DOCA Ethernet
Table of contents

Introduction

Prerequisites

Changes From Previous Releases

Changes in 2.8.0

Added

Changed

Environment

Architecture

DOCA ETH RXQ

Operating Modes

Working with DOCA Flow

DOCA ETH TXQ

Operating Modes

Offloads

Objects

Configurations Phase

Configurations

Mandatory Configurations

DOCA ETH RXQ

DOCA ETH TXQ

Optional Configurations
Execution Phase

Tasks

Event Batch

State Machine

Idle
Starting
Running
Stopping

Alternative Datapath Options
DOCA ETH Samples

Running the Samples

Samples

ETH TXQ Send Ethernet Frames
ETH TXQ LSO Send Ethernet Frames
ETH TXQ Batch Send Ethernet Frames
ETH TXQ Batch LSO Send Ethernet Frames
ETH RXQ Regular Receive
ETH RXQ Managed Receive
ETH RXQ Batch Managed Receive
This guide provides an overview and configuration instructions for the DOCA ETH API.

**Introduction**

**Note**

The DOCA Ethernet library is supported at alpha level.

DOCA ETH comprises of two APIs, DOCA ETH RXQ and DOCA ETH TXQ. The control path is always handled on the host/DPU CPU side by the library. The datapath can be managed either on the CPU by the DOCA ETH library or on the GPU by the GPUNetIO library.

DOCA ETH RXQ is an RX queue. It defines a queue for receiving packets. It also supports receiving Ethernet packets on any memory mapped by `doca_mmap`.

The memory location to which packets are scattered is agnostic to the processor which manages the datapath (CPU/DPU/GPU). For example, the datapath may be managed on the CPU while packets are scattered to GPU memory.

DOCA ETH TXQ is an TX queue. It defines a queue for sending packets. It also supports sending Ethernet packets from any memory mapped by `doca_mmap`.

To free the CPU from managing the datapath, the user can choose to manage the datapath from the GPU. In this mode of operation, the library collects user configurations and creates a receive/send queue object on the GPU memory (using the DOCA GPU sub-device) and coordinates with the network card (NIC) to interact with the GPU processor.

**Prerequisites**

This library follows the architecture of a DOCA Core Context. It is recommended to read the following sections:

- DOCA Core Execution Model
- DOCA Core Device
Changes From Previous Releases

Changes in 2.8.0

The following subsection(s) detail the doca_eth library updates in version 2.8.0.

Added

- `doca_error_t doca_eth_rxq_set_notification_moderation(struct doca_eth_rxq *eth_rxq, uint16_t period_usec, uint16_t comp_count)`
- `doca_error_t doca_eth_txq_task_send_num_expand(struct doca_eth_txq *eth_txq, uint32_t task_send_num)`
- `doca_error_t doca_eth_txq_task_lso_send_num_expand(struct doca_eth_txq *eth_txq, uint32_t task_lso_send_num)`
- `doca_error_t doca_eth_txq_task_batch_send_num_expand(struct doca_eth_txq *eth_txq, uint16_t task_batches_num)`
- `doca_error_t doca_eth_txq_task_batch_lso_send_num_expand(struct doca_eth_txq *eth_txq, uint16_t task_batches_num)`

Changed

- `doca_eth_rxq_task_recv_allocate_init(struct doca_eth_rxq *eth_rxq, union doca_data user_data, struct doca_buf *pkt, struct doca_eth_rxq_task_recv **task_recv)`
  - `doca_eth_rxq_task_recv_allocate_init(struct doca_eth_rxq *eth_rxq, struct doca_buf *pkt, union doca_data user_data, struct doca_eth_rxq_task_recv **task_recv)`
Environment

DOCA ETH based applications can run either on the Linux host machine or on the NVIDIA® BlueField® DPU target. The following is required:

- Applications should run with root privileges
- To run DOCA ETH on the DPU, applications must supply the library with SFs as a `doca_dev`. See OpenvSwitch Offload and BlueField DPU Scalable Function to see how to create SFs and connect them to the appropriate ports.
- Applications need to use DOCA Flow to forward incoming traffic to DOCA ETH RXQ's queue. See DOCA Flow and DOCA ETH RXQ samples for reference.

