DOCA Flow Connection Tracking

Table of contents

Introduction

Architecture

- Aging
- Autonomous Mode
- Managed Mode

Prerequisites

- DPU
- ConnectX

Actions

- Shared Actions
- Non-shared Actions
- Action Sets in Pipe Creation
- Feature Enable
- Using Actions in Autonomous Mode
- Init
- Create DOCA Flow CT Pipe
- Create Shared Actions
- Implement Worker Callbacks
- Using Actions in Managed Mode
- Init
- Create DOCA Flow CT Pipe
- Create Shared Actions
- Add Entry

Remove Entry

Update Entry

Changeable Forward

Using Changeable Forward in Managed Mode

Using Changeable Forward in Autonomous Mode

API

enum doca_flow_ct_flags

enum doca_flow_ct doca_flow_ct_entry_flags

enum doca_flow_ct_rule_opr

struct direction_cfg

struct doca_flow_ct_worker_callbacks

struct doca_flow_ct_cfg

struct doca_flow_ct_actions

> **Note**
>
> This feature is not supported in this DOCA release. It will be re-enabled in DOCA version 3.0.

This guide provides an overview and configuration instructions for DOCA Flow CT API.

Introduction

DOCA Flow Connection Tracking (CT) is a 5-tuple table which supports the following:

- Track 5-tuple sessions (or 6-tuple when a zone is available)
- Zone based – virtual tables
- Aging (i.e., removes idle connections)
- Sets metadata for a connection
- Bidirectional packet handling
- High rate of connections per second (CPS)

The CT module makes it simple and efficient to track connections by leveraging hardware resources. The module supports both autonomous and managed mode.

Architecture

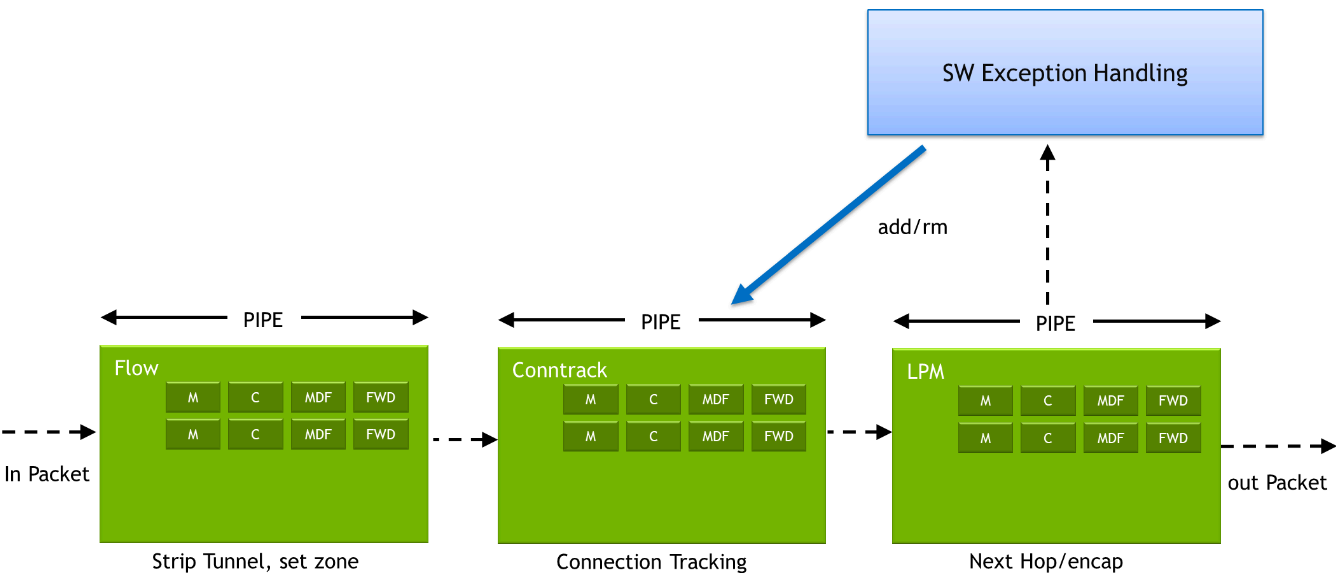
DOCA Flow CT pipe handles non-encapsulated TCP and UDP packets. The CT pipe only supports forward to next pipe or miss to next pipe actions:

- All packets matching known connection 6-tuples are forwarded to the CT's forward pipe
- Non-matching packets are forwarded to the miss pipe

The user application must handle packets accordingly.

