DPA Development

Table of contents

Toolchain

FlexIO

List of Figures

Overview

DOCA Libs and Drivers

The NVIDIA DOCA framework is the key for unlocking the potential of NVIDIA® BlueField®-3 platforms.

DOCA's software environment allows developers to program the DPA to accelerate workloads. Specifically, DOCA includes:

- DOCA DPA SDK – a high-level SDK for application-level protocol acceleration
- DOCA FlexIO SDK – a low-level SDK to load DPA programs into the DPA, manage the DPA memory, create the execution handlers and the needed hardware rings and contexts
- DPACC – DPA toolchain for compiling and ELF file manipulation of the DPA code

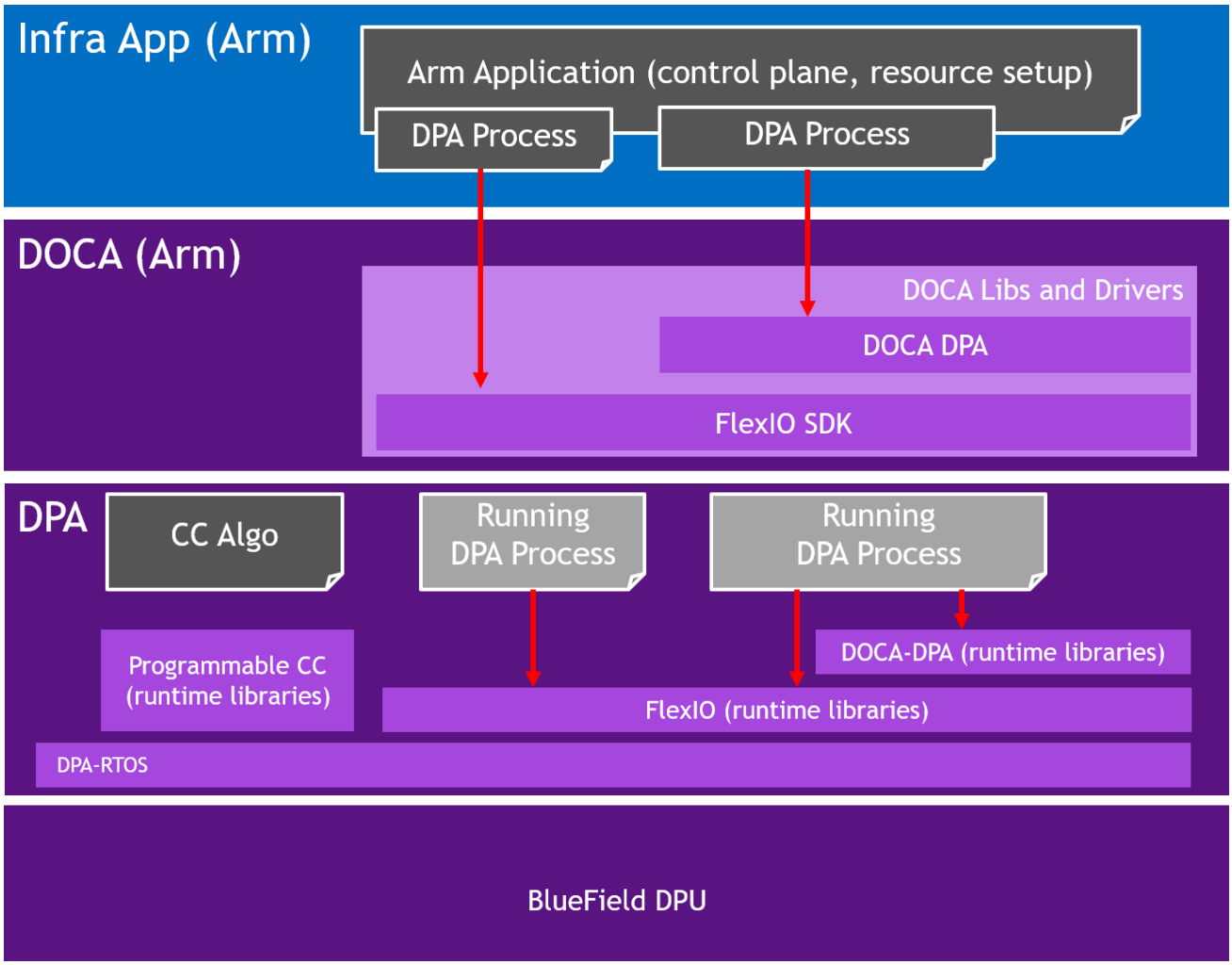
Programming Model

The DPA is intended to accelerate datapath operations for the DPU and host CPU. The accelerated portion of the application using DPA is presented as a library for the host application. The code within the library is invoked in an event-driven manner in the context of a process that is running on the DPA. One or many DPA execution units may work to handle the work associated with network events. The programmer specifies different conditions when each function should be called using the appropriate SDK APIs on the host or DPU.

The DPA cannot be used as a standalone CPU.

Management of the DPA, such as loading processes and allocating memory, is performed from a host or DPU process. The host process discovers the DPA capabilities on the device and drives the control plane to set up the different DPA objects. The DPA objects exist as long as the host process exists. When the host process is destroyed, the DPA objects are freed. The host process decides which functions it wants to accelerate using the DPA: Either its entire data plane or only a part of it.

The following diagram illustrates the different processes that exist in the system:



Compiler

DPACC is a compiler for the DPA processor. It compiles code targeted for the DPA processor into an executable and generates a DPA program. A DPA program is a host library with interfaces encapsulating the DPA executable.

This DPA program is linked with the host application to generate a host executable. The host executable can invoke the DPA code through the DPA SDK's runtime.

Compiler Keywords

DPACC implements the following keywords:

Keyword	Application Usage	Comment
__dpa_global__	Annotate all event handlers that execute on the DPA and all common user-defined datatypes (including user-defined structures) which are passed from the host to the DPA as arguments.	Used by the compiler to generate entry points in the DPA executable and automatically replicate user-defined datatypes between the host and DPA.
__dpa_rpc__	Annotate all RPC calls which are invoked by the host and execute on the DPA. RPC calls return a value of uint64_t.	Used by the compiler to generate RPC specific entry points.

Please refer to NVIDIA DOCA DPACC Compiler for more details.

FlexIO

Supported at beta level.

FlexIO is a low-level event-driven library to program and accelerate functions on the DPA.

FlexIO Execution Model

To load an application onto the DPA, the user must create a process on the DPA, called a FlexIO process. FlexIO processes are isolated from each other like standard host OS processes.

FlexIO supports the following options for executing a user-defined function on the DPA:

1. FlexIO event hander – the event handler executes its function each time an event occurs. An event on this context is a completion event (CQE) received on the NIC completion queue (CQ) when the CQ was in the armed state. The event triggers an internal DPA interrupt that activates the event handler. When the event handler is activated, it is provided with a user-defined argument. The argument in most cases is a pointer to the software execution context of the event handler.

   The following pseudo-code example describes how to create an event handler and attach it to a CQ:

```
// Device code
__dpa_global__ void myFunc(flexio_uintptr_t myArg){
    struct my_db *db = (struct my_db *)myArg;
    get_completion(db->myCq)
    work();
    arm_cq(myCq);
    // reschedule the thread
    flexio_dev_thread_reschedule();
}

// Host code
main() {

    /* Load the application code into the DPA */
    flexio_process_create(device, application, &myProcess);

    /* Create event handler to run my_func with my_arg */
    flexio_event_handler_create(myProcess, myFunc, myArg,
&myEventHandler);

    /* Associate the event hanlder with a specific CQ */
    create_cq(&myCQ,... , myEventHandler)

    /* Start the event handler */
    flexio_event_handler_run(myEventHandler)
    ...
}
```

2. RPC – remote, synchronous, one-time call of a specific function. RPC is mainly used for the control path to update DPA memory contexts of a process. The RPC's return value is reported back to the host application.

   The following pseudo-code example describes how to use the RPC:

```
// Device code
```

```
__dpa_rpc__ uint64_t myFunc(myArg) {
    struct my_db *db = (struct my_db *)myArg;
    if (db->flag) return 1;
    db->flag = 1;
    return 0;
}

// Host code
main() {
    ...

    /* Load the application code into the DPA */
    flexio_process_create(device, application, &myProcess);

    /* run the function */
    flexio_process_call(myProcess, myFunc, myArg, &returnValue);
    ...
}
```

FlexIO Memory Management

The DPA process can access several memory locations:

- Global variables defined in the DPA process.
- Stack memory – local to the DPA execution unit. Stack memory is not guaranteed to be preserved between different execution of the same handler.
- Heap memory – this is the process' main memory. The heap memory contents are preserved as long as the DPA process is active.
- External registered memory – remote to the DPA but local to the server. The DPA can access any memory location that can be registered to the local NIC using the provided API. This includes BlueField DRAM, external host DRAM, GPU memory, and more.

The heap and external registered memory locations are managed from the host process. The DPA execution units can load/store from stack/heap and external memory locations. Note that for external memory locations, the window should be configured appropriately using FlexIO Window APIs.

FlexIO allows the user to allocate and populate heap memory on the DPA. The memory can later be used by in the DPA application as an argument to the execution context (RPC and event handler):

```
/* Load the application code into the DPA */
flexio_process_create(device, application, &myProcess);

/* allocate some memory */
flexio_buf_dev_alloc(process, size, ptr)

/* populate it with user defined data */
flexio_host2dev_memcpy(process, src, size, ptr)

/* run the function */
flexio_process_call(myProcess, function, ptr, &return value);
```

FlexIO allows accessing external registered memory from the DPA execution units using FlexIO Window. FlexIO Window maps a memory region from the DPA process address space to an external registered memory. A memory key for the external memory region is required to be associated with the window. The memory key is used for address translation and protection. FlexIO window is created by the host process and is configured and used by the DPA handler during execution. Once configured, LD/ST from the DPA execution units access the external memory directly.

The access for external memory is not coherent. As such, an explicit memory fencing is required to flush the cached data to maintain consistency. See section "Memory Fences" for more.

The following example code demonstrates the window management:

```
// Device code
__dpa_rpc__ uint64_t myFunc(arg1, arg2, arg3)
{
```

```
    struct flexio_dev_thread_ctx *dtctx;
    flexio_dev_get_thread_ctx(&dtctx);
    uint32_t windowId = arg1;
    uint32_t mkey = arg2;
    uint64_t *dev_ptr;
    flexio_dev_window_config(dtctx, windowId, mkey );
    /* get ptr to the external memory (arg3) from the DPA process address space */
    flexio_dev_status status = flexio_dev_window_ptr_acquire (dtctx, arg3, dev_ptr);
    /* will set the external memory */
    *dev_ptr = 0xff;
    /* flush the data out */
    __dpa_thread_window_writeback();
    return 0;
}

// Host code
main() {
     /* Load the application code into the DPA */
     flexio_process_create(device, application, &myProcess);
     /* define an array on host */
     uint64_t var= {0};
     /* register host buffer  */
     mkey =ibv_reg_mr(&var, ...)
     /* create the window */
     flexio_window_create(process, doca_device->pd, mkey, &window_ctx);
     /* run the function */
     flexio_process_call(myProcess, myFunc, flexio_window_get_id(window_ctx),
mkey, &var, &returnValue);
}
```

Send and Receive Operation

A DPA process can initiate send and receive operations using the FlexIO outbox object. The FlexIO outbox contains memory-mapped IO registers that enable the DPA application to issue device doorbells to manage the send and receive planes. The DPA outbox can be

configured during run time to perform send and receive from a specific NIC function exposed by the DPU. This capability is not available for Host CPUs that can only access their assigned NIC function.

Each DPA execution engine has its own outbox. As such, each handler can efficiently use the outbox without needing to lock to protect against accesses from other handlers. To enforce the required security and isolation, the DPA outbox enables the DPA application to send and receive only for queues created by the DPA host process and only for NIC functions the process is allowed to access.