📍 **Info**

Make sure the system has free huge pages for DOCA Flow.

Architecture

DOCA ETH is comprised of two parts: DOCA ETH RXQ and DOCA ETH TXQ.

**DOCA ETH RXQ**

Operating Modes

DOCA ETH RXQ can operate in the three modes, each exposing a slightly different control/datapath.

Regular Receive

📍 **Info**
This mode is supported only for CPU datapath.

In this mode, the received packet buffers are managed by the user. To receive a packet, the user should submit a receive task containing a `doca_buf` to write the packet into.

The application uses this mode if it wants to:

- Run on CPU
- Manage the memory of received packet and the packet's exact place in memory
- Forward the received packets to other DOCA libraries

Cyclic Receive
In this mode, the library scatters packets to the packet buffer (supplied by the user as `doca_mmap`) in a cyclic manner. Packets acquired by the user may be overwritten by the library if not processed fast enough by the application.

In this mode, the user must provide DOCA ETH RXQ with a packet buffer to be managed by the library (see `doca_eth_rxq_set_pkt_buf()`). The buffer should be large enough to avoid packet loss (see `doca_eth_rxq_estimate_packet_buf_size()`).

The application uses this mode if:

- It wants to run on GPU
- It has a deterministic packet processing time, where a packet is guaranteed to be processed before the library overwrites it with a new packet
- It wants best performance

Managed Memory Pool Receive

This mode is supported only for GPU datapath.
In this mode, the library uses various optimizations to manage the packet buffers. Packets acquired by the user cannot be overwritten by the library unless explicitly freed by the application. Thus, if the application does not release the packet buffers fast enough, the library would run out of memory and packets would start dropping.

Unlike Cyclic Receive mode, the user can pass the packet to other libraries in DOCA with the guarantee that the packet is not overwritten while being processed by those libraries.

In this mode, the user must provide DOCA ETH RXQ with a packet buffer to be managed by the library (see `doca_eth_rxq_set_pkt_buf()`). The buffer should be large enough to avoid packet loss (see `doca_eth_rxq_estimate_packet_buf_size()`).

The application uses this mode if:

- It wants to run on CPU
- It has a deterministic packet processing time, where a packet is guaranteed to be processed before the library runs out of memory and packets start dropping
- It wants to forward the received packets to other DOCA libraries
- It wants best performance

**Info**

This mode is supported only for CPU datapath.
Working with DOCA Flow

In order to route incoming packets to the desired DOCA ETH RXQ, applications need to use DOCA Flow. Applications need to do the following:

- Create and start DOCA Flow on the appropriate port (device)
- Create pipes to route packets into
- Get the queue ID of the queue (inside DOCA ETH RXQ) using `doca_eth_rxq_get_flow_queue_id()`
- Add an entry to a pipe which routes packets into the RX queue (using the queue ID we obtained)
For more details see **DOCA ETH RXQ samples** and **DOCA Flow**.

**DOCA ETH TXQ**

**Operating Modes**

DOCA ETH TXQ can only operate in one mode.

**Regular Send**

For the CPU datapath, the user should submit a send task containing a `doca_buf` of the packet to send.

For information regarding the datapath on the GPU, see **DOCA GPUNetIO**.
Offloads

DOCA ETH TXQ supports:

- Large Segment Offloading (LSO) – the hardware supports LSO on transmitted TCP packets over IPv4 and IPv6. LSO enables the software to prepare a large TCP message for sending with a header template (the application should provide this header to the library) which is updated automatically for every generated segment. The hardware segments the large TCP message into multiple TCP segments. Per each such segment, device updates the header template accordingly (see LSO Send Task).