The DOCA Flow CT API is built around four major parts:

- CT module manipulation – configuring CT module resources
- CT connection entry manipulation – adding, removing, or updating connection entries
- Callbacks – handling asynchronous entry processing result
- Pipe and entry statistics



Aging

Aging time is a time in seconds that sets the maximum allowed time for a session to be maintained without a packet seen. If that time elapses with no packet being detected, the session is terminated.

To support aging, a dedicated aging thread is started to poll and check counters for all connections.

Autonomous Mode

In this mode, DOCA runs multiple CT workers internally, to handle connections in parallel.

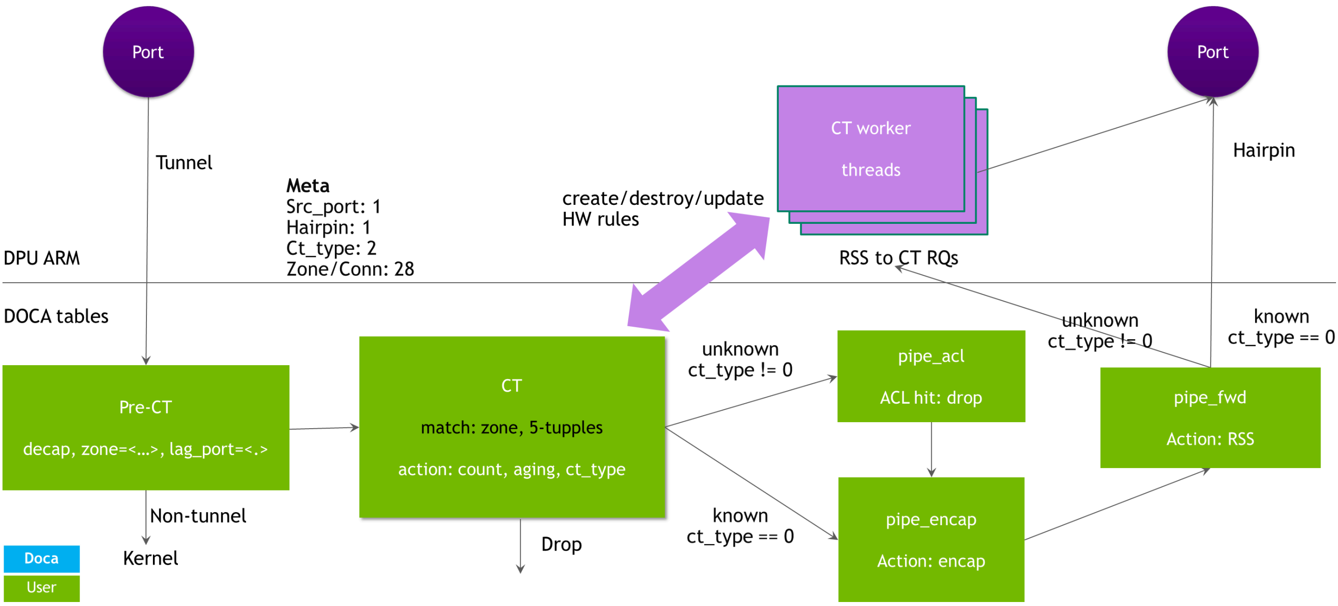
A connection's lifecycle is controlled by the connection state encapsulated in the packet and time-based aging.

