NVIDIA DOCA FLOW

Programming Guide

MLNX-15-060488 _v1.1 | October 2021

Table of Contents

Chapter 1. Introduction

DOCA FLOW is the most fundamental API for building generic execution pipes in HW.

The library provides an API for building a set of pipes, where each pipe consists of match criteria, monitoring, and a set of actions. Pipes can be connected where after pipe-defined actions are executed, the packet may proceed to another pipe.

Using DOCA FLOW API, it is easy to develop HW-accelerated applications that have a match on up to two layers of packets (tunneled).

- MAC/VLAN
- IPv4/IPv6
- TCP/UDP/ICMP
- GRE/VXLAN

The execution pipe may include packet modification actions:

- Modify MAC address
- Modify IP address
- Modify L4 (ports, TCP sequences and acknowledgments)
- Strip tunnel
- Add tunnel

The execution pipe may also have monitoring actions:

- Count
- Policers
- Mirror

The pipe also has a forwarding target which may be any of the following:

- Software (RSS to subset of queues)
- Port
- Another pipe

This document is intended for software developers writing network function application that focus on packet processing such as gateways. The document assumes familiarity with network stack and DPDK.

Chapter 2. DOCA FLOW Architecture

2.1. INIT

Before using any DOCA FLOW, it is mandatory to call DOCA FLOW initialization.

```
int doca_flow_init(const struct doca_flow_cfg *cfg, struct doca_flow_error *error);
```

The struct `doca_flow_cfg` contains the following elements:

```
struct doca_flow_cfg {
 uint32_t total_sessions;
 /**< total flows count */
 uint16_t queues;
 /**< queue id for each offload thread */
 bool is_hairpin;
 /**< when true, the fwd will be hairpin queue*/
 bool aging;
 /**< when true, aging is handled by doca */
};
```

Where:

- `total_sessions` - refers to the estimated scale of HW rules
- `queues` – the number of HW acceleration controls queues. It is expected that the same core always uses the same queue_id. In cases where multiple cores are accessing the API with the same queue_id, it is up to the application to lock in between cores/threads.

2.2. PORT

DOCA FLOW API serves as an abstraction layer API for network acceleration. The packet processing in-network function is described from ingress to egress, and therefore a PIPE must be attached to the origin port. Once a packet arrives to the origin port, it will start the HW execution as defined by the DOCA API.

```
/**
 * @brief doca flow port struct
 */
struct doca_flow_port;
```

`doca_flow_port` is an opaque object since the DOCA FLOW API is not bound to a specific packet delivery API such as DPDK. The first step is to start the DOCA FLOW port. The purpose of this step is to attach user application ports to the DOCA ports.

```
struct doca_flow_port *doca_flow_port_start(struct doca_flow_port_cfg *cfg,
              struct doca_flow_error *error);
```

Port configuration contains the following:

- `port_id` – chosen by the user. IDs must start with 0 and be consecutive.
- `type` - depends on underlying API
- `devargs` – a string containing the exact configuration needed according to the type
- `priv_data_size` – per port, users may define private data where application-specific info can be stored

```
struct doca_flow_port_cfg {
        uint16_t port_id;
        /**< dpdk port id*/
        enum doca_flow_port_type type;
        /**< mapping type of port */
        const char *devargs;
        /**< specific per port type cfg */
        uint16_t priv_data_size;
        /**< user private data */
};
```

When DPDK is used, the following configuration must be provided:

```
enum doca_flow_port_type type = DOCA_FLOW_PORT_DPDK_BY_ID
const char *devargs = "1"
```