- L3/L4 checksum offloading – the hardware supports calculation of checksum on transmitted packets and validation of received packet checksum. Checksum calculation is supported for TCP/UDP running over IPv4 and IPv6. (In case of tunneling, the hardware calculates the checksum of the outer header.) The hardware does not require any pseudo header checksum calculation, and the value placed in TCP/UDP checksum is ignored when performing the calculation. See doca_eth_txq_set_l3_chksum_offload()/doca_eth_txq_set_l4_chksum_offload().
Objects

- **doca_mmap** – in Cyclic Receive and Managed Memory Pool Receive modes, the user must configure DOCA ETH RXQ with packet buffer to write the received packets into as a `doca_mmap` (see DOCA Core Memory Subsystem).

- **doca_buf** – in Regular Receive mode, the user must submit receive tasks that includes a buffer to write the received packet into as a `doca_buf`. Also, In Regular Send mode, the user must submit send tasks that include a buffer of the packet to send as a `doca_buf` (see DOCA Core Memory Subsystem).

Configurations Phase

To start using the library, the user must first go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

**Note**

DOCA ETH in GPU datapath does not need to be associated with a DOCA PE (since the datapath is not on the CPU).

Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or the min/max value for it, refer to Device Support.

Mandatory Configurations
These configurations are mandatory and must be set by the application before attempting to start the context.

**DOCA ETH RXQ**

- At least one task/event/event_batch type must be configured. Refer to Tasks/Events/Event Batch for more information.

- Max packet size (the maximum size of packet that can be received) must be provided at creation time of the DOCA ETH RXQ context.

- Max burst size (the maximum number of packets that the library can handle at the same time) must be provided at creation time of the DOCA ETH RXQ context.

- A device with appropriate support must be provided upon creation.

- When in Cyclic Receive or Managed Memory Pool Receive modes, a `doca_mmap` must be provided in-order write the received packets into (see `doca_eth_rxq_set_pkt_buf()`).

- In case of a GPU datapath, a DOCA GPU sub-device must be provided using `doca_ctx_set_datapath_on_gpu()`.

**DOCA ETH TXQ**

- At least one task/task_batch type must be configured. Refer to Tasks/Task Batch for more information.

- Max burst size (the maximum number of packets that the library can handle at the same time) must be provided at creation time of the DOCA ETH TXQ context.

- A device with appropriate support must be provided on creation.

- In case of a GPU datapath, a DOCA GPU sub-device must be provided using `doca_ctx_set_datapath_on_gpu()`.

**Optional Configurations**
The following configurations are optional. If they are not set, then a default value is used.

**DOCA ETH RXQ**

- RXQ mode – User can set the working mode using \texttt{doca\_eth\_rxq\_set\_type()}. The default type is Regular Receive.

- Max receive buffer list length – User can set the maximum length of buffer list/chain as a receive buffer using \texttt{doca\_eth\_rxq\_set\_max\_recv\_buf\_list\_len()}. The default value is 1.

**DOCA ETH TXQ**

- TXQ mode – User can set the working mode using \texttt{doca\_eth\_txq\_set\_type()}. The default type is Regular Send.

- Max send buffer list length – User can set the maximum length of buffer list/chain as a send buffer using \texttt{doca\_eth\_txq\_set\_max\_send\_buf\_list\_len()}. The default value is 1.

- L3/L4 offload checksum – User can enable/disable L3/L4 checksum offloading using \texttt{doca\_eth\_txq\_set\_l3\_chksum\_offload()}/\texttt{doca\_eth\_txq\_set\_l4\_chksum\_offload()}. They are disabled by default.

- MSS – User can set MSS (maximum segment size) value for LSO send task/task\_batch using \texttt{doca\_eth\_txq\_set\_mss()}. The default value is 1500.

- Max LSO headers size – User can set the maximum LSO headers size for LSO send task/task\_batch using \texttt{doca\_eth\_txq\_set\_max\_lso\_header\_size()}. The default value is 74.

