NVIDIA DOCA Flow

Programming Guide

MLNX-15-060488 _v1.5.1 | December 2022

Table of Contents

Chapter 1. Introduction

DOCA Flow is the most fundamental API for building generic packet processing pipes in hardware.

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 chained so that after a pipe-defined action is 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/ETHERTYPE
- IPv4/IPv6
- TCP/UDP/ICMP
- GRE/VXLAN/GTP-U
- Metadata

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
- Set metadata

The execution pipe may also have monitoring actions:

- Count
- Policers

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

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

- Drop packets

This document is intended for software developers writing network function applications that focus on packet processing (e.g., gateways). The document assumes familiarity with network stack and DPDK.

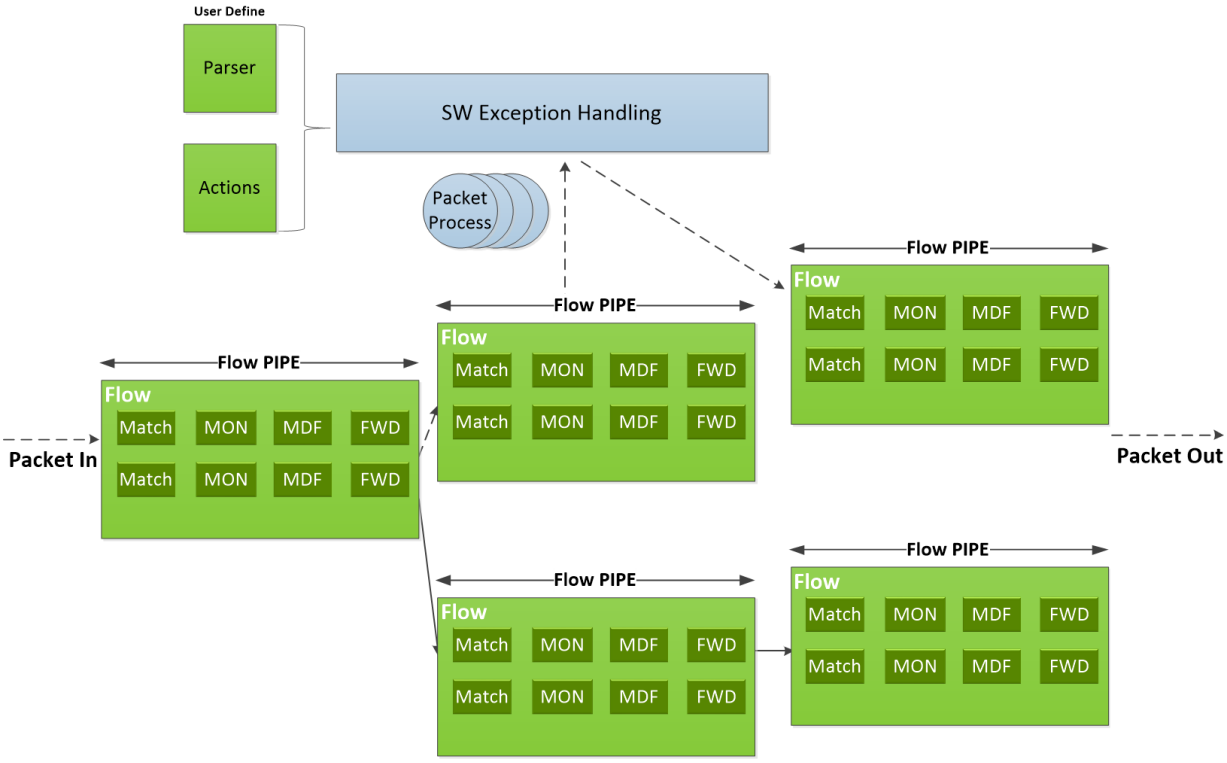
Chapter 2. Prerequisites

A DOCA Flow-based application can run either on the host machine or on the NVIDIA® BlueField® DPU target. Since it is based on DPDK, Flow-based programs require an allocation of huge pages:

```
sudo echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
sudo mkdir /mnt/huge
sudo mount -t hugetlbfs nodev /mnt/huge
```

Chapter 3. Architecture

The following diagram shows how the DOCA Flow library defines a pipe template, receives a packet for processing, creates the a pipe entry, and offloads the flow rule in NIC HW.



- MON: Monitor, can be count or meter
- MDF: Modify, can modify a field
- FWD: Forward to the next stage in packet processing
- User-defined set of matches parser and actions
- DOCA Flow pipes can be created or destroyed dynamically
- Packet processing is fully accelerated by hardware with a specific entry in a flow pipe
- Packets that do not match any of the pipe entries in hardware can be sent to Arm cores for exception handling and then reinjected back to hardware

Chapter 4. API

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

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

The following sections provide additional details about the library API.

4.1. doca_flow_cfg

This structure is required input for the DOCA Flow global initialization function, doca_flow_init.

```
struct doca_flow_cfg {
 uint16_t queues;
 struct doca_flow_resources resource;
 const char *mode_args;
 bool aging;
 uint32_t nr_shared_resources[DOCA_FLOW_SHARED_RESOURCE_MAX];
 unit32_t queue_depth;
 doca_flow_entry_process_cb cb;
};
```

queues
The number of hardware acceleration control queues. It is expected that the same core always uses the same `queue_id`. In cases where multiple cores access the API using the same `queue_id`, it is up to the application to use locks between different cores/threads.

resource
Resource quota. This field includes the flow resource quota defined in the following structs:

- `uint32_t nb_counters` – number of counters to configure
- `uint32_b nb_meters` – number of traffic meters to configure

mode_args
Mandatory, set the DOCA Flow architecture mode.

aging
Aging is handled by DOCA Flow while it is set to true. Default is false. See Setting Pipe Monitoring for information on the aging algorithm.

nr_shared_resources
Total shared resource per type. See section Shared Counter Resource for more information.

- Index `DOCA_FLOW_SHARED_RESOURCE_METER` - number of meters that can be shared among flows
- Index `DOCA_FLOW_SHARED_RESOURCE_COUNT` - number of counters that can be shared among flows
- Index `DOCA_FLOW_SHARED_RESOURCE_RSS` – number of RSS that can be shared among flows
- Index `DOCA_FLOW_SHARED_RESOURCE_CRYPTO` – number of crypto actions that can be shared among flows

queue_depth
Number of flow rule operations a queue can hold. This value is preconfigured at port start (`queue_size`). Default is 128. Configuring `0` sets default value.

cb
Callback function for entry create/destroy.

4.2. doca_flow_port_cfg

This struct is required input for the DOCA Flow port initialization function, doca_flow_port_start.

```
struct doca_flow_port_cfg {
 uint16_t port_id;
 enum doca_flow_port_type type;
 const char *devargs;
 uint16_t priv_data_size;
};
```

port_id
Port ID for the given type of port. For example, the following is a DPDK port ID for type `DOCA_FLOW_PORT_DPDK_BY_ID`.

type
Determined by the data plane in use.

- `DOCA_FLOW_PORT_DPDK_BY_ID` for DPDK dataplane.

devargs
String containing the exact configuration needed according to the type.

Note: For usage information of the type and devargs fields, refer to Start Port.

priv_data_size
Per port, if this field is not set to zero, it means users want to define private data where application-specific information can be stored. See doca_flow_port_priv_data for more information.

4.3. doca_flow_pipe_cfg

This is a pipe configuration that contains the user-defined template for the packet process.

```
struct doca_flow_pipe_attr {
    const char *name;
```

```
    enum doca_flow_pipe_type type;
    bool is_root;
    uint32_t nb_flows;
    uint8_t nb_actions;
    uint8_t nb_ordered_lists;
};
```

name
A string containing the name of the pipeline.

type
Type of pipe (`enum doca_flow_pipe_type`). This field includes the following pipe types:

- `DOCA_FLOW_PIPE_BASIC` – flow pipe
- `DOCA_FLOW_PIPE_CONTROL` – control pipe
- `DOCA_FLOW_PIPE_LPM` – LPM pipe
- `DOCA_FLOW_PIPE ORDERED_LIST` – ordered list pipe

is_root
Determines whether or not the pipeline is root. If true, then the pipe is a root pipe executed on packet arrival.

Note: Only one root pipe is allowed per port of any type.

nb_flows
Maximum number of flow rules. Default is 8k if not set.

nb_actions
Maximum number of DOCA Flow action array. Default is 1 if not set.

nb ordered_lists
Number of ordered lists in the array; default 0; mutually exclusive with `nb_actions`.

```
struct doca_flow_ordered_list {
        uint32_t idx;
        uint32_t size;
        const void **elements;
        enum doca_flow_ordered_list_element_type *types;
};
```

idx
List index among the lists of the pipe.

- At pipe creation, it must match the list position in the array of lists
- At entry insertion, it determines which list to use

size
Number of elements in the list.

elements
An array of DOCA flow structure pointers, depending on the `types`.

types
Types of DOCA Flow structures each of the elements is pointing to. This field includes the following ordered list element types:

- `DOCA_FLOW_ORDERED_LIST_ELEMENT_ACTIONS` – ordered list element is `struct doca_flow_actions`. The next element is `struct doca_flow_action_descs` which is associated with the current element.

- DOCA_FLOW_ORDERED_LIST_ELEMENT_ACTION_DESCS – ordered list element is struct doca_flow_action_descs. If the previous element type is ACTIONS, the current element is associated with it. Otherwise, the current element is ordered with regards to the previous one.
- DOCA_FLOW_ORDERED_LIST_ELEMENT_MONITOR – ordered list element is struct doca_flow_monitor.

```
struct doca_flow_pipe_cfg {
        struct doca_flow_pipe_attr attr;
        struct doca_flow_port *port;
        struct doca_flow_match *match;
        struct doca_flow_match *match_mask;
        struct doca_flow_actions **actions;
        struct doca_flow_action_descs **action_descs;
        struct doca_flow_monitor *monitor;
        struct doca_flow_ordered_list **ordered_lists;
};
```

attr
Attributes for the pipeline.

port
Port for the pipeline.

match
Matcher for the pipeline.

match_mask
Match mask for the pipeline and only for DOCA_FLOW_PIPE_BASIC, DOCA_FLOW_PIPE_CONTROL and DOCA_FLOW_PIPE_ORDERED_LIST.

actions
Actions array for the pipeline and only for DOCA_FLOW_PIPE_BASIC and DOCA_FLOW_PIPE_CONTROL.

action_descs
Action descriptions array and only for DOCA_FLOW_PIPE_BASIC and DOCA_FLOW_PIPE_CONTROL.

monitor
Monitor for the pipeline and only for DOCA_FLOW_PIPE_BASIC and DOCA_FLOW_PIPE_CONTROL.

ordered_lists
Array of ordered list types; only for DOCA_FLOW_PIPE_ORDERED_LIST.

4.4. doca_flow_meta

This is a maximum 20-byte scratch area which exists throughout the pipeline.

The user can set a value to metadata, copy from a packet field, then match in later pipes. Mask is supported in both match and modification actions.

The user can modify the metadata in different ways based on its description type:

AUTO
Set metadata value from an action of a specific entry. Pipe action is used as a mask.

CONSTANT
Set metadata value from a pipe action. Which is masked by the description mask.

SET
Set metadata value from an action of a specific entry which is masked by the description as a mask.

ADD
Set metadata scratch value from a pipe action or an action of a specific entry. Width is specified by the description.

Note: In a real application, it is encouraged to create a union of `doca_flow_meta` defining the application's scratch fields to use as metadata.

```
struct doca_flow_meta {
    union {
        uint32_t pkt_meta; /**< Shared with application via packet. */
        struct {
            uint32_t lag_port :2; /**< Bits of LAG member port. */
            uint32_t type :2; /**< 0: traffic 1: SYN 2: RST 3: FIN. */
            uint32_t zone :28; /**< Zone ID for CT processing. */
        } ct;
    };
    uint32_t u32[DOCA_FLOW_META_MAX / 4 - 1]; /**< Programmable user data. */
    uint32_t port_meta; /**< Programmable source vport. */
    uint32_t mark; /**< Mark id. */
    uint8_t nisp_syndrome; /**< NISP decrypt/authentication syndrome. */
    uint8_t ipsec_syndrome; /**< IPsec decrypt/authentication syndrome. */
    uint8_t align[2]; /**< Structure alignment. */
};
```

pkt_meta
Metadata can be received along with packet.

u32[]
Scratch area.

Note: If `encap` action is used, `pkt_meta` should not be defined by the user as it is defined internally in DOCA to reference the encapsulated tunnel ID.

4.5. doca_flow_match

This structure is a match configuration that contains the user-defined fields that should be matched on the pipe.

```
struct doca_flow_match {
 uint32_t flags;
 struct doca_flow_meta meta;
 uint8_t out_src_mac[DOCA_ETHER_ADDR_LEN];
 uint8_t out_dst_mac[DOCA_ETHER_ADDR_LEN];
 doca_be16_t out_eth_type;
 doca_be16_t out_vlan_tci;
 struct doca_flow_ip_addr out_src_ip;
 struct doca_flow_ip_addr out_dst_ip;
 uint8_t out_l4_type;
 uint8_t out_tcp_flags;
 doca_be16_t out_src_port;
 doca_be16_t out_dst_port;
 struct doca_flow_tun tun;
 uint8_t in_src_mac[DOCA_ETHER_ADDR_LEN];
 uint8_t in_dst_mac[DOCA_ETHER_ADDR_LEN];
 doca_be16_t in_eth_type;
 doca_be16_t in_vlan_tci;
```

```
 struct doca_flow_ip_addr in_src_ip;
 struct doca_flow_ip_addr in_dst_ip;
 uint8_t in_l4_type;
 uint8_t in_tcp_flags;
 doca_be16_t in_src_port;
 doca_be16_t in_dst_port;
};
```

flags
Match items which are no value needed.

meta
Programmable meta data.

out_src_mac
Outer source MAC address.

out_dst_mac
Outer destination MAC address.

out_eth_type
Outer Ethernet layer type.

out_vlan_tci
Outer VLAN TCI field.

out_src_ip
Outer source IP address.

out_dst_ip
Outer destination IP address.

out_l4_type
Outer layer 4 protocol type.

out_tcp_flags
Outer TCP flags.

out_src_port
Outer layer 4 source port.

out_dst_port
Outer layer 4 destination port.

tun
Tunnel info.

in_src_mac
Inner source MAC address if tunnel is used.

in_dst_mac
Inner destination MAC address if tunnel is used.

in_eth_type
Inner Ethernet layer type if tunnel is used.

in_vlan_tci
Inner VLAN TCI field if tunnel is used.

in_src_ip
Inner source IP address if tunnel is used.

in_dst_ip
Inner destination IP address if tunnel is used.

in_l4_type
Inner layer 4 protocol type if tunnel is used.

in_tcp_flags
Inner TCP flags if tunnel is used.

in_src_port
Inner layer 4 source port if tunnel is used.

in_dst_port
Inner layer 4 destination port if tunnel is used.

