NVIDIA DOCA Telemetry

Programming Guide

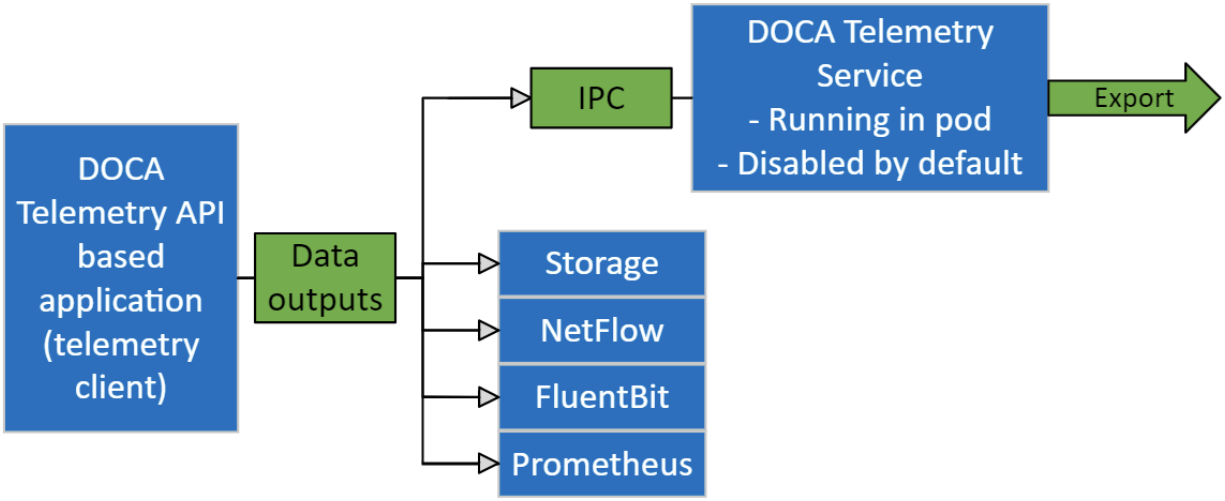
MLNX-15-060506 _v1.5.1 | December 2022

Table of Contents

Chapter 1. Introduction

DOCA Telemetry API offers a fast and convenient way to transfer user-defined data to DOCA Telemetry Service (DTS). In addition, the API provides several built-in outputs for user convenience, including saving data directly to storage, NetFlow, Fluent Bit forwarding, and Prometheus endpoint.

The following figure shows an overview of the telemetry API. The telemetry client side, based on the telemetry API, collects user-defined telemetry and sends it to the DTS which runs as a container on BlueField. DTS does further data routing, including export with filtering. DTS can process several user-defined telemetry clients and can collect pre-defined counters by itself. Additionally, telemetry API has built-in data outputs that can be used from telemetry client applications.



Several scenarios are available

- Send data via IPC transport to DTS. For IPC, refer to Inter-process Communication.
- Write data as binary files to storage (for debugging data format).
- Export data directly from DOCA Telemetry API application using the following options:
  - Fluent Bit exports data through forwarding
  - NetFlow exports data from NetFlow API. Available from both API and DTS. See details in Data Outputs.

- Prometheus creates Prometheus endpoint and keeps the most recent data to be scraped by Prometheus.

Users can either enable or disable any of the data outputs mentioned above. See Data Outputsto see how to enable each output.