CT workers establish and close connections automatically based on the connection's state stored in packet meta.

Packet meta is defined as follows:

```
uint32_t src : 1;          /**< Source port in multi-port E-Switch mode */
uint32_t hairpin : 1;      /**< Subject to forward using hairpin. */
uint32_t type : 2;         /**< CT packet type: New, End or Update */
uint32_t data : 28;        /**< Zone set by user or reserved after CT pipe. */
```

- `data` – CT table matches on packet meta (zone) and 5-tuples
- `type` – can have the following values:
  - `NONE` – (known) if packet hit any connection rule
  - `NEW` – if new TCP or UDP connection
  - `END` – if TCP connection closed
- `src` and `hairpin` – used for forwarding pipe and worker to deliver packet



Managed Mode

The application is responsible for managing the worker threads in this mode, parsing and handling the connection's lifecycle.

Managed mode uses DOCA Flow CT management APIs to create or destroy the connections.

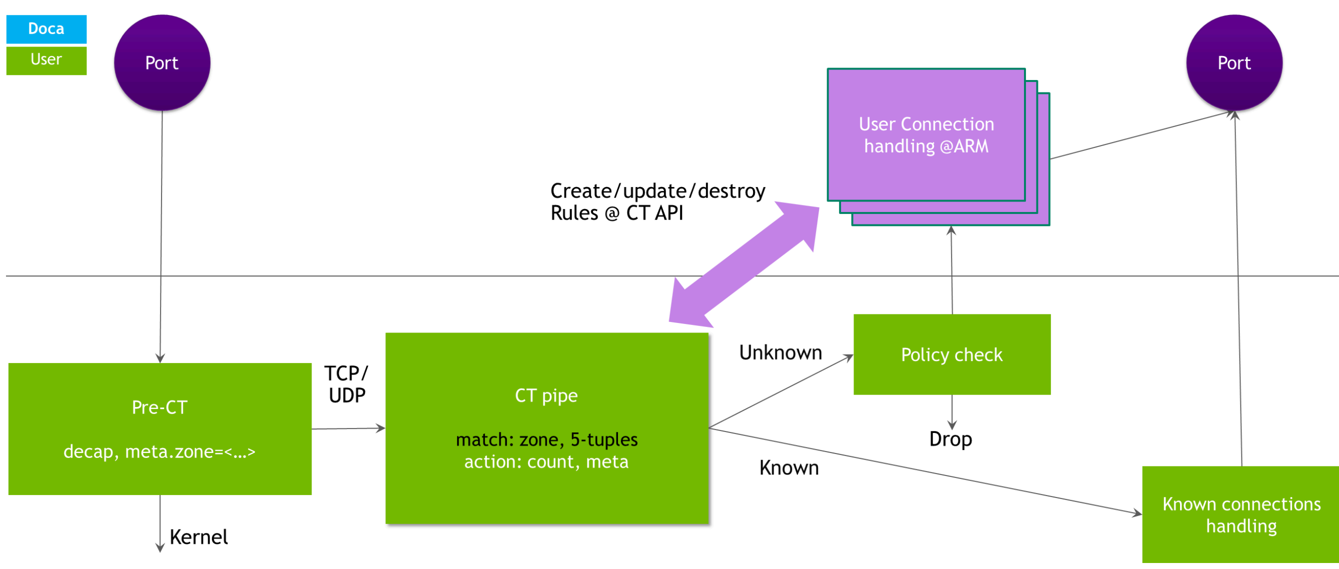
The CT aging module notifies on aged out connections by calling callbacks.

Users can create connection rules with a different pattern, meta, or counter, for each packet direction.

Info

Users are responsible for defining meta and mask to `match` and `modify`.

Users can create one rule of a connection first, then create another rule using API `doca_flow_ct_entry_add_dir()`.



DOCA Flow API can be used to process CT entries with a CT-dedicated queue.