4.6. doca_flow_actions

This structure is a flow actions configuration.

```
struct doca_flow_actions {
    uint8_t action_idx;
    uint32_t flags;
    bool decap;
    struct doca_flow_meta meta;
    uint8_t mod_src_mac[DOCA_ETHER_ADDR_LEN];
    uint8_t mod_dst_mac[DOCA_ETHER_ADDR_LEN];
    doca_be16_t mod_vlan_id;
    struct doca_flow_ip_addr mod_src_ip;
    struct doca_flow_ip_addr mod_dst_ip;
    uint8_t ttl;
    doca_be16_t mod_src_port;
    doca_be16_t mod_dst_port;
    bool has_encap;
    struct doca_flow_encap_action encap;
    struct {
        enum doca_flow_crypto_protocol_type proto_type;
        uint32_t crypto_id;
    } security;
};
```

action_idx
Index according to place provided on creation.

flags
Action flags.

decap
Decap while it is set to true.

meta
Mask value if description type is AUTO, specific value if description type is CONSTANT.

mod_src_mac
Modify source MAC address.

mod_dst_mac
Modify destination MAC address.

mod_vlan_id
Modify VLAN ID.

mod_src_ip
Modify source IP address.

mod_dst_ip
Modify destination IP address.

ttl
TTL value to add if the field description type is ADD.

mod_src_port
Modify layer 4 source port.

mod_dst_port
Modify layer 4 destination port.

has_encap
Encap while it is set to true.

encap
Encap data information.

security
Contains crypto action type and ID.

4.7. doca_flow_action_desc

This structure is an action description.

```
struct doca_flow_action_desc {
    enum doca_flow_action_type type;
    union {
        union {
            uint32_t u32;
            uint64_t u64;
            uint8_t u8[16];
        } mask;
        struct {
            unit16_t doca_flow_action_field src;
            unit16_t doca_flow_action_field dst;
            unit16_t width;
        } copy;
        struct {
            struct doca_flow_action_field dst; /* destination info. */
            uint32_t width; /* Bit width to add */
        } add;
 };
};
```

type
Action type.

mask
Mask of modification action type `CONSTANT` and `SET`. Big-endian for network fields, host-endian for meta field.

copy
Field copy source and destination description.

add
Field add description. User can use the `dst` field to locate the destination field. Add always applies from field bit 0.

The `type` field includes the following forwarding modification types:

- `DOCA_FLOW_ACTION_AUTO` – modification type derived from pipe action
- `DOCA_FLOW_ACTION_CONSTANT` – modify action field with the constant value from pipe
- `DOCA_FLOW_ACTION_SET` – modify action field with the value of pipe entry
- `DOCA_FLOW_ACTION_ADD` – add field value. Supports meta scratch, ipv4_ttl, ipv6_hop, tcp_seq, and tcp_ack.
- `DOCA_FLOW_ACTION_COPY` – copy field

Refer to Setting Pipe Actions for more information.

4.8. doca_flow_monitor

This structure is a monitor configuration.

```
struct doca_flow_monitor {
        uint8_t flags;
        struct {
                uint64_t cir;
                uint64_t cbs;
        };
        uint32_t shared_meter_id;
        uint32_t shared_counter_id;
        uint32_t aging;
        void *user_data;
};
```

flags
Indicate actions to be included.

cir
Committed information rate in bytes per second. Defines maximum bandwidth.

cbs
Committed burst size in bytes. Defines maximum local burst size.

shared_meter_id
Meter ID that can be shared among multiple pipes.

shared_counter_id
Counter ID that can be shared among multiple pipes.

aging
Number of seconds from the last hit after which an entry is aged out.

user_data
Aging user data input.

The `flags` field includes the following monitor types:

- `DOCA_FLOW_ACTION_METER` – set monitor with meter action
- `DOCA_FLOW_ACTION_COUNT` – set monitor with counter action
- `DOCA_FLOW_ACTION_AGING` – set monitor with aging action

```
enum {
        DOCA_FLOW_MONITOR_NONE = 0,
        DOCA_FLOW_MONITOR_METER = (1 << 1),
        DOCA_FLOW_MONITOR_COUNT = (1 << 2),
        DOCA_FLOW_MONITOR_AGING = (1 << 3),
};
```

`T(c)` is the number of available tokens. For each packet where `b` equals the number of bytes, if `t(c)-b≥0` the packet can continue, and tokens are consumed so that `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 the last update, but it is limited by the CBS value.

CIR is the maximum bandwidth at which packets continue being confirmed. Packets surpassing this bandwidth are dropped. CBS is the maximum bytes allowed to exceed the CIR to be still CIR confirmed. Confirmed packets are handled based on the `fwd` parameter.

The number of `<cir,cbs>` pair different combinations is limited to 128.

Metering packets can be individual (i.e., per entry) or shared among multiple entries:

- For the individual case, set bit `DOCA_FLOW_MONITOR_METER` in flags
- For the shared case, use a non-zero `shared_meter_id`

Counting packets can be individual (i.e., per entry) or shared among multiple entries:

- For the individual case, set bit `DOCA_FLOW_MONITOR_COUNT` in flags
- For the shared case, use a non-zero `shared_counter_id`

4.9. doca_flow_fwd

This structure is a forward configuration which directs where the packet goes next.

```
struct doca_flow_fwd {
 enum doca_flow_fwd_type type;
 union {
  struct {
   unit32_t rss_flags;
   unit32_t *rss_queues;
   int num_of_queues;
  };
  struct {
   unit16_t port_id;
  };
  struct {
   struct doca_flow_pipe *next_pipe;
               };
               struct {
         struct doca_flow_pipe *pipe;
         uint32_t idx;
        } ordered_list_pipe;
 };
};
```

type
Indicates the forwarding type.

rss_flags
RSS offload types.

rss_queues
RSS queues array.

num_of_queues
Number of queues.

port_id
Destination port ID.

next_pipe
Next pipe pointer.

ordered_list_pipe.pipe
Ordered list pipe to select an entry from.

ordered_list_pipe.idx
Index of the ordered list pipe entry.

The `type` field includes the forwarding action types defined in the following enum:

- `DOCA_FLOW_FWD_RSS` – forwards packets to RSS
- `DOCA_FLOW_FWD_PORT` – forwards packets to port
- `DOCA_FLOW_FWD_PIPE` – forwards packets to another pipe
- `DOCA_FLOW_FWD_DROP` – drops packets
- `DOCA_FLOW_FWD_ORDERED_LIST_PIPE` – forwards packet to a specific entry in an ordered list pipe

The `rss_flags` field is a bitwise OR of the RSS fields defined in the following enum:

- `DOCA_FLOW_RSS_IP` – RSS by IP header
- `DOCA_FLOW_RSS_UDP` – RSS by UDP header
- `DOCA_FLOW_RSS_TCP` – RSS by TCP header

4.10. doca_flow_query

This struct is a flow query result.

```
struct doca_flow_query {
 uint64_t total_bytes;
 uint64_t total_pkts;
};
```

total_bytes
Total bytes hit this flow.

total_pkts
Total packets hit this flow.

4.11. doca_flow_aged_query

This structure is an aged flow callback context.

```
struct doca_flow_aged_query {
 uint64_t user_data;
};
```

user_data
The user input context. Otherwise, the `doca_flow_pipe_entry` pointer be returned.

4.12. doca_flow_init

This function is the global initialization function for DOCA Flow.

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

cfg [in]
A pointer to flow config structure.

error [out]
A pointer to flow error output.

Returns
0 on success, a negative `errno` value otherwise and error is set.

Note: Must be invoked first before any other function in this API. This is a one-time call used for DOCA Flow initialization and global configurations.

4.13. doca_flow_port_start

This function starts a port with its given configuration. It creates one port in the DOCA Flow layer, allocates all resources used by this port, and creates the default offload flow rules to redirect packets into software queues.

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

cfg [in]
A pointer to flow port config structure.

error [out]
A pointer to flow error output.

Returns
Port handler on success, NULL otherwise an error is set.

4.14. doca_flow_port_priv_data

This function get the pointer of user private data. User can manage the specific data in DOCA port, the size of the private data is given on port configuration.

```
uint8_t *doca_flow_port_priv_data(struct doca_flow_port *port);
```

port [in]
A pointer to the DOCA Flow port structure.

Returns
Private data head pointer.

4.15. doca_flow_port_pair

This function pairs two DOCA ports. If two ports are not representor ports, after performing a physical hairpin bind, this API notifies DOCA that these two ports are hairpin peers. If FWD to the hairpin port, DOCA builds a hairpin queue action. If one of the two ports is a representor, DOCA creates a miss flow with a port action to redirect the traffic from one port to the other. Those two paired ports have no order, and a port cannot be paired with itself.

```
int *doca_flow_port_pair(struct doca_flow_port *port,
                         struct doca_flow_port *pair_port);
```

port [in]
A pointer to DOCA Flow port structure.

pair_port [in]
A pointer to another DOCA Flow port structure.

Returns
0 on success, negative value on failure.

4.16. doca_flow_pipe_create

This function creates a new pipeline to match and offload specific packets. The pipeline configuration is defined in the `doca_flow_pipe_cfg`. The API creates a new pipe but does not start the hardware offload.

When `cfg` type is `DOCA_FLOW_PIPE_CONTROL`, the function creates a special type of pipe that can have dynamic matches and forwards with priority.

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

cfg [in]
A pointer to flow pipe config structure.

fwd [in]
A pointer to flow forward config structure.

fwd_miss [in]
A pointer to flow forward miss config structure. NULL for no `fwd_miss`. When creating a pipe, if there is a miss and `fwd_miss` is configured, then packet steering should jump to it.

error [out]
A pointer to flow error output.

Returns
Pipe handler on success, NULL otherwise and error is set.

4.17. doca_flow_pipe_add_entry

This function add a new entry to a pipe. When a packet matches a single pipe, it starts hardware offload. The pipe defines which fields to match. This API does the actual hardware offload, with the information from the fields of the input packets.

```
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 *monitor,
                         const struct doca_flow_fwd *fwd,
      unit32_t flags,
      void *usr_ctx,
                         struct doca_flow_error *error);
```

pipe_queue [in]
Queue identifier.

pipe [in]
A pointer to flow pipe.

match [in]
A pointer to flow match. Indicates specific packet match information.

actions [in]
A pointer to modify actions. Indicates specific modify information.

monitor [in]
A pointer to monitor profiling or aging.

fwd [in]
A pointer to flow forward actions.

flags [in]
Can be set as DOCA_FLOW_WAIT_FOR_BATCH or DOCA_FLOW_NO_WAIT.
DOCA_FLOW_WAIT_FOR_BATCH means that this entry waits to be pushed to hardware.
DOCA_FLOW_NO_WAIT means that this entry is pushed to hardware immediately.

usr_ctx [in]
A pointer to user context.

error [out]
A pointer to flow error output.

Returns
Pipe entry handler on success, NULL otherwise and error is set.

4.18. doca_flow_pipe_control_add_entry

This function adds a new entry to a control pipe. When a packet matches a single pipe, it starts hardware offload. The pipe defines which fields to match. This API does the actual hardware offload with the information from the fields of the input packets.

```
struct doca_flow_pipe_entry *
doca_flow_pipe_control_add_entry(uint16_t pipe_queue,
                         struct doca_flow_pipe *pipe,
                         const struct doca_flow_match *match,
                         const struct doca_flow_match *match_mask,
                         const struct doca_flow_actions *actions,
                         const struct doca_flow_action_descs *action_descs,
                         const struct doca_flow_monitor *monitor,
                         const struct doca_flow_fwd *fwd,
                         struct doca_flow_error *error);
```

pipe_queue [in]
Queue identifier.

priority [in]
Priority value.

pipe [in]
A pointer to flow pipe.

match [in]
A pointer to flow match. Indicates specific packet match information.

match_mask [in]
A pointer to flow match mask information.

actions [in]
A pointer to modify actions. Indicates specific modify information.

action_descs
A pointer to action descriptions.

monitor [in]
A pointer to monitor actions.

fwd [in]
A pointer to flow FWD actions.

error [out]
A pointer to flow error output.

Returns
Pipe entry handler on success, NULL otherwise and error is set.