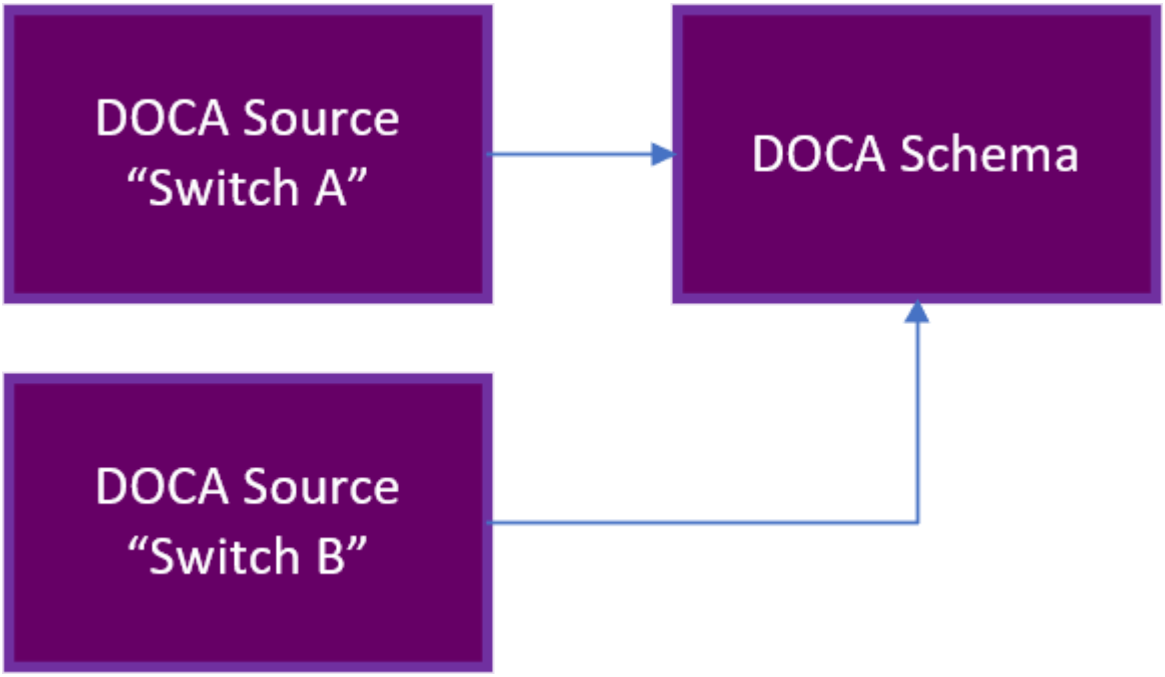
The library stores data in an internal buffer and flushes it to DTS/exporters in the following scenarios:

- Once the buffer is full. Buffer size is configurable with different attributes.
- When `doca_telemetry_source_flush(void *doca_source)` function is invoked.
- When the telemetry client terminates. If the buffer has data, it is processed before the library's context cleanup.

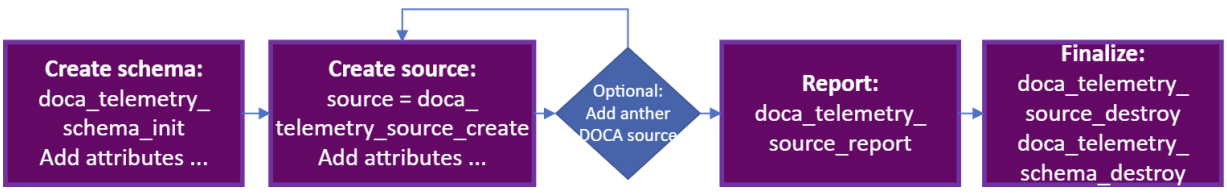
Chapter 2. Architecture

DOCA Telemetry API is fundamentally built around four major parts:

- DOCA schema – defines a reusable structure (refer to `doca_telemetry_type`) of telemetry data which can be used by multiple sources



- Source – the unique identifier of the telemetry source that periodically reports telemetry data.
- Report – exports the information to the DTS
- Finalize – releases all the resources



2.1. DOCA Telemetry API Walkthrough

Here is a basic walkthrough of the needed steps for using the DOCA Telemetry API.

1. Create `doca_schema`.

   a). Initialize an empty schema with default attributes:

   ```
   struct doca_telemetry_schema *doca_schema;
   doca_telemetry_schema_init("example_doca_schema_name", &doca_schema);
   ```

   b). Set the following attributes if needed:

   - `doca_telemetry_schema_set_buffer_*(…)`
   - `doca_telemetry_schema_set_file_write_*(…)`
   - `doca_telemetry_schema_set_ipc_*(…)`

   c). Add user event types:

   Event type (`struct doca_telemetry_type`) is the user-defined data structure that describes event fields. The user is allowed to add multiple fields to the event type. Each field has its own attributes that can be set (see example). Each event type is allocated an index (`doca_telemetry_type_index_t`) which can be used to refer to the event type in future API calls.