- `doca_flow_entries_process` – process pipe entries in queue

- `doca_flow_aging_handle` – handle pipe entries aging

Info

Other DOCA Flow APIs like CT entry status query and pipe miss query are not supported.

Prerequisites

DPU

To enable DOCA Flow CT on the DPU, perform the following on the Arm:

1. Enable `iommu.passthrough` in Linux boot commands (or disable SMMU from the DPU BIOS):
    1. Run:

       ```
       sudo vim /etc/default/grub
       ```

    2. Set `GRUB_CMDLINE_LINUX="iommu.passthrough=1"`.
    3. Run:

       ```
       sudo update-grub
       sudo reboot
       ```

2. Configure DPU firmware with `LAG_RESOURCE_ALLOCATION=1`:

```
sudo mlxconfig -d <device-id> s LAG_RESOURCE_ALLOCATION=1
```

Info

Retrieve `device-id` from the output of the `mst status -v` command. If, under the MST tab, the value is N/A, run the `mst start` command.

3. Update `/etc/mellanox/mlnx-bf.conf` as follows:

```
ALLOW_SHARED_RQ="no"
```

4. Perform power cycle on the host and Arm sides.
5. If working with a single port, set the DPU into e-switch mode:

```
sudo devlink dev eswitch set pci/<pcie-address> mode
switchdev
sudo devlink dev param set pci/<pcie-address> name
esw_multiport value false cmode runtime
```

Info

Retrieve `pcie-address` from the output of the `mst status -v` command.

6. If working with two PF ports, set the DPU into multi-port e-switch mode (for the 2 PCIe devices):

```
sudo devlink dev param set pci/<pcie-address> name
esw_multiport value true cmode runtime
```

Info

Retrieve `pcie-address` from the output of the `mst status -v` command.

7. Define huge pages (see DOCA Flow prerequisites).

ConnectX

To enable DOCA Flow CT on the NVIDIA® ConnectX®, perform the following:

1. Configure firmware with `LAG_RESOURCE_ALLOCATION=1`:

```
sudo mlxconfig -d <device-id> s LAG_RESOURCE_ALLOCATION=1
```

Info

Retrieve `device-id` from the output of the `mst status -v` command. If, under the MST tab, the value is N/A, run the `mst start` command.

2. Perform power cycle.

3. If working with a single port:

```
sudo devlink dev eswitch set pci/<pcie-address> mode
switchdev
sudo devlink dev param set pci/<pcie-address> name
esw_multiport value false cmode runtime
```

> **Info**
>
> Retrieve `pcie-address` from the output of the `mst status -v` command.

4. If working with two PF ports:

```
sudo devlink dev eswitch set pci/<pcie-address0> mode
switchdev
sudo devlink dev eswitch set pci/<pcie-address1> mode
switchdev
sudo devlink dev param set pci/<pcie-address0> name
esw_multiport value true cmode runtime
sudo devlink dev param set pci/<pcie-address1> name
esw_multiport value true cmode runtime
```

> **Info**

Retrieve `pcie-address` from the output of the `mst status -v` command.

5. Define huge pages (see DOCA Flow prerequisites).

Actions

DOCA Flow CT supports actions based on meta and NAT operations. Each action can be defined as either shared or non-shared.

Note

Action descriptors are not supported.

Shared Actions

Actions that can be shared between entries. Shared actions are predefined and reused in multiple entries.

The user gets a handle per shared action created and uses this handle as a reference to the action where required.

Info

It is user responsibility to track shared actions and to remove them when they become irrelevant.

Shared actions are defined using a control queue (see struct doca_flow_ct_cfg).

Non-shared Actions

Actions provided with their data during entry create/update.

These actions are completely managed by DOCA Flow CT and cannot be reused in multiple flows (i.e., NAT operations).

Action Sets in Pipe Creation

When creating a DOCA Flow CT pipe, users must define action sets, just as they would for any other pipe.

Fields in the CT pipe must be marked as `CHANGEABLE` during pipe creation. This allows the actual criteria for these fields to be specified later during entry creation.