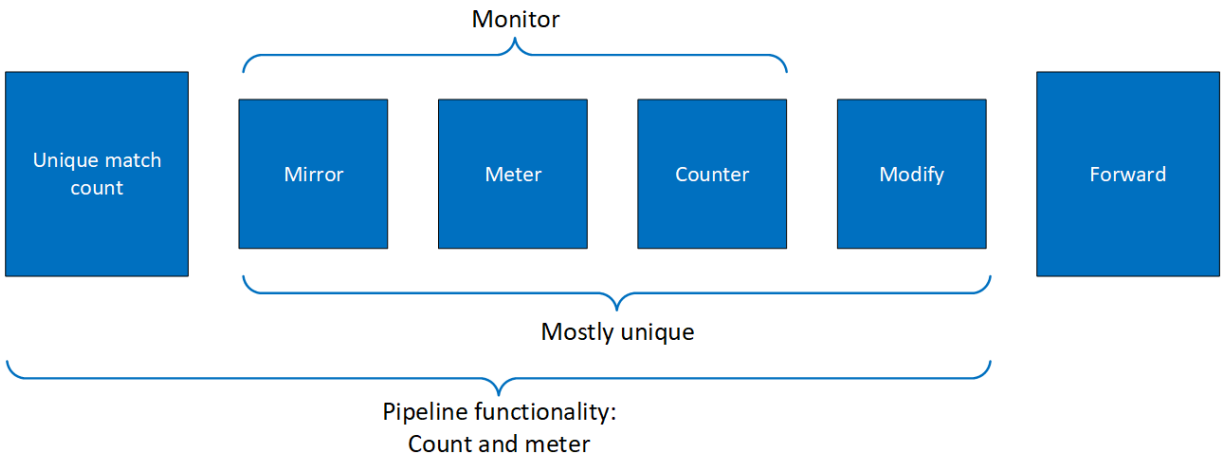
The `devargs` parameter points to a string that has the numeric value of the DPDK `port_id` in decimal. The port must be configured and started before calling this API. Mapping the DPDK port to the DOCA port is required to synchronize application ports with HW ports.

2.3. PIPE

PIPE is a template that defines packet processing without adding any specific HW rule. A PIPE consists of a template that includes the following elements:

- Match
- Monitor
- Actions
- Forward

The following diagram illustrates a PIPE structure.



The creation phase allows the HW to efficiently build the execution PIPE. After the PIPE is created, specific entries can be added. Only a subset of the PIPE can be used (e.g. skipping the monitor completely, or just using the counter, etc).

2.3.1. Setting PIPE Match

Match is a mandatory field when creating a PIPE. Using the following struct, users must define the fields that should be matched on the PIPE.

```
struct doca_flow_match {
 uint32_t flags;
 uint8_t out_src_mac[DOCA_ETHER_ADDR_LEN];
 /**< outer source mac address*/
 uint8_t out_dst_mac[DOCA_ETHER_ADDR_LEN];
 /**< outer destination mac address*/
 doca_be16_t vlan_id;
 /**< outer vlan id*/
 struct doca_flow_ip_addr out_src_ip;
 /**< outer source ip address*/
 struct doca_flow_ip_addr out_dst_ip;
 /**< outer destination ip address*/
 uint8_t out_l4_type;
 /**< outer layer 4 protcol type*/
 doca_be16_t out_src_port;
 /**< outer layer 4 source port*/
 doca_be16_t out_dst_port;
 /**< outer layer 4 destination port*/
 struct doca_flow_tun tun;
 /**< tunnel info*/
 struct doca_flow_ip_addr in_src_ip;
 /**< inner source ip address if tunnel is used*/
 struct doca_flow_ip_addr in_dst_ip;
 /**< inner destination ip address if tunnel is used*/
 uint8_t in_l4_type;
 /**< inner layer 4 protocol type if tunnel is used*/
 doca_be16_t in_src_port;
 /**< inner layer 4 source port if tunnel is used*/
 doca_be16_t in_dst_port;
 /**< inner layer 4 destination port if tunnel is used*/
};
```

For each field, users choose whether the field may be:

- Ignored (wild card) – value of the field is ignored
- Constant – all entries in the PIPE must have the same value for this field. Users should not put a value for each entry.
- Changeable – per entry, the user must provide the value to match

> **Note:** L4 type, L3 type, and tunnel type cannot be changeable.

The match field type can be defined either implicitly or explicitly.

2.3.1.1. Implicit Match

To match implicitly, the following considerations should be taken into account.

2.3.1.1.1. Ignored Fields

- Field is zeroed
- Pipeline has no comparison on the field

2.3.1.1.2. Constant Fields

These are fields that have a constant value. For example, as shown in the following, the tunnel type is VXLAN.

```
match.tun.type = DOCA_FLOW_TUN_VXLAN;
```

These fields only need to be configured once, not once per new pipeline entry.