   ```
   struct doca_telemetry_type *doca_type;
   struct doca_telemetry_field *field1;

   doca_telemetry_type_create(&doca_type);
   doca_telemetry_field_create(&field1);

   doca_telemetry_field_set_name(field1, "sport");
   doca_telemetry_field_set_description(field1, "Source port")
   doca_telemetry_field_set_type_name(field1, DOCA_TELEMETRY_FIELD_TYPE_UINT16);
   doca_telemetry_field_set_array_length(field1, 1);

   /* The user loses ownership on field1 after a successful invocation of the
    function */
   doca_telemetry_type_add_field(type, field1);

   /* Add more fields if needed */

   /* The user loses ownership on doca_type after a successful invocation of the
    function */
   doca_telemetry_schema_add_type(doca_schema, "example_event", doca_type,
    &type_index);
   ```

   d). Apply attributes and types to start using the schema:

   ```
   doca_telemetry_schema_start(doca_schema)
   ```

2. Create `doca_source`:

   a). Initialize:

   ```
   struct doca_telemetry_source *doca_source;
   doca_telemetry_source_create(doca_schema, &doca_source);
   ```

   b). Set source ID and tag:

   ```
   doca_telemetry_source_set_id(doca_source, "example id");
   doca_telemetry_source_set_tag(doca_source, "example tag");
   ```

c). Apply attributes to start using the source:

```
doca_telemetry_source_start(doca_source)
```

You may optionally add more `doca_sources` if needed.

3. Collect the data per source and use:

```
doca_telemetry_source_report(source, type_index, &my_app_test_ev1, num_events)
```

4. Finalize:

a). For every source:

```
doca_telemetry_source_destroy(source)
```

b). Destroy:

```
doca_telemetry_schema_destroy(doca_schema)
```

Please find example implementation in the `telemetry_export.c` DOCA sample.

2.2. DOCA Telemetry NetFlow API Walkthrough

The DOCA telemetry API also supports NetFlow using DOCA Telemetry NetFlow API. This API is designed to allow customers to easily support the NetFlow protocol at the endpoint side. Once an endpoint produces NetFlow data using the API, the corresponding exporter can be used to send the data to a NetFlow collector.

The NVIDIA DOCA Telemetry Netflow API's definitions can be found in the `doca_telemetry_netflow.h` file.

The following are the steps to use the NetFlow API:

1. Initiate the API with an appropriate source ID:

```
doca_telemetry_netflow_init(source_id)
```

2. Set the relevant attributes:

- `doca_telemetry_netflow_set_buffer_*(…)`
- `doca_telemetry_netflow_set_file_write_*(…)`
- `doca_telemetry_netflow_set_ipc_*(…)`
- `doca_telemetry_netflow_source_set_*(…)`

3. Start the API to use the configured attributes:

```
doca_telemetry_netflow_start();
```

4. Form a desired NetFlow template and the corresponding NetFlow records.
5. Collect the NetFlow data.

```
doca_telemetry_netflow_send(…)
```

6. (Optional) Flush the NetFlow data to send data immediately instead of waiting for the buffer to fill:

```
doca_telemetry_netflow_flush()
```

7. Clean up the API:

```
doca_telemetry_netflow_destroy()
```

You may find example implementation in the `telemetry_netflow_export.c` DOCA sample.

Chapter 3. API

Refer to NVIDIA DOCA Libraries API Reference Manual, for more detailed information on DOCA Telemetry API.

Note: The pkg-config (`*.pc` file) for the telemetry library is named `doca-telemetry`.

The following sections provide additional details about the library API.

Some attributes are optional as they are initialized with default values. Refer to `DOCA_TELEMETRY_DEFAULT_*` in `doca_telemetry_netflow_types.h`.

3.1. DOCA Telemetry Buffer Attributes

Buffer attributes are used to set the internal buffer size and data root used by all DOCA sources in the schema.

Configuring the attributes is optional as they are initialized with default values.

```
doca_telemetry_schema_set_buffer_size(doca_schema, 16 * 1024); /* 16KB - arbitrary
 value */
doca_telemetry_schema_set_buffer_data_root(doca_schema, "/opt/mellanox/doca/
services/telemetry/data/");
```

Where:

- `buffer_size [in]` – the size of the internal buffer which accumulates the data before sending it to the outputs. Data is sent automatically once the internal buffer is full. Larger buffers mean fewer data transmissions and vice versa.
- `data_root [in]` – the path to where data is stored (if `file_write_enabled` is set to true). See section DOCA Telemetry File Write Attributes for more.

