NVIDIA DOCA SHA

Programming Guide

MLNX-15-060575 _v2.2.1 | October 2023

Table of Contents

Chapter 1. Introduction

The DOCA SHA library provides a flexible and unified API to leverage the SHA offload engine present in the NVIDIA® BlueField® DPU. For more information on SHA (secure hash standard algorithm), please review the FIPS 180-4 specifications.

Important: SHA hardware acceleration is only available on the BlueField-2 DPU. The library will automatically fall back to a software-accelerated solution for BlueField-3 DPUs.

SHA is commonly used in cryptography to generate a given hash value for a supplied input buffer. Depending on the SHA algorithm used, the message length may vary: Any length less than 2^64 bits for SHA-1, SHA-224, and SHA-256, or less than 2^128 bits for SHA-384, SHA-512, SHA-512/224, and SHA-512/256. The resulting output from a SHA operation is called a message digest. The message digests range in length from 160 to 512 bits depending on the selected SHA algorithm. As expected from any cryptography algorithm, any change to a message will, with a very high probability, result in a different message digest and verification failure.

SHA is typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers.

The DOCA SHA library supports three SHA algorithms, SHA-1, SHA-256, and SHA-512, and aims to comply with the OpenSSL SHA implementation standard. It supports both one-shot and stateful SHA calculations.

- One-shot means that the input message is composed of a single segment of data and, therefore, the SHA operation is completed in a single step (i.e., one single SHA engine enqueue and dequeue operation)
- Stateful means that the input message is composed of many segments of data and, therefore, its SHA calculation needs more than one SHA enqueue and dequeue operation to finish. During any stateful operation, other SHA operations can also be executed.

Chapter 2. Prerequisites

DOCA SHA applications can run either on the host machine or directly on the crypto-enabled DPU target. As the DOCA SHA leverages the SHA engine, users must make sure it is enabled:

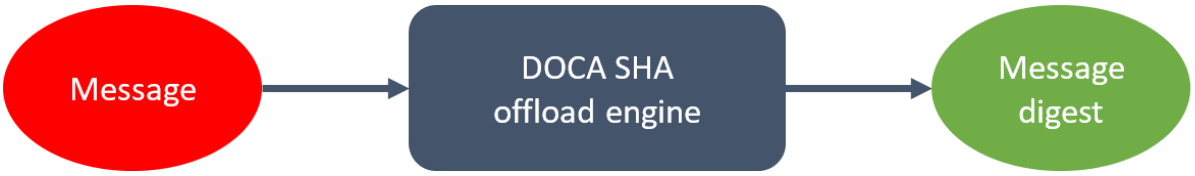
```
$ sudo mlxfwmanager
```

In the output, make sure that `Crypto Enabled` appears in the command output in the `Description` line.

Chapter 3. Architecture

The following diagram shows how the DOCA SHA library receives a message and outputs a message digest.

From an application level, the DOCA SHA library can be seen as a black box. DOCA SHA outputs a response regardless of the nature of the input message.



- In a one-shot SHA situation, the single output is the correct message digest
- In a stateful SHA situation, multiple outputs are expected corresponding to multiple inputs but only the last output is the correct message digest

Chapter 4. API

In the following sections, additional details about the library API are provided. For the library API reference, refer to the NVIDIA DOCA Libraries API Reference Manual.

4.1. doca_sha_job_type

The enum defines six job types in the DOCA SHA library.

```
enum doca_sha_job_type {
    DOCA_SHA_JOB_SHA1 = DOCA_ACTION_SHA_FIRST + 1,
    DOCA_SHA_JOB_SHA256,
    DOCA_SHA_JOB_SHA512,
    DOCA_SHA_JOB_SHA1_PARTIAL,
    DOCA_SHA_JOB_SHA256_PARTIAL,
    DOCA_SHA_JOB_SHA512_PARTIAL,
};
```

DOCA_SHA_JOB_SHA1; DOCA_SHA_JOB_SHA256; DOCA_SHA_JOB_SHA512
Used to specify a one-shot SHA calculation.

DOCA_SHA_JOB_SHA1_PARTIAL; DOCA_SHA_JOB_SHA256_PARTIAL; DOCA_SHA_JOB_SHA512_PARTIAL
Used to specify a stateful SHA calculation.

4.2. DOCA SHA Output Length Macro

These macros define the smallest SHA response buffer length corresponding to different job types.

```
#define DOCA_SHA1_BYTE_COUNT       20
#define DOCA_SHA256_BYTE_COUNT     32
#define DOCA_SHA512_BYTE_COUNT     64
```

DOCA_SHA1_BYTE_COUNT
Number of message digest bytes for `SHA1_PARTIAL` and `SHA1_PARTIAL`.

