calling any `*_create_from_export` function, users must first verify it is supported by calling the corresponding create from

the export capability getter:
`doca_sync_event_cap_is_create_from_export_supported`
or
`doca_sync_event_cap_remote_net_is_create_from_export_supported`
.

Note

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

Warning

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

Device Support

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

Info

Both NVIDIA® BlueField ® -2 and BlueField ® -3 devices are supported as well as any `doca_dev` is supported.

Info

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

As device capabilities may change in the future (see DOCA Capability Check), it is recommended to choose your device using any relevant capability method (starting with the prefix `doca_sync_event_cap_*`).

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

- `doca_sync_event_cap_is_create_from_export_supported`
- `doca_sync_event_cap_remote_net_is_create_from_export_supported`

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

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

Capability APIs to query whether an asynchronous task is supported:

- `doca_sync_event_cap_task_get_is_supported`

- `doca_sync_event_cap_task_notify_set_is_supported`
- `doca_sync_event_cap_task_notify_add_is_supported`
- `doca_sync_event_cap_task_wait_eq_is_supported`
- `doca_sync_event_cap_task_wait_neq_is_supported`

Execution Phase

This section describes execution on CPU. For additional execution environments refer to section "Alternative Datapath Options".

DOCA Sync Event Data Path Operations

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

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

Info

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

Note

Prior to asynchronous task submission, users must check if the job is supported using

`doca_error_t doca_sync_event_cap_task_<task_type>_is_supported`.

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

Publishing on DOCA Sync Event

Setting DOCA Sync Event Value

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

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

Adding to DOCA Sync Event Value

Users can atomically increment the value of a DOCA SE:

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

Subscribe on DOCA Sync Event

Getting DOCA Sync Event Value

Users can get the value of a DOCA SE:

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

Waiting on DOCA Sync Event

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

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

Synchronously

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

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

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

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

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

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

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

Asynchronously

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

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

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

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

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

Tasks

DOCA SE context exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

Get Task

The get task retrieves the value of a DOCA SE.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_sync_event_task_get_set_conf`	`doca_sync_event_cap_task_get_is_supported`
Number of tasks	`doca_sync_event_task_get_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Return value	8-bytes memory pointer to hold the DOCA SE value

Task Output

Common output described in DOCA Core Task .

Task Completion Success

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

Task Completion Failure

If the task fails midway:

- The context may enter a stopping state if a fatal error occurs

- The return value memory may be modified

Task Limitations

All limitations are described in DOCA Core Task.

Notify Set Task

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

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_sync_event_task_notify_set_set_conf`	`doca_sync_event_cap_task_notify_set_is_supported`
Number of tasks	`doca_sync_event_task_notify_set_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Set value	64-bit value to set the DOCA SE value to

Task Output

Common output described in DOCA Core Task.

Task Completion Success

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

Task Completion Failure

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

Task Limitations

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

Notify Add Task

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

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_sync_event_task_notify_add_set_conf`	`doca_sync_event_cap_task_notify_add_is_supported`
Number of tasks	`doca_sync_event_task_notify_add_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Increment value	64-bit value to atomically increment the DOCA SE value by

Name	Description
Fetched value	8-bytes memory pointer to hold the DOCA SE value before the increment

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the following occurs:

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

Task Completion Failure

If the task fails midway:

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

Task Limitations

All limitations are described in DOCA Core Task.

Wait Equal-to Task

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

Task Configuration

Description	API to set the configuration	API to query support
Enable the task	`doca_sync_event_task_wait_eq_set_conf`	`doca_sync_event_cap_task_wait_eq_is_supported`
Number of tasks	`doca_sync_event_task_wait_eq_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Wait threshold	64-bit value to wait for the DOCA SE value to be equal to
Mask	64-bit mask to apply on the DOCA SE value before comparing with the wait threshold

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the following occurs:

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

Task Completion Failure

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

Task Limitations

Other limitations are described in DOCA Core Task.

Wait Not-equal-to Task

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

Task Configuration

Description	API to set the configuration	API to query support
Enable the task	`doca_sync_event_task_wait_neq_set_conf`	`doca_sync_event_cap_task_wait_neq_is_supported`
Number of tasks	`doca_sync_event_task_wait_neq_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Wait threshold	64-bit value to wait for the DOCA SE value to be not equal to
Mask	64-bit mask to apply on the DOCA SE value before comparing with the wait threshold

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the following occurs:

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

Task Completion Failure

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

Task Limitations

Limitations are described in DOCA Core Task.

Events

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

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

State Machine

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

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

Idle

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

- Destroy the context; or
- Start the context

Allowed operations in this state:

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

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and then freed

Starting

This state cannot be reached.

Running

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

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

Allowed operations in this state:

- Allocate previously configured task
- Submit an allocated task
- Call stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

Stopping

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

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

Allowed operations in this state:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

DOCA Sync Event Tear Down

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

Note

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

Stopping DOCA Sync Event

To stop a DOCA SE:

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

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

Destroying DOCA Sync Event

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

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

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

Alternative Datapath Options

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

GPU Datapath

DOCA SE does not currently support GPU related features.