3.2. DOCA Telemetry File Write Attributes

File write attributes are used to enable and configure data storage to the file system in binary format.

Configuring the attributes is optional as they are initialized with default values.

```
doca_telemetry_schema_set_file_write_enabled(doca_schema);
doca_telemetry_schema_set_file_write_max_size(doca_schema, 1 * 1024 * 1024); /* 1 MB
 */
```

```
doca_telemetry_schema_set_file_write_max_age(doca_schema, 60 * 60 * 1000000L); /* 1
 Hour */
```

Where:

- `file_write_enable [in]` – use this function to enable storage. Storage/FileWrite is disabled by default.
- `file_write_max_size [in]` – maximum file size (in bytes) before a new file is created.
- `file_write_max_age [in]` – maximum file age (in mircoseconds) before a new file is created.

3.3. DOCA Telemetry IPC Attributes

IPC attributes are used to enable and configure IPC transport. IPC is disabled by default.

Configuring the attributes is optional as they are initialized with default values.

Note: It is important to make sure that the IPC location matches the IPC location used by DTS, otherwise IPC communication will fail.

```
doca_telemetry_schema_set_ipc_enabled(doca_schema);
doca_telemetry_schema_set_ipc_sockets_dir(doca_schema, "/path/to/sockets/");
doca_telemetry_schema_set_ipc_reconnect_time(doca_schema, 100); /* 100 milliseconds
 */
doca_telemetry_schema_set_ipc_reconnect_tries(doca_schema, 3);
doca_telemetry_schema_set_ipc_socket_timeout(doca_schema, 3 * 1000) /* 3 seconds */
```

Where:

- `ipc_enabled [in]` – use this function to enable communication. IPC is disabled by default.
- `ipc_sockets_dir [in]` – a directory that contains UDS for IPC messages. Both the telemetry program and DTS must use the same folder. DTS that runs on BlueField as a container has the default folder `/opt/mellanox/doca/services/telemetry/ipc_sockets`.
- `ipc_reconnect_time [in]` – maximum reconnection time in milliseconds after which the client is considered disconnected.
- `ipc_reconnect_tries [in]` – maximum reconnection attempts.
- `ipc_socket_timeout [in]` – timeout for the IPC socket.

3.4. DOCA Telemetry Source Attributes

Source attributes are used to create proper folder structure. All the data collected from the same host is written to the `source_id` folder under data root.

Note: Sources attributes are mandatory and must be configured before invoking `doca_telemetry_source_start()`.

```
doca_telemetry_source_set_id(doca_source, "example_source");
```

```
doca_telemetry_source_set_tag(doca_source, "example_tag");
```

Where:

- `source_id [in]` – describes the data's origin. It is recommended to set it to the hostname. In later dataflow steps, data is aggregated from multiple hosts/DPUs and `source_id` helps navigate in it.
- `source_tag [in]` – a unique data identifier. It is recommended to set it to describe the data collected in the application. Several telemetry apps can be deployed on a single node (host/DPU). In that case, each telemetry data would have a unique tag and all of them would share a single `source_id`.

3.5. DOCA Telemetry Netflow Collector Attributes

DOCA Telemetry NetFlow API attributes are optional and should only be used for debugging purposes. They represent the NetFlow collector's address while working locally, effectively enabling the local NetFlow exporter.

```
doca_telemetry_netflow_set_collector_addr("127.0.0.1");
doca_telemetry_netflow_set_collector_port(6343);
```

Where:

- `collector_addr [in]` – NetFlow collector's address (IP or name). Default value is `NULL`.
- `collector_port [in]` – NetFlow collector's port. Default value is `0`.