2.3.1.1.3. Changeable Fields

These are fields that may change per entry. For example, the following shows an inner 5-tuple which are set with a full mask.

```
match.in_dst_ip.a.ipv4_addr = 0xffffffff;
```

If this is the constant value required by user, then they should set zero on the field when adding a new entry.

2.3.1.1.4. Example

The following is an example of a match on the VXLAN tunnel, when for each entry there is a specific IPv4 destination address, and an inner 5-tuple.

```
static void build_underlay_overlay_match(struct doca_flow_match *match)
{
    match->out_dst_ip.ipv4_addr = 0xffffffff;
    match->out_l4_type = DOCA_PROTO_UDP;
    match->out_dst_port = DOCA_VXLAN_DEFAULT_PORT;


    match->tun.type = DOCA_FLOW_TUN_VXLAN;
    match->tun.vxlan.tun_id = 0xffffffff;

    //inner
    match->in_dst_ip.ipv4_addr = 0xffffffff;
    match->in_src_ip.ipv4_addr = 0xffffffff;
    match->in_src_ip.type = DOCA_FLOW_IP4_ADDR;
```

```
    match->in_l4_type = DOCA_PROTO_TCP;
    match->in_src_port = 0xffff;
    match->in_dst_port = 0xffff;
}
```

2.3.1.2. Explicit Match

Users may provide a mask on match. In this case, there are two `doca_flow_match` items: The first will contain constant values, and the second will contain masks.

2.3.1.2.1. Ignored Fields

- Field is zeroed
- Pipeline has no comparison on the field

```
Match_mask.in_dst_ip.ipv4_addr = 0;
```

2.3.1.2.2. Constant Fields

These are fields that have a constant value. For example, as shown in the following, the tunnel type is VXLAN and the mask should be full.

```
match.tun.type = DOCA_FLOW_TUN_VXLAN;
match_mask.tun.type = 0xffffffff;
```

Once a field is defined as constant, the field's value cannot be changed per entry. Users must set constant fields to zero when adding entries to avoid ambiguity.

2.3.1.2.3. Changeable Fields

These are fields that may change per entry (e.g. inner 5-tuple). Their value should be zero and the mask should be full.

```
match.in_dst_ip.ipv4_addr = 0;
match_mask.in_dst_ip.ipv4_addr = 0xffffffff;
```

Note that for IPs, the prefix mask can be used as well.

2.3.2. Setting PIPE Actions

Similarly to setting PIPE match, actions also have a template definition with the following data structure:

```
struct doca_flow_actions {
 bool decap;
 /**< when true, will do decap*/
 uint8_t mod_src_mac[DOCA_ETHER_ADDR_LEN];
 /**< modify source mac address*/
 uint8_t mod_dst_mac[DOCA_ETHER_ADDR_LEN];
 /**< modify destination mac address*/
 struct doca_flow_ip_addr mod_src_ip;
 /**< modify source ip address*/
 struct doca_flow_ip_addr mod_dst_ip;
 /**< modify destination ip address*/
 doca_be16_t mod_src_port;
 /**< modify layer 4 source port*/
 doca_be16_t mod_dst_port;
 /**< modify layer 4 destination port*/
 bool dec_ttl;
```

```
 /**< decrease TTL value*/
 bool has_encap;
 /**< when true, will do encap*/
 struct doca_flow_encap_action encap;
 /**< encap data information*/
};
```

Similarly to `doca_flow_match` in the creation phase, only the subset of actions that should be executed per packet are defined. This is done in a similar way to match, namely by classifying a field to one of the following:

- Ignored field – field is zeroed, modify is not used
- Constant fields – when a field should be modified per packet, but the value is the same for all packets, a one-time value on action definitions can be used
- Changeable fields – fields that may have more than one possible value, and the exact values is set by the user per entry

  ```
  match_mask.in_dst_ip.ipv4_addr = 0xffffffff;
  ```
- Boolean fields – Boolean values, encap and decap are considered as constant values. It is not allowed to generate actions with `encap=true` and to then have an entry without an encap value.