Like the FlexIO window, the FlexIO outbox is created by the host process and configured and used at run time by the DPA process.

```
// Device code
__dpa_rpc__ uint64_t myFunc(arg1,arg2,arg3) {

   struct flexio_dev_thread_ctx *dtctx;

   flexio_dev_get_thread_ctx(&dtctx);

   uint32_t outbox = arg1;
   flexio_dev_outbox_config (dtctx, outbox);

   /* Create some wqe and post it on sq */

   /* Send DB on sq*/

   flexio_dev_qp_sq_ring_db(dtctx, sq_pi,arg3);

   /* Poll CQ (cq number is in arg2) */
   return 0;
}

// Host code
main() {

    /* Load the application code into the DPA */
```

```
    flexio_process_create(device, application, &myProcess);

    /* Allocate uar */
    uar = ibv_alloc_uar(ibv_ctx);

    /* Create queues*/
    flexio_cq_create(myProcess, ibv_ctx, uar, cq_attr, &myCQ);
    my_hwcq = flexio_cq_get_hw_cq (myCQ);

    flexio_sq_create(myProcess, ibv_ctx, myCQ, uar, sq_attr, &mySQ);
    my_hwsq = flexio_sq_get_hw_sq(mySQ);

    /* Outbox will allow access only for queues created with the same UAR*/
    flexio_outbox_create(process, ibv_ctx, uar, &myOutbox);

    /* Run the function */
    flexio_process_call(myProcess, myFunc, myOutbox, my_hwcq->cq_num,
my_hwsq->sq_num,  &return_value);
}
```

Synchronization Primitives

The DPA execution units support atomic instructions to protect from concurrent access to the DPA process heap memory. Using those instructions, multiple synchronization primitives can be designed.

FlexIO currently supports basic spin lock primitives. More advanced thread pipelining can be achieved using DOCA DPA events.

DOCA DPA

Supported at beta level.

The DOCA DPA SDK eases DPA code management by providing high-level primitives for DPA work offloading, synchronization, and communication. This leads to simpler code but lacks the low-level control that FlexIO SDK provides.

User-level applications and libraries wishing to utilize the DPA to offload their code may choose DOCA DPA. Use-cases closer to the driver level and requiring access to low-level NIC features would be better served using FlexIO.

The implementation of DOCA DPA is based on the FlexIO API. The higher level of abstraction enables the user to focus on their program logic and not the low-level mechanics.

> **Info**
>
> Refer to DOCA DPA documentation for more details.

Memory Model

The DPA offers a coherent but weakly ordered memory model. The application is required to use fences to impose the desired memory ordering. Additionally, where applicable, the application is required to write back data for the data to be visible to NIC engines (see the coherency table).

The memory model offers "same address ordering" within a thread. This means that, if a thread writes to a memory location and subsequently reads that memory location, the read returns the contents that have previously been written.

The memory model offers 8-byte atomicity for aligned accesses to atomic datatypes. This means that all eight bytes of read and write are performed in one indivisible transaction.

The DPA does not support unaligned accesses, such as accessing N bytes of data from an address not evenly divisible by N.

The DPA processes memory can be divided into the following memory spaces:

Memory Space	Definition
Heap	Memory locations within the DPA process heap. Referenced as __DPA_HEAP in the code.
Memory	Memory locations belonging to the DPA process (including stack, heap, BSS and data segment) except the memory-mapped IO. Referenced as __DPA_MEMORY in the code.
MMIO (memory-mapped I/O)	External memory outside the DPA process accessed via memory-mapped IO. Window and Outbox accesses are considered MMIO. Referenced as __DPA_MMIO in the code.
System	All memory locations accessible to the thread within Memory and MMIO spaces as described above. Referenced as __DPA_SYSTEM in the code.

The coherency between the DPA threads and NIC engines is described in the following table:

Producer	Observer	Coherency	Comments
DPA thread	NIC engine	Not coherent	Data to be read by the NIC must be written back using the appropriate intrinsic (see section "Memory Fence and Cache Control Usage Examples").
NIC engine	DPA Thread	Coherent	Data written by the NIC is eventually visible to the DPA threads. The order in which the writes are visible to the DPA threads is influenced by the ordering configuration of the memory region (see IBV_ACCESS_RELAXED_ORDERING). In a typical example of the NIC writing data and generating a completion entry (CQE), it is guaranteed that when the write to the CQE is visible, the DPA thread can read the data without additional fences.
DPA thread	DPA thread	Coherent	Data written by a DPA thread is eventually visible to the other DPA threads without additional fences. The order in which writes made by a thread are visible to other threads is undefined when fences are not used. Programmers can enforce ordering of updates using fences (see section "Memory Fences") .

Memory Fences

Fence APIs are intended to impose memory access ordering. The fence operations are defined on the different memory spaces. See information on memory spaces under section "Memory Model".

The fence APIs apply ordering between the operations issued by the calling thread. As a performance note, the fence APIs also have a side effect of writing back data to the memory space used in the fence operation. However, programmers should not rely on this side effect. See section "Cache Control" for explicit cache control operations. The fence APIs have an effect of a compiler-barrier which means that memory accesses are not reordered around the fence API invocation by the compiler.

A fence applies between the "predecessor" and the "successor" operations. The predecessor and successor ops can be refenced using __DPA_R, __DPA_W, and __DPA_RW in the code.

The generic memory fence operation can operate on any memory space and any set of predecessor and successor operations. The other fence operations are provided as convenient shortcuts that are specific to the use case. It is preferable for programmers to use the shortcuts when possible.

Fence operations can be included using the dpaintrin.h header file.

Generic Fence

```
void __dpa_thread_fence(memory_space, pred_op, succ_op);
```

This fence can apply to any DPA thread memory space. Memory spaces are defined under section "Memory Model". The fence ensures that all operations (pred_op) performed by the calling thread, before the call to __dpa_thread_fence(), are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before all operations (succ_op) to the memory space after the call to __dpa_thread_fence().

System Fence

```
void __dpa_thread_system_fence();
```

This is equivalent to calling __dpa_thread_fence(__DPA_SYSTEM, __DPA_RW, __DPA_RW).

Outbox Fence

```
void __dpa_thread_outbox_fence(pred_op, succ_op);
```

This is equivalent to calling __dpa_thread_fence(__DPA_MMIO, pred_op, succ_op).

Window Fence

```
void __dpa_thread_window_fence(pred_op, succ_op);
```

This is equivalent to calling __dpa_thread_fence(__DPA_MMIO, pred_op, succ_op).

Memory Fence

```
void __dpa_thread_memory_fence(pred_op, succ_op);
```

This is equivalent to calling __dpa_thread_fence(__DPA_MEMORY, pred_op, succ_op).

Cache Control

Cache control operations allow the programmer to exercise fine-grained control over data resident in the DPA's caches. They have an effect of a compiler-barrier. The operations can be included using the dpaintrin.h header file.

Window Read Contents Invalidation

```
void __dpa_thread_window_read_inv();
```

The DPA can cache data that was fetched from external memory using a window. Subsequent memory accesses to the window memory location may return the data that is already cached. In some cases, it is required by the programmer to force a read of external memory (see example under "Polling Externally Set Flag"). In such a situation, the window read contents cached must be dropped.

This function ensures that contents in the window memory space of the thread before the call to `__dpa_thread_window_read_inv()` are invalidated before read operations made by the calling thread after the call to `__dpa_thread_window_read_inv()`.

Window Writeback

```
void __dpa_thread_window_writeback();
```

Writes to external memory must be explicitly written back to be visible to external entities.

This function ensures that contents in the window space of the thread before the call to `__dpa_thread_window_writeback()` are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before any write operation after the call to `__dpa_thread_window_writeback()`.

Memory Writeback

```
void __dpa_thread_memory_writeback();
```

Writes to DPA memory space may need to be written back. For example, the data must be written back before the NIC engines can read it. Refer to the coherency table for more.

This function ensures that the contents in the memory space of the thread before the call to `__dpa_thread_writeback_memory()` are performed and made visible to all threads in the DPA,

host, NIC engines, and peer devices as occurring before any write operation after the call to __dpa_thread_writeback_memory().

Memory Fence and Cache Control Usage Examples

These examples illustrate situations in which programmers must use fences and cache control operations.

In most situations, such direct usage of fences is not required by the application using FlexIO or DOCA DPA SDKs as fences are used within the APIs.

Issuing Send Operation

In this example, a thread on the DPA prepares a work queue element (WQE) that is read by the NIC to perform the desired operation.

The ordering requirement is to ensure the WQE data contents are visible to the NIC engines read it. The NIC only reads the WQE after the doorbell (MMIO operation) is performed. Refer to coherency table.

#	User Code – WQE Present in DPA Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = __DPA_MEMORY)
2	__dpa_thread_memory_writeback();	Cache control operation
3	Write doorbell	MMIO operation via Outbox

In some cases, the WQE may be present in external memory. See the description of flexio_qmem below. The table of operations in such a case is below.

#	User Code – WQE Present in External Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = __DPA_MMIO)
2	__dpa_thread_window_writeback();	Cache control operation

#	User Code – WQE Present in External Memory	Comment
3	Write doorbell	MMIO operation via Outbox

Posting Receive Operation

In this example, a thread on the DPA is writing a WQE for a receive queue and advancing the queue's producer index. The DPA thread will have to order its writes and writeback the doorbell record contents so that the NIC engine can read the contents.

#	User Code – WQE Present in DPA Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = __DPA_MEMORY)
2	__dpa_thread_memory_fence(__DPA_W, __DPA_W);	Order the write to the doorbell record with respect to WQE
3	Write doorbell record	Write to memory locations in the DPA (memory space = __DPA_MEMORY)
4	__dpa_thread_memory_writeback();	Ensure that contents of doorbell record are visible to the NIC engine

Polling Externally Set Flag

In this example, a thread on the DPA is polling on a flag that will be updated by the host or other peer device. The memory is accessed by the DPA thread via a window. The DPA thread must invalidate the contents so that the underlying hardware performs a read.

User Code – Flag Present in External Memory	Comment

`while (!flag) {`
`  __dpa_thread_window_read_inv();`
`}`
	flag is a memory location read using a window

Thread-to-thread Communication

In this example, a thread on the DPA is writing a data value and communicating that the data is written to another thread via a flag write. The data and flag are both in DPA memory.

User Code – Thread 1	User Code – Thread 2	Comment
		Initial condition, flag = 0
var1 = x;	while(*((volatile int *)&flag) !=1);	• Thread 1 - write to var1 • Thread 2 - flag is accessed as a volatile variable, so the compiler preserves the intended program order of reads
__dpa_thread_memory_fence(__DPA_W, __DPA_W);		Thread 1 – write to flag cannot bypass write to var1
	var_t2 = var1;	
flag = 1;	assert(var_t2 == x);	var_t2 must be equal to x

Setting Flag to be Read Externally

In this example, a thread on the DPA sets a flag that is observed by a peer device. The flag is written using a window.

User Code – Flag Present in External Memory	Comment
flag = data;	flag is updated in local DPA memory
__dpa_thread_window_writeback();	Contents from DPA memory for the window are written to external memory

Polling Completion Queue

In this example, a thread on the DPA reads a NIC completion queue and updates its consumer index.

First, the DPA thread polls the memory location for the next expected CQE. When the CQE is visible, the DPA thread processes it. After processing is complete, the DPA thread updates the CQ's consumer index. The consumer index is read by the NIC to determine whether a completion queue entry has been read by the DPA thread. The consumer index is used by the NIC to monitor a potential completion queue overflow situation.

User Code – CQE in DPA Memory	Comment
while(*((volatile uint8_t *)&cq op_own) & 0x1 == hw_owner);	Poll CQ owner bit in DPA memory until the value indicates the CQE is in software ownership. Coherency model ensures update to the CQ is visible to the DPA execution unit without additional fences or cache control operations. Coherency model ensures that data in the CQE or referenced by it are visible when the CQE changes ownership to software.
process_cqe();	User processes the CQE according to the application's logic.
cq cq_index++; // next CQ index. Handle wraparound if necessary	Calculate the next CQ index taking into account any wraparound of the CQ depth.
update_cq_dbr(cq, cq_index); // writes cq_index to DPA memory	Memory operation to write the new consumer index.
__dpa_thread_memory_writeback();	Ensures that write to CQ's consumer index is visible to the NIC. Depending on the application's logic, the __dpa_thread_memory_writeback() may be coalesced or eliminated if the CQ is configured in overrun ignore mode.
arm_cq();	Arm the CQ to generate an event if this handler is going to call flexio_dev_thread_reschedule(). Arming the CQ is not required if the handler calls flexio_dev_thread_finish().