3.6. doca_telemetry_source_report

The source report function is the heart of communication with the DTS. The report operation causes event data to be allocated to the internal buffer. Once the buffer is full, data is forwarded onward according to the set configuration.

```
doca_error_t doca_telemetry_source_report(struct doca_telemetry_source *doca_source,
                                          doca_telemetry_type_index_t index,
                                          void *data,
                                          int count);
```

Where:

- `doca_source [in]` – a pointer to the `doca_telemetry_source` which reports the event
- `index [in]` – the event type index received when the schema was created
- `data [in]` – a pointer to the data buffer that needs to be sent
- `count [in]` – numbers of events to be written to the internal buffer

The function returns `DOCA_SUCCESS` if successful, or a `doca_error_t` if an error occurs. If a memory-related error occurs, try a larger buffer size that matches the event's size.

3.7. doca_telemetry_schema_add_type

This function allows adding a reusable telemetry data struct, also known as a schema. The schema allows sending a predefined data structure to the telemetry service. Note that it is mandatory to define a schema for proper functionality of the library. After adding the scehmas, one needs to invoke the schema start function.

```
doca_error_t doca_telemetry_schema_add_type(struct doca_telemetry_schema
 *doca_schema,
                                             const char *new_type_name,
                                             struct doca_telemetry_type *type,
                                             doca_telemetry_type_index_t
 *type_index);
```

Where:

- `doca_schema [in]` – a pointer to the schema to which the type is added
- `new_type_name [in]` – name of the new type
- `fields [in]` – user-defined fields to be used for the schema. Multiple fields can (and should) be added.
- `type_index [out]` – type index for the created type is written to this output variable

The function returns `DOCA_SUCCESS` if successful, or `doca_error_t` if an error occurs.

Chapter 4. Telemetry Data Format

The internal data format consists of 2 parts: a schema containing metadata, and the actual binary data. When data is written to storage, the data schema is written in JSON format, and the data is written as binary files. In the case of IPC transport, both schema and binary data are sent to DTS. In the case of export, data is converted to the formats required by exporter.

Adding custom event types to the schema can be done using `doca_telemetry_schema_add_type` API call.

Note: See available `DOCA_TELEMETRY_FIELD_TYPE`s in `doca_telemetry.h`. See example of usage in `/opt/mellanox/doca/samples/doca_telemetry/telemetry_export/telemetry_export.c`.

Note: It is highly recommended to have the `timestamp` field as the first field since it is required by most databases. To get the current timestamp in the correct format use:

```
doca_error_t doca_telemetry_timestamp_get(doca_telemetry_timestamp_t
 *timestamp);
```

Chapter 5. Data Outputs

This section describes available exporters:

- IPC
- NetFlow
- Fluent Bit
- Prometheus

Fluent Bit and Prometheus exporters are presented in both API and DTS. Even though DTS export is preferable, the API has the same possibilities for development flexibility.

5.1. Inter-process Communication

IPC transport automatically transfers the data from the telemetry-based program to DTS service.

It is implemented as a UNIX domain socket (UDS) sockets for short messages and shared memory for data. DTS and the telemetry-based program must share the same `ipc_sockets` directory.

When IPC transport is enabled, the data is sent from the DOCA-telemetry-based application to the DTS process via shared memory.

To enable IPC, use the `doca_telemetry_schema_set_ipc_enabled` API function.

Note: IPC transport relies on system folders. For the host's usage, run the DOCA-telemetry-API-based application with `sudo` to be able to use IPC with system folders.

To check the status of IPC for current context, use:

```
doca_error_t doca_telemetry_check_ipc_status(struct doca_telemetry_source
 *doca_source,
                                              doca_telemetry_ipc_status_t *status)
```

If IPC is enabled and for some reason connection is lost, it would try to automatically reconnect on every report's function call.

5.1.1. Using IPC During Development

When developing and testing a DOCA Telemetry based program and its IPC interaction with DTS, some modifications are necessary in DTS's deployment for the program to be able to interact with DTS over IPC:

- Shared memory mapping should be removed: `telemetry-ipc-shm`
- Host IPC should be enabled: `hostIPC`

Before the change:

```
spec:
  hostNetwork: true
  volumes:
  - name: telemetry-service-config
    hostPath:
      path: /opt/mellanox/doca/services/telemetry/config
      type: DirectoryOrCreate
  ...
  - name: telemetry-ipc-shm
    hostPath:
      path: /dev/shm/telemetry
      type: DirectoryOrCreate
  containers:
  ...
      volumeMounts:
      - name: telemetry-service-config
        mountPath: /config
  ...
      - name: telemetry-ipc-shm
        mountPath: /dev/shm
```

After the change:

```
spec:
  hostNetwork: true
  hostIPC: true
  volumes:
  - name: telemetry-service-config
    hostPath:
      path: /opt/mellanox/doca/services/telemetry/config
      type: DirectoryOrCreate
  ...
  containers:
  ...
      volumeMounts:
      - name: telemetry-service-config
        mountPath: /config
```

These changes ensure that a DOCA-based program running outside of a container is able to communicate with DTS over IPC.