4.19. doca_flow_pipe_lpm_add_entry

This function adds a new entry to an LPM pipe. This API does the actual hardware offload all entries when flags is set to DOCA_FLOW_NO_WAIT.

```
struct doca_flow_pipe_entry *
doca_flow_pipe_lpm_add_entry(uint16_t pipe_queue,
                             uint8_t priority,
                             struct doca_flow_pipe *pipe,
                             const struct doca_flow_match *match,
                             const struct doca_flow_match *match_mask,
                             const struct doca_flow_fwd *fwd,
                             unit32_t flags,
                             void *usr_ctx,
                             struct doca_flow_error *error);
```

pipe_queue [in]
Queue identifier.

priority [in]
Priority value.

pipe [in]
A pointer to flow pipe.

match [in]
A pointer to flow match. Indicates specific packet match information.

match_mask [in]
A pointer to flow match mask information.

fwd [in]
A pointer to flow FWD actions.

flags [in]
Can be set as DOCA_FLOW_WAIT_FOR_BATCH or DOCA_FLOW_NO_WAIT.

- DOCA_FLOW_WAIT_FOR_BATCH – LPM collects this flow entry
- DOCA_FLOW_NO_WAIT – LPM adds this entry, builds the LPM software tree, and pushes all entries to hardware immediately

usr_ctx [in]
A pointer to user context.

error [out]
A pointer to flow error output.

Returns
Pipe entry handler on success, NULL otherwise and error is set.

4.20. doca_flow_pipe_ordered_list_add_entry

This function adds a new entry to an order list pipe. When a packet matches a single pipe, it starts hardware offload. The pipe defines which fields to match. This API does the actual hardware offload, with the information from the fields of the input packets.

```
struct doca_flow_pipe_entry *
doca_flow_pipe_ordered_list_add_entry(uint16_t pipe_queue,
  struct doca_flow_pipe *pipe,
  uint32_t idx,
  const struct doca_flow_ordered_list *ordered_list,
  const struct doca_flow_fwd *fwd,
  enum doca_flow_flags_type flags,
  void *user_ctx,
  struct doca_flow_error *error);
```

pipe_queue [in]
Queue identifier.

pipe [in]
A pointer to flow pipe.

idx [in]
A unique entry index. It is the user's responsibility to ensure uniqueness.

ordered_list [in]
A pointer to an ordered list structure with pointers to `struct doca_flow_actions` and `struct doca_flow_monitor` at the same indices as they were at the pipe creation time. If the configuration contained an element of `struct doca_flow_action_descs`, the corresponding array element is ignored and can be NULL.

fwd [in]
A pointer to flow FWD actions.

flags [in]
Can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT`.

- `DOCA_FLOW_WAIT_FOR_BATCH` – this entry waits to be pushed to hardware
- `DOCA_FLOW_NO_WAIT` – this entry is pushed to hardware immediately

usr_ctx [in]
A pointer to user context.

error [out]
A pointer to flow error output.

Returns
Pipe entry handler on success, NULL otherwise and error is set.

4.21. doca_flow_entries_process

This function processes entries in the queue. The application must invoke this function to complete flow rule offloading and to receive the flow rule's operation status.

```
int
doca_flow_entries_process(struct doca_flow_port *port,
                          uint16_t pipe_queue,
                          uint64_t timeout,
```

```
                                uint32_t max_processed_entries);
```

port [in]
A pointer to the flow port structure.

pipe_queue [in]
Queue identifier.

timeout [in]
Timeout value.

max_processed_entries [in]
A pointer to the flow pipe.

Returns
>0 – the number of entries processed
0 – no entries are processed
<0 – failure

4.22. doca_flow_entries_process

This function get the status of pipe entry.

```
enum doca_flow_entry_status
doca_flow_entry_get_status(struct doca_flow_entry *entry);
```

entry [in]
A pointer to the flow pipe entry to query.

Returns
Entry's status, defined in the following enum:

- `DOCA_FLOW_ENTRY_STATUS_IN_PROCESS` – the operation is in progress
- `DOCA_FLOW_ENTRY_STATUS_SUCCESS` – the operation completed successfully
- `DOCA_FLOW_ENTRY_STATUS_ERROR` – the operation failed

4.23. doca_flow_entry_query

This function queries packet statistics about a specific pipe entry.

Note: The pipe must have been created with the `DOCA_FLOW_MONITOR_COUNT` flag or the query will return an error.

```
int doca_flow_entry_query(struct doca_flow_pipe_entry *entry, struct doca_flow_query
 *query_stats);
```

entry [in]
A pointer to the flow pipe entry to query.

query_stats [out]
A pointer to the data retrieved by the query.

Returns
0 on success. Otherwise, a negative errno value is returned and error is set.

4.24. doca_flow_query_pipe_miss

This function queries packet statistics about a specific pipe miss flow.

Note: The pipe must have been created with the `DOCA_FLOW_MONITOR_COUNT` flag or the query will return an error.

```
int doca_flow_query_pipe_miss(struct doca_flow_pipe *pipe, struct doca_flow_query
 *query_stats);
```

pipe [in]
A pointer to the flow pipe to query.

query_stats [out]
A pointer to the data retrieved by the query.

Returns
0 on success. Otherwise, a negative errno value is returned and error is set.

4.25. doca_flow_aging_handle

This function handles the aging of all the pipes of a given port. It goes over all flows and releases aged flows from being tracked. The entries array is filled with aged flows. Since the number of flows can be very large, it can take a significant amount of time to go over all flows, so this function is limited by a time quota. This means it might return without handling all flows which requires the user to call it again.

Note: The pipe must have been created with the `DOCA_FLOW_MONITOR_COUNT` flag or the query will return an error.

```
int doca_flow_aging_handle(struct doca_flow_port *port,
                           uint16_t queue,
                           uint64_t quota,
                           struct doca_flow_aged_query *entries,
                           int len);
```

queue [in]
Queue identifier.

quota [in]
Max time quota in microseconds for this function to handle aging.

entries [in]
User input entry array for the aged flows.

len [in]
User input length of entries array.

Returns
>0 – the number of aged flows filled in entries array.
0 – no aged entries in current call.
-1 – full cycle is done.

Chapter 5. Shared Counter Resource

A shared counter can be used in multiple pipe entries. The following are the steps involved in configuring and using shared counters.

5.1. On doca_flow_init()

Specify the total number of shared counters to be used, `nb_shared_counters`.

This call implicitly defines the shared counters IDs in the range of 0-`nb_shared_counters-1`.

```
.nr_shared_resources = {
    [DOCA_FLOW_SHARED_RESOURCE_COUNT] = nb_shared_counters
},
```

5.2. On doca_flow_shared_resource_cfg()

This call can be skipped for shared counters.

5.3. On doca_flow_shared_resource_bind()

This call binds a bulk of shared counters IDs to a specific pipe or port.

```
int
doca_flow_shared_resources_bind(enum doca_flow_shared_resource_type type, uint32_t
 *res_array,
                                uint32_t res_array_len, void *bindable_obj,
                                struct doca_flow_error *error);
```

res_array [in]
Array of shared counters IDs to be bound.

res_array_len [in]
Array length.

bindable_obj
Pointer to either a pipe or port.

This call allocates the counter's objects. A counter ID specified in this array can only be used later by the corresponding bindable object (pipe or port).

The following example binds counter IDs 2, 4, and 7 to a pipe. The counters' IDs must be within the range 0-nb_shared_coutners-1.

```
uint32_t shared_counters_ids = {2, 4, 7};
struct doca_flow_pipe *pipe = ...

doca_flow_shared_resources_bind(
        DOCA_FLOW_SHARED_RESOURCE_COUNT,
        shared_counters_ids, 3, pipe, &error);
```

5.4. On doca_flow_pipe_add_entry() or Pipe Configuration (struct doca_flow_pipe_cfg)

The shared counter ID is included in the monitor parameter. It must be bound to the pipe object in advance.

```
struct doca_flow_monitor {
        ...
        uint32_t shared_counter_id;
        /**< shared counter id */
        ...
}
```

Packets matching the pipe entry are counted on the `shared_counter_id`. In pipe configuration, the `shared_counter_id` can be changeable (all FFs) and then the pipe entry holds the specific shared counter ID.

5.5. Querying Bulk of Shared Counter IDs

Use this API:

```
int
      doca_flow_shared_resources_query(enum doca_flow_shared_resource_type type,

      uint32_t *res_array,
      struct doca_flow_shared_resource_result *query_results_array,
      uint32_t array_len,
      struct doca_flow_error *error);
```

res_array [in]
Array of shared counters IDs to be queried.

res_array_len [in]
Array length.

query_results_array [out]
Query results array. Must be allocated prior to calling this API.

The `type` parameter is `DOCA_FLOW_SHARED_RESOURCE_COUNT`.

5.6. On doca_flow_pipe_destroy() or doca_flow_port_destroy()

All bound resource IDs of this pipe or port are destroyed.

Chapter 6. Shared Meter Resource

A shared meter can be used in multiple pipe entries (hardware steering mode support only). The following are the steps involved in configuring and using shared meters.

6.1. On doca_flow_init()

Specify the total number of shared meters to be used, `nb_shared_meters`.

The following call is an example how to initialize both shared counters and meter ranges. This call implicitly defines the shared counter IDs in the range of 0-`nb_shared_counters-1` and the shared meter IDs in the range of 0-`nb_shared_meters-1`.

```
struct doca_flow_cfg cfg = {
                  .queues = queues,
                  ...
                  .nr_shared_resources = {nb_shared_meters, nb_shared_counters},
}
doca_flow_init(&cfg, &error);
```

6.2. On doca_flow_shared_resource_cfg()

This call binds a specific meter ID with its committed information rate (CIR) and committed burst size (CBS):

```
struct doca_flow_resource_meter_cfg {
        uint64_t cir;
        /**< Committed Information Rate (bytes/second). */
        uint64_t cbs;
        /**< Committed Burst Size (bytes). */
};

struct doca_flow_shared_resource_cfg {
        union {
                struct doca_flow_resource_meter_cfg meter_cfg;
       ...
        };
};

int
doca_flow_shared_resource_cfg(enum doca_flow_shared_resource_type type, uint32_t id,
                              struct doca_flow_shared_resource_cfg *cfg,
                              struct doca_flow_error *error);
```

The following example configures the shared meter ID 5 with a CIR of 0x1000 bytes per second and a CBS of 0x600 bytes:

```
struct doca_flow_shared_resource_cfg shared_cfg = { 0 };
shared_cfg.meter_cfg.cir = 0x1000;
shared_cfg.meter_cfg.cbs = 0x600;
doca_flow_shared_resource_cfg(DOCA_FLOW_SHARED_RESOURCE_METER, 0x5, &shared_cfg,
 &error);
```

6.3. On doca_flow_shared_resource_bind()

This call binds a bulk of shared meter IDs to a specific pipe or port.

```
int
doca_flow_shared_resources_bind(enum doca_flow_shared_resource_type type, uint32_t
 *res_array,
                                uint32_t res_array_len, void *bindable_obj,
                                struct doca_flow_error *error);
```

res_array [in]
Array of shared meter IDs to be bound.

res_array_len [in]
Array length.

bindable_obj
Pointer to either a pipe or port.

This call allocates the meter's objects. A meter ID specified in this array can only be used later by the corresponding bindable object (pipe or port).

The following example binds meter IDs 5 and 14 to a pipe. The meter IDs must be within the range 0-`nb_shared_meters-1`.

```
uint32_t shared_meters_ids = {5, 14};
struct doca_flow_pipe *pipe = ...

doca_flow_shared_resources_bind(
        DOCA_FLOW_SHARED_RESOURCE_METER,
        shared_meters_ids, 2, pipe, &error);
```

6.4. On doca_flow_pipe_add_entry() or Pipe Configuration (struct doca_flow_pipe_cfg)

The shared meter ID is included in the monitor parameter. It must be bound in advance to the pipe object.

```
struct doca_flow_monitor {
        ...
        uint32_t shared_meter_id;
        /**< shared meter id */
        ...
}
```

Packets matching the pipe entry are metered based on the `cir` and the `cbs` parameters related to the `shared_meter_id`. In the pipe configuration, the `shared_meter_id` can be changeable (all FFs) and then the pipe entry must hold the specific shared meter ID for that entry.

6.5. Querying Bulk of Shared Meter IDs

There is no direct API to query a shared meter ID. To count the number of packets before a meter object, add a counter (shared or single) and use an API to query it. For an example, see section Querying Bulk of Shared Meter IDs.

6.6. On doca_flow_pipe_destroy() or doca_flow_port_destroy()

All bound resource IDs of this pipe or port are destroyed.

Chapter 7. Shared RSS Resource

A shared RSS can be used in multiple pipe entries.

7.1. On doca_flow_init()

Specify the total number of shared RSS to be used, `nb_shared_rss`.

This call implicitly defines the shared RSS IDs in the range of 0 to `nb_shared_rss`-1.

```
struct doca_flow_cfg cfg;

      cfg.nr_shared_resources[DOCA_FLOW_SHARED_RESOURCE_RSS] = nb_shared_rss;
      doca_flow_init(&cfg, &error);
```

7.2. On doca_flow_shared_resource_cfg()

This call configures shared RSS resource.

```
struct doca_flow_shared_resource_cfg res_cfg;

      for (uint8_t i = 0; i < nb_shared_rss; i++) {
      res_cfg.rss_cfg.nr_queues = nr_queues;
      res_cfg.rss_cfg.flags = flags;
      res_cfg.rss_cfg.queues_array = queues_array;
      doca_flow_shared_resource_cfg(DOCA_FLOW_SHARED_RESOURCE_RSS, i, &rss_cfg,
 &error);
      }
```

7.3. On doca_flow_shared_resource_bind()

This call binds a bulk of shared RSS to a specific port.

```
uint32_t shared_rss_ids = {2, 4, 7};
struct doca_flow_port *port;

doca_flow_shared_resources_bind(
        DOCA_FLOW_SHARED_RESOURCE_RSS,
        shared_rss_ids, 3, port, &error);
```

7.4. On doca_flow_pipe_add_entry()

On `doca_flow_pipe_create`, the user can input NULL as fwd. On `doca_flow_pipe_add_entry`, the user can input preconfigured shared RSS as `fwd` by specifying the `shared_rss_id`.

```
struct doca_flow_fwd;

fwd.shared_rss_id = 2;
fwd.type = DOCA_FLOW_FWD_RSS;
doca_flow_pipe_add_entry(queue, pipe, match, action, mon, &fwd, flag, usr_ctx,
 &error);
```

7.5. On doca_flow_port_destroy()

All bound `shared_rss` resource IDs of this port are destroyed.