Info

Only actions related to meta and NAT, as defined in struct doca_flow_ct_actions , are supported.

During entry creation or update, different actions can be specified for each direction, allowing variations in action content and/or action type.

Feature Enable

To enable user actions, configure the following parameters:

- User action templates during DOCA Flow CT pipe creation
- Maximum number of user actions (`nb_user_actions` on DOCA Flow CT init)

Using Actions in Autonomous Mode

Init

Configure the following parameters on `doca_flow_ct_init()`:

- `nb_ctrl_queues` – number of control queues for defining shared actions
- `nb_user_actions` – maximum number of actions (shared and non-shared)
- `worker_cb` – callbacks required to communicate with the user

Create DOCA Flow CT Pipe

Configure actions sets on `doca_flow_pipe_create()`.

Create Shared Actions

Use `doca_flow_ct_actions_add_shared()` with one of the control queues.

Shared actions can be added at any time before use.

Implement Worker Callbacks

Callbacks are called from each worker thread to acquire synchronization with the user code and on the first packet of a flow.

On `doca_flow_ct_rule_pkt_cb`:

- Determine how the packet should be treated
- If rules are required, return the actions handles to use

Using Actions in Managed Mode

Init

Configure the following parameters on `doca_flow_ct_init()`:

- `nb_ctrl_queues` – number of control queues for defining shared actions
- `nb_user_actions` – maximum number of user actions. Must align to 64. Both shared control queues and non-shared control queues cache action IDs to speed up ID allocation. Each queue may cache a maximum of 1024 IDs. Users must configure the expected number of actions + total queues * 1024. This number cannot exceed the number of actions hardware supports.

Create DOCA Flow CT Pipe

Configure actions sets on `doca_flow_pipe_create()`.

Create Shared Actions

Use `doca_flow_ct_actions_add_shared()` with one of the control queues.

Shared actions can be added at any time before use.

Add Entry

Entry can be created in one of the following ways:

- Using an action handle of a predefined shared action
- Using action data, which is specific to the flow, not sharable (e.g., for NAT operations)

The entry can have different actions and/or different action types per direction.

Remove Entry

Non-shared actions associated with an entry are implicitly destroyed by DOCA Flow CT.

Shared actions are not destroyed. They can be used by the user until they decide to remove them.

Update Entry

Entry actions can be updated per direction. All combinations of shared/non-shared actions are applicable (e.g., update from shared to non-shared).

Changeable Forward

DOCA Flow CT allows using a different forward pipe per flow direction.

DOCA Flow CT supports the forward pipe in two levels:

- Pipe level – a single forward pipe defined during DOCA Flow CT pipe creation and used for all entries
- Entry level – forward pipe defined during entry create
- DOCA Flow CT operates in one of the two levels

DOCA CT forward in entry level has the following characteristics:

- Supports only `DOCA_FLOW_FWD_PIPE` (up to 4 different forward pipes)
- Supports forward pipe per flow direction (both directions can have same/different forward pipe)
- Must set forward pipes on each entry create (no default forward pipe)

Turn on the feature:

1. Create DOCA Flow CT pipe with forward type = `DOCA_FLOW_FWD_PIPE` and `next_pipe` = `NULL` .
2. Call to `doca_flow_ct_fwd_register` to register forward pipes and get `fwd_handles` in return.

Using Changeable Forward in Managed Mode

1. Initialize DOCA Flow CT (`doca_flow_ct_init`).
2. Register forward pipes (`doca_flow_ct_fwd_register`).
    - Define pipes that can be used for forward
3. Create DOCA Flow CT pipe (`doca_flow_pipe_create`) w ith definition of possible forward pipes.
4. Add entry (`doca_flow_ct_add_entry`).
    - Set origin and/or reply `fwd_handles` returned from `doca_flow_ct_fwd_register`.
5. Update forward for entry direction (`doca_flow_ct_update_entry`).

> **Note**
>
> Updating forward handle requires setting all other parameters with their previous values.