For example:

```
static void
create_decap_inner_modify_actions(struct doca_flow_actions *actions)
{
        actions->decap = true;
        actions->mod_dst_ip.ipv4_addr = 0xffffffff;
}
```

2.3.3. Setting PIPE Monitor

If a policer should be used, then it is possible to have the same configuration for all policers on the PIPE or to have a specific configuration per entry.

```
struct doca_flow_monitor {
 uint8_t flags;
 /**< indicate which actions be included*/
 struct {
  uint32_t id;
  /**< meter id */
  uint64_t cir;
  /**< Committed Information Rate (bytes/second). */
  uint64_t cbs;
  /** Committed Burst Size (bytes). */
 };
 /**< meter action configuration*/
 uint32_t aging;
 /**< aging time in seconds*/
};
```

Where:

- Committed information rate (CIR) – defines the maximum bandwidth
- Committed burst size (CBS) – maximum local burst size

$T(c)$ is the number of available tokens. For each packet where "b" equals the number of bytes, if $t(c)-b \geq 0$ the packet can continue, and tokens are consumed so $t(c)=t(c)-b$. If $t(c)-b<0$, the packet is dropped.

`T(c)` tokens are increased according to time, configured CIR, configured CBS, and packet arrival. When a packet is received prior to anything else, the `t(c)` tokens are filled. The number of tokens is a relative value that relies on the total time passed since last update, but it is limited by CBS value.

2.3.4. Setting PIPE Forwarding

The last "action" in a PIPE directs a packet on where to go next. Users may configure one of the following:

- Send to software (representor)
- Send to wire
- Jump to next PIPE

The FORWARDING action may be set for PIPE create, but it can also be unique per entry.

A PIPE can be defined with constant forwarding (e.g., always send packets on a specific port). In this case, all entries will have the exact same forwarding. If forwarding is not defined when PIPE is created, however, users must define forwarding for each entry. In this instance PIPEs may have different forwarding actions.

```
struct doca_flow_fwd {
        enum doca_flow_fwd_type type;
        /**< indicate the forwarding type*/
        union {
                struct {
                        uint32_t rss_flags;
                        /**< rss offload types*/
                        uint16_t *rss_queues;
                        /**< rss queues array*/
                        int num_of_queues;
                        /**< number of queues*/
                        uint32_t rss_mark;
                        /**< markid of each queues*/
                };
                /**< rss configuration information*/
                struct {
                        uint16_t port_id;
                        /**< destination port id*/
                };
                /**< port configuration information*/
                struct {
                        struct doca_flow_pipe *next_pipe;
                        /**< next pipe pointer*/
                };
                /**< next pipe configuration information*/
        };
};
```

The following is an RSS forwarding example:

```
struct {
   uint32_t rss_flags;
   /**< rss offload types*/
   uint16_t *rss_queues;
   /**< rss queues array*/
   int num_of_queues;
   /**< number of queues*/
   uint32_t rss_mark;
   /**< markid of each queues*/
 };
```

Flags include the RSS fields defined in the following enum:

```
enum doca_rss_type {
        DOCA_FLOW_RSS_IP = (1 << 0),
        /**< rss by ip head*/
        DOCA_FLOW_RSS_UDP = (1 << 1),
        /**< rss by udp head*/
        DOCA_FLOW_RSS_TCP = (1 << 2),
};
```

Queues point to the uint16_t array that contains the queue numbers. When a port is started, the number of queues is defined, starting from zero up to the number of queues minus 1. RSS queue numbers may contain any subset of those predefined queue numbers. For specific match packet can be directed to a single queue by having RSS forwarding with a single queue.

MARK is an optional parameter that may be communicated to the software. If MARK is set and the packet arrives to the software, the value can be examined using the software API. When DPDK is used, MARK is placed on the struct `rte_mbuf`. (See "Action: MARK" section in official DPDK documentation.) When using the Kernel, the MARK value is placed on the struct `sk_buff` MARK field.

The port_id is given in struct `doca_flow_port_cfg`.

The packet will be directed to the port. In many instances the complete PIPE is executed in the HW, including the forwarding of the packet back to the wire. The packet never arrives to the SW.