Chapter 8. Shared Crypto Resource

A shared crypto resource can be used in multiple pipe entries and is intended to perform combinations of crypto offloads and crypto protocol header operations.

The following subsections expand on the steps involved in configuring and using shared crypto actions.

8.1. On doca_flow_init()

Specify the total number of shared crypto operations to be used, `nb_shared_crypto`.

This call implicitly defines the shared counter IDs in the range of 0-`nb_shared_crypto`-1.

```
struct doca_flow_cfg cfg = {
                  .queues = queues,
                  ...
                  .nr_shared_resources = {0, 0, 0, nb_shared_crypto},
}
doca_flow_init(&cfg, &error);
```

8.2. On doca_flow_shared_resource_cfg()

This call binds a specific crypto ID with its committed information rate (CIR) and committed burst size (CBS).

```
struct doca_flow_resource_crypto_cfg {
 enum doca_flow_crypto_protocol_type proto_type;
 /**< packet reformat action */
 enum doca_flow_crypto_action_type action_type;
 /**< crypto action */
 enum doca_flow_crypto_reformat_type reformat_type;
 /**< packet reformat action */
 enum doca_flow_crypto_net_type net_type;
 /**< packet network mode type */
 enum doca_flow_crypto_header_type header_type;
 /**< packet header type */
 uint16_t reformat_data_sz;
 /**< reformat header length in bytes */
 uint8_t reformat_data[DOCA_FLOW_CRYPTO_REFORMAT_LEN_MAX];
 /**< reformat header buffer */
 union {
  struct {
   uint16_t key_sz;
   /**< key size in bytes */
   uint8_t key[DOCA_FLOW_CRYPTO_KEY_LEN_MAX];
   /**< Crypto key buffer */
```

```
  };
  void* security_ctx;
  /**< crypto object handle */
 };
 struct doca_flow_fwd fwd;
 /**< Crypto action continuation */
};

struct doca_flow_shared_resource_cfg {
        union {
                struct doca_flow_crypto_cfg crypto_cfg;
       ...
        };
};

int
doca_flow_shared_resource_cfg(enum doca_flow_shared_resource_type type, uint32_t id,
                              struct doca_flow_shared_resource_cfg *cfg,
                              struct doca_flow_error *error);
```

8.3. On doca_flow_shared_resource_bind()

This call binds a bulk of shared crypto IDs to a specific pipe or port.

```
int
doca_flow_shared_resources_bind(enum doca_flow_shared_resource_type type, uint32_t
 *res_array,
                                uint32_t res_array_len, void *bindable_obj,
                                struct doca_flow_error *error);
```

res_array [in]
Array of shared crypto IDs to be bound.

res_array_len [in]
Array length.

bindable_obj
Pointer to either a pipe or port.

This call allocates the crypto's objects. A crypto ID specified in this array can only be used later by the corresponding bindable object (pipe or port).

The following example binds crypto IDs 2, 5, and 7 to a pipe. The cryptos' IDs must be within the range 0-nb_shared_crypto-1.

```
uint32_t shared_crypto_ids = {2, 5, 7};
struct doca_flow_pipe *pipe = ...

doca_flow_shared_resources_bind(
        DOCA_FLOW_SHARED_RESOURCE_CRYPTO,
        shared_crypto_ids, 3, pipe, &error);
```

8.4. On doca_flow_pipe_add_entry() or Pipe Configuration (struct doca_flow_pipe_cfg)

The shared crypto ID is included in the action parameter. It must be bound in advance to the pipe object.

```
struct doca_flow_actions {
        ...
        struct {
            enum doca_flow_crypto_protocol_type proto_type;
            /**< Crypto shared action type */
            uint32_t crypto_id;
            /**< Crypto shared action id */
        } security;
}
```

Crypto and header reformat operations are performed over the packets matching the pipe entry according on the `crypto_id` configuration. Afterwards, the flow continues from the point specified in the forward part of the crypto action configuration. In pipe configuration, the `crypto_id` can be changeable (all FFs) and then the pipe entry holds the specific shared crypto ID.

8.5. On doca_flow_pipe_destroy() or doca_flow_port_destroy()

All bound crypto resource IDs of this pipe or port are destroyed.

Chapter 9. Flow Life Cycle

9.1. Initialization Flow

Before using any DOCA Flow function, it is mandatory to call DOCA Flow initialization, `doca_flow_init()`, which initializes all resources used by DOCA Flow.

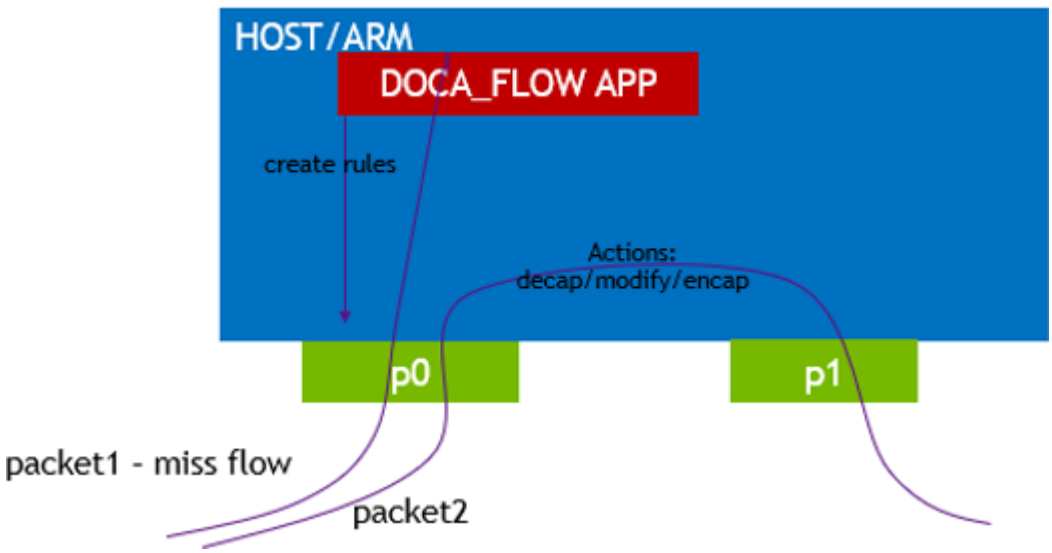
9.1.1. Pipe Mode

This mode (`mode_args`) defines the basic traffic in DOCA. It creates some miss rules when the DOCA port initialized. Currently, DOCA supports 3 types:

- `vnf`

  The packet arrives from one side of the application, is processed, and sent from the other side. The miss packet by default goes to the RSS of all queues.

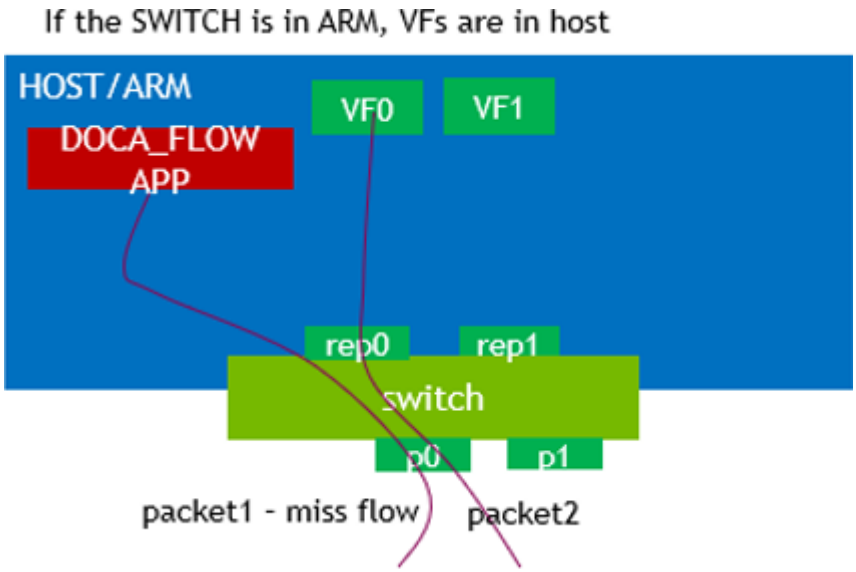
  The following diagram shows the basic traffic flow in `vnf` mode. Packet1 firstly misses to host RSS queues. The app captures this packet and decides how to process it and then creates a pipe entry. Packet2 will hit this pipe entry and do the action, for example, for VXLAN, will do decap, modify, and encap, then is sent out from P1.



- `switch`

  Used for internal switching, only representor ports are allowed, for example, uplink representors and SF/VF representors. Packet is forwarded from one port to another. If a packet arrives from an uplink and does not hit the rules defined by the user's pipe. Then the packet is received on all RSS queues of the representor of the uplink.

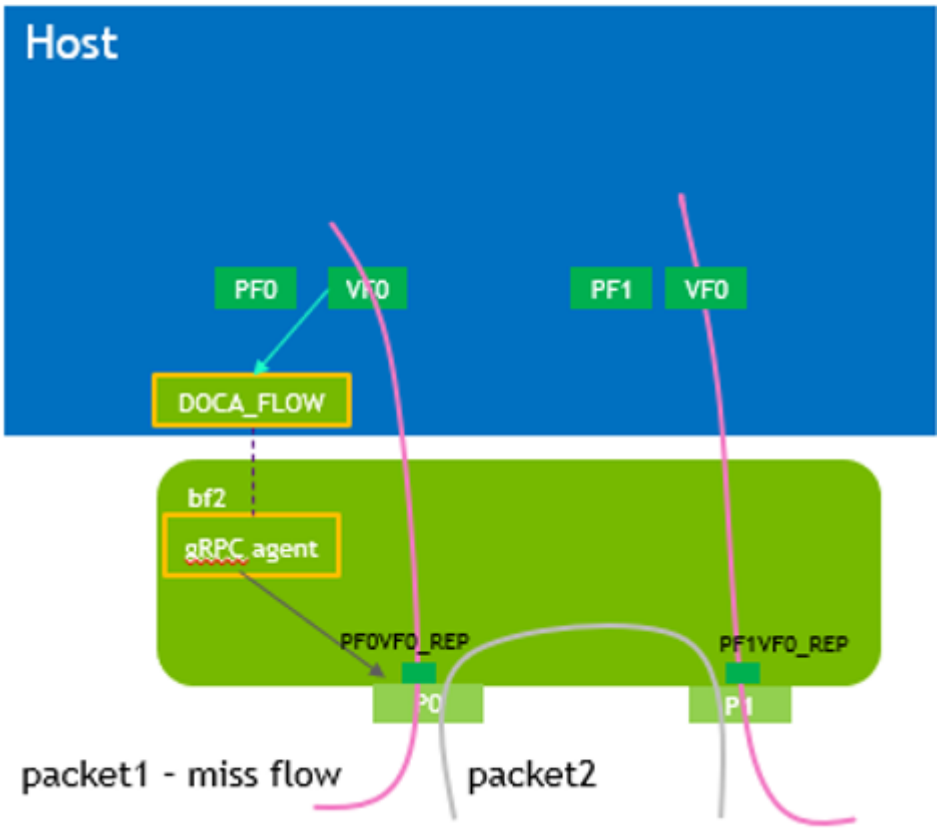
  The following diagram shows the basic flow of traffic in `switch` mode. Packet1 firstly misses to host RSS queues. The app captures this packet and decides which representor goes, and then sets the rule. Packets hit this rule and go to representor0.



- `remote-vnf`

  Remote mode is a BlueField mode only, with two physical ports (uplinks). Users must use `doca_flow_port_pair` to pair one physical port and one of its representors. A packet from this uplink, if it does not hit any rules from the users, is firstly received on this representor. Users must also use `doca_flow_port_pair` to pair two physical uplinks. If a packet is received from one uplink and hits the rule whose FWD action is to another uplink, then the packets are sent out from it.

  The following diagram shows the basic traffic flow in remote-vnf mode. Packet1, from BlueField uplink P0, firstly misses to host VF0. The app captures this packet and decides whether to drop it or forward it to another uplink (P1). Then, using gRPC to set rules on P0, packet2 hits the rule, then is either dropped or is sent out from P1.



9.2. Start Point

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 ingress port, it starts the hardware execution as defined by the DOCA API.

`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 by calling `doca_flow_port_start()`. The purpose of this step is to attach user application ports to the DOCA Flow ports.