DOCA_SHA256_BYTE_COUNT
Number of message digest bytes for `SHA256_PARTIAL` and `SHA256_PARTIAL`.

DOCA_SHA512_BYTE_COUNT
Number of message digest bytes for `SHA512_PARTIAL` and `SHA512_PARTIAL`.

4.3. doca_sha_job_flags

The enum defines flags used for `doca_sha_job` construction.

```
enum doca_sha_job_flags {
    DOCA_SHA_JOB_FLAGS_NONE = 0,
    DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_FINAL
};
```

DOCA_SHA_JOB_FLAGS_NONE
The default flag suitable for all SHA jobs.

DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_FINAL
Signifies that the current input is the final segment of a whole stateful job.

4.4. doca_sha_job

This is the DOCA SHA job definition, suitable for one-shot SHA job types, `DOCA_JOB_SHA1/256/512`.

```
struct doca_sha_job {
    struct doca_job base;
    struct doca_buf *req_buf;
    struct doca_buf *resp_buf;
    uint64_t flags;
};
```

base
An opaque `doca_job` structure.

req_buf
The `doca_buf` containing the input message.

resp_buf
The `doca_buf` used for the output message digest.

flags
the `doca_sha_job_flags`.

4.5. doca_sha_partial_session

An opaque structure used in a stateful SHA job.

```
struct doca_sha_partial_session;
```

4.6. doca_sha_partial_job

This is the DOCA SHA job definition, suitable for stateful SHA job types, `DOCA_JOB_SHA1/256/512_PARTIAL`.

```
struct doca_sha_partial_job {
    struct doca_sha_job sha_job;
    struct doca_sha_partial_session *session;
};
```

sha_job
Contain the fields for the input message, output message digest, and flags.
session
Contain the state information for a stateful SHA calculation.

4.7. doca_sha

An opaque structure for DOCA SHA API.

```
struct doca_sha;
```

4.8. doca_sha_create

Before performing any SHA operation, it is essential to create a `doca_sha` object.

```
doca_error_t doca_sha_create(struct doca_sha **ctx);
```

ctx [in/out]
`doca_sha` object to be created.
Returns
`DOCA_SUCCESS` on success, error code otherwise.

4.9. doca_sha_destroy

Used to destroy a `doca_sha` object after a SHA operation is done:

```
doca_error_t doca_sha_destroy(struct doca_sha *ctx);
```

ctx [in]
`doca_sha` object to be destroyed; it is created by `doca_sha_create()`.
Returns
`DOCA_SUCCESS` on success, error code otherwise.

4.10. doca_sha_job_get_supported

Check whether a device can perform `doca_sha` jobs.

```
doca_error_t doca_sha_destroy(struct doca_sha *ctx);
```

devinfo [in]
A pointer to the `doca_devinfo` object.
job_type [in]
`doca_sha` job type enum.
Returns
`DOCA_SUCCESS` on success, error code otherwise.

4.11. doca_sha_get_max_list_buf_num_elem

Get the maximum linked list `doca_buf` count for the source buffer in a `doca_sha` job.

```
doca_error_t doca_sha_get_max_list_buf_num_elem(const struct doca_devinfo *devinfo,
 uint32_t *max_list_num_elem);
```

devinfo [in]
A pointer to the `doca_devinfo` object.

max_list_num_elem [out]
Maximum linked list `doca_buf` count.

Returns
`DOCA_SUCCESS` on success, error code otherwise.

4.12. doca_sha_get_max_src_buffer_size

Get the maximum buffer byte count for the source buffer in a `doca_sha` job.

```
doca_error_t doca_sha_get_max_src_buffer_size(const struct doca_devinfo *devinfo,
 uint64_t *max_buffer_size);
```

devinfo [in]
A pointer to the `doca_devinfo` object.

max_buffer_size [out]
Maximum buffer byte count.

Returns
`DOCA_SUCCESS` on success, error code otherwise.

4.13. doca_sha_get_min_dst_buffer_size

Get the minimum buffer byte count for the destination buffer in a `doca_sha` job.

```
doca_error_t doca_sha_get_max_src_buffer_size(const struct doca_devinfo *devinfo,
 uint64_t *max_buffer_size);
```

devinfo [in]
A pointer to the `doca_devinfo` object.

job_type [in]
`doca_sha` job type enum.

min_buffer_size [out]
Minimum buffer byte count.

Returns
`DOCA_SUCCESS` on success, error code otherwise.