Using Changeable Forward in Autonomous Mode

1. Initialize DOCA Flow CT (`doca_flow_ct_init`).
2. Register forward pipes (`doca_flow_ct_fwd_register`).
    - Define pipes that can be used for forward.
3. Create DOCA Flow CT pipe (`doca_flow_pipe_create`) w ith definition of possible forward pipes.
4. CT workers start to handle traffic.

5. On the first flow packet, `doca_flow_ct_rule_pkt` callback is called.
    - In this callback, determine if the entry should be created, and which actions and/or forward handles should be used for this entry.

Info

Update forward for entry direction is not supported.

API

For the library API reference, refer to DOCA Flow and CT API documentation in the DOCA Library APIs.

Note

The pkg-config (`*.pc` file) for the Flow CT library is included in DOCA's regular definitions : `doca`.

The following sections provide additional details about the library API.

enum doca_flow_ct_flags

DOCA Flow CT configuration optional flags.

Flag	Description
`DOCA_FLOW_CT_FLAG_STATS = 1u << 0`	Enable internal pipe counters for packet tracking purposes. Call `doca_flow_pipe_dump(<ct_pipe>)` to dump counter values. Each call dumps values changed.

Flag	Description
`DOCA_FLOW_CT_FLAG_WORKER_STATS = 1u << 1,`	Enable worker thread internal debug counter periodical dump. Autonomous mode only.
`DOCA_FLOW_CT_FLAG_NO_AGING = 1u << 2,`	Disable aging
`DOCA_FLOW_CT_FLAG_SW_PKT_PARSING = 1u << 3,`	Enable CT worker software packet parsing to support VLAN, IPv6 options, or special tunnel types
`DOCA_FLOW_CT_FLAG_MANAGED = 1u << 4,`	Enable managed mode in which user application is responsible for managing packet handling, and calling the CT API to manipulate CT connection entries
`DOCA_FLOW_CT_FLAG_ASYMMETRIC = 1u << 5,`	Allows different 6-tuple table definitions for the origin and reply directions. Default to symmetric mode, uses same meta and reverse 5-tuples for reply direction. Managed mode only.
`DOCA_FLOW_CT_FLAG_ASYMMETRIC_COUNTER = 1u << 6,`	Enable different counters for the origin and reply directions. Managed mode only.
`DOCA_FLOW_CT_FLAG_NO_COUNTER = 1u << 7,`	Disable counter and aging to save aging thread CPU cycles
`DOCA_FLOW_CT_FLAG_DEFAULT_MISS = 1u << 8,`	Check TCP SYN flags and UDP in CT miss flow to identify ADD type packets.

Flag	Description
`DOCA_FLOW_CT_FLAG_WIRE_TO_WIRE = 1u << 9,`	Hint traffic comes from uplink wire and forwards to uplink wire. **Note** If this flag is set, the direction info must be `DOCA_FLOW_DIRECTION_NETWORK_TO_HOST` .
`DOCA_FLOW_CT_FLAG_CALC_TUN_IP_CHKSUM = 1u << 10,`	Enable hardware to calculate and set the checksum on L3 header (IPv4)
`DOCA_FLOW_CT_FLAG_DUP_FILTER_UDP_ONLY= 1u << 11,`	Apply the connection duplication filter for UDP connections only

enum doca_flow_ct doca_flow_ct_entry_flags

DOCA Flow CT Entry optional flags.