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";
```

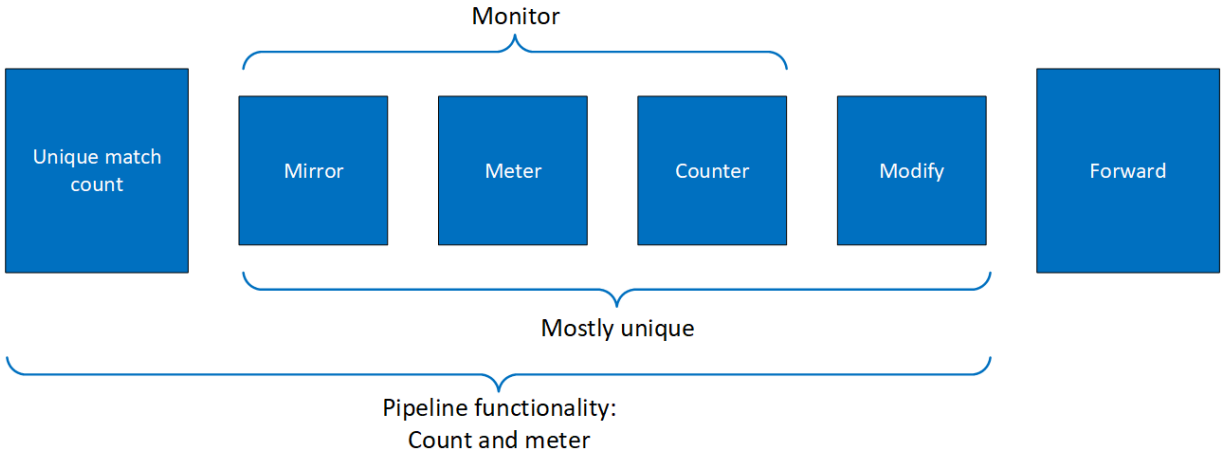
The `devargs` parameter points to a string that has the numeric value of the DPDK `port_id` in decimal format. The port must be configured and started before calling this API. Mapping the DPDK port to the DOCA port is required to synchronize application ports with hardware ports.

9.3. Create Pipe and Pipe Entry

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, just using the counter, etc).

9.3.1. 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.

For each `doca_flow_match` field, users choose whether the field is:

- Ignored (wild card) – the 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 using the `doca_flow_pipe_cfg.match_mask` pointer. `match_mask==NULL` is implicit. Otherwise, it is explicit.

9.3.1.1. Implicit Match

Match Type	Pipe Value	Pipe Mask, match_mask	Entry Value
Wildcard (match any)	0	Null pointer	N/A
Constant	Pipe value	Null pointer	N/A
Variable (per entry)	Full mask (0xff...)	Null pointer	Per-entry value

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

- Ignored fields:
  - Field is zeroed
  - Pipeline has no comparison on the field
- 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.
- 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.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.
- Example

  The following is an example of a match on the VXLAN tunnel, where 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)
  {
      //outer
      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_dst_ip.type = DOCA_FLOW_IP4_ADDR;
      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;
  }
  ```

9.3.1.2. Explicit Match

Match Type	Pipe Value	Pipe Mask, match_mask	Entry Value
Wildcard (match any)	0	0	N/A
Constant	Pipe value	Full mask (0xff...)	N/A
Variable (per entry)	0	Mask	Per-entry value

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

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

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

- 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 so as to avoid ambiguity.

- 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.

9.3.2. Setting Pipe Actions

9.3.2.1. Auto-modification

Similarly to setting pipe match, actions also have a template definition.

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 of `doca_flow_match` to one of the following:

- Ignored field – field is zeroed, modify is not used
- Constant fields – when a field must 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 are set by the user per entry

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

  Metadata is considered as per-packet changeable fields, pipe action is used as a mask.
- 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;
}
```

9.3.2.2. Explicit Modification Type

It is possible to force constant modification or per-entry modification with action description type (`CONSTANT` or `SET`) and mask. For example:

```
static void
create_constant_modify_actions(struct doca_flow_actions *actions#
                               struct doca_flow_action_descs *descs)
{
        actions->mod_src_port = 0x1234;
        descs->src_port.type = DOCA_FLOW_ACTION_CONSTANT;
  descs->outer.src_port.mask.u64 = 0xffff;
}
```

9.3.2.3. Copy Field

Action description can be used to copy between packet field and metadata. For example:

```
static void
create_copy_packet_to_meta_actions(struct doca_flow_match *match#
                                   struct doca_flow_action_descs *descs)
{
        descs->src_ip.type = DOCA_FLOW_ACTION_COPY;
        descs->src_ip.copy.dst = &match->meta.u32[1];
}
```

9.3.2.4. Multiple Actions List

Creating a pipe is possible using a list of multiple actions. For example:

```
static void
create_multi_actions_for_pipe_cfg()
{
    struct doca_flow_actions *actions_arr[2];
    struct doca_flow_actions actions_0 = {0}, actions_1 = {0};
    struct doca_flow_pipe_cfg pipe_cfg = {0};

    /* input configurations for actions_0 and actions_1 */

    actions_arr[0] = &actions_0;
    actions_arr[1] = &actions_1;
    pipe_cfg.attr.nb_actions = 2;
    pipe_cfg.actions = actions_arr;
}
```

9.3.2.5. Summary of Action Types

Pipe Creation			Entry Creation
action_desc		Pipe Actions	Entry Actions
doca_flow_action_type	Configuration		
DOCA_FLOW_ACTION_AUTO Derived from pipe actions.	No specific config	0 – field ignored, no modification	N/A
		val != 0 – apply this val to all entries	N/A
		val = 0xfff – changeable field	Define val per entry
		Specific for Metadata - the meta field in the actions is used as a mask.	Define val per entry
DOCA_FLOW_ACTION_CONSTANT Pipe action is constant.	Define the mask	Define val to apply for all entries	N/A
DOCA_FLOW_ACTION_SET Set value from entry action.	Define the mask	N/A	Define val per entry
DOCA_FLOW_ACTION_ADD Add field value.	Define the dst field and width	N/A	Define val per entry
DOCA_FLOW_ACTION_COPY Copy field to another field.	Define the source and destination fields. ▶ Meta field → header field ▶ Header field → meta field ▶ Meta field → meta field	N/A	N/A

9.3.2.6. Summary of Fields

Field	Match	Modification	Add	Copy
meta.pkt_meta	x	x		x
meta.u32	x	x		x
Packet outer fields	x (field list)	x (field list)	TTL	Between meta[1]
Packet tunnel	x			To meta
Packet inner fields	x (field list)			To meta[1]

[1] Copy from meta to IP is not supported.

9.3.3. Setting Pipe Monitoring

If a meter 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. The meter policer is determined by the FWD action. If an entry has NULL FWD action, the policer FWD action is taken from the pipe.

The monitor also includes the aging configuration, if the aging time is set, this entry ages out if timeout passes without any matching on the entry. User data is used to map user usage. If the `user_data` field is set, when the entry ages out, query API returns this `user_data`. If `user_data` is not configured by the application, the aged pipe entry handle is returned.

For example:

```
static void build_entry_monitor(struct doca_flow_monitor *monitor, void *user_ctx)
{
 monitor->flags |= DOCA_FLOW_MONITOR_AGING;
 monitor->aging = 10;
 monitor->user_data = (uint64_t)user_ctx;
}
```

Refer to Pipe Entry Aged Query for more information.

9.3.4. Setting Pipe Forwarding

The FWD (forwarding) action is the last action in a pipe, and it directs where the packet goes next. Users may configure one of the following destinations:

- Send to software (representor)
- Send to wire
- Jump to next pipe
- Drop packets

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 a pipe is created, users must define forwarding per entry. In this instance, pipes may have different forwarding actions.

When a pipe includes meter monitor `<cir, cbs>`, it must have `fwd` defined as well as the policer.

The following is an RSS forwarding example:

```
fwd->type = DOCA_FLOW_FWD_RSS;
fwd->rss_queues = queues;
fwd->rss_flags = DOCA_FLOW_RSS_IP | DOCA_FLOW_RSS_UDP;
fwd->num_of_queues = 4;
```

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 a specific match, a packet may be directed to a single queue by having RSS forwarding with a single queue.

Changeable RSS forwarding is supported. When creating the pipe, the `num_of_queues` must be set to `0xffffffff`, then different forwarding RSS information can be set when adding each entry.

```
fwd->num_of_queues = 0xffffffff;
```

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 is 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`.

Changeable port forwarding is also supported. When creating the pipe, the `port_id` must be set to `0xffff`, then different forwarding `port_id` values can be set when adding each entry.

```
fwd->port_id = 0xffff;
```

9.3.5. Basic Pipe Create

Once all parameters are defined, the user should call `doca_flow_pipe_create` to create a pipe.

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.

Refer to the NVIDIA DOCA Libraries API Reference Manual to see which fields are optional and may be skipped. It is typically recommended to set optional fields to 0 when not in use. See Miss Pipe and Control Pipe for more information.

Once a 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.

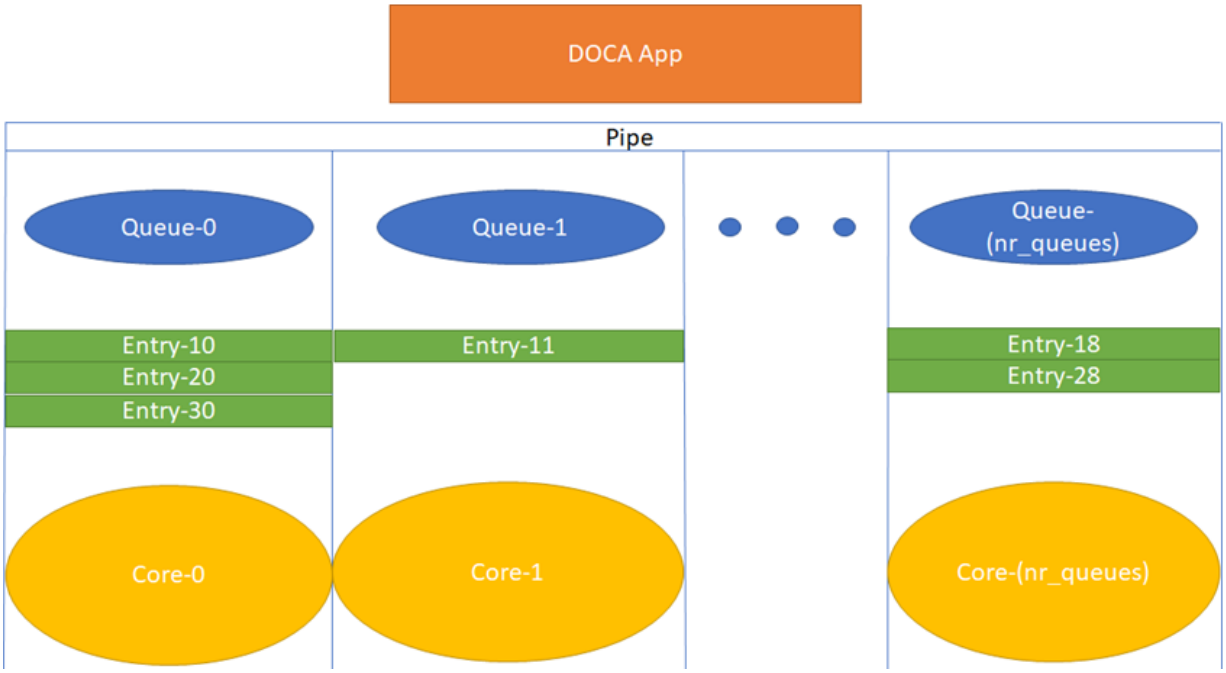
There is no priority between pipes or entries. The way that priority can be implemented is to match the highest priority first, and if a miss occurs, to jump to the next PIPE. There can be more than one PIPE on a root as long the pipes are not overlapping. If entries overlap, the priority is set according to the order of entries added. So, if two root pipes have overlapping matching and PIPE1 has higher priority than PIPE2, users should add an entry to PIPE1 after all entries are added to PIPE2.

9.3.6. 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 FWD action.

When a user calls `doca_flow_pipe_add_entry()` to add an entry, they should define the values that are not constant among all entries in the pipe. And if FWD is not defined then that is also mandatory.

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 mandatory to use a similar architecture as the 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.



Each core is expected to use its own dedicated `pipe_queue` number when calling `doca_flow_pipe_entry`. Using the same `pipe_queue` from different cores causes a race condition and has unexpected results.

Upon success, a handle is returned. If a failure occurs, a NULL value is returned, and an error message is 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, void *usr_ctx, struct
 doca_flow_pipe_entry *entry);
```

9.3.6.1. 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 by calling `doca_flow_query`.

The retrieved statistics are stored in struct `doca_flow_query`.

9.3.6.2. Pipe Entry Aged Query

When a user calls `doca_flow_aged_query()`, this query is used to get the aged-out entries by the time quota in microseconds. The entry handle or the `user_data` input is returned by this API.

Since the number of flows can be very large, the query of aged flows is limited by a quota in microseconds. This means that it may return without all flows and requires the user to call it again. When the query has gone over all flows, a full cycle is done.

The struct `doca_flow_aged_query` contains the element `user_data` which contains the aged-out flow contexts.

9.3.7. Pipe Entry With Multiple Actions

Users can define multiple actions per pipe. This gives the user the option to define different actions per entry in the same pipe by providing the `action_idx` in `struct doca_flow_actions`.

For example, to create two flows with the same match but with different actions, users can provide two actions upon pipe creation, `Action_0` and `Action_1`, which have indices 0 and 1 respectively in the actions array in the pipe configuration. `Action_0` has `modify_mac`, and `Action_1` has `modify_ip`.