4.14. doca_sha_get_hardware_supported

Check a `doca_sha` engine is hardware-based or openssl-sha-fallback-based.

```
doca_error_t doca_sha_get_hardware_supported(const struct doca_devinfo *devinfo);
```

devinfo [in]
A pointer to the doca_devinfo object.

Returns
DOCA_SUCCESS on success, error code otherwise.

4.15. doca_sha_as_ctx

Convert a doca_sha object into a doca object.

```
struct doca_ctx *doca_sha_as_ctx(struct doca_sha *ctx);
```

ctx [in]
A pointer to the doca_sha object.

doca_ctx [out]
A pointer to the doca object

Returns
A pointer to the doca object on success, NULL otherwise

4.16. doca_sha_partial_session_create

Before doing any stateful SHA calculation, it is necessary to create a doca_sha_partial_session object to keep the state information:

```
doca_error_t doca_sha_partial_session_create(
        struct doca_sha *ctx,
        struct doca_workq *workq,
        struct doca_sha_partial_session **session);
```

ctx [in]
A pointer to the doca_sha object.

workq [in]
A pointer to the doca_workq object.

session [in/out]
A pointer to the doca_sha_partial_session object to be created.

Returns
DOCA_SUCCESS on success, error code otherwise.

4.17. doca_sha_partial_session_destroy

Free stateful SHA session resource:

```
doca_error_t doca_sha_partial_session_destroy(
    struct doca_sha *ctx,
    struct doca_workq *workq,
    struct doca_sha_partial_session *session);
```

ctx [in]
A pointer to the doca_sha object.

workq [in]
A pointer to the doca_workq object.

session [in]
A pointer to the doca_sha_partial_session object to be freed.

Returns

`DOCA_SUCCESS` on success, error code otherwise.

4.18. doca_sha_partial_session_copy

Copy the stateful SHA session resource:

```
doca_error_t doca_sha_partial_session_copy(
        struct doca_sha *ctx,
        struct doca_workq *workq,
        struct doca_sha_partial_session *from,
        struct doca_sha_partial_session *to);
```

ctx [in]

A pointer to the `doca_sha` object.

workq [in]

A pointer to the `doca_workq` object.

from [in]

A pointer to the source `doca_sha_partial_session` object to be copied.

to [out]

A pointer to the destination `doca_sha_partial_session` object.

session [in]

A pointer to the `doca_sha_partial_session` object to be freed.

Returns

`DOCA_SUCCESS` on success, error code otherwise.

Chapter 5. Capabilities and Limitations

Supported SHA algorithms:

- SHA1
- SHA256
- SHA512

Output message digest length:

- 20B for SHA1
- 32B for SHA256
- 64B for SHA512

Maximum single job size:

- For one-shot SHA calculation, the input message size must be ≤ 2^{31}
- For stateful SHA calculation, the accumulated input message size must be ≤ 2^{31}

Stateful SHA job length requirement:

- For `SHA1/256_PARTIAL`, only the last segment allows its `byte_count` != multiple-of-64
- For `SHA512_PARTIAL`, only the last segment allows its `byte_count` != multiple-of-128

Chapter 6. Troubleshooting

6.1. Performing One-shot SHA Calculation

1. Construct a `doca_sha_job`:

```
struct doca_sha_job job = {
 .base.type = DOCA_SHA_JOB_SHA1,
  .req_buf   = user_req_buf,
 .resp_buf  = user_resp_buf,
 .flags     = DOCA_SHA_JOB_FLAGS_NONE
};
```

2. Submit the job until `DOCA_SUCCESS` is received:

```
In synchronous mode, we can use:
ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
 error_exit;
```

   In asynchronous mode, `doca_workq_submit()` may return `DOCA_ERROR_NO_MEMORY`. In that case, you must first call `doca_workq_progress_retrieve()` to receive a response so that the job resource can be freed, then retry calling `doca_workq_submit()`.

   Possible `doca_workq_submit()` return codes:

   - `DOCA_SUCCESS`
   - `DOCA_ERROR_INVALID_VALUE`
   - `DOCA_ERROR_NO_MEMORY`
   - `DOCA_ERROR_BAD_STATE`

   If `doca_workq_submit()` returns `DOCA_ERROR_INVALID_VALUE`, it means the job construction has a problem. If it returns `DOCA_ERROR_BAD_STATE`, it indicates a fatal internal error and the whole engine must be reinitialized.