**Device Support**

DOCA ETH requires a device to operate. For picking a device, see DOCA Core Device Discovery.

To check if a device supports a specific mode, use the type capabilities functions (see \texttt{doca\_eth\_rxq\_cap\_is\_type\_supported()} and \texttt{doca\_eth\_txq\_cap\_is\_type\_supported()}).
Devices can allow the following capabilities:

- The maximum burst size
- The maximum buffer chain list (only for Regular Receive/Regular Send modes)
- The maximum packet size (only for DOCA ETH RXQ)
- L3/L4 checksum offloading capability (only for DOCA ETH TXQ)
- Maximum LSO message/header size (only for DOCA ETH TXQ)
- Wait-on-time offloading capability (only for DOCA ETH TXQ in GPU datapath)

**Buffer Support**

DOCA ETH support buffers (doca_mmap or doca_buf) with the following features:

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Send Task</th>
<th>LSO Send Task</th>
<th>Receive Task</th>
<th>Managed Receive Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local mmap buffer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from PCIe export buffer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from RDMA export buffer</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Linked list buffer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

For buffer support in the case of GPU datapath, see [DOCA GPUNetIO Programming Guide](#).

**Execution Phase**

This section describes execution on CPU (unless stated otherwise) using [DOCA Core Progress Engine](#).
DOCA Ethernet 17

17

DOCA Ethernet

Tasks

DOCA ETH exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

DOCA ETH RXQ

Receive Task

This task allows receiving packets from a doca_dev.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>calling doca_eth_rxq_task_recv_set_conf()</td>
<td>doca_eth_rxq_cap_is_type_supported() checking support for Regular Receive mode</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>task_recv_num in doca_eth_rxq_task_recv_set_conf()</td>
<td>-</td>
</tr>
<tr>
<td>Max receive buffer list length</td>
<td>doca_eth_rxq_set_max_recv_buf_list_len() (default value is 1)</td>
<td>doca_eth_rxq_cap_get_max_recv_buf_list_len()</td>
</tr>
<tr>
<td>Maximal packet size</td>
<td>max_packet_size in doca_eth_rxq_create()</td>
<td>doca_eth_rxq_cap_get_max_packet_size()</td>
</tr>
</tbody>
</table>

Task Input

Common input as described in DOCA Core Task.

Note

For information regarding GPU datapath, see DOCA GPUNetIO.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet buffer</td>
<td>Buffer pointing to the memory where received packet are to be written</td>
<td>The received packet is written to the tail segment extending the data segment</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in [DOCA Core Task](#).

Additionally:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3 checksum result</td>
<td>Value indicating whether the L3 checksum of the received packet is valid or not</td>
<td>Can be queried using <code>doca_eth_rxq_task_recv_get_l3_ok()</code></td>
</tr>
<tr>
<td>L4 checksum result</td>
<td>Value indicating whether the L4 checksum of the received packet is valid or not</td>
<td>Can be queried using <code>doca_eth_rxq_task_recv_get_l4_ok()</code></td>
</tr>
</tbody>
</table>

**Task Completion Success**

After the task is completed successfully the following will happen:

- The received packet is written to the packet buffer
- The packet buffer data segment is extended to include the received packet

**Task Completion Failure**

If the task fails midway:

- The context enters stopping state
• The packet buffer `doca_buf` object is not modified

• The packet buffer contents may be modified

**Task Limitations**

All limitations described in **DOCA Core Task**

Additionally:

• The operation is not atomic.

• Once the task has been submitted, then the packet buffer should not be read/written to.

**DOCA ETH TXQ**

**Send Task**

This task allows sending packets from a `doca_dev`.