Flag	Description
`DOCA_FLOW_CT_ENTRY_FLAGS_NO_WAIT = (1 << 0)`	Entry is not buffered; send to hardware immediately
`DOCA_FLOW_CT_ENTRY_FLAGS_DIR_ORIGIN = (1 << 1)`	Apply flags to origin direction
`DOCA_FLOW_CT_ENTRY_FLAGS_DIR_REPLY = (1 << 2)`	Apply flags to reply direction
`DOCA_FLOW_CT_ENTRY_FLAGS_IPV6_ORIGIN = (1 << 3)`	Origin direction is IPv6; origin match union in struct `doca_flow_ct_match` is IPv6
`DOCA_FLOW_CT_ENTRY_FLAGS_IPV6_REPLY = (1 << 4)`	Reply direction is IPv6; reply match union in struct `doca_flow_ct_match` is IPv6

Flag	Description
`DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_ORIGIN = (1 << 5)`	Apply counter to origin direction
`DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_REPLY = (1 << 6)`	Apply counter to reply direction
`DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_SHARED = (1 << 7)`	Counter is shared for both direction (origin and reply)
`DOCA_FLOW_CT_ENTRY_FLAGS_FLOW_LOG = (1 << 8)`	Enable flow log on entry removed
`DOCA_FLOW_CT_ENTRY_FLAGS_ALLOC_ON_MISS = (1 << 9)`	Allocate on entry not found when calling `doca_flow_ct_entry_prepare()` API
`DOCA_FLOW_CT_ENTRY_FLAGS_DUP_FILTER_ORIGIN = (1 << 10)`	Enable duplication filter on origin direction
`DOCA_FLOW_CT_ENTRY_FLAGS_DUP_FILTER_REPLY = (1 << 11)`	Enable duplication filter on reply direction

enum doca_flow_ct_rule_opr

Options for handling flows in autonomous mode with shared actions. The decision is taken on the first flow packet.

Operation	Description
`DOCA_FLOW_CT_RULE_OK`	Flow should be defined in the CT pipe using the required shared actions handles
`DOCA_FLOW_CT_RULE_DROP`	Flow should not be defined in the CT pipe. The packet should be dropped.
`DOCA_FLOW_CT_RULE_TX_ONLY`	Flow should not be defined in the CT pipe. The packet should be transmitted.

struct direction_cfg

Managed mode configuration for origin or reply direction.

Field	Description
`bool match_inner`	5-tuple match pattern applies to packet inner layer
`struct doca_flow_meta *zone_match_mask`	Mask to indicate meta field and bits to match
`struct doca_flow_meta *meta_modify_mask`	Mask to indicate meta field and bits to modify on connection packet match

struct doca_flow_ct_worker_callbacks

Set of callbacks for using shared actions in autonomous mode.

Field	Description
`doca_flow_ct_sync_acquire_cb worker_init`	Called at the start of a worker thread to sync with the user context
`doca_flow_ct_sync_release_cb worker_release`	Called at the end of a worker thread
`doca_flow_ct_rule_pkt_cb rule_pkt`	Called on the first packet of a flow

struct doca_flow_ct_cfg

DOCA Flow CT configuration.

```
uint32_t nb_arm_queues;
uint32_t nb_ctrl_queues;
uint32_t nb_user_actions;
uint32_t nb_arm_sessions[DOCA_FLOW_CT_SESSION_MAX];
uint32_t flags;
uint16_t aging_core;
```

```
uint16_t aging_query_delay_s;
doca_flow_ct_flow_log_cb flow_log_cb;
struct doca_flow_ct_aging_ops *aging_ops;
uint32_t base_core_id;
uint32_t dup_filter_sz;
union {
                /* Managed mode configuration for origin and reply direction. */
                struct direction_cfg direction[2];

                /* Below fields are dedicate for autonomous mode */
                struct {
                                                uint16_t
tcp_timeout_s;
                                                uint16_t
tcp_session_del_s;
                                                uint16_t
udp_timeout_s;
                        enum doca_flow_tun_type tunnel_type;
                        uint16_t vxlan_dst_port;
                        enum doca_flow_ct_hash_type hash_type;
                        uint32_t meta_user_bits;
                        uint32_t meta_action_bits;
                        struct doca_flow_meta *meta_zone_mask;
                        struct doca_flow_meta
*connection_id_mask;
                        struct doca_flow_ct_worker_callbacks
worker_cb;
               };
       };
```

Where:

Field	Description
`uint32_t nb_arm_queues`	Number of CT queues. In autonomous mode, also the number of worker threads.