Example code for forwarding to port:

```
struct doca_flow_fwd *fwd = malloc(sizeof(struct doca_flow_fwd));

memset(fwd, 0, sizeof(struct doca_flow_fwd));
fwd->type = DOCA_FLOW_FWD_PORT;
fwd->port_id = port_cfg->port_id;
```

The type of forwarding is `DOCA_FLOW_FWD_PORT` and the only data required is the `port_id` as defined in `DOCA_FLOW_PORT`.

2.3.5. PIPE Create

Once all parameters are defined, the create function is called.

```
struct doca_flow_pipe *
doca_flow_create_pipe(const struct doca_flow_pipe_cfg *cfg,
        const struct doca_flow_fwd *fwd,
        struct doca_flow_error *error);
```

The return value of the function is a handle to the PIPE. This handle should be given when adding entries to PIPE. If a failure occurs, the function returns `NULL`, and the error reason and message are put in the error argument if provided by the user.

It is possible to not have all options fields. For example, `fwd` is not mandatory, and in PIPE configuration some of the fields might be zeroed when not used.

Once PIPE is created, a new entry can be added to it. These entries are bound to a PIPE, so when a PIPE is destroyed, all the entries in the PIPE are removed. Please refer to section PIPE Entry for more information.

There is no priority between PIPEs or entries. The way that priority can be implemented is to match the highest priority first, and if a miss occurs, to jump to the next PIPE. There can be more than one PIPE on a root as long the PIPEs are not overlapping. If entries are overlapping,

the priority is set according to the order of entries added. So, if two PIPEs have overlapping matching and PIPE1 has higher priority than PIPE2, users should add an entry to PIPE1 after any entry is added to PIPE2.

2.3.6. PIPE Entry (doca_flow_pipe_add_entry)

An entry is a specific instance inside of a PIPE. When defining a PIPE, users define match criteria (subset of fields to be matched), the type of actions to be done on matched packets, monitor, and, optionally, the FORWARDING action.

When adding an entry, users should define the values that are not constant among all entries in the PIPE, and if FORWARDING is not defined then that is also mandatory.

```
struct doca_flow_pipe_entry *doca_flow_pipe_add_entry(uint16_t pipe_queue,
                          struct doca_flow_pipe *pipe,
                          const struct doca_flow_match *match,
                          const struct doca_flow_actions *actions,
                          const struct doca_flow_monitor *mon,
                          const struct doca_flow_fwd *fwd,
                          struct doca_flow_error *error)
```

2.3.6.1. PIPE Entry Queue

DOCA FLOW is designed to support concurrency in an efficient way. Since the expected rate is going to be in millions of new entries per second, it is required to use similar architecture as data path. Having a unique queue ID per core saves the DOCA engine from having to lock the data structure and enables the usage of multiple queues when interacting with HW.

It is expected for a single thread/core to use a unique `pipe_queue` and that the `pipe_queue` is not shared with other threads/cores.

```
struct doca_flow_pipe_entry
```

Upon success, a handle is returned. If a failure occurs, a NULL value is returned, and an error message will be filled. The application can keep this handle and call remove on the entry using its handle.

```
int doca_flow_pipe_rm_entry(uint16_t pipe_queue,
           struct doca_flow_pipe_entry *entry);
```

2.3.6.2. PIPE Entry Counting

By default, no counter is added. If defined in monitor, a unique counter is added per entry.

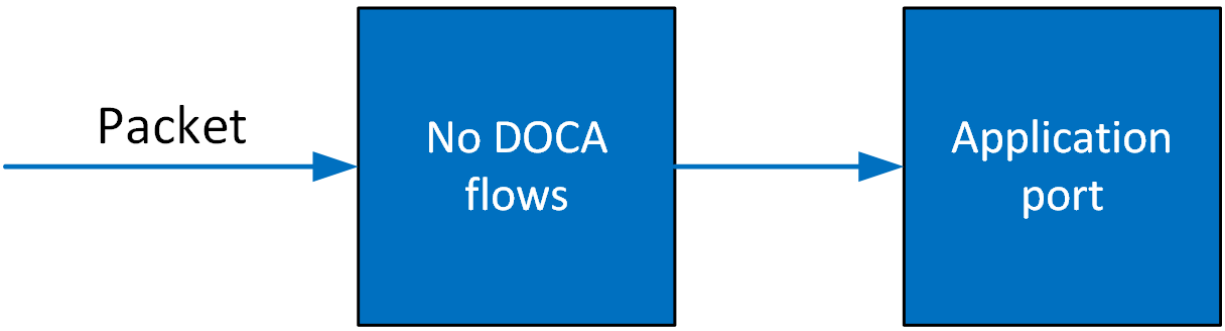
Note: Having a counter per entry affects performance and should be avoided if it is not required by the application.