DPA-specific Operations

The DPA supports some platform-specific operations. These can be accessed using the functions described in the following subsections. The operations can be included using the dpaintrin.h header file.

Clock Cycles

```
uint64_t __dpa_thread_cycles();
```

Returns a counter containing the number of cycles from an arbitrary start point in the past on the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This function also acts as a compiler barrier, preventing the compiler from moving instructions around the location where it is used.

Timer Ticks

```
uint64_t __dpa_thread_time();
```

Returns the number of timer ticks from an arbitrary start point in the past on the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This intrinsic also acts as a compiler barrier, preventing the compiler from moving instructions around the location where the intrinsic is used.

Instructions Retired

```
uint64_t __dpa_thread_inst_ret();
```

Returns a counter containing the number of instructions retired from an arbitrary start point in the past by the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the software thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This intrinsic also acts as a compiler barrier, preventing the compiler from moving instructions around the location where the intrinsic is used.

Fixed Point Log2

```
int __dpa_fxp_log2(unsigned int);
```

This function evaluates the fixed point Q16.16 base 2 logarithm. The input is an unsigned integer.

Fixed Point Reciprocal

```
int __dpa_fxp_rcp(int);
```

This function evaluates the fixed point Q16.16 reciprocal (1/x) of the value provided.

Fixed Point Pow2

```
int __dpa_fxp_pow2(int);
```

This function evaluates the fixed point Q16.16 power of 2 of the provided value.

FlexIO

This chapter provides an overview and configuration instr uctions for DOCA FlexIO SDK API.

The DPA processor is an auxiliary processor designed to accelerate packet processing and other data-path operations. The FlexIO SDK exposes an API for managing the DPA device and executing native code over it.

The DPA processor is supported on NVIDIA® BlueField®-3 DPUs and later generations.

After DOCA installation, FlexIO SDK headers may be found under `/opt/mellanox/flexio/include` and libraries may be found under `/opt/mellanox/flexio/lib/`.

Prerequisites

DOCA FlexIO applications can run either on the host machine or on the target DPU.

Developing programs over FlexIO SDK requires knowledge of DPU networking queue usage and management.

Architecture

FlexIO SDK library exposes a few layers of functionality:

- `libflexio` – library for Host-side operations. It is used for resource management.
- `libflexio_dev` – library for DPA-side operations. It is used for data path implementation.
- `libflexio_libc` – a lightweight C library for DPA device code. `libflexio_libc` may expose very partial functionality compared to a standard `libc`.

A typical application is composed of two parts: One running on the host machine or the DPU target and another running directly over the DPA.

API

Please refer to the *NVIDIA DOCA Driver APIs*.

Resource Management

DPA programs cannot create resources. The responsibility of creating resources, such as FlexIO process, thread, outbox and window, as well as queues for packet processing (completion, receive and send), lies on the DPU program. The relevant information should be communicated (copied) to the DPA side and the address of the copied information should be passed as an argument to the running thread.

Example

Host side:

1. Declare a variable to hold the DPA buffer address.

   ```
   flexio_uintptr_t app_data_dpa_daddr;
   ```

2. Allocate a buffer on the DPA side.

   ```
   flexio_buf_dev_alloc(flexio_process, sizeof(struct my_app_data),
   &app_data_dpa_daddr);
   ```

3. Copy application data to the DPA buffer.

   ```
   flexio_host2dev_memcpy(flexio_process, (uintptr_t)app_data, sizeof(struct
   my_app_data), app_data_dpa_daddr);
   ```

   `struct my_app_data` should be common between the DPU and DPA applications so the DPA application can access the struct fields.

The event handler should get the address to the DPA buffer with the copied data:

```
flexio_event_handler_create(flexio_process, net_entry_point,
app_data_dpa_daddr, NULL, flexio_outbox, &app_ctx.net_event_handler)
```

DPA side:

```
__dpa_rpc__ uint64_t event_handler_init(uint64_t thread_arg)
{
    struct my_app_data *app_data;
    app_data = (my_app_data *)thread_arg;
    ...
}
```

DPA Memory Management

As mentioned previously, the DPU program is responsible for allocating buffers on the DPA side (same as resources). The DPU program should allocate device memory in advance for the DPA program needs (e.g., queues data buffer and rings, buffers for the program functionality, etc).

The DPU program is also responsible for releasing the allocated memory. For this purpose, the FlexIO SDK API exposes the following memory management functions:

```
flexio_status flexio_buf_dev_alloc(struct flexio_process *process, size_t buff_bsize,
flexio_uintptr_t *dest_daddr_p);
flexio_status flexio_buf_dev_free(flexio_uintptr_t daddr_p);
flexio_status flexio_host2dev_memcpy(struct flexio_process *process, void
*src_haddr, size_t buff_bsize, flexio_uintptr_t dest_daddr);
flexio_status flexio_buf_dev_memset(struct flexio_process *process, int value, size_t
buff_bsize, flexio_uintptr_t dest_daddr);
```

Allocating NIC Queues for Use by DPA

The FlexIO SDK exposes an API for allocating work queues and completion queues for the DPA. This means that the DPA may have direct access and control over these queues, allowing it to create doorbells and access their memory.

When creating a FlexIO SDK queue, the user must pre-allocate and provide memory buffers for the queue's work queue elements (WQEs). This buffer may be allocated on the DPU or the DPA memory.

To this end, the FlexIO SDK exposes the flexio_qmem struct, which allows the user to provide the buffer address and type (DPA or DPU).

Memory Allocation Best Practices

To optimize process device memory allocation, it is recommended to use the following allocation sizes (or closest to it):

- Up to 1 page (4KB)
- 2^6 pages (256KB)
- 2^{11} pages (8MB)
- 2^{16} pages (256MB)

Using these sizes minimizes memory fragmentation over the process device memory heap. If other buffer sizes are required, it is recommended to round the allocation up to one of the listed sizes and use it for multiple buffers.

DPA Window

DPA windows are used to access external memory, such as on the DPU's DDR or host's memory. DPA windows are the software mechanism to use the Memory Apertures mentioned in section "DPA Memory and Caches". To use the window functionality, DPU or host memory must be registered for the device using the ibv_reg_mr() call.

Both the address and size provided to this call must be 64 bytes aligned for the window to operate. This alignment may be obtained using the posix_memalign() allocation call.

DPA Event Handler

Default Window/Outbox

The DPA event handler expects a DPA window and DPA outbox structs upon creation. These are used as the default for the event handler thread. Users may choose to set one or both to NULL, in which case there would be no valid default value for one/both of them.

Upon thread invocation on the DPA side, the thread context is set for the provided default IDs. If, at any point, the outbox/window IDs are changed, then the thread context on the next invocation is restored to the default IDs. This means that the DPA Window MKey must be configured each time the thread is invoked, as it has no default value.

Execution Unit Management

DPA execution units (EUs) are the equivalent to logical cores. For a DPA program to execute, it must be assigned an EU.

It is possible to set EU affinity for an event handler upon creation. This causes the event handler to execute its DPA program over specific EUs (or a group of EUs).

DPA supports three types of affinity: none , strict, group .

The affinity type and ID, if applicable, are passed to the event handler upon creation using the affinity field of the flexio_event_handler_attr struct.

For more information, please refer to *NVIDIA DOCA DPA Execution Unit Management Tool*.

Execution Unit Partitions

To work over DPA, an EU partition must be created for the used device. A partition is a selection of EUs marked as available for a device. For the DPU ECPF, a default partition is created upon boot with all EUs available in it. For any other device (i.e., function), the user

must create a partition. This means that running an application on a non-ECPF function without creating a partition would result in failure.

FlexIO SDK uses strict and none affinity for internal threads, which require a partition with at least one EU for the participating devices. Failing to comply with this assumption may cause failures.

Virtual Execution Units

Users should be aware that beside the default EU partition, which is exposed to the real EU numbers, all other partitions created use virtual EUs.

For example, if a user creates a partition with the range of EUs 20-40, querying the partition info from one of its virtual HCAs (vHCAs) it would display EUs from 0-20. So, the real EU number, 39 in this example, would correspond to the virtual EU number 19.

Version API and Backward Compatibility

FlexIO SDK supports partial backward compatibility. The may follow one of the following options:

1. Work only with the latest version. The user must align their entire code according to the changes in the FlexIO SDK API listed in the document accompanying each version.

2. Ensure partial backward compatibility for the working code. The user must inform the SDK which version they intend to work with. The SDK provides a set of tools that ensure backward compatibility. The set consists of compile-time and runtime tools.

Version API Toolkit

To support backward compatibility, the FlexIO SDK uses the macros FLEXIO_VER for the host and FLEXIO_DEV_VER for the DPA device. The macros have 3 parameters, where the first is the major version (year), the second is the minor version (month), and the third is the sub-minor version (not used, always 0).

Compile-time

This toolkit is available for both the host and DPA device. The header files flexio_ver.h and flexio_dev_ver.h contain the macros FLEXIO_VER and FLEXIO_VER_LATEST for the host and FLEXIO_DEV_VER and FLEXIO_DEV_VER_LATEST for the DPA device. For example, to set backward compatibility for version 24.04, the user must declare the following construct for the host:

```
#include <libflexio/flexio_ver.h>
#define FLEXIO_VER_USED FLEXIO_VER(24, 4, 0)
#include <libflexio/flexio.h>
```

And the user must declare the following construct for the DPA device:

```
#include <libflexio-dev/flexio_dev_ver.h>
#define FLEXIO_DEV_VER_USED FLEXIO_DEV_VER(24, 4, 0)
#include <libflexio-dev/flexio_dev.h>
```

Where 24 is the major version, and 4 is the minor version.

> ⚠ **Warning**
>
> The files flexio.h and flexio_dev.h have the macros FLEXIO_CURRENT_VERSION and FLEXIO_LAST_SUPPORTED_VERSION for the host FLEXIO_DEV_CURRENT_VERSION and FLEXIO_DEV_LAST_SUPPORTED_VERSION for the DPA device. These versions are provided for internal use and user information. The user should not use these macros.

Runtime

This toolkit is only present for the host. For backward compatibility in runtime, the user can call the function flexio_status flexio_version_set(uint64_t version); in flexio.h once before calling

any other function from the API, with the version parameter they wish to work with. The function returns an error in the following cases:

- If the specified version is less than FLEXIO_LAST_SUPPORTED_VERSION
- If it exceeds FLEXIO_CURRENT_VERSION
- If the function is called again with a version value different from the previous one

```
status = flexio_version_set(FLEXIO_VER(24, 4, 0));
if (status == FLEXIO_STATUS_FAILED)
{
return ERROR;
}
```

It is recommended to use the FLEXIO_VER_USED macro as a parameter :

```
flexio_version_set(FLEXIO_VER_USED);
```

End of Backward Compatibility

The backward compatibility tools are designed to have an endpoint. With each new version, it is possible to gradually raise the value of FLEXIO_LAST_SUPPORTED_VERSION for the host and FLEXIO_DEV_LAST_SUPPORTED_VERSION for the DPA device. If FLEXIO_VER_USED equals FLEXIO_LAST_SUPPORTED_VERSION, then the compiler will issue a warning. This is a sign for the user to start transitioning to a newer version. This way the user has time at least until the next version to modify their code to comply with the older version. If FLEXIO_VER_USED is lower than FLEXIO_LAST_SUPPORTED_VERSION, then the compiler will issue errors. This is a sign for the user to immediately transition to a newer version. T he same behavior for the DPA device.

Application Debugging

Because application execution is divided between the host side and the DPA processor services, debugging may be somewhat challenging, especially since the DPA side does not have a terminal allowing the use of the C stdio library printf services.

Using Device Messaging Stream API

Another logging (messaging) option is to use FlexIO SDK infrastructure to send strings or formatted text in general, from the DPA side to the host side console or file. The host side's flexio.h file provides the flexio_msg_stream_create API function for initializing the required infrastructures to support this. Once initialized, the DPA side must have the thread context, which can be obtained by calling flexio_dev_get_thread_ctx. flexio_dev_msg can then be called to write a string generated on the DPA side to the stream created (using its ID) on the host side, where it is directed to the console or a file, according to user configuration in the creation stage.

It is important to call flexio_msg_stream_destroy when exiting the DPU application to ensure proper clean-up of the print mechanism resources.