3. To retrieve a job response until `DOCA_SUCCESS` is received:

```
while ((ret = doca_workq_progress_retrieve(workq, &event,
 DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
 error_exit;
```

   Possible `doca_workq_progress_retrieve()` return codes:

- DOCA_SUCCESS
- DOCA_ERROR_INVALID_VALUE
- DOCA_ERROR_NO_MEMORY
- DOCA_ERROR_BAD_STATE

If `doca_workq_progress_retrieve()` returns `DOCA_ERROR_INVALID_VALUE` it means invalid input is received. If it returns `DOCA_ERROR_IO_FAILED`, it signifies fatal internal error and the whole engine needs reinitialized.

6.2. Performing Stateful SHA Calculation

This section describes the steps to finish a stateful SHA1 calculation, assuming the whole job is composed of three or more segments.

1. Obtain a `doca_sha_partial_session`:
```
doca_sha_partial_session *session;
doca_sha_partial_session_create(ctx, workq, &session);
```
2. Construct a `doca_sha_partial_job` for the first segment:
```
struct doca_sha_partial_job job = {
 .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
  .sha_job.req_buf   = user_req_buf_of_1st_segment,
 .sha_job.resp_buf  = user_resp_buf,
 .sha_job.flags     = DOCA_SHA_JOB_FLAGS_NONE,
 .session           = session,
};
```
3. Submit the job for the first segment:
```
ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
 error_exit;
```
4. Wait until first segment processing is done:
```
while ((ret = doca_workq_progress_retrieve(workq, &event,
 DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
 error_exit;
```
   The purpose of this call is to make sure the first segment processing is finished before continuing to send the next segment, as it is necessary to sequentially process all segments for a correct message digest generation. The `user_resp_buf` at this moment contains garbage values.
5. For the second segment, repeat the previous three steps:
```
struct doca_sha_partial_job job = {
 .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
  .sha_job.req_buf   = user_req_buf_of_2nd_segment,
 .sha_job.resp_buf  = user_resp_buf,
 .sha_job.flags     = DOCA_SHA_JOB_FLAGS_NONE,
 .session           = session,
};

ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
 error_exit;
```

```
while ((ret = doca_workq_progress_retrieve(workq, &event,
 DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
 error_exit;
```

The purpose of this call is still to make sure the second segment processing is finished. The user `user_resp_buf` at this moment still contains garbage values.

6. All subsequent segments repeat the same process.
7. For the last segment, repeat the same process while setting the special flag for the last segment:

```
struct doca_sha_partial_job job = {
 .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
  .sha_job.req_buf   = user_req_buf_of_the_last_segment,
 .sha_job.resp_buf  = user_resp_buf,
 .sha_job.flags     = DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_LAST,
 .session           = session,
};

ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
 error_exit;

while ((ret = doca_workq_progress_retrieve(workq, &event,
 DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
 error_exit;
```

After the `DOCA_SUCCESS` event of the last segment is received the processing of the whole job is done now. You can get the expected SHA message digest from the `user_resp_buf` now.

8. Release the session object:

```
doca_sha_partial_session_destroy(ctx, workq, session);
```

Notes:

- Before submitting the first segment, call `doca_sha_partial_session_create()` to obtain a "session" object.
- During the whole process, make sure to use the same `doca_sha_partial_session` object used for all segments of the entire job.
- If a session object is released before the whole stateful SHA is finished, or if different objects are used for a stateful SHA, the job submission may fail due to job validity check failure. Even the job submission succeeds, a wrong SHA message digest is expected.
- The session resource is limited, it is the user's responsibility to properly call `doca_sha_partial_session_destroy()` to make sure all allocated session objects are released.
- For the last segment, the `DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_FINAL` flag must be set.
- If `DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_FINAL` is not properly set, the engine assumes an intermediate partial SHA calculation and returns an invalid SHA message digest. As only the user knows when the last segment arrives, it is their responsibility to properly set this flag.

- Make sure the `SHA_PARTIAL` segment length requirements are In this example, the first and second segments' byte count must be a multiple of 64. Otherwise, the job submission may fail due to job validity check failure.

6.3. Using Session Copy

This section describes the steps for `utilizing session_copy()` to reduce the stateful SHA calculation overhead.

The example assumes there are two whole jobs, `job_0` and `job_1`, where `job_0` is composed of several segments, `{header_segment, job_0's other segments}`, and `job_1` is composed of `{header_segment, job_1' other segments}`.