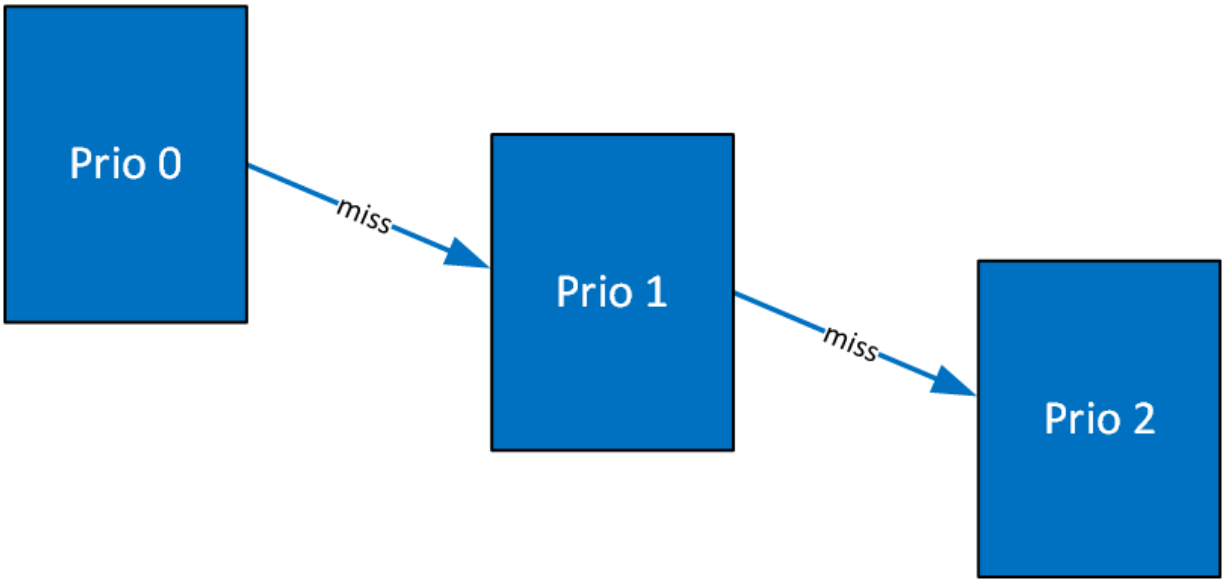
Users can also add two kinds of entries to the pipe, the first one with `Action_0` and the second with `Action_1`. This is done by assigning 0 in the `action_idx` field in `struct doca_flow_actions` when creating the first entry and 1 when creating the second one.

9.3.8. Miss Pipe and Control Pipe

Note: Only one root pipe is allowed. If more than one is needed, create a control pipe as root and forward the packets to relevant non-root pipes.

To set priority between pipes, users must use miss-pipes. Miss pipes allow to look up entries associated with pipe X, and if there are no matches, to jump to pipe X+1 and perform a lookup on entries associated with pipe X+1.

The following figure illustrates the HW table structure:



The first lookup is performed on the table with priority 0. If no hits are found, then it jumps to the next table and performs another lookup.

The way to implement a miss pipe in DOCA Flow is to use a miss pipe in FWD. In struct `doca_flow_fwd`, the field `next_pipe` signifies that when creating a pipe, if a `fwd_miss` is configured then if a packet does not match the specific pipe, steering should jump to `next_pipe` in `fwd_miss`.

Note: `fwd_miss` is of type `struct doca_flow_fwd` but it only implements two forward types of this struct:

- `DOCA_FLOW_FWD_PIPE` – forwards the packet to another pipe
- `DOCA_FLOW_FWD_DROP` – drops the packet

Other forwarding types (e.g., forwarding to port or sending to RSS queue) are not supported.

`next_pipe` is defined as `doca_flow_pipe` and created by `doca_flow_pipe_create`. To separate `miss_pipe` and a general one, `is_root` is introduced in struct `doca_flow_pipe_cfg`. If `is_root` is true, it means the pipe is a root pipe executed on packet arrival. Otherwise, the pipe is `next_pipe`.

When `fwd_miss` is not null, the packet that does not match the criteria is handled by `next_pipe` which is defined in `fwd_miss`.

In internal implementations of `doca_flow_pipe_create`, if `fwd_miss` is not null and the forwarding action type of `miss_pipe` is `DOCA_FLOW_FWD_PIPE`, a flow with the lowest priority is created that always jumps to the group for the `next_pipe` of the `fwd_miss`. Then the flow of `next_pipe` can handle the packets, or drop the packets if the forwarding action type of `miss_pipe` is `DOCA_FLOW_FWD_DROP`.

For example, VXLAN packets are forwarded as RSS and hairpin for other packets. The `miss_pipe` is for the other packets (non-VXLAN packets) and the match is for

general Ethernet packets. The `fwd_miss` is defined by `miss_pipe` and the type is `DOCA_FLOW_FWD_PIPE`. For the VXLAN pipe, it is created by `doca_flow_create()` and `fwd_miss` is introduced.

Since, in the example, the jump flow is for general Ethernet packets, it is possible that some VXLAN packets match it and cause conflicts. For example, VXLAN flow entry for `ipA` is created. A VXLAN packet with `ipB` comes in, no flow entry is added for `ipB`, so it hits `miss_pipe` and is hairpinned.

A control pipe is introduced to handle the conflict. When a user calls `doca_flow_create_control_pipe()`, the new control pipe is created without any configuration except for the port. Then the user can add different matches with different forwarding and priorities when there are conflicts.

The user can add a control entry by calling `doca_flow_control_pipe_add_entry()`.

`priority` must be defined as higher than the lowest priority (3) and lower than the highest one (0).

The other parameters represent the same meaning of the parameters in `doca_flow_pipe_create`. In the example above, a control entry for VXLAN is created. The VLXAN packets with `ipB` hit the control entry.

9.3.9. doca_flow_pipe_lpm

`doca_flow_pipe_lpm` uses longest prefix match (LPM) matching. LPM matching is limited to a single field of the `doca_flow_match` (e.g., the outer destination IP). Each entry is consisted of a value and a mask (e.g., 10.0.0.0/8, 10.10.0.0/16, etc). The LPM match is defined as the entry that has the maximum matching bits. For example, using the two entries 10.7.0.0/16 and 10.0.0.0/8, the IP 10.1.9.2 matches on 10.0.0.0/8 and IP 10.7.9.2 matches on 10.7.0.0/16 because 16 bits match.

The monitor, actions, and FWD of the DOCA Flow LPM pipe works the same as the basic DOCA Flow pipe.

`doca_flow_pipe_lpm` insertion max latency can be measured in milliseconds in some cases and, therefore, it is better to insert it from the control path. To get the best insertion performance, entries should be added in large batches.

Note: An LPM pipe cannot be a root pipe. You must create a pipe as root and forward the packets to the LPM pipe.

9.3.10. doca_flow_pipe_ordered_list

`doca_flow_pipe_ordered_list` allows the user to define a specific order of actions and multiply the same type of actions (i.e., specific ordering between counter/meter and encap/decap).

An ordered list pipe is defined by an array of actions (i.e., sequences of actions). Each entry can be an instance one of these sequences. An ordered list pipe may consist of up to an array of 8 different actions. The maximum size of each action array is 4 elements. Resource allocation may be optimized when combining multiple action arrays in one ordered list pipe.

9.3.11. Hardware Steering Mode

Users can enable hardware steering mode by setting devarg `dv_flow_en` to `2`.

The following is an example of running DOCA with hardware steering mode:

```
.... -a 03:00.0, dv_flow_en=2 -a 03:00.1, dv_flow_en=2....
```

The following is an example of running DOCA with software steering mode:

```
.... -a 03:00.0 -a 03:00.1 ....
```

The `dv_flow_en=2` means that hardware steering mode is enabled.

In the struct `doca_flow_cfg`, the member `mode_args` represents DOCA applications. If it is defined with `hws` (e.g., `"vnf,hws"`, `"switch,hws"`, `"remote_vnf,hws"`) then hardware steering mode is enabled.

To create an entry by calling `doca_flow_pipe_add_entry`, the parameter `flags` can be set as `DOCA_FLOW_WAIT_FOR_BATCH` or `DOCA_FLOW_NO_WAIT`. `DOCA_FLOW_WAIT_FOR_BATCH` means that this flow entry waits to be pushed to hardware. Batch flows then can be pushed only at once. This reduces the push times and enhances the insertion rate. `DOCA_FLOW_NO_WAIT` means that the flow entry is pushed to hardware immediately.

The parameter `usr_ctx` is handled in the callback defined in struct `doca_flow_cfg`.

`doca_flow_entries_process` processes all the flows in this queue. After the flow is handled and the status is returned, the callback is executed with the status and `usr_ctx`.

If the user does not define the callback in `doca_flow_cfg`, the user can get the status using `doca_flow_entry_get_status` to check if the flow has completed offloading or not.

9.3.12. Isolated Mode

In non-isolated mode (default) any received packets (e.g., following an RSS forward) can be processed by the DOCA application, bypassing the kernel. In the same way, the DOCA application can send packets to the NIC without kernel knowledge. This is why, by default, no replies are received when pinging a host with a running DOCA application. If only specific packet types (e.g., DNS packets) should be processed by the DOCA application, while other packets (e.g., ICMP ping) should be handled directly the kernel, then isolated mode becomes useful.

In isolated mode, packets that match root pipe entries are steered to the DOCA application (as usual) while other packets are received/sent directly by the kernel.

To activate isolated mode, in the struct `doca_flow_cfg`, the member `mode_args` represents DOCA applications. If it is defined with isolated (i.e., `"vnf,hws,isolated"`, `"switch,isolated"`) then isolated mode is enabled.

If you plan to create a pipe with matches followed by action/monitor/forward operations, due to functional/performance considerations, it is advised that root pipe entries include the matches followed by a next pipe forward operation. In the next pipe, all the planned matches actions/ monitor/forward operations could be specified. Unmatched packets are received and sent by the kernel.

9.4. Teardown

9.4.1. Pipe Entry Teardown

When an entry is terminated by the user application or ages-out, the user should call the entry destroy function, `doca_flow_pipe_rm_entry()`. This frees the pipe entry and cancels hardware offload.

9.4.2. Pipe Teardown

Whena pipe is terminated by the user application, the user should call the pipe destroy function, `doca_flow_destroy_pipe()`. This destroys the pipe and the pipe entries that match it.

When all pipes of a port are terminated by the user application, the user should call the pipe flush function, `doca_flow_port_pipe_flush()`. This destroys all pipes and all pipe entries belonging to this port.

9.4.3. Port Teardown

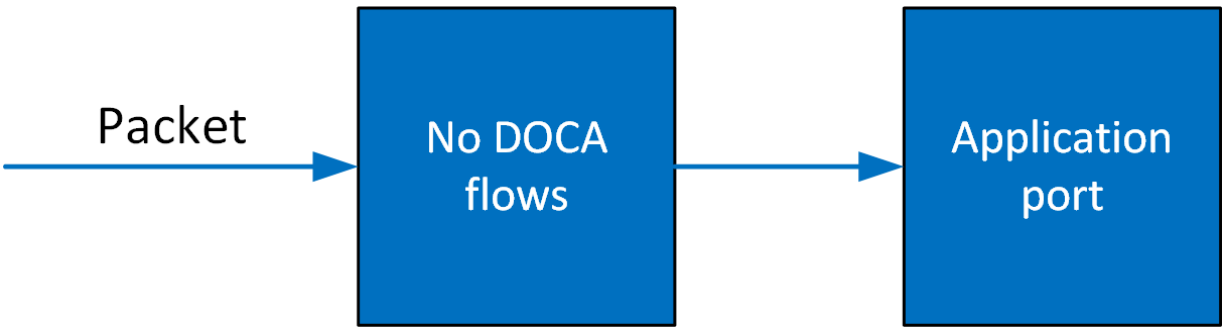
When the port is not used anymore, the user should call the port destroy function, `doca_flow_destroy_port()`. This destroys the DOCA port and frees all resources of the port.

9.4.4. Flow Teardown

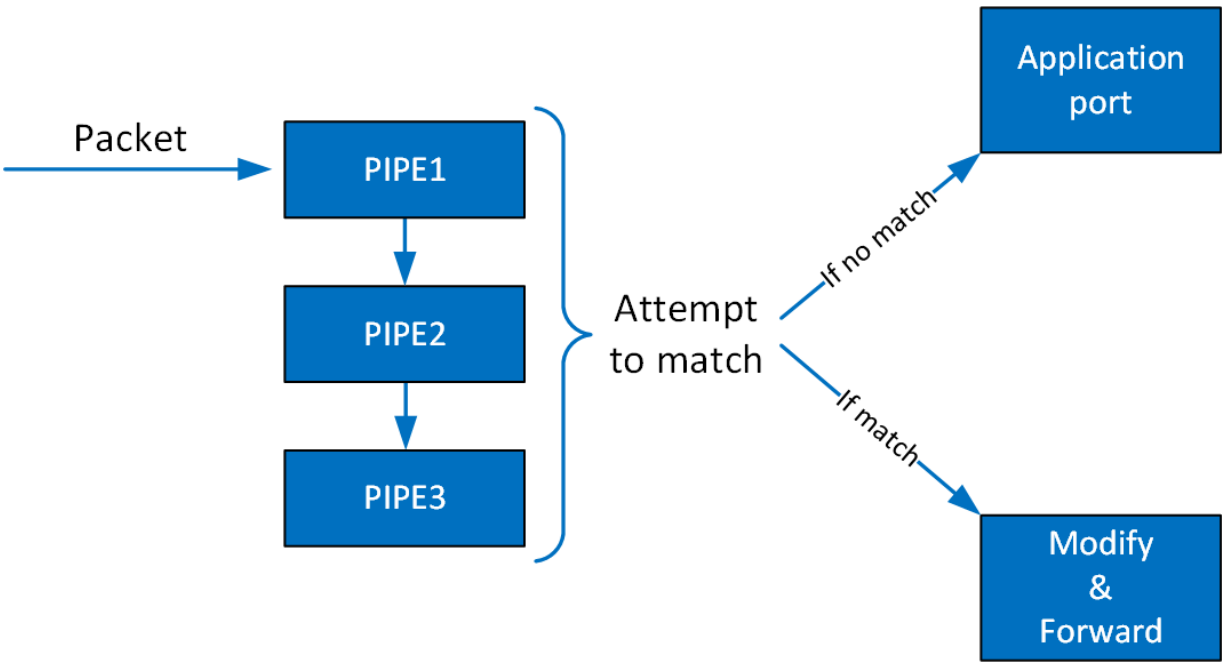
When the DOCA Flow is not used anymore, the user should call the flow destroy function, `doca_flow_destroy()`. This releases all the resources used by DOCA Flow.

Chapter 10. 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 a port in the 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 is matched, 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, the 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 that the packet goes through, the longer it takes the 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.