When a counter is present, it is possible to query the flow and get the counter's data.

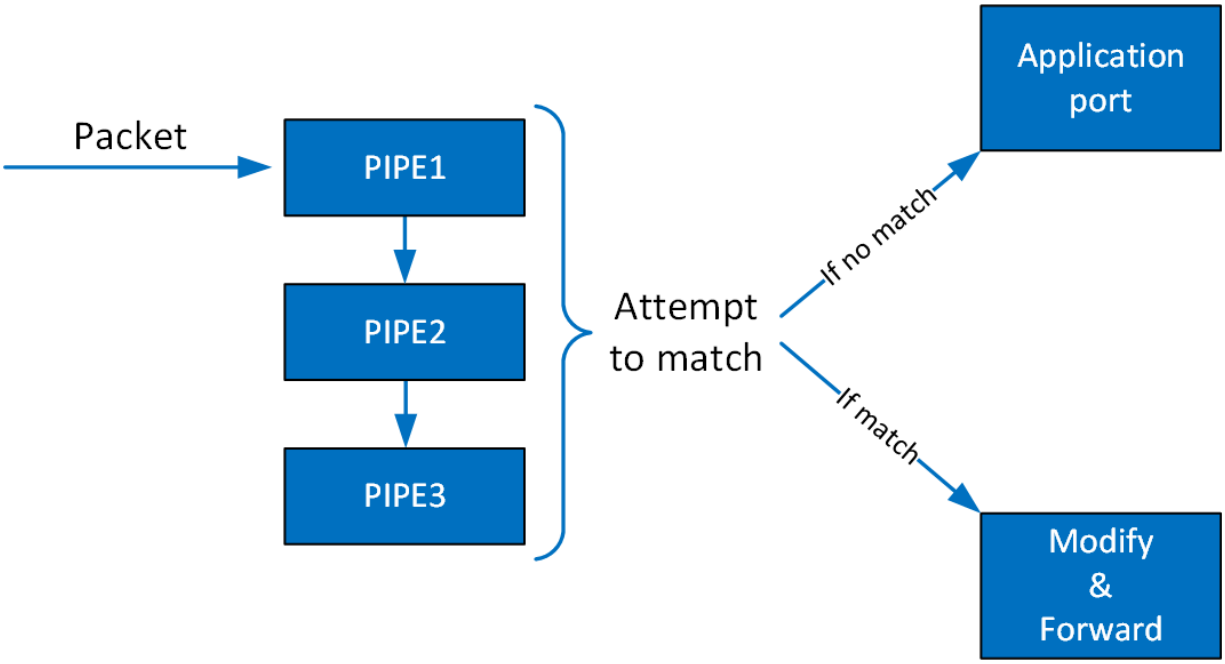
```
struct doca_flow_query {
 uint64_t total_bytes;
 /**< total bytes hit this flow*/
 uint64_t total_pkts;
 /**< total packets hit this flow*/
};

int doca_flow_query(struct doca_flow_pipe_entry *pe,
                    struct doca_flow_query *q);
```

Chapter 3. Packet Processing

In situations where there is a port without a PIPE defined, or with a PIPE defined but without any entry. The default behavior is that all packets arrive to port in software.



Once entries are added to the PIPE, if a packet has no match then it continues to the port in the software. If it does match, then the rules defined in the PIPE are executed.



If the packet is forwarded in RSS, the packet is forwarded to software according to the RSS definition. If the packet is forwarded to a port, packet is redirected back to the wire. If the packet is forwarded to the next PIPE, then the software attempts to match it with the next PIPE.

Note that the number of PIPEs impacts performance. The longer the number of matches and actions the packet goes through, the more time it takes HW to process it. When there is a very large number of entries, the HW needs to access the main memory to retrieve the entry context which increases latency.

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. 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