**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>calling <code>doca_eth_txq_task_send_set_conf()</code></td>
<td><code>doca_eth_txq_cap_is_type_supported()</code> checking support for Regular Send mode</td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>task_send_num</code> in <code>doca_eth_txq_task_send_set_conf()</code></td>
<td><code>--</code></td>
</tr>
<tr>
<td>Max send buffer list length</td>
<td><code>doca_eth_txq_set_max_send_buf_list_len()</code> (default value is 1)</td>
<td><code>doca_eth_txq_cap_get_max_send_buf_list_len()</code></td>
</tr>
</tbody>
</table>
### Task Input

Common input as described in [DOCA Core Task](#).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet buffer</td>
<td>Buffer pointing to the packet to send</td>
<td>The sent packet is the memory in the data segment</td>
</tr>
</tbody>
</table>

### Task Output

Common output as described in [DOCA Core Task](#).

### Task Completion Success

The task finishing successfully does not guarantee that the packet has been transmitted onto the wire. It only signifies that the packet has successfully entered the device’s TX hardware and that the packet buffer `doca_buf` is no longer in the library's ownership and it can be reused by the application.

### Task Completion Failure

If the task fails midway:

- The context enters stopping state
- The packet buffer `doca_buf` object is not modified

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3/L4 offload checksum</td>
<td><code>doca_eth_txq_set_l3_chksum_offload()</code></td>
<td><code>doca_eth_txq_cap_is_l3_chksum_offload_supported()</code></td>
</tr>
<tr>
<td></td>
<td><code>doca_eth_txq_set_l4_chksum_offload()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disabled by default.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>doca_eth_txq_cap_is_l4_chksum_offload_supported()</code></td>
<td></td>
</tr>
</tbody>
</table>
Task Limitations

- The operation is not atomic
- Once the task has been submitted, the packet buffer should not be written to
- Other limitations are described in DOCA Core Task

LSO Send Task

This task allows sending "large" packets (larger than MTU) from a doca_dev (hardware splits the packet into several packets smaller than the MTU and sends them).

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>calling</td>
<td>doca_eth_txq_cap_is_type_supported()</td>
</tr>
<tr>
<td></td>
<td>doca_eth_txq_task_lso_send_set_conf()</td>
<td>checking support for Regular Send mode</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>task_lso_send_num in</td>
<td></td>
</tr>
<tr>
<td></td>
<td>doca_eth_txq_task_lso_send_set_conf()</td>
<td></td>
</tr>
<tr>
<td>Max send buffer list length</td>
<td>doca_eth_txq_set_max_send_buf_list_len() (default value is 1)</td>
<td>doca_eth_txq_cap_get_max_send_buf_list_len()</td>
</tr>
<tr>
<td>L3/L4 offload checksum</td>
<td>doca_eth_txq_set_l3_chksum_offload()</td>
<td>doca_eth_txq_cap_is_l3_chksum_offload_supported()</td>
</tr>
<tr>
<td></td>
<td>doca_eth_txq_set_l4_chksum_offload()</td>
<td>doca_eth_txq_cap_is_l4_chksum_offload_supported()</td>
</tr>
<tr>
<td></td>
<td>(disabled by default)</td>
<td></td>
</tr>
<tr>
<td>MSS</td>
<td>doca_eth_txq_set_mss() (default value is 1500)</td>
<td>-</td>
</tr>
<tr>
<td>Max LSO headers size</td>
<td>doca_eth_txq_set_max_lso_header_size() (default value is 74)</td>
<td>doca_eth_txq_cap_get_max_lso_header_size()</td>
</tr>
</tbody>
</table>
Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet payload buffer</td>
<td>Buffer that points to the &quot;large&quot; packet's payload (does not include headers) to send</td>
<td>The sent packet is the memory in the data segment</td>
</tr>
<tr>
<td>Packet headers buffer</td>
<td>Gather list that when combined includes the &quot;large&quot; packet's headers to send</td>
<td>See struct doca_gather_list</td>
</tr>
</tbody>
</table>

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

The task finishing successfully does not guarantee that the packet has been transmitted onto the wire. It only means that the packet has successfully entered the device's TX hardware and that the packet payload buffer and the packet headers buffer is no longer in the library's ownership and it can be reused by the application.