Device messages use an internal QP for communication between the DPA and the DPU. When running over an InfiniBand fabric, the user must ensure that the subnet is well-configured, and that the relevant device's port is in active state.

Message Stream Functionality

The user can create as many streams as they see fit, up to a maximum of FLEXIO_MSG_DEV_MAX_STREAMS_AMOUNT as defined in flexio.h.

Every stream has its own messaging level which serves as a filter where messages with a level below that of the stream are filtered out.

The first stream created is the default_stream gets stream ID 0, and it is created with messaging level FLEXIO_MSG_DEV_INFO by default.

The stream ID defined by FLEXIO_MSG_DEV_BROADCAST_STREAM serves as a broadcast stream which means it messagaes all open streams (with the proper messaging level).

A stream can be configured with a synchronization mode attribute according to the following options:

- sync – displays the messages as soon as they are sent from the device to the host side using the verb SEND.

- async – uses the verb RDMA write. When the programmer calls the stream's flush functionality, all the messages in the buffer are displayed (unless there was a wraparound due to the size of messages being bigger than the size allocated for them). In this synchronization mode, the flush should be called at the end of the run.

- batch – uses RDMA write and RDMA write with immediate. It works similarly to the async mode, except the fact each batch size of messages is being flushed and therefore displayed automatically in every batch. The purpose is to allow the host to use fewer resources for device messaging.

Device Messaging Assumptions

Device messaging uses RPC calls to create, modify, and destroy streams. By default, these RPC calls run with affinity none, which requires at least one available EU on the default group. If the user wants to set the management affinity of a stream to a different option (any affinity option is supported, including forcing none, which is the default behavior) they should specify this in the stream attributes using the mgmt_affinity field.

Printf Support

Only limited functionality is implemented for printf. Not all libc printf is supported.

Please consult the following list for supported modifiers:

- Formats – %c, %s, %d, %ld, %u, %lu, %i, %li, %x, %hx, %hxx, %lx, %X, %lX, %lo, %p, %%
- Flags – ., *, -, +, #
- General supported modifiers:
    - "0" padding
    - Min/max characters in string
- General unsupported modifiers:
    - Floating point modifiers – %e, %E, %f, %lf, %LF

- Octal modifier %o is partially supported
- Precision modifiers

Core Dump

If the DPA process encounters a fatal error, the user can create a core dump file to review the application's status at that point using a GDB app.

Creating a core dump file can be done after the process has crashed (as indicated by the flexio_err_status API) and before the process is destroyed by calling the flexio_coredump_create API.

Recommendations for opening DPA core dump file using GDB:

- Use the gdb-multiarch application
- The Program parameter for GDB should be the device-side ELF file
    - Use the dpacc-extract tool (provided with the DPACC package) to extract the device-side ELF file from the application's ELF file

FlexIO Samples

This section describes samples based on the FlexIO SDK. These samples illustrate how to use the FlexIO API to configure and execute code on the DPA.

Running FlexIO Sample

The FlexIO SDK samples serve as a reference for building and running FlexIO-based DPA applications. They provide a collection of out-of-the-box working DPA applications that encompass the basic functionality of the FlexIO SDK.

Documentation

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

Minimal Requirements

The user must have the following installed:

- DOCA DPACC package
- DOCA RDMA package
- pkg-config package
- Python3 package
- Gcc with version 7.0 or higher
- Meson package with version 0.53.0 or higher
- Ninja package
- DOCA FlexIO SDK

Sample Structure

Each sample is situated in its own directory and is accompanied by a corresponding description in README files. Every sample comprises two applications:

- The first, located in the `device` directory, is designed for DPA
- The second, found in the `host` directory, is intended for execution on the DPU or host in a Linux OS environment

Additionally, there is a `common` directory housing libraries for the examples. These libraries are further categorized into `device` and `host` directories to facilitate linking with

similar applications. Beyond containing functions and macros, these libraries also serve as illustrative examples for how to use them.

The list of the samples:

- flexio_rpc – sample demonstrating how to run RPC functions from DPA
- packet_processor – sample demonstrating how to process a package

Building the Samples

```
cd /opt/mellanox/fleio/samples/
./build.sh --check-compatibility --rebuild
```

Samples

flexio_rpc

This sample application executes FlexIO with a remote process call.

The device program calculates the sum of 2 input parameters, prints the result, and copies the result back to the host application.

This sample demonstrates how applications are built (DPA and host), how to create processes and message streams, how to open the IBV device, and how to use RPC from the host to DPA function.

Compilation

```
cd /opt/mellanox/flexio/samples/
./build.sh --check-compatibility --rebuild
```

The output path:

```
/opt/mellanox/flexio/samples/build/flexio_rpc/host/flexio_rpc
```

Usage

```
<sample_root>/build/flexio_rpc/host/flexio_rpc <mlx5_device> <arg1> <arg2>
```

Where:

- mlx5_device – IBV device with DPA
- arg1 – first numeric argument
- arg2 – second numeric argument

Example:

```
$/opt/mellanox/flexio/samples/build/flexio_rpc/host/flexio_rpc mlx5_0 44 55
Welcome to 'Flex IO RPC' sample
Registered on device mlx5_0
/ 2/Calculate: 44 + 55 = 99
Result: 99
Flex IO RPC sample is done
```

flexio_packet_process

This example demonstrates packet processing handling.

The device application implements a handler for flexio_pp_dev that receives packets from the network, swaps MAC addresses, inserts some text into the packet, and sends it back.

This allows the user to send UDP packets (with a packet length of 65 bytes) and check the content of returned packets. Additionally, the console displays the execution of packet processing, printing each new packet index. Device messaging operates in synchronous mode (i.e., each message from the device received by the host is output immediately).

This sample illustrates how applications work with libraries (DPA and host), how to create SQ, RQ, CQ, memory keys, and doorbell rings, how to create and use DPA memory buffers, how to use UAR, and how to create and run event handlers.

Compilation

```
cd /opt/mellanox/flexio/samples/
./build.sh --check-compatibility --rebuild
```

The output path:

```
/opt/mellanox/flexio/samples/build/packet_processor/host/flexio_packet_processor
```

Usage

```
<sample_root>/build/packet_processor/host/flexio_packet_processor
<mlx5_device>
```

Where:

- mlx5_device – name of IB device with DPA
- --nic-mode – optional parameter indicating that the application is run from the host. If the application is run from DPU, then the parameter should not be used.

For example

```
$sudo /build/packet_processor/host/flexio_packet_processor mlx5_0
```

The application must run with root privileges.

Running with Traffic

Run host-side sample:

```
$ cd <sample_root>
$ sudo ./build/packet_processor/host/flexio_packet_processor mlx5_0
```

Use another machine connected to the setup running the application. Bring the interface used as packet generator up:

```
$ sudo ifconfig my_interface up
```

Use scapy to run traffic to the device the application is running on:

```
$ python

>>> from scapy.all import *
>>> from scapy.layers.inet import IP, UDP, Ether

>>> sendp(Ether(src="02:42:7e:7f:eb:02",
dst='52:54:00:79:db:d3')/IP()/UDP()/Raw(load="==============12345678"),
iface="my_interface")
```

Note

Source MAC must be same as above as the application defines a steering rule for it. Destination MAC can be anything.

Note

The load should be kept the same as above, as the application looks for this pattern and changes it during processing.

> **Note**
>
> Interface name should be changed to the interfaced used for traffic generation.

The packets can be viewed using tcpdump:

```
$ sudo tcpdump -i my_interface -en host 127.0.0.1 -X
```

Example output

```
Example output:
11:53:51.422075 02:42:7e:7f:eb:02 > 52:54:00:12:34:56, ethertype IPv4 (0x0800), length 65:
127.0.0.1.domain > 127.0.0.1.domain: 15677 op7+% [b2&3=0x3d3d] [15677a] [15677q]
[15677n] [15677au][|domain]
0x0000: 4500 0033 0001 0000 4011 7cb7 7f00 0001 E..3....@.|.....
0x0010: 7f00 0001 0035 0035 001f 42c6 3d3d 3d3d .....5.5..B.==== <-- Original data
0x0020: 3d3d 3d3d 3d3d 3d3d 3d3d 3d31 3233 3435 ============12345
0x0030: 3637 38 678
11:53:51.700038 52:54:00:12:34:56 > 02:42:7e:7f:eb:02, ethertype IPv4 (0x0800), length 65:
127.0.0.1.domain > 127.0.0.1.domain: 26144 op8+% [b2&3=0x4576] [29728a] [25966q]
[25701n] [28015au][|domain]
0x0000: 4500 0033 0001 0000 4011 7cb7 7f00 0001 E..3....@.|.....
0x0010: 7f00 0001 0035 0035 001f 42c6 6620 4576 .....5.5..B.f.Ev <-- Modified data
0x0020: 656e 7420 6465 6d6f 2a2a 2a2a 2a2a 2a2a ent.demo********
0x0030: 2a2a 2a ***
```

DPA Application Authentication

DPA Application Authentication is supported at beta level for BlueField-3.

DPA Application Authentication is currently only supported with statically linked libraries. Dynamically linked libraries are currently not supported.

This section provides instructions for developing, signing, and using authenticated BlueField-3 data-path accelerator (DPA) applications. It includes information on:

- Principles of root of trust and structures supporting it
- Device ownership transfer/claiming flow (i.e., how the user should configure the device so that it will authenticate the DPA applications coming from the user)
- Crypto signing flow and ELF file structure and tools supporting it

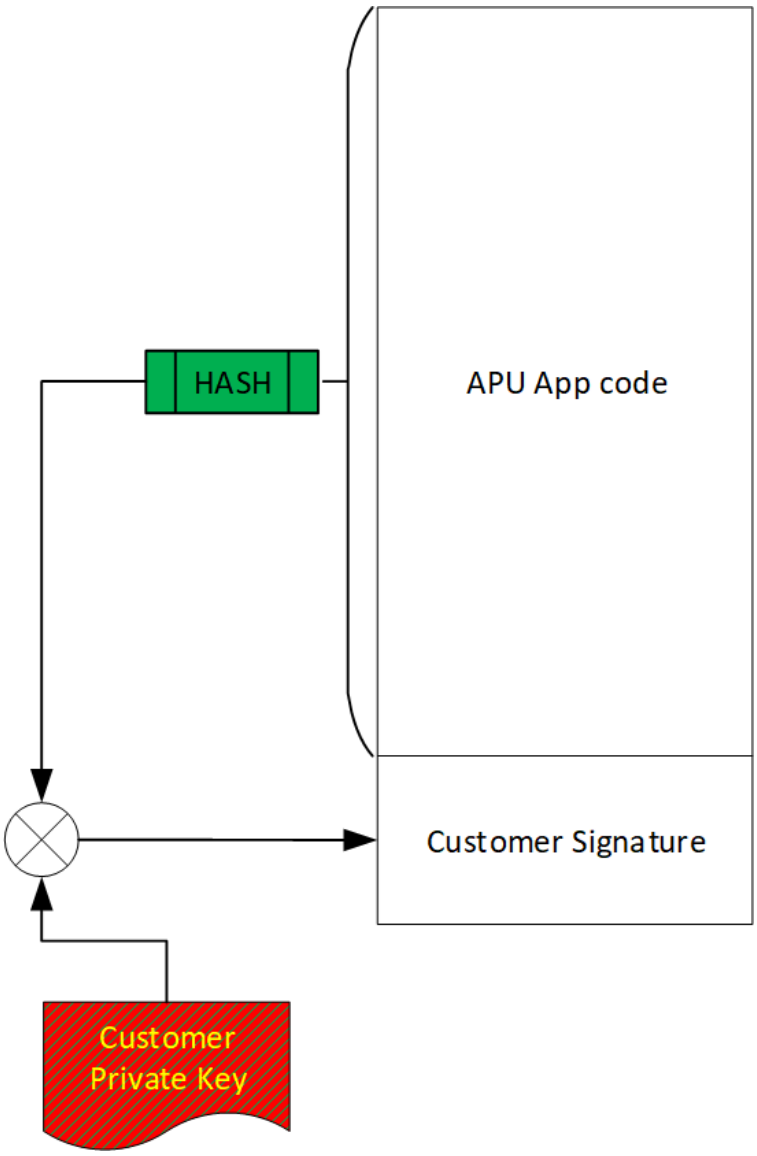
Root of Trust Principles

Signing of 3rd Party DPA App Code

NVIDIA® BlueField®-3 introduces the ability for customers/device owners to sign applications running on the DPA with their private key and have it authenticated by a device-embedded certificate chain. This provides the benefit of ensuring that only code permitted by the customer can run on the DPA. The customer can be any party writing code intended to run on the DPA (e.g., a cloud service provider, OEM, etc).