Field	Description
`uint32_t nb_ctrl_queues`	Number of CT control queues used for defining shared actions
`uint32_t nb_user_actions`	Maximum number of user actions supported (shared and non-shared) Minimum value is 1K * (`nb_ctrl_queues + nb_arm_queues`)
`uint32_t nb_arm_sessions[DOCA_FLOW_CT_SESSION_MAX]`	Maximum number of IPv4 and IPv6 CT connections
`uint32_t flags`	CT configuration flags
`uint16_t aging_core`	CPU core ID for CT aging thread to bind.
`uint16_t aging_core_delay`	CT aging code delay.
`doca_flow_ct_flow_log_cb flow_log_cb`	Flow log callback function, when set
`struct doca_flow_ct_aging_ops *aging_ops`	User-defined aging logic callback functions. Fallback to default aging logic
`uint32_t base_core_id`	Base core ID for the workers
`uint32_t dup_filter_sz`	Number of connections to cache in the duplication filter
`struct direction_cfg direction`	Managed mode configuration for origin or reply direction
`uint16_t tcp_timeout_s`	TCP timeout in seconds
`uint16_t tcp_session_del_s`	Time to delay or kill TCP session after RST/FIN
`enum doca_flow_tun_type tunnel_type`	Encapsulation tunnel type
`uint16_t vxlan_dst_port`	VXLAN outer UDP destination port in big endian
`enum doca_flow_ct_hash_type hash_type`	Type of connection hash table type: `NONE` or `SYMMETRIC_HASH`
`uint32_t meta_user_bits`	User packet meta bits to be owned by the user

Field	Description
`uint32_t meta_action_bits`	User packet meta bits to be carried by identified connection packet
`struct doca_flow_meta *meta_zone_mask`	Mask to indicate meta field and bits saving zone information
`struct doca_flow_meta *connection_id_mask`	Mask to indicate meta field and bits for CT internal connection ID
`struct doca_flowct_worker_callbacks worker_cb`	Worker callbacks to use shared actions

struct doca_flow_ct_actions

This structure is used in the following cases:

- For defining shared actions. In this case, action data is provided by the user. The action handle is returned by DOCA Flow CT.
- For defining an entry with actions. The structure can be filled with two options:
    - With action handle of a previously created shared action
    - With non-shared action data

DOCA Flow CT action structure.

```
enum doca_flow_resource_type  resource_type;
union {
                        /* Used when creating an entry with a shared action. */
                uint32_t action_handle;

                /* Used when creating an entry with non-shared action or when creating a shared
action. */
                struct {
                        uint32_t action_idx;
                        struct doca_flow_meta meta;
```

```
                                        struct doca_flow_header_l4_port
l4_port;
                                        union {
                                                struct doca_flow_ct_ip4 ip4;
                                                struct doca_flow_ct_ip6 ip6;
                                         };
                        } data;
            };
```

Where:

Field	Description
`enum doca_flow_resource_type resource_type`	Shared/non-shared action
`uint32_t action_handle`	Shared action handle
`uint32_t action_idx`	Actions template index
`struct doca_flow_meta meta`	Modify meta values
`struct doca_flow_header_l4_port l4_port`	UDP or TCP source and destination port
`struct doca_flow_ct_ip4 ip4`	Source and destination IPv4 addresses
`struct doca_flow_ct_ip6 ip6`	Source and destination IPv6 addresses

Info

The value in `meta` , `l4_port` , `ip4` , and `ip6` should start from `bit0`, the least significant bit, regardless of which bits are set in mask. For example, `action_val.meta.u32[0] = DOCA_HTOBE32(0x12)`,

`action_mask.meta.u32[0] = DOCA_HTOBE32(0x0000FF00)`
sets bits 15-8 to `0x12`.

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 (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes 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 and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

© Copyright 2025, NVIDIA. PDF Generated on 05/05/2025