1. Obtain two `doca_sha_partial_session`:
```
doca_sha_partial_session *session_0;
doca_sha_partial_session_create(ctx, workq, &session_0);
doca_sha_partial_session *session_1;
doca_sha_partial_session_create(ctx, workq, &session_1);
```
2. Construct a `doca_sha_partial_job` for the `header_segment`:
```
struct doca_sha_partial_job job = {
 .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
  .sha_job.req_buf   = user_req_buf_of_header_segment,
 .sha_job.resp_buf  = user_resp_buf,
 .sha_job.flags     = DOCA_SHA_JOB_FLAGS_NONE,
 .session           = session_0,
};
```
3. Submit the `header_segment` of `job_0`:
```
ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
 error_exit;
```
4. Wait until the processing of `header_segment` is done:
```
while ((ret = doca_workq_progress_retrieve(workq, &event,
 DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
 error_exit;
```
5. Perform the session copy so that `job_1` does not need to calculate its `header_segment`:
```
doca_sha_partial_session_copy(ctx, workq, session_0, session_1);
```
6. Continue to calculate `job_0` and `job_1`'s other segments until final segment using normal `partial_sha` calculation process:
```
struct doca_sha_partial_job job = {
 .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
  .sha_job.req_buf   = user_req_buf_of_job_0_other_segment,
 .sha_job.resp_buf  = user_resp_buf,
 .sha_job.flags     = DOCA_SHA_JOB_FLAGS_NONE,
 .session           = session_0,
};

ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
 error_exit;

while ((ret = doca_workq_progress_retrieve(workq, &event,
 DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
 error_exit;
```

```
struct doca_sha_partial_job job = {
 .sha_job.base.type = DOCA_SHA_JOB_SHA1_PARTIAL,
  .sha_job.req_buf   = user_req_buf_of_job_1_other_segment,
 .sha_job.resp_buf  = user_resp_buf,
 .sha_job.flags     = DOCA_SHA_JOB_FLAGS_NONE,
 .session           = session_1,
};

ret = doca_workq_submit(workq, &job.base);
if (ret != DOCA_SUCCESS)
 error_exit;

while ((ret = doca_workq_progress_retrieve(workq, &event,
 DOCA_WORKQ_RETRIEVE_FLAGS_NONE)) == DOCA_ERROR_AGAIN);
if (ret != DOCA_SUCCESS)
 error_exit;
```

7. Release the session object:

```
doca_sha_partial_session_destroy(ctx, workq, session_0);
doca_sha_partial_session_destroy(ctx, workq, session_1);
```

6.4. Quick Start

Please refer to the NVIDIA DOCA SHA Sample Guide for instructions on how to test the DOCA SHA library.

Chapter 7. DOCA SHA Samples

his section describes SHA samples based on the DOCA SHA library. These samples illustrate how to use the DOCA SHA API to calculate secure hash algorithm on a given message.

7.1. Running the 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 applications.
2. To build a given sample:

```
cd /opt/mellanox/doca/samples/doca_sha/<sample_name>
meson build
ninja -C build
```

> Note: The `doca_<sample_name>` will be created under `./build/`.

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

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

Program Flags:
  -p, --pci-addr                PCI device address
  -d, --data                    User data
```

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

```
./build/doca_<sample_name> -h
```

7.2. Samples

7.2.1. SHA Create

This sample illustrates how to send A SHA job and retrieve the result.

The sample logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA core structures.
3. Populating DOCA memory map with two relevant buffers; one for the source data and one for the result.
4. Allocating the element in DOCA buffer inventory for each buffer.
5. Initializing a DOCA SHA job object.
6. Submitting the SHA job into work queue.
7. Retrieving the SHA job from the queue once it is done.
8. Printing the job result.
9. Destroying all SHA and DOCA core structures.

References:

- `/opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_sample.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_main.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_create/meson.build`

7.2.2. SHA Partial Create

This sample illustrates how to send partial SHA jobs and retrieve the result. Each job source buffer (except the final) will be 64 bytes.

The sample logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA core structures.
3. Initializing a partial session for all the jobs.
4. Populating DOCA memory map with two relevant buffers; one for the source data and one for the result.
5. Allocating the element in DOCA buffer inventory for the result buffer.
6. Calculating total jobs; user data length divided by 64.
7. For each job:
   a). Allocating the element in DOCA buffer inventory for the relevant part in the source buffer.
   b). Initializing the DOCA SHA job object. If it is the final job, send `DOCA_SHA_JOB_FLAGS_SHA_PARTIAL_FINAL` flag.
   c). Submitting SHA job into work queue.
   d). Retrieving SHA job from the queue once it is done.
8. Printing the final job result.

9. Destroying all SHA and DOCA core structures.

References:

- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partial_create_sample.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partial_create_main.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/meson.build`

Notice

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

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

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

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

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

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

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

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

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF 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

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

Copyright

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