The following figure illustrates the signature of customer code. This signature will allow NVIDIA firmware to authenticate the source of the application's code.

Example of Customer DPA Code Signed by Customer for Authentication



The high-level scheme is as follows (see figure "Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device"):

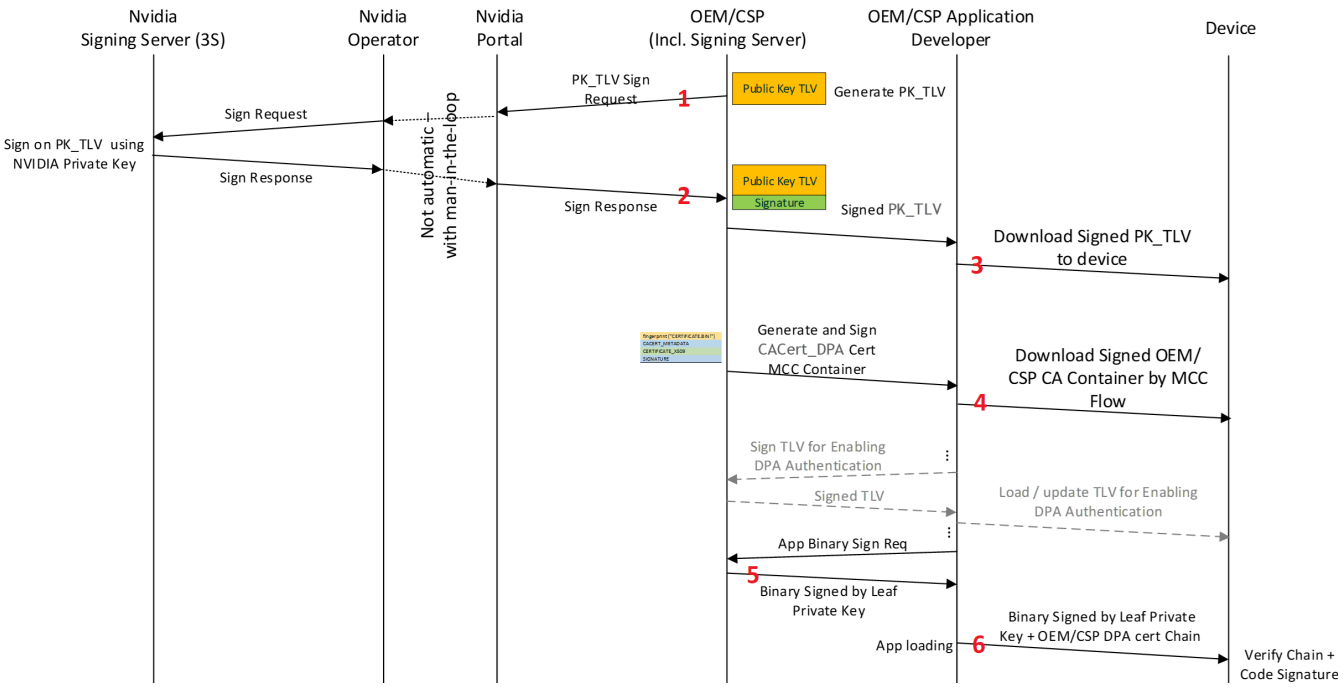
The numbers of these steps correspond to the numbers indicated in the figure below.

1. Customer provides NVIDIA Enterprise Support the public key for device ownership.
2. NVIDIA signs the customer's public key and sends it back to the customer.
3. Customer uploads the NVIDIA-signed public key to the device, enabling "Transfer of Ownership" to the customer (from NVIDIA).
4. Using the private key corresponding to the public key uploaded to the device, the customer can now enable DPA authentication and load the root certificate used for authentication of DPA App code.

5. DPA app code crypto-signed by the customer serves to authenticate the source of the app code.

   The public key used to authenticate the DPA app is provided as part of the certificate chain (leaf certificate), together with the DPA firmware image.
6. App code and the owner signature serves to authorize the app execution by the NVIDIA firmware (similar to NVIDIA own signature).

Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device



The following sections provide more details about this high-level process.

Verification of Authenticity of DPA App Code

Authentication of application firmware code before authorization to execute shall consist of validation of the customer certificate chain and customer signature using the

customer's public key.

Public Keys (Infrastructure, Delivery, and Verification)

For the purposes of the authentication verification of the application firmware, the public key must be securely provided to the hardware. To do so, a secure Management Component Control (MCC) Flow shall be used. Using this, the content of the downloaded certificate is enveloped in an MCC Download Container and signed by NVIDIA Private Key.

The following is an example of how to use the MCC flow describes in detail the procedures, tools and structures supporting this (Section "Loading of CSP CA Certificates and Keys and Provisioning of DPA Firmware to Device" describes the high-level flow for this).

The following command burns the certificate container:

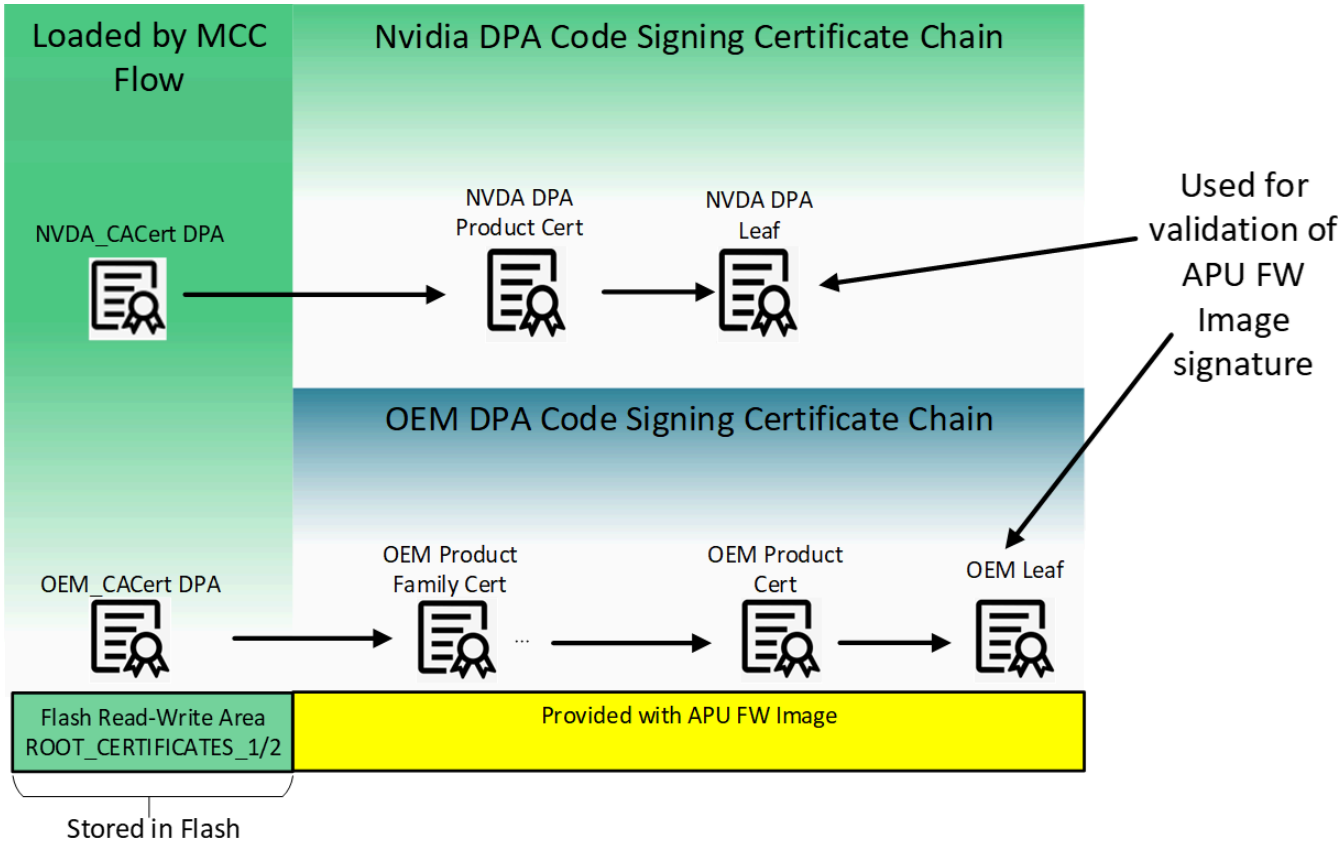
```
flint -d <mst device> -i <signed-certificate-container> burn
```

Two use cases are possible:

- The DPA application is developed internally in NVIDIA, and the authentication is based on internal NVIDIA keys and signing infrastructure
- The DPA application is developed by a customer, and the authentication is based on the customer certificate chain

In either case, the customer must download the relevant CA certificate to the device.

ROT Certificate Chain



This figure illustrates the build of the certificate chain used for validation of DPA app images. The leaf certificate of these chains is used to validate the DPA application supplied by the customer (with ROT from customer CA). The NVIDIA certificate chain for validation of DPA applications (built internally in NVIDIA) is structured in a very similar way. OEMDpaCert CA is the root CA which can be used by the customer to span their certificate chain up to the customer leaf certificate which is used for validating the signature of the application's image. Similarly, NVDADpaCert CA is the root CA used internally in NVIDIA to build the DPA certificate chain for validation of NVIDIA DPA apps.

Customer private keys must be kept secure and are the sole responsibility of the customer to maintain. It is recommended to have a set of keys ready and usable by customer for redundancy purposes.The whole customer certificate chain, including root CA and leaf, must not exceed 4 certificates.

The NVDA_CACert_DPA and OEM_CACert_DPA certificates are self-signed and trusted because they are loaded by the secure MCC flow and authenticated by the firmware.

The customer certificate chain beyond OEM_CACert_DPA is delivered with the DPA image, including the leaf certificate that is used for validating the cryptographic signature of the DPA firmware (see table "ELF Crypto Data Section Fields Description").

For more details on the certificates and their location in the flash, contact NVIDIA

Enterprise Support to obtain the *Flash Application Note*. The rest of the certificate chain used for the DPA firmware authentication includes:

- For NVIDIA-signed images (e.g., figure "ROT Certificate Chain"): NVDA DPA root certificate (NVDA_CACert_DPA can be downloaded here)
- For customer-signed images (e.g., figure "ROT Certificate Chain"): Customer CA certificate, customer product, and customer leaf certificates

In both cases (NVIDIA internal and customer-signed) these parts of the certificate chain are attached to the DPA firmware image.

Loading of CSP CA Certificates and Keys and Provisioning of DPA Firmware to Device

The figure "Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device" shows, at high-level, the procedures for loading user public keys to the device, signing and loading of customer certificates MCC container, and downloading the DPA firmware images.

For clarity, the hierarchy of ROT validation is as follows:

1. Customer public key to be used for customer TLVs and CACert_DPA certificate validation, PK_TLV (i.e., NV_LC_NV_PUBLIC_KEY):
    1. For a device whose DPA authentication ability the customer wishes to enable for the first time, the customer must get it signed and authenticated by NVIDIA keys by reaching out to NVIDIA Enterprise Support. The complete flow is described in "Device Ownership Claiming Flow".
    2. After PK_TLV is loaded, it can be updated by authenticating the update using either the same PK_TLV. The complete flow is described in "Device Ownership Claiming Flow".
    3. Authentication of TLV for enabling/disabling DPA authentication is also validated by the PK_TLV. The complete flow is described in section "DPA Authentication Enablement".
2. Loading of CA certificate (CACert_DPA) to be used for DPA code validation. It is authenticated using the same PK_TLV.

    The complete flow is described in "Uploading DPA Root CA Certificate".

3. The public key in the leaf of the certificate chain anchored by CACert_DPA is used for authentication of the DPA firmware Image.

   The structure of the ELF file containing the DPA app and the certificate chain is described in "ELF File Structure".