Chapter 11. DOCA Flow gRPC

This chapter describes gRPC support for DOCA Flow. The DOCA Flow gRPC-based API allows users on the host to leverage the HW offload capabilities of the BlueField DPU using gRPCs from the host itself.

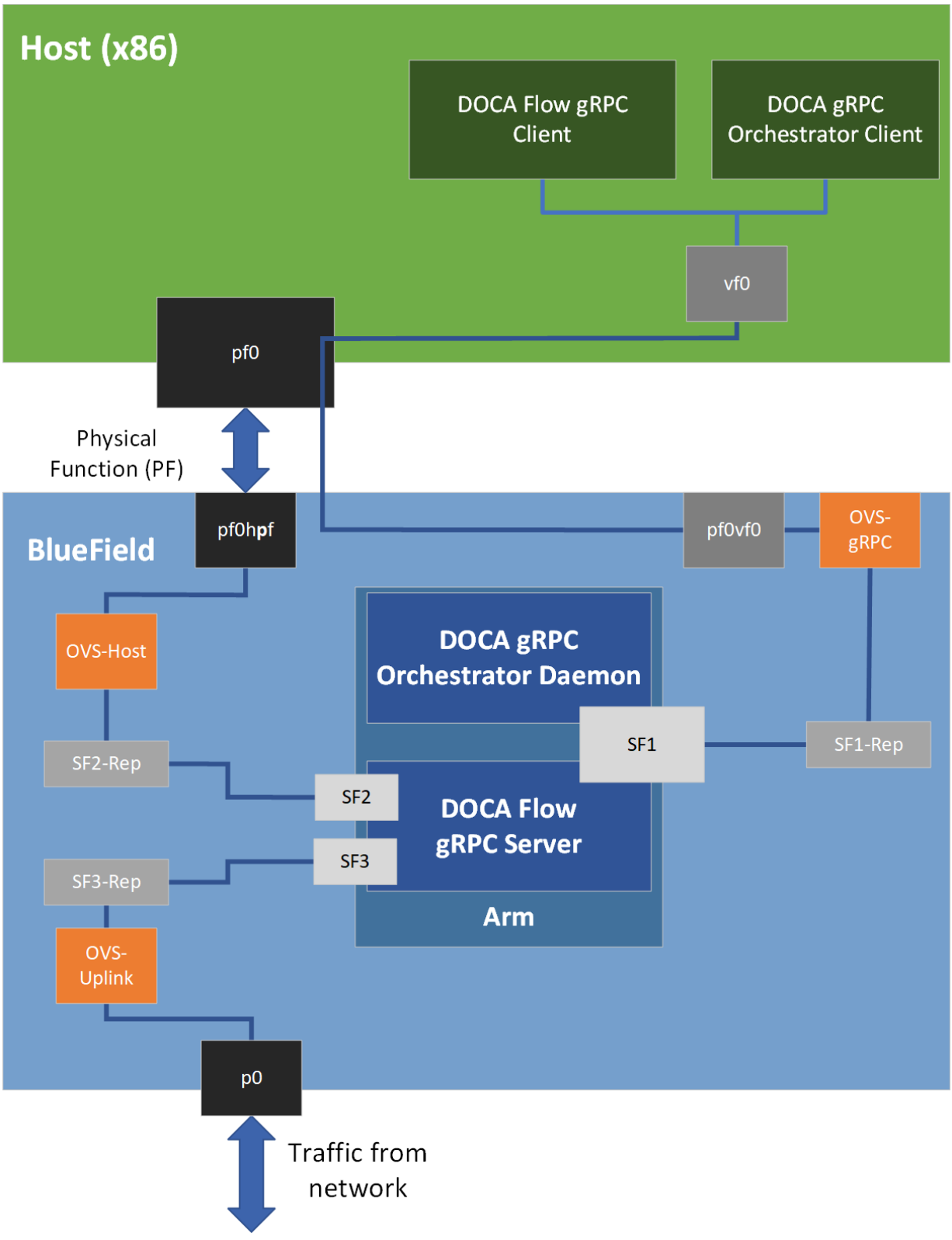
DOCA Flow gRPC server implementation is based on gRPC's `async` API to maximize the performance offered to the gRPC client on the host. In addition, the gRPC support in the DOCA Flow library provides a client interface which gives the user the ability to send/receive messages to/from the client application in C.

This section is divided into the following parts:

- proto-buff – this section details the messages defined in the proto-buff
- Client interface – this section details the API for communicating with the server
- Usage – this section explains how to use the client interface to develop your own client application based on DOCA Flow gRPC support

Refer to NVIDIA DOCA gRPC Infrastructure User Guide for more information about DOCA gRPC support.

The following figure illustrates the DOCA Flow gRPC server-client communication when running in VNF mode.


Host (x86)
DOCA Flow gRPC Client
DOCA gRPC Orchestrator Client
vf0
pf0
Physical Function (PF)
BlueField
pf0hpf
pf0vf0
OVS-gRPC
OVS-Host
DOCA gRPC Orchestrator Daemon
SF1
SF1-Rep
SF2-Rep
SF2
DOCA Flow gRPC Server
SF3
SF3-Rep
Arm
OVS-Uplink
p0
Traffic from network

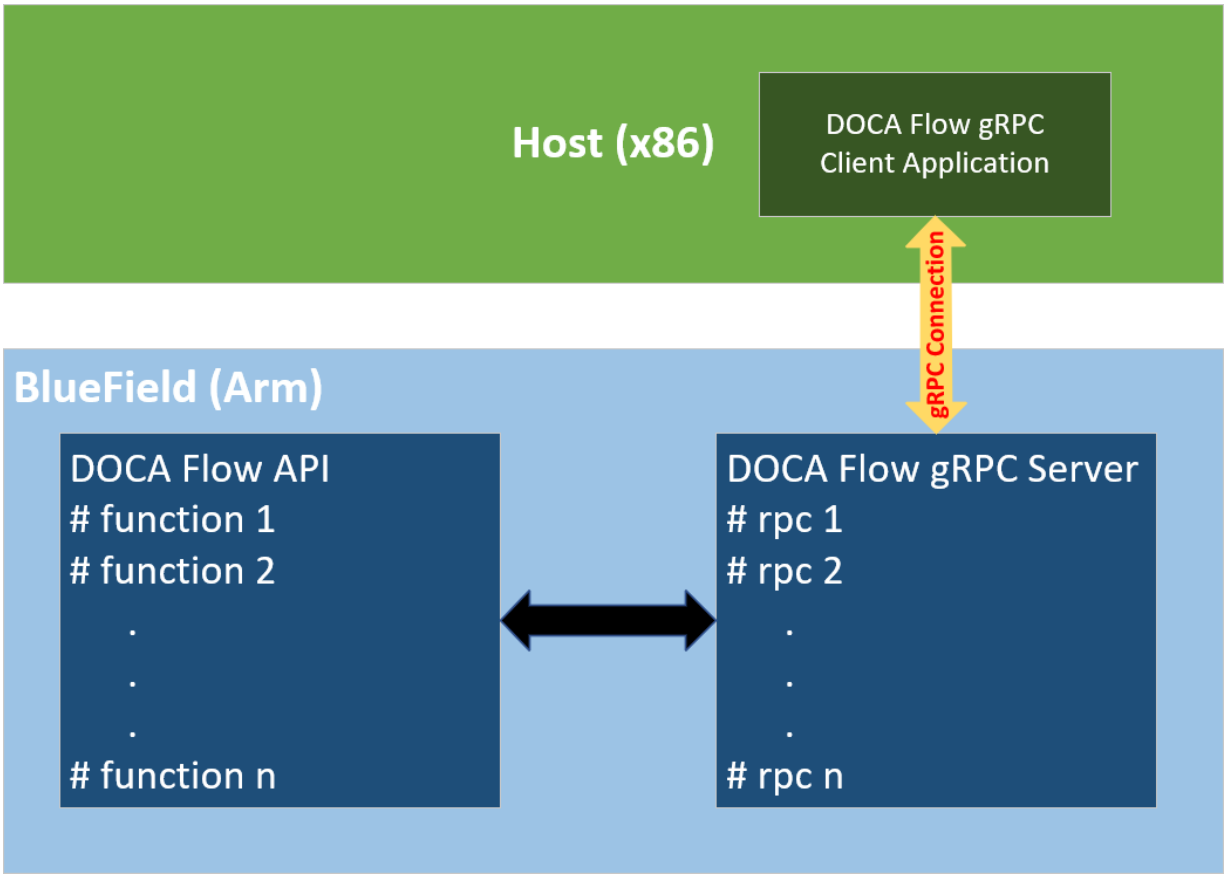
11.1. Proto-Buff

As with every gRPC proto-buff, DOCA Flow gRPC proto-buff defines the services it introduces, and the messages used for the communication between the client and the server. Each proto-buff DOCA Flow method:

- Represents exactly one function in DOCA Flow API
- Has its request message, depending on the type of the service
- Has the same response message (`DocaFlowResponse`)

In addition, DOCA Flow gRPC proto-buff defines several of messages that are used for defining request messages, the response message, or other messages.

Each message defined in the proto-buff represents either a struct or an enum defined by DOCA Flow API. The following figure illustrates how DOCA Flow gRPC server represents the DOCA Flow API.



The proto-buff path for DOCA Flow gRPC is `/opt/mellanox/doca/infrastructure/doca_grpc/doca_flow/doca_flow.proto`.

11.1.1. Response Message

All services have the same response message. `DocaFlowResponse` contains all types of results that the services may return to the client.

```
/** General DOCA Flow response message */
message DocaFlowResponse{
    bool success = 1;    /* True in case of success */
    DocaFlowError error = 2; /* Otherwise, this field contains the error information
 */
    /* in case of success, one or more of the following may be used */
        uint32 port_id = 3;
        uint64 pipe_id = 4;
        uint64 entry_id = 5;
        string port_pipes_dump = 6;
        DocaFlowQueryRes query_stats = 7;
        bytes priv_data = 8;
        DocaFlowHandleAgingRes handle_aging_res = 9;
  uint64 nb_entries_processed = 10;
  DocaFlowEntryStatus status = 11;
}
```

11.1.2. DocaFlowCfg

The `DocaFlowCfg` message represents the `doca_flow_cfg` struct.

11.1.3. DocaFlowPortCfg

The `DocaFlowPortCfg` message represents the `doca_flow_port_cfg` struct.

11.1.4. DocaFlowPipeCfg

The `DocaFlowPipeCfg` message represents the `doca_flow_pipe_cfg` struct.

11.1.5. DocaFlowMeta

The `DocaFlowMeta` message represents the `doca_flow_meta` struct.

The `DocaFlowMatch` message contains fields of types `DocaFlowIPAddress` and `DocaFlowTun`. These types are messages which are also defined in the `doca_flow.proto` file and represent `doca_flow_ip_address` and `doca_flow_tun` respectively.

11.1.6. DocaFlowMatch

The `DocaFlowMatch` message represents the `doca_flow_match` struct.

The `DocaFlowMatch` message contains fields of types `DocaFlowIPAddress` and `DocaFlowTun`. These types are messages which are also defined in the `doca_flow.proto` file and represents `doca_flow_ip_address` and `doca_flow_tun` respectively.

11.1.7. DocaFlowActions

The `DocaFlowActions` message represents the `doca_flow_actions` struct.

11.1.8. DocaFlowActionDesc

The `DocaFlowActionDesc` message represents the `doca_flow_action_desc` struct.

The`DocaFlowActionDesc` message contains fields of type `DocaFlowActionField` which are also defined in the `doca_flow.proto` file and represent `doca_flow_action_field`.

11.1.9. DocaFlowMonitor

The `DocaFlowMonitor` message represents the `doca_flow_monitor` struct.

11.1.10. DocaFlowFwd

The `DocaFlowFwd` message represents the `doca_flow_fwd` struct.

11.1.11. DocaFlowQueryStats

The `DocaFlowQueryStats` message represents the `doca_flow_query` struct.

11.1.12. DocaFlowHandleAgingRes

The DocaFlowHandleAgingRes message contains all the parameters needed to save the result of an aging handler.

11.1.13. DocaFlowInit

DOCA Flow initialization gRPC:

```
rpc DocaFlowInit(DocaFlowCfg) returns (DocaFlowResponse);
```

If successful, the `success` field in the response message is set to `true`. Otherwise, the `error` field is populated with the error information.

11.1.14. DocaFlowPortStart

The service for starting the DOCA flow ports:

```
rpc DocaFlowPortStart(DocaFlowPortCfg) returns (DocaFlowResponse);
```

If successful, the `success` field in the `DocaFlowResponse` is set to `true`. Otherwise, the `error` field is populated with the error information.

11.1.15. DocaFlowPortPair

The `DocaFlowPortPairRequest` message contains all the necessary information for port pairing:

```
message DocaFlowPortPairRequest {
    uint32 port_id = 1;         /* port identefier of doca flow port. */
    uint32 pair_port_id = 2;    /* port identefier to the pair port. */
}
```

Once all the parameters are defined, a "port pair" service can be called. The service for DOCA Flow port pair is as follows:

```
rpc DocaFlowPortPair(DocaFlowPortPairRequest) returns (DocaFlowResponse);
```

If successful, the success field in the `DocaFlowResponse` is set to `true`. Otherwise, the `error` field is populated with the error information.