5.2. NetFlow

When the NetFlow exporter is enabled (NetFlow Collector Attributes are set), it sends the NetFlow data to the NetFlow collector specified by the attributes: Address and port. This exporter must be used when using DOCA Telemetry Netflow API.

5.3. Fluent Bit

Fluent Bit export is based on `fluent_bit_configs` with `.exp` files for each destination. Every export file corresponds to one of Fluent Bit's destinations. All found and enabled `.exp` files are used as separate export destinations. Examples can be found after running DTS container under its configuration folder (`/opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/`).

All `.exp` files are documented in-place.

```
dpu# ls -l /opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/
/opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/:
total 56
-rw-r--r-- 1 root root  528 Oct 11 07:52 es.exp
-rw-r--r-- 1 root root  708 Oct 11 07:52 file.exp
-rw-r--r--pu 1 root root 1135 Oct 11 07:52 forward.exp
-rw-r--r-- 1 root root  719 Oct 11 07:52 influx.exp
-rw-r--r-- 1 root root  571 Oct 11 07:52 stdout.exp
-rw-r--r-- 1 root root  578 Oct 11 07:52 stdout_raw.exp
-rw-r--r-- 1 root root 2137 Oct 11 07:52 ufm_enterprise.fset
```

Fluent Bit `.exp` files have 2-level data routing:

- `source_tags` in `.exp` files (documented in-place)
- Token-based filtering governed by `.fset` files (documented in `ufm_enterprise.fset`)

To run with Fluent Bit exporter, set `enable=1` in required `.exp` files and set the environment variables before running the application:

```
export FLUENT_BIT_EXPORT_ENABLE=1
export FLUENT_BIT_CONFIG_DIR=/path/to/fluent_bit_configs
export LD_LIBRARY_PATH=/opt/mellanox/collectx/lib
```

5.4. Prometheus

Prometheus exporter sets up endpoint (HTTP server) which keeps the most recent events data as text records.

The Prometheus server can scrape the data from the endpoint while the DOCA-Telemetry-API-based application stays active.

Check the generic example of Prometheus records:

```
event_name_1{label_1="label_1_val", label_2="label_2_val", label_3="label_3_val",
 label_4="label_4_val"} counter_value_1 timestamp_1
event_name_2{label_1="label_1_val", label_2="label_2_val", label_3="label_3_val",
 label_4="label_4_val"} counter_value_2 timestamp_2
...
```

Labels are customizable metadata which can be set from data file. Events names could be filtered by token-based name-match according to .fset files.

Set the following environment variables before running.

```
# Set the endpoint host and port to enable export.
export PROMETHEUS_ENDPOINT=http://0.0.0.0:9101

# Set indexes as a comma-separated list to keep data for every index field. In
```

```
# this example most recent data will be kept for every record with unique
# `port_num`. If not set, only one data per source will be kept as the most
# recent.
export PROMETHEUS_INDEXES=Port_num

# Set path to a file with Prometheus custom labels. Use labels to store
# information about data source and indexes. If not set, the default labels
# will be used.
export CLX_METADATA_FILE=/path/to/labels.txt

# Set the folder which contains fset-files. If set, Prometheus will scrape
# only filtered data according to fieldsets.
export PROMETHEUS_CSET_DIR=/path/to/prometheus_cset
```

Prometheus labels can be obtained from file.

Note: To scrape the data without Prometheus server use:

```
curl -s http://0.0.0.0:9101/metrics
```

Or:

```
curl -s http://0.0.0.0:9101/{fset_name}
```

Chapter 6. Samples

Please refer to the NVIDIA DOCA Telemetry Sample Guide for more information about the API of this DOCA library.

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