A scalable and reliable infrastructure is required to support many users. The customer must also have an infrastructure to support their own code signing process according to their organization's security policy. Both matters are out of the scope of this document.

> **Note**
>
> Trying to utilize the DPA signing flow in a firmware version prior to DOCA 2.2.0 is not supported.

Device Ownership Claiming Flow

NVIDIA networking devices allow the user of the device to customize the configurations, and in some cases change the behavior of the device. This set of available customizations is controlled by higher level NVIDIA configurations that come either as part of the device firmware or as a separate update file. To allow customers/device owners to change the set of available configurations and allowed behaviors, each device can have a device owner who is allowed to change the default behaviors and configurations of the device, and to change what configurations are exposed to the user.

The items controlled by the customer/device owner are:

- Device configurations: The customer/device owner can change the default value of any configuration available to users. They can also prevent users from changing the value.

- Trusted root certificates: The customer/device owner can control what root certificates the device trusts. These certificates control various behaviors (e.g., what 3rd party code the BlueField DPA accepts).

After the device has the public key of the owner, whenever an NVconfig file is signed with this key, one of two things must be true:

- The nv_file_id field in the NVconfig file must have the parameter keep_same_priority as True; or
- The NVconfig file must contain the public key itself (so the public key is rewritten to the device)

Otherwise, the public key is removed from the device, and as such will not accept files signed by the matching private key.

Detailed Ownership Claiming Flow

1. Customer generates a private-public key pair, and a UUID for the key pair.
    1. Generating UUID for the key pair:

        ```
        uuidgen -t
        ```

        Example output:

        ```
        77dd4ef0-c633-11ed-9e20-001dd8b744ff
        ```
    2. Generating an RSA key pair:

        ```
        openssl genrsa -out OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.pem 4096
        ```

        Example output:

        ```
        Generating RSA private key, 2048 bit long modulus
        ..........+++
        ..............+++
        ```

```
e is 65537 (0x10001)
```

3. Extracting the public key file from the RSA key pair:

```
openssl rsa -in OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.pem -out
OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.public -pubout -outform
PEM
```

Output:

```
writing RSA key
```

The public key should look similar to the following:

```
-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAxfijde+27A3pQ7MoZ
mtpyuHO1JY9AUeKaHUXkWRiopL9Puswx1KcGfWJSNzlEPZRevTHraYlLQCru4
W9NBE/qIwS2n7kiFwCCvZK6FKUUqZAuMJTpfuNtv9o4C4v0ZiX4TQqWDND8
hPf3QLRiJ/ux4G6uHIFwENSwagershuKD0RI6BaZ1g9S9IxdXcD0vTdEuDPqQ0
CwEs/3xnksNRLUM+TiPEZoc5MoEoKyJv4GFbGttabhDCt5sr9RqAqTNUSDI9B
XoQBQQpqRgYd3lQ31Fhh3G9GjtoAcUQ6l0Gct3DXKFTAADV3Lyo1vjFNrOKU
pjDKzNmZAsxyIZI0buc24TCgj1yPyFboJtpnHmltyxfm9e+EJsdSIpRiX8YTWwkN
aIzNj08VswULwbKow5Gu5FFpE/uXDE3cXjLOUNnKihszFv4qkqsQjKaK4GszXg
jfiEwsDKwS+cuWd9ihnyLrIWF23+OX0S5xjFXDJE8UthOD+3j3gGmP3kze1Iz2Y
Qvh3ITPRsqQltaiYh+CivqaCHC0voIMOP1ilAEZ/rW85pi6LA8EsudNMG2ELrUy
SznBzZI/OxMk4qKx9nGgjaP2YjmcPw2Ffc9zZcwl57ThEOhlyS6w3E9xwBvZIN
gMuOIWsu1FK3lIGxMSCUZQsCAwEAAQ==
-----END PUBLIC KEY-----
```

2. Customer provides NVIDIA Enterprise Support the public key for device ownership with its UUID.
3. NVIDIA generates a signed NVconfig file with this public key and sends it to the customer. This key may only be applied to devices that do not have a device ownership key installed yet.
4. Customer uses mlxconfig to install the OEM key on the needed devices.

```
mlxconfig -d /dev/mst/<dev> apply oem_public_key_nvconfig.bin
```

To check if the upload process has been successful, the customer can use mlxconfig to query the device and check if the new public key has been applied. The relevant parameters to query are LC_NV_PUB_KEY_EXP, LC_NV_PUB_KEY_UUID, and LC_NV_PUB_KEY_0_255.

Example of query command and expected response:

```
mlxconfig -d <dev>-e q LC_NV_PUB_KEY_0_255
```

Uploading DPA Root CA Certificate

After uploading a device ownership public key to the device, the owner can upload DPA root CA certificates to the device. There can be multiple DPA root CA certificates on the device at the same time.

If the owner wants to upload authenticated DPA apps developed by NVIDIA, they must upload the NVIDIA DPA root CA certificate found here.

If the owner wants to sign their own DPA apps, they must create another public-private key pair (in addition to the device ownership key pair), create a certificate containing the DPA root CA public key, and create a container with this certificate using mlxdpa.

To upload a signed container with a DPA root CA certificate to the device, mlxdpa must be used. This can be done both for either NVIDIA or customer-created certificates.

Generating DPA Root CA Certificate

1. Create a DER encoded certificate containing the public key used to validate DPA apps.

    1. Generating a certificate and a new key pair:

        ```
        openssl req -x509 -newkey rsa:4096 -keyout OEM-DPA-root-CA-key.pem -outform der -out OEM-DPA-root-CA-cert.crt -sha512 -nodes -subj
        ```

```
"/C=XX/ST=OEMStateName/L=OEMCityName/O=OEMCompanyName/OU=C
-days 3650
```

> **Note**
> Both SHA256 and SHA512 are supported in cert. Only a RSA 4096 key is supported. The size of each certificate in DER format must be less than 1792 bytes.

Output:

```
Generating a 4096 bit RSA private key
......++
.....................++
writing new private key to 'OEM-DPA-root-CA-key.pem'
-----
```

2. Create a container for the certificate and sign it with the device ownership private key.

    1. To create and add a container:

    ```
    mlxdpa --cert_container_type add -c <cert.der> -o <path to output> --
    life_cycle_priority <Nvidia/OEM/User> create_cert_container
    ```

    Output example:

    ```
    Certificate container created successfully!
    ```

    2. To sign a container:

    ```
    mlxdpa --cert_container <path to container> -p <key file> --keypair_uuid
    <uuid> --cert_uuid <uuid> --life_cycle_priority <Nvidia/OEM/User> -o
    <path-to-output> sign_cert_container
    ```

```
Certificate container signed successfully!
```

Manually Signing Container

If the server holding the private key cannot run mlxdpa, it is possible to manually sign the certificate container and add the signature to the container. In that case, the following process should be followed:

1. Generate unsigned cert container:

```
mlxdpa --cert_container_type add -c <.DER-formatted-certificate> -o <unsigned-container-path> --keypair_uuid <uuid> --cert_uuid <uuid> --life_cycle_priority OEM create_cert_container
```

2. Generate signature field header:

```
echo "90 01 02 0C 10 00 00 00 00 00 00 00" | xxd -r -p - <signature-header-path>
```

3. Generate signature of container (in whatever way, this is an example only):

```
openssl dgst -sha512 -sign <private-key-pem-file> -out <container-signature-path> <unsigned-container-path>
```

4. Concatenate unsigned container, signature header, and signature into one file:

```
cat <unsigned-container-path> <signature-header-path> <container-signature-path> > <signed-container-path>
```

Uploading Certificates

Upload each signed container containing the desired certificates for the device.

```
flint -d <dev> -i  <signed-container> -y b
```

Output example:

```
-I- Downloading FW ...
FSMST_INITIALIZE -   OK
Writing DIGITAL_CACERT_REMOVAL component -   OK
-I- Component FW burn finished successfully.
```

Removing Certificates

To remove root CA certificates from the device, the user must apply a certificate removal container signed by the device ownership private key.

There are two ways to remove certificates, either removing all certificates, or removing all installed certificates:

- Removing all root CA certificates from the device:
  1. Create a certificate container.

     ```
     mlxdpa --cert_container_type remove --remove_all_certs -o <path-to-output> --life_cycle_priority <Nvidia/OEM/User> create_cert_container
     ```

     Output example:

     ```
     Certificate container created successfully!
     ```

  2. Sign the certificate container.

     ```
     mlxdpa --cert_container <path-to-container> -p <key-file> --keypair_uuid <uuid> --life_cycle_priority <Nvidia/OEM/User> -o <path-to-signed-container> sign_cert_container
     ```

     Output example:

     ```
     Certificate container signed successfully!
     ```

  3. Apply the container to the device.

     ```
     flint -d <dev> -i <signed-container> -y b
     ```

     Output example:

     ```
     -I- Downloading FW ...
     FSMST_INITIALIZE -   OK
     ```

```
Writing DIGITAL_CACERT_REMOVAL component -   OK
-I- Component FW burn finished successfully.
```

- Removing specific root CA certificates according to their UUID:

  1. Generate a signed container to remove certificate based on UUID.

     ```
     mlxdpa --cert_container_type remove --cert_uuid <uuid> -o <path to output> --life_cycle_priority <Nvidia/OEM/User> create_cert_container
     Certificate container created successfully!

     mlxdpa --cert_container <path to container> -p <key file> --keypair_uuid <uuid> --cert_uuid <uuid> --life_cycle_priority <Nvidia/OEM/User> -o <path to output> sign_cert_container
     Certificate container signed successfully!
     ```

  2. Apply the container to the device:

     ```
     flint -d <dev> -i <signed container> -y b
     ```

     Output:

     ```
     -I- Downloading FW ...
     FSMST_INITIALIZE -   OK
     Writing DIGITAL_CACERT_REMOVAL component -   OK
     -I- Component FW burn finished successfully.
     ```

DPA Authentication Enablement

After the device has a device ownership key and DPA root CA certificates installed, the owner of the device can enable DPA authentication. To do this, they must create an NVconfig file, sign it with the device ownership private key, and upload the NVconfig to the device.

Generating NVconfig Enabling DPA Authentication

1. Create XML with TLVs to enable DPA authentication.

    1. Get list of available TLVs for this device:

        ```
        mlxconfig -d /dev/mst/<dev> gen_tlvs_file enable_dpa_auth.txt
        ```

        Output:

        ```
        Saving output...
        Done!
        ```

        Example part of the generated text file:

        ```
        file_applicable_to                  0
        file_comment                        0
        file_signature                      0
        file_dbg_fw_token_id                0
        file_cs_token_id                    0
        file_btc_token_id                   0
        file_mac_addr_list                  0
        file_public_key                     0
        file_signature_4096_a               0
        file_signature_4096_b               0
        ...
        ```

    2. Edit the text file to contain the following TLVs:

        ```
        file_applicable_to                  1
        nv_file_id_vendor                   1
        nv_dpa_auth                         1
        ```

    3. Convert the .txt file to XML format with another mlxconfig command:

        ```
        mlxconfig -a gen_xml_template enable_dpa_auth.txt
        enable_dpa_auth.xml
        ```

Output:

```
Saving output...
Done!
```

The generated .xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.mellanox.com/config">
<file_applicable_to ovr_en='1' rd_en='1' writer_id='0'>
   <psid></psid>
   <psid_branch></psid_branch>
</file_applicable_to>

<nv_file_id_vendor ovr_en='1' rd_en='1' writer_id='0'>

   <!-- Legal Values: False/True -->
   <disable_override></disable_override>

   <!-- Legal Values: False/True -->
   <keep_same_priority></keep_same_priority>

   <!-- Legal Values: False/True -->
   <per_tlv_priority></per_tlv_priority>

   <!-- Legal Values: False/True -->
   <erase_lower_priority></erase_lower_priority>
   <file_version></file_version>
   <day></day>
   <month></month>
   <year></year>
   <seconds></seconds>
   <minutes></minutes>
   <hour></hour>

</nv_file_id_vendor>
```

```
<nv_dpa_auth ovr_en='1' rd_en='1' writer_id='0'>
   <!-- Legal Values: False/True -->
   <dpa_auth_en></dpa_auth_en>

</nv_dpa_auth>
</config>
```

4. Edit the XML file and add the information for each of the TLVs, as seen in the following example XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.mellanox.com/config">

<file_applicable_to ovr_en='0' rd_en='1' writer_id='0'>
   <psid>TODO</psid>
   <psid_branch>TODO</psid_branch>
</file_applicable_to>

<nv_file_id_vendor ovr_en='0' rd_en='1' writer_id='0'>
   <disable_override>False</disable_override>
   <keep_same_priority>True</keep_same_priority>
   <per_tlv_priority>False</per_tlv_priority>
   <erase_lower_priority>False</erase_lower_priority>
   <file_version>TODO</file_version>
   <day>TODO</day>
   <month>TODO</month>
   <year>TODO</year>
   <seconds>TODO</seconds>
   <minutes>TODO</minutes>
   <hour>TODO</hour>
</nv_file_id_vendor>

<nv_dpa_auth ovr_en='0' rd_en='1' writer_id='0'>
   <dpa_auth_en>True</dpa_auth_en>
</nv_dpa_auth>
```

```
</config>
```

> **Note**
> In nv_file_id_vendor, keep_same_priority must be True to avoid removing the ownership public key from the device. More information they can be found in section "Device Ownership Claiming Flow".