11.1.16. DocaFlowPipeCreate

The `DocaFlowPipeCreateRequest` message contains all the necessary information for pipe creation as the DOCA Flow API suggests:

```
message DocaFlowPipeCreateRequest {
        DocaFlowPipeCfg cfg = 1;            /* the pipe configurations */
        DocaFlowFwd fwd = 2;                /* the pipe's FORWARDING component  */
        DocaFlowFwd fwd_miss = 3;           /* The FORWARDING miss component */
}
```

Once all the parameters are defined, a "create pipe" service can be called:

```
rpc DocaFlowPipeCreate (DocaFlowPipeCreateRequest) returns (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true` and the `pipe_id` field is populated with the ID of the added entry. This ID should be given when adding entries to the pipe. Otherwise, the error field is filled accordingly.

11.1.17. DocaFlowPipeAddEntry

The `DocaFlowPipeAddEntryRequest` message contains all the necessary information for adding an entry to the pipe:

```
message DocaFlowPipeAddEntryRequest{
        uint32 pipe_queue = 2;              /* the pipe queue */
        uint64 pipe_id = 3;                 /* the pipe ID to add the entry to */
        DocaFlowMatch match = 4;            /* matcher for the entry */
        DocaFlowActions actions = 5;        /* actions for the entry */
        DocaFlowMonitor monitor = 6;        /* monitor for the entry */
        DocaFlowFwd fwd = 7;                /* The entry's FORWARDING component */
        uint32 flags = 1;                   /* whether the flow entry is pushed to HW
 immediately or not */
}
```

Once all the parameters are defined, an "add entry to pipe" service can be called:

```
rpc DocaFlowPipeAddEntry(DocaFlowPipeAddEntryRequest) returns (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`, and the `entry_id` field is populated with the ID of the added entry. This ID should be given when adding entries to the pipe. Otherwise, the error field is filled accordingly.

11.1.18. DocaFlowPipeControlAddEntry

The `DocaFlowPipeControlAddEntryRequest` message contains the required arguments for adding entries to the control pipe:

```
message DocaFlowPipeControlAddEntryRequest{
        uint32 priority = 2;                /* he priority of the added entry to the
 filter pipe */
        uint32 pipe_queue = 3;              /* the pipe queue */
        uint64 pipe_id = 4;                 /* the pipe ID to add the entry to */
        DocaFlowMatch match = 5;            /* matcher for the entry */
        DocaFlowMatch match_mask = 6;       /* matcher mask for the entry */
```

```
        DocaFlowFwd fwd = 7;                /* The entry's FORWARDING component */
}
```

Once all the parameters are defined, an "add entry to pipe" service can be called:

```
rpc DocaFlowPipeControlAddEntry(DocaFlowPipeControlAddEntryRequest) returns
 (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`, and the `entry_id` field is populated with the ID of the added entry. This ID should be given when adding entries to the pipe. Otherwise, the error field is filled accordingly.

11.1.19. DocaFlowPipeLpmAddEntry

The `DocaFlowPipeLpmAddEntryRequest` message contains the required arguments for adding entries to the LPM pipe:

```
message DocaFlowPipeLpmAddEntryRequest{
        uint32 pipe_queue = 1;              /* the pipe queue */
        uint64 pipe_id = 2;                 /* the pipe ID to add the entry to */
        DocaFlowMatch match = 3;            /* matcher for the entry */
        DocaFlowMatch match_mask = 4;       /* matcher mask for the entry */
        DocaFlowActions actions = 5;        /* actions for the entry */
        DocaFlowMonitor monitor = 6;        /* monitor for the entry */
        DocaFlowFwd fwd = 7;                /* The entry's FORWARDING component */
        uint32 flag = 8;                    /* whether the flow entry will be pushed
 to HW immediately or not */
}
```

Once all the parameters are defined, an "add entry to LPM pipe" service can be called:

```
rpc DocaFlowPipeLpmAddEntry(DocaFlowPipeLpmAddEntryRequest) returns
 (DocaFlowResponse);
```

If successful, the success field in `DocaFlowResponse` is set to `true`, and the `entry_id` field is populated with the ID of the added entry. This ID should be given when adding entries to the pipe. Otherwise, the error field is filled accordingly.

11.1.20. DocaFlowEntriesProcess

The `DocaFlowEntriesProcessRequest` contains the required arguments for processing the entries in the queue.

```
message DocaFlowEntriesProcessRequest{
        uint32 port_id = 1;          /* the port ID of the entries to process. */
        uint32 pipe_queue = 2;       /* the pipe queue of the entries to process.
 */
        /* max time in micro seconds for the actual API to process entries. */
        uint64 timeout = 3;
  /* An upper bound for the required number of entries to process. */
        uint32 max_processed_entries = 4;
}
```

Once all the parameters are defined, the "entries process" service can be called:

```
rpc DocaFlowEntriesProcess(DocaFlowEntriesProcessRequest) returns
 (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`, and the `nb_entries_processed` field is populated with the ID of the number of processed entries.

11.1.21. DocaFlowEntyGetStatus

The `DocaFlowEntryGetStatusRequest` contains the required arguments for fetching the status of a given entry.

```
message DocaFlowEntryGetStatusRequest{
        /* the entry identifier of the requested entry's status. */
        uint64 entry_id = 1;
}
```

Once all the parameters are defined, the "entry get status" service can be called:

```
rpc DocaFlowEntriesProcess(DocaFlowEntriesProcessRequest) returns
 (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`, and the `status` field is populated with the status of the requested entry. This field's type is `DocaFlowEntryStatus`, which is an enum defined in the proto-buff, and represents the enum `doca_flow_entry_status`, defined in the DOCA Flow header.

11.1.22. DocaFlowQuery

`DocaFlowQueryRequest` contains the required arguments for querying a given entry.

```
message DocaFlowQueryRequest{
    uint64 entry_id = 3;              /* the entry id. */
}
```

Once all the parameters are defined, the "query" service can be called:

```
rpc DocaFlowQuery(DocaFlowQueryRequest) returns (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`, and the `query_stats` field is populated with the query result of the requested entry. This field's type is `DocaFlowQueryStats`, which is an enum defined in the proto-buff, and represents the `doca_flow_query` struct.

11.1.23. DocaFlowAgingHandle

`DocaFlowAgingHandleRequest` contains the required arguments for handling aging by DOCA Flow.

```
message DocaFlowAgingHandleRequest{
    uint32 port_id = 1;               /* the port id handle aging to. */
    uint32 queue = 2;                 /* the queue identifier */
    uint64 quota = 3;                 /* the max time quota in micro seconds for
 this function to handle aging. */
    uint64 user_data = 4;             /* the user input context, otherwise the
 doca_flow_pipe_entry pointer */
    uint32 len = 5;                   /* the user input length of entries array. */
}
```

Once all the parameters are defined, the "handle aging" service can be called:

```
rpc DocaFlowAgingHandle(DocaFlowAgingHandleRequest) returns (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true` and the `handle_aging_res` field is populated with the aging handler result. This field's type is `DocaFlowHandleAgingRes`.

11.1.24. DocaFlowSharedResourceCfg

The `DocaFlowSharedResourceCfgRequest` contains the required arguments for configuring a shared resource by DOCA Flow.

```
message DocaFlowSharedResourceCfgRequest {
  DocaFlowSharedResourceType type = 1;                /* Shared resource type */
 uint32 id = 2;                         /* Shared resource id */
 SharedResourceCfg cfg= 3;                      /* Shared resource configuration */
}
```

Once all the parameters are defined, the "config shared resource" service can be called:

```
rpc DocaFlowSharedResourceCfg(DocaFlowSharedResourceCfgRequest ) returns
 (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`.

11.1.25. DocaFlowSharedResourcesBind

The `DocaFlowSharedResourcesBindRequest` contains the required arguments for configuring a shared resource by DOCA Flow.

```
message DocaFlowSharedResourcesBindRequest {
  DocaFlowSharedResourceType type = 1;                /* Shared resource type */
 repeated uint32 resource_arr = 2;                   /* Repeated shared resource IDs
 */
 /* id of allowed bindable object, use 0 to bind globally */
 oneof bindable_obj_id {
  uint64 port_id = 3;        /* Used if the bindable object is port */
  uint64 pipe_id = 4;        /* Used if the bindable object is pipe */
 }
}
```

Once all the parameters are defined, the "bind shared resources" service can be called:

```
rpc DocaFlowSharedResourcesBind(DocaFlowSharedResourcesBindRequest ) returns
 (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`.

11.1.26. DocaFlowSharedResourcesQuery

The `DocaFlowSharedResourcesQueryRequest` contains the required arguments for configuring a shared resource by DOCA Flow.

```
message DocaFlowSharedResourcesQueryRequest {
    DocaFlowSharedResourceType type = 1;    /* Shared object type */
    repeated uint32 res_array = 2;      /*  Array of shared objects IDs to query */
}
```

Once all the parameters are defined, the "query shared resources" service can be called:

```
rpc DocaFlowSharedResourcesBind(DocaFlowSharedResourcesBindRequest ) returns
 (DocaFlowResponse);
```

If successful, the `success` field in `DocaFlowResponse` is set to `true`, and the `query_result` is populated with the query shared resources result. This field's type is DocaFlowQueryStats which represents the `doca_flow_query` struct.

11.2. DOCA Flow gRPC Client API

This section describes the recommended way for C developers to utilize gRPC support for DOCA Flow API. Refer to the DOCA Flow gRPC API in NVIDIA DOCA Libraries API Reference Manual for the library API reference.

The following sections provide additional details about the library API.

The DOCA installation includes `libdoca_flow_grpc` which is a library that provides a C API wrapper to the C++ gRPC, while mimicking the regular DOCA Flow API, for ease of use, and allowing smooth transition to the Arm.

This library API is exposed in `doca_flow_grpc_client.h` and is essentially the same as `doca_flow.h`, with the notation differences detailed in the following subsections. In general, the client interface API usage is almost identical to the regular API (i.e., DOCA Flow API). The arguments of each function in DOCA Flow API, are almost identical to the arguments of each function defined in the client API, except that each pointer is replaced with an ID representing the pointer.

For example, when creating a pipe or adding an entry, the original API returns a pointer to the created pipe or the added entry. However, when adding an entry or creating a pipe using the client interface, an ID representing the added entry or the created pipe is returned to the client application instead of the pointer.

11.2.1. doca_flow_grpc_response

`doca_flow_grpc_response` is a general response struct that holds information regarding the function result. Each API returns this struct. If an error occurs, the `error` field is populated with the error's information, and the `success` field is set to `false`. Otherwise, the `success` field is set to `true` and one of the other fields may hold a return value depending on the called function.

For example, when calling `doca_flow_grpc_create_pipe()` the `pipe_id` field is populated with the ID of the created pipe in case of success.

```
struct doca_flow_grpc_response {
      bool success;
      struct doca_flow_error error;
      uint64_t pipe_id;
      uint64_t entry_id;
      uint32_t aging_res;
      uint64_t nb_entries_processed;
      enum doca_flow_entry_status entry_status;
      };
```

success
In case of success, the value should be `true`.

error
In case of error, this struct should contain the error information.

pipe_id
Pipe ID of the created pipe.

entry_id
Entry ID of the created entry.

aging_res
Return value from handle aging.

nb_entries_processed
Return value from entries process.

entry_status
Return value from entry get status.

11.2.2. doca_flow_grpc_pipe_cfg

`doca_flow_grpc_pipe_cfg` is a pipeline configuration wrapper.

```
struct doca_flow_grpc_pipe_cfg {
 struct doca_flow_pipe_cfg cfg;
 uint16_t port_id;
};
```

cfg
Pipe configuration containing the user-defined template for the packet process.

port_id
Port ID for the pipeline.

11.2.3. doca_flow_grpc_fwd

`doca_flow_grpc_fwd` is a forwarding configuration wrapper.

```
struct doca_flow_grpc_fwd {
 struct doca_flow_fwd fwd;
 uint64_t next_pipe_id;
};
```

fwd
Forward configuration which directs where the packet goes next.

next_pipe_id
When using `DOCA_FLOW_FWD_PIPE`, this field contains the next pipe's ID.

11.2.4. doca_flow_grpc_client_create

This function initializes a channel to DOCA Flow gRPC server.

This must be invoked first before any other function in this API. This is a one-time call.

```
void doca_flow_grpc_client_create(char *grpc_address);
```

grpc_address [in]
String representing the server IP.

11.3. DOCA Flow gRPC Usage

A DOCA flow gRPC based server is implemented using the `async` API of gRPC. This is because the `async` API gives the server the ability to expose DOCA flow's concurrency support. Therefore, it is very important to use the client interface API for communicating with the DOCA Flow gRPC server because it hides all gRPC-related details from the users, which eases the use of the server, and exposes to the client applications the efficiency of DOCA Flow, in terms of flow insertion rates.

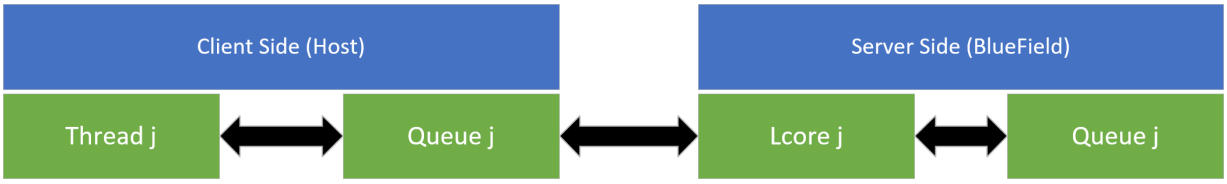
The following phases demonstrate a basic flow of client applications:

- Init Phase – client interface and environment initializations
- Flow life cycle – this phase is the same phase described in chapter Flow Life Cycle

It is important to emphasize that the number of threads for adding entries should be the same as the number of queues used when starting the server and initializing the environment (DPDK) and DOCA Flow API. This is to prevent bottlenecks on the server side.

If a client application starts the server on BlueField with N cores (through EAL arguments), this means that environment and DOCA Flow initialization should be done with N queues. As a result, the server launches N lcores, each one responsible for exactly one queue that is accessed only by it. Therefore, the client application should launch N threads as well, each being responsible for adding entries to a specific queue which is accessed by it only as well.

The following illustration demonstrates the relation between thread "j" on the client side and lcore "j" on the server side:



Chapter 12. Samples

Please refer to NVIDIA DOCA Flow 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