DPA Datapath

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

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

DOCA Sync Event Sample

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

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

Running DOCA Sync Event Sample

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

```
cd
/opt/mellanox/doca/samples/doca_common/sync_event_<local|remot
```

```
meson /tmp/build
ninja -C /tmp/build
```

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

3. Sample usage:

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

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

Program Flags:
  -d, --pci-addr                        Device PCI address
  -r, --rep-pci-addr                DPU representor PCI
address
```

```
  --async                                 Start DOCA Sync Event in
asynchronous mode (synchronous mode by default)
  --async_num_tasks                       Async num tasks 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.

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

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

Samples

Sync Event Remote PCIe

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

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

The sample logic includes:

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

Reference:

- `/opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/sync`
- `/opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/sync`
- `/opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/mesc`

Sync Event Local PCIe

Note

This sample should run (on the DPU or on the Host) only after Sync Event Remote PCIe has been started.

This sample demonstrates how to initialize a SE to be shared with a remote PCIe (host or the DPU) how to export it to a remote PCIe, and how to interact with the SE to achieve synchronization between the host and DPU.

The sample logic includes:

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

Reference:

- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/sync_`
- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/sync_`
- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/meson`

Mmap Advise

Introduction

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

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

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

Prerequisites

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

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

Architecture

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

DOCA Mmap Advise currently supports the following list of advised operations:

- Cache Invalidate Operation

Cache Invalidate Operation

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

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

Environment

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

Objects

Device and Device Representor

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

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

> **Note**
>
> The device must stay valid for as long as the MMAP Advise instance is not destroyed.

Memory Buffers

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

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

Configuration Phase

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

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

Configurations

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

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

Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

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

Device Support

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

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

- `doca_mmap_advise_cap_task_cache_invalidate_is_supported`

Some devices expose different capabilities as follows:

- Maximum cache invalidate buffer size may differ.

Buffer Support

Tasks support buffers with the following features:

Buffer Type	Buffer
Local mmap buffer	Yes
MMAP from PCIe export buffer	No
MMAP from RDMA export buffer	No
Linked list buffer	No

Execution Phase

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

Tasks

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

Cache Invalidate Task

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

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_mmap_advise_task_invalidate_cache_set_conf`	`doca_mmap_advise_cap_task_cache_invalidate_is_supported`
Number of tasks	`doca_mmap_advise_task_invalidate_cache_set_conf`	–
Maximal buffer size	–	`doca_mmap_advise_task_cache_invalidate_get_max_buf_size`
Maximal buffer list size	–	–

Task Input

Common input as described in DOCA Core Task.

Name	Description
buffer	Buffer that points to the memory to be invalidated

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task is completed successfully:

- The cache is invalidated

Task Completion Failure

If the task fails midway:

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

Task Limitations

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

Events

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

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

State Machine

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

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

Idle

In this state it is expected that the application:

- Destroys the context
- Starts the context

Allowed operations:

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

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

Starting

This state cannot be reached.

Running

In this state, it is expected that the application:

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

Allowed operations:

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

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

Stopping

In this state it is expected that the application:

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

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

Alternative Datapath Options

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

Samples

Cache Invalidate Sample

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

The sample logic includes:

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

Reference:

- `/opt/mellanox/doca/samples/doca_common/cache_invalidate/cache_inv`
- `/opt/mellanox/doca/samples/doca_common/cache_invalidate/cache_inv`
- `/opt/mellanox/doca/samples/doca_common/cache_invalidate/meson.bui`

DOCA Log

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

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

Log Verbosity Level Enumerations

The following verbosity levels are supported by the DOCA logging:

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

Note

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

See `doca_log.h` for more information.

Logging Backends

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

The following backend types are supported:

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

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

- Lower level – `DOCA_LOG_LEVEL_INFO`
- Upper level – `DOCA_LOG_LEVEL_CRIT`

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

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

Enabling DOCA SDK Libraries Logging

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

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

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

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

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

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

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

Enabling DOCA Application Logging

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

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

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

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

`doca_log_backend_create_standard()` creates a default non-configurable set of two backends:

- `stdout` prints the range from a global minimum level up to `DOCA_LOG_LEVEL_INFO`

- `stderr` prints the range from `DOCA_LOG_LEVEL_WARNING` level up to `DOCA_LOG_LEVEL_CRIT`

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

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

`doca_log_level_set_global_lower_limit()` sets the lower limit for all existing backends not marked as strict and sets the global application lower limit.

`doca_log_level_set_global_upper_limit()` sets the upper limit for all existing backends not marked as strict and sets the global application upper limit.

Logging DOCA Application Messages

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

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

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

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

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 (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes 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 NONINFRINGEMENT, 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