> **Note**
> The ovr_en should be set to 0. This can ignore user priority changing nv_dpa_auth.

2. Convert XML file to binary NVconfig file and sign it using mlxconfig:

```
mlxconfig -p OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.pem -u 77dd4ef0-c633-11ed-9e20-001dd8b744ff create_conf enable_dpa_auth.xml enable_dpa_auth.bin
```

Output of create_conf command:

```
Saving output...
Done!
```

3. Upload NVconfig file to device by writing the file to the device:

```
mlxconfig -d /dev/mst/<dev> apply enable_dpa_auth.bin
```

Output:

```
Saving output...
Done!
```

4. Verify that the device has DPA authentication enabled by reading the status of DPA authentication from the device:

```
mlxconfig -d /dev/mst/<dev> -e q DPA_AUTHENTICATION
```

Output:

```
Device #1:
----------

Device type:    BlueField3
...
...
Configurations:                                 Default         Current         Next Boot
 RO     DPA_AUTHENTICATION                              True(1)         True(1)         True(1)
```

The DPU's factory default setting is configured with dpa_auth_en=0 (i.e., DPA applications can run without authentication). To prevent configuration change by any user, it is strongly recommended for the customer to generate and install NVconfig with dpa_auth_en=0/1, according to their preferences, with ovr_en=0.

Manually Signing NVconfig File

If the server holding the private key cannot run mlxconfig, it is possible to manually sign the binary NVconfig file and add the signature to the file. In this case, the following process should be followed instead of step 2:

1. Generate unsigned NVconfig bin file from the XML file:

```
mlxconfig create_conf <xml-nvconfig-path> <unsigned-nvconfig-path>
```

2. Generate random UUID for signature:

```
uuidgen -r | xxd -r -p - <signature-uuid-path>
```

3. Generate signature of NVconfig bin file (in whatever way, this is an example only):

```
openssl dgst -sha512 -sign <private-key-pem-file> -out <nvconfig-signature-path> <unsigned-nvconfig-path>
```

4. Split the signature into two parts:

```
head -c 256 <nvconfig-signature-path> > <signature-part-1-path> && tail -c 256 <nvconfig-signature-path> > <signature-part-2-path>
```

5. Add signing key UUID:

```
echo "<signing-key-UUID>" | xxd -r -p - <signing-key-uuid-path>
```

   Use the signing key UUID, which must have a length of exactly 16 bytes, in a format like aa9c8c2f-8b29-4e92-9b76-2429447620e0.
6. Generate headers for signature struct:

```
echo "03 00 01 20 06 00 00 0B 00 00 00 00" | xxd -r -p - <signature-1-header-path>
echo "03 00 01 20 06 00 00 0C 00 00 00 00" | xxd -r -p - <signature-2-header-path>
```

7. Concatenate everything:

```
cat <unsigned-nvconfig-path> <signature-1-header-path> <signature-uuid-path> <signing-key-uuid-path> <signature-part-1-path> <signature-2-header-path> <signature-uuid-path> <signing-key-uuid-path> <signature-part-2-path> > <signed-nvconfig-path>
```

Device Ownership Transfer

The device owner may change the device ownership key to change the owner of the device or to remove the owner altogether.

First Installation

To install the first OEM_PUBLIC_KEY on the device, the user must upload an NVCONFIG file signed by NVIDIA. This file would contain the 3 FILE_OEM_PUBLIC_KEY TLVs of the current user.

Removing Device Ownership Key

Before removing the device ownership key completely, it is recommended that the device owner reverts any changes made to the device since it is not possible to undo them after the key is removed. Mainly, the root CA certificates installed by the owner should be removed.

1. To remove device ownership key completely, follow the steps in section "Generating NVconfig Enabling DPA Authentication" to create an XML file with TLVs.
2. Edit the XML file to contain the following TLVs:

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.mellanox.com/config">

<file_applicable_to ovr_en='0' rd_en='1' writer_id='0'>
   <psid> MT_0000000911</psid>
   <psid_branch> </psid_branch>
</file_applicable_to>

<nv_file_id_vendor ovr_en='0' rd_en='1' writer_id='0'>
   <disable_override>False</disable_override>
   <keep_same_priority>False</keep_same_priority>
   <per_tlv_priority>False</per_tlv_priority>
   <erase_lower_priority>False</erase_lower_priority>
   <file_version>0</file_version>
   <day>17</day>
   <month>7</month>
   <year>7e7</year>
   <seconds>1</seconds>
   <minutes>e</minutes>
```

```
    <hour>15</hour>
</nv_file_id_vendor>
</config>
```

The TLVs in this file are the only TLVs that will have OEM priority after this file is applied, and as the device ownership key will no longer be on the device, the OEM will no longer be able to change the TLVs. To have OEM priority TLVs on the device after removing the device ownership key, add to this XML any TLV that must stay as default on the device.

3. Convert the XML file to a binary NVconfig TLV file signed by the device ownership key as described in section "Generating NVconfig Enabling DPA Authentication".
4. Apply the NVconfig file to the device as described in section "Generating NVconfig Enabling DPA Authentication".

Changing Device Ownership Key

To transfer ownership of the device to another entity, the previous owner can change the device ownership public key to the public key of the new owner.

To do this, they can use an NVconfig file, and include in it the following TLVs:

```
<nv_ls_nv_public_key_0 ovr_en='0' rd_en='1' writer_id='0'>
    <public_key_exp>65537</public_key_exp>
    <keypair_uuid>77dd4ef0-c633-11ed-9e20-001dd8b744ff</keypair_uuid>
</nv_ls_nv_public_key_0>

<nv_ls_nv_public_key_1 ovr_en='0' rd_en='1' writer_id='0'>
    <key>
    c5:f8:a3:75:ef:b6:ec:0d:e9:43:b3:28:66:79:
    66:9a:da:72:b8:73:b5:25:8f:40:51:e2:9a:1d:45:
    e4:59:18:a8:a4:bf:4f:ba:cc:31:d4:a7:06:7d:62:
    52:37:39:44:3d:94:5e:bd:31:eb:69:89:4b:40:2a:
    ee:e2:87:eb:5b:d3:41:13:fa:88:c1:2d:a7:ee:48:
    85:c0:20:af:64:ae:85:29:45:2a:64:0b:8c:25:3a:
    5f:b8:db:6f:f6:8e:02:e2:fd:19:89:7e:13:42:a5:
    83:34:3f:21:cb:ed:4b:84:f7:f7:40:b4:62:27:fb:
```

```
    b1:e0:6e:ae:1c:81:70:10:d4:b0:6a:07:ab:b2:1b:
    8a:0f:44:48:e8:16:99:d6:0f:52:f4:8c:5d:5d:c0:
    f4:bd:37:44:b8:33:ea:43:49:b8:0b:01:2c:ff:7c:
    67:92:c3:51:2d:43:3e:4e:23:c4:66:87:39:32:81:
    28:2b:22:6f:e0:61:5b:1a:db:5a:6e:10:c2:b7:9b:
    2b:f5:1a:80:a9:33:54:48:32:3d:07:48:eb:5e:84:
    01:41:0a:6a:46:06:1d:de:54:37:d4:58:61:dc:6f:
    46:8e:da:00:71:44:3a:97:41:9c:b7:70:d7:28:54:
    c0:00:35:77:2f:2a:35:be:31:4d:ac:e2:94:85:d8:
    53:a6:
    </key>
</nv_ls_nv_public_key_1>

<nv_ls_nv_public_key_2 ovr_en='0' rd_en='1' writer_id='0'>
    <key>
    30:ca:cc:d9:99:02:cc:72:21:92:34:6e:e7:
    36:e1:30:a0:8f:5c:8f:c8:56:e8:26:da:67:1e:69:
    6d:cb:17:e6:f5:ef:84:26:c7:52:22:94:62:5f:c6:
    13:5b:09:0d:68:8c:cd:8f:4f:15:b3:05:0b:c1:b2:
    a8:c3:91:ae:e4:51:69:13:fb:97:0c:4d:dc:5e:32:
    ce:50:d9:ca:8a:1b:33:16:fe:2a:92:ab:10:8c:a6:
    8a:e0:6b:33:5e:07:be:8d:f8:84:c2:c0:ca:c1:2f:
    9c:b9:67:7d:8a:19:f2:2e:b2:16:17:6d:fe:39:7d:
    12:e7:18:c5:5c:32:44:f1:4b:61:38:3f:b7:8f:78:
    06:98:fd:e4:cd:ed:48:cf:66:0f:42:f8:77:21:33:
    d1:b2:a4:25:b5:a8:98:87:e0:a2:be:a6:82:1c:2d:
    2f:a0:83:0e:3f:58:a5:00:46:7f:ad:6f:39:a6:2e:
    8b:03:c1:2c:b9:d3:4c:1b:61:0b:ad:4c:a5:4b:39:
    c1:cd:92:3f:3b:13:24:e2:a2:b1:f6:71:a0:8d:a3:
    f6:62:39:9c:3f:0d:85:7d:cf:73:65:cc:25:e7:b4:
    e1:10:e8:65:c9:2e:b0:dc:4f:71:c0:1b:d9:20:d2:
    de:80:cb:8e:21:6b:2e:d4:52:b7:94:81:b1:31:20:
    94:65:0b
    </key>
</nv_ls_nv_public_key_2>
```

If the transfer is internal, the owner should set keep_same_priority=True in nv_file_id_vendor TLV and only include the 3 nv_ls_nv_public_key_* TLVs, file_applicable_to and nv_file_id_vendor TLVs in the NVconfig file.

If the transfer is to another OEM/CSP, the owner should clean the device (similarly to removing the device ownership key) and set keep_same_priority=False in nv_file_id_vendor TLV.

ELF File Structure

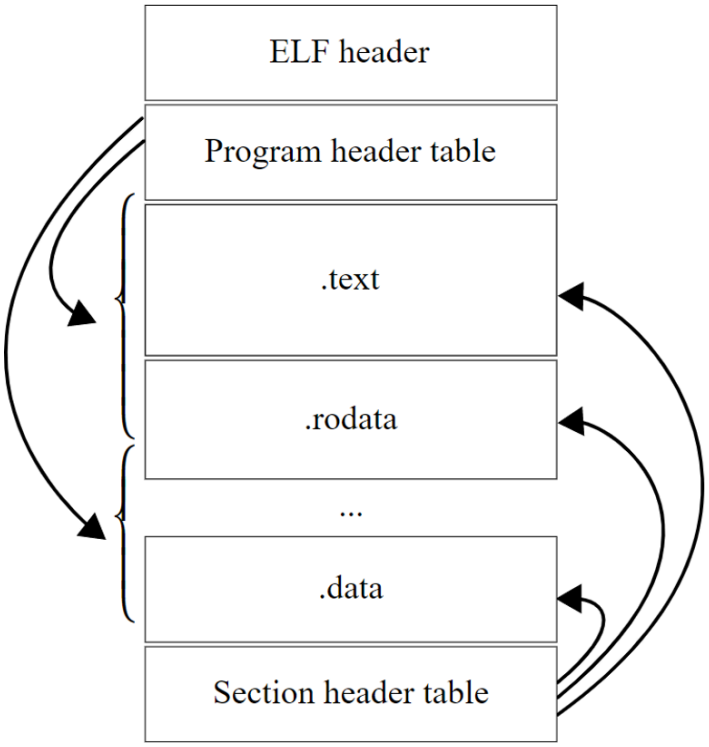
For maximal firmware code reuse, the format of the DPA image loaded from driver should be the same as for the file loaded from flash. As for files loaded from the host, ELF is the default file format. This is chosen as the format for the DPA image, both for flash and for files loaded from the host.

The following figure shows, schematically, a generic ELF file structure.

To support DPA Code authentication additional information needs to be presented to firmware. This info must include:

- Cryptographic signature of the DPA code
- Customer certificate chain including a Leaf Certificate with the public key to be used for signature validation (as described in section "Public Keys (Infrastructure, Delivery, and Verification)")

ELF File Structure Schematic



Crypto Signing Flow

The host ELF includes parts which run on the host, and those that run on DPA. DPA code files are incorporated in the "big" host ELF as binaries. Each host file may include several DPA applications.

When it is required to sign the DPA applications, the following steps need to be performed by the MFT Signing Tool (also see figure "Crypto Signing Flow"):

1. Using ELF manipulation library APIs of DPACC, extract Apps List Table
    1. Input – host ELF
    2. Output – apps list data table to include:
        1. DPA app index
        2. DPA app name
        3. Offset in host ELF
        4. Size of app
        5. Name of corresponding crypto data section

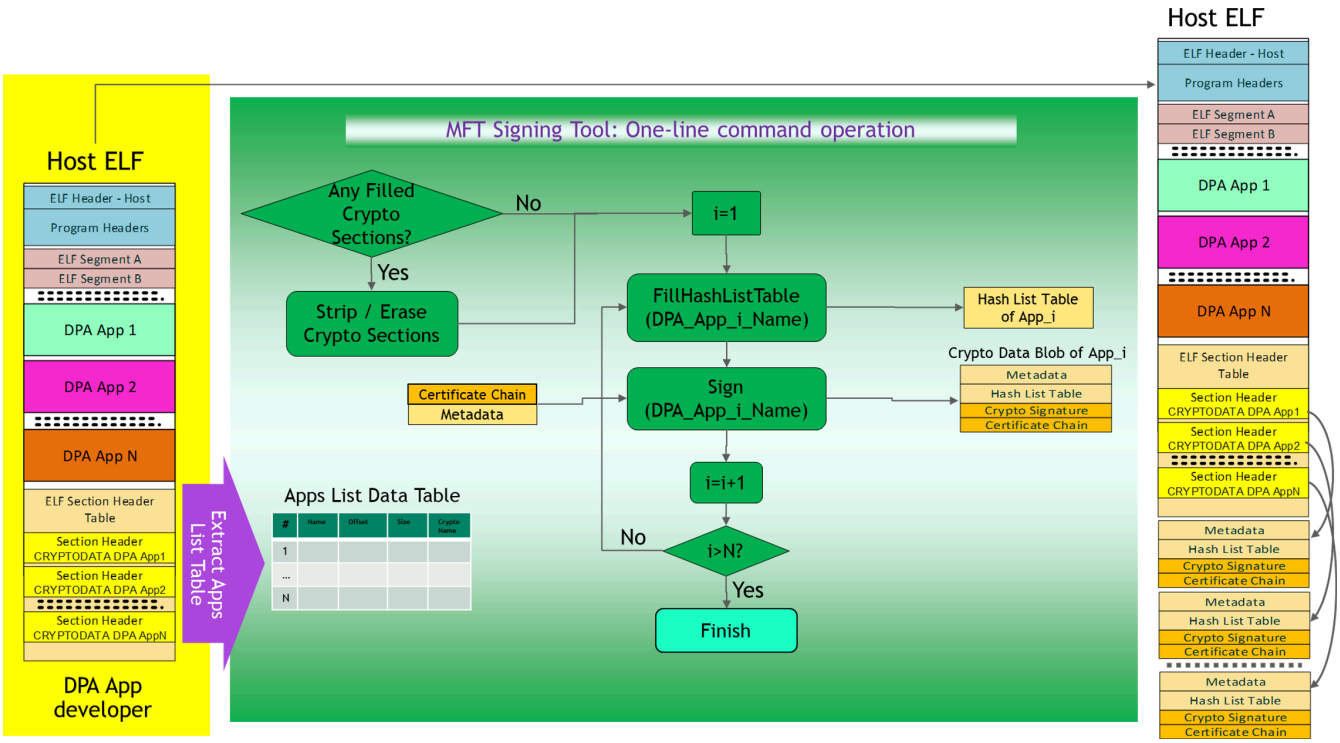
For each DPA application (from i=1 to i=N, N- number of DPA apps in the host ELF) run steps 2 and 3.

2. Fill hash list table:
    - Input: Dpa_App_i
    - Output: Hash list table
3. Sign the crypto data:
    - Input: {Metadata, Hash List Table}, key handle (e.g., UUID from leaf of the Certificate Chain)
    - Output: Crypto_Data "Blob", including: Metadata, Hash List Table, Crypto Signature, Certificate Chain
4. Add crypto data section to host ELF:
    - Inputs: Host ELF, crypto data section name to use
    - Output: File name of host ELF with signature added

The structures used in the flow (hash list table, metadata, etc.) are described in sections "ELF Crypto Data Section Content" and "Hash List Table Layout".

Signing the crypto data may be done using a signing server or a locally stored key.

Crypto Signing Flow



ELF Cryptographic Data Section

This figure shows, schematically, the layout of the cryptographic data section, and the following subsections provide details about the ELF section header and the rest of the structures.

ELF Cryptographic Data Section Layout

Crypto Data ELF Section Header

Defined according to the ELF section header format.

ELF Section Header

Name	Offset	Range	Description
sh_name	0x0	4B	&("Cryptographic Data Section DPA App X") An offset to a string (in the .shstrtab section of ELF) which represents the name of this section
sh_type	0x4	4B	0x70000666 SHT_CRYPTODATA – the section is proprietary and holds crypto information defined in this document
sh_flags	0X8	8B	0 – no flags
sh_addr	0x10	8B	Virtual address of the section in memory, for sections that are loaded
sh_offset	0x18	8B	Offset of the section in the file image

Name	Offset	Range	Description
sh_size	0x20	8B	Size in bytes of the section in the file image. Depends on the content (e.g., presence and type of public key certificate chain and signature).
sh_link	0x28	4B	0 – =SHN_UNDEF, no link information
sh_info	0x2C	4B	0 – no extra information about the section
sh_addralign	0x30	8B	Contains the required alignment of the section. This field must be a power of two.
sh_entsize	0x38	8B	0
	0x40		End of section header (size)

ELF Crypto Data Section Content

ELF Crypto Data Section Fields Description

Name	Offset	Range	Description
metadata_version	0x0	15:0	Version metadata structure format. Initial version is 0.
Reserved (DPA_fw_type)	0x4	15:8	Reserved
Reserved	0x8	31:0	Reserved
Reserved	0xC	31:0	Reserved. Shall be set to all zeros.
Reserved	0x10	16B	Reserved. Shall be set to all zeros.
Reserved	0x20	4 bytes	Reserved. Shall be set to all zeros.
Reserved	0x24	24B	Reserved. Shall be set to all zeros.
signature_type	0x3c	15:0	Signature Type. Only relevant for signed firmware:

Name	Offset	Range	Description
			• 0, 1 – Reserved • 2 – RSA_ SHA_512 • >3 – Reserved
Hash List Table	0x40	HashTableLength	
Crypto Signature	0x40 + HashTableLength	Signature_Length	Signature_Length depends on the signature_type.
Certificate_ Chain	0x40 + HashTableLength + Signature_Length	CrtChain _Length	Structure given the table under section "Certificate Chain Layout".
Padding			FF-padding to align the full size of the data to multiples of DWords (DWs)

The full length of the ELF crypto data section shall be a multiple of DWs (due to firmware legacy implementation). Thus, the MFT (as part of the flow described in figure "Crypto Signing Flow") shall add FF-padding for this structure to align to multiple of DW.

Hash List Table Layout

This table specifies the hash table layout (proposal).

The table contains two parts:

- The 1st part corresponds to the segments of the ELF file, as referenced by the Program Header Table of the EFL file
- The 2nd part corresponds to the sections of the ELF file, as referenced by the Section Header Table

The hash algorithm to be used is SHA-256.

Hash List Table Layout (Proposal)

Name	Offset	Range	Description
Hash Table Magic Pattern	0x0	8 bytes	ASCII "HASHLIST' string: 0x0: 31:24 – "H", 23:16 – "A", 15:8 – "S", 7:0 – "H" 0x4: 31:24 – "L", 23:16 – "I", 15:8 – "S", 7:0 – "T"
Number of Entries – Segments	0x8	7:0	Number of entries in Hashes Segments part, N_Segments.
Reserved	0x8	31:8	Reserved
Number of Entries – Sections	0xc	7:0	Number of entries in Hashes Sections part, N_Sections. Minimum – 0
Reserved	0xc	31:8	Reserved
Reserved	0x10	16 bytes	Reserved
DPA Application ELF Hash	0x20	32 bytes	Hash of the full ELF App file
ELF Header Hash	0x40	32 bytes	Hash of the ELF Header
Program Header Hash	0x60	32 bytes	Hash of the program header
Hash of 1st Segment referenced in the Program Header Table	0x80	32 bytes	Hash of 1st segment referenced in the Program Header Table
Hash of 2nd Segment referenced in the Program Header Table	0xA0	32 bytes	Hash of 2nd Segment referenced in the Program Header Table
……	……	…..	……

Name	Offset	Range	Description
Hash of N_Segments (last) Segment referenced in the Program Header Table	0x60 + N_Segments*0x20	32 bytes	Hash of 2nd segment referenced in the Program Header Table
Section Header Table Hash	0x80 + N_Segments*0x20	32 bytes	Hash of the Section Header Table
Hash of 1st Section referenced in the Section Header Table	+ 0x20	32 bytes	Hash of 1st section referenced in the Section Header Table
Hash of 2nd Section referenced in the Section Header Table	+ 0x20	32 bytes	Hash of 2nd section referenced in the Section Header Table
.....	.....	.....	.....
Hash of N_Sections (last) Section referenced in the Section Header Table	+ 0x20	32 bytes	Hash of N_Sections (last) section referenced in the Section Header Table

The 32-bytes hash fields of different sections/segments in the previous table shall follow Big-Endian convention, as illustrated here:

Hash Fields (Big Endian) Bytes Alignment

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	offset
hash[0]								hash[1]								hash[2]								hash[3]								0h
hash[4]								hash[5]								hash[6]								hash[7]								4h
...								...								...								...								...
hash[28]								hash[29]								hash[30]								hash[31]								1Ch

Certificate Chain Layout

The following table specifies the certificate chain layout. The leaf (the last certificate) of the chain is used as the public key for authentication of the DPA code. This structure is aligned with the certificate chain layout as defined in the *Flash Application Note*.

Certificate Chain Layout

Name	Offset	Range	Description
Type	0x0	3:0	Chain type. Shall be set to 1. 3rd party code authentication certificate chain.
Count	0x0	7:4	Number of certificates in this chain
Length	0x0	23:8	Total length of the certificate chain, in bytes, including all fields in this table
Reserved	0x4	31:0	31:0 – Reserved
CRC	0x8	15:0	The CRC of the header, for header integrity check, covering DWs in 0x0, 0x4
Certificates	0xC-0x1000		One or more ASN.1 DER-encoded X509v3 certificates. The ASN.1 DER encoding of each individual certificate can be analyzed to determine its length. The certificates shall be listed in hierarchical order, with the leaf certificate being the last on the list.

Known Limitations

Supported Devices

- BlueField-3 based DPUs

Supported Host OS

- Windows is not supported

Supported SDKs

- DOCA FlexIO at beta level
- DOCA DPA at beta level

Toolchain

- DPA image-signing and signature-verification are not currently supported
- Debugger (GDB) is currently not supported

FlexIO

- When flexio_dev_outbox_config_uar_extension API is called with a device_id parameter different than PF/ECPF ID (i.e., move to SF/VF outbox) and the APIs flexio_dev_yield(), flexio_dev_print(), or flexio_dev_msg() are called, then when either of those 3 APIs return, the user cannot work with the SF/VF queues.