Task Completion Failure

If the task fails midway:

- The context enters stopping state
- The packet payload buffer `doca_buf` object and the packet header buffer `doca_gather_list` are not modified
**Task Limitations**

- The operation is not atomic
- Once the task has been submitted, the packet payload buffer and the packet headers buffer should not be written to
- Other limitations are described in DOCA Core Task

**Events**

DOCA ETH exposes asynchronous events to notify about changes that happen asynchronously, according to the DOCA Core architecture. See DOCA Core Event.

In addition to common events as described in DOCA Core Event, DOCA ETH exposes an extra events:

**DOCA ETH RXQ**

**Managed Receive Event**

This event allows receiving packets from a doca_dev (without requiring the application to manage the memory the packets are written to).

**Event Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event</td>
<td>doca_eth_rxq_event_managed_recv_register()</td>
<td>doca_eth_rxq_cap_is_type_supported() checking support for Managed Memory Pool Receive mode</td>
</tr>
</tbody>
</table>
**Event Trigger Condition**

The event is triggered every time a packet is received.

---

**Event Success Handler**

The success callback (provided in the event registration) is invoked and the user is expected to perform the following:

- Use the `pkt` parameter to process the received packet
- Use `event_user_data` to get the application context
- Query L3/L4 checksum results of the packet
- Free the `pkt` (a `doca_buf` object) and return it to the library

---

**Note**

Not freeing the `pkt` may cause scenario where packets are lost.

---

**Event Failure Handler**

The failure callback (provided in the event registration) is invoked, and the following happens:

- The context enters stopping state
- The `pkt` parameter becomes NULL
- The `event_user_data` parameter contains the value provided by the application when registering the event
DOCA ETH TXQ

Error Send Packet

This event is relevant when running DOCA ETH on GPU datapath (see DOCA GPUNetIO). It allows detecting failure in sending packets.

Event Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event</td>
<td>doca_eth_txq_gpu_event_error_send_packet_register()</td>
<td>Always supported</td>
</tr>
</tbody>
</table>

Event Trigger Condition

The event is triggered when sending a packet fails.

Event Handler

The callback (provided in the event registration) is invoked and the user can:

- Get the position (index) of the packet that TXQ failed to send

Notify Send Packet

This event is relevant when running DOCA ETH on GPU datapath (see DOCA GPUNetIO). It notifies user every time a packet is sent successfully.
### Event Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event</td>
<td><code>doca_eth_txq_gpu_event_notify_send_packet_register()</code></td>
<td>Always supported</td>
</tr>
</tbody>
</table>

### Event Trigger Condition

The event is triggered when sending a packet fails.

### Event Handler

The callback (provided in the event registration) is invoked and the user can:

- Get the position (index) of the packet was sent
- Timestamp of sending the packet

### Task Batch

DOCA ETH exposes asynchronous task batches that leverage the BlueField Platform hardware according to the DOCA Core architecture.

**DOCA ETH RXQ**

There are no task batches in ETH RXQ at the moment.

**DOCA ETH TXQ**
Send Task Batch

This is an extended task batch for Send Task which allows batched sending of packets from a doca_dev.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task batch</td>
<td>calling doca_eth_txq_task_batch_send_set_conf()</td>
<td>doca_eth_txq_cap_is_type_supported() checking support for Regular Send mode</td>
</tr>
<tr>
<td>Number of task batches</td>
<td>num_task_batches in doca_eth_txq_task_batch_send_set_conf()</td>
<td>-</td>
</tr>
<tr>
<td>Max number of tasks per task batch</td>
<td>max_tasks_number in doca_eth_txq_task_batch_send_set_conf()</td>
<td>-</td>
</tr>
<tr>
<td>Max send buffer list length</td>
<td>doca_eth_txq_set_max_send_buf_list_len() (default value is 1)</td>
<td>doca_eth_txq_cap_get_max_send_buf_list_len()</td>
</tr>
<tr>
<td>L3/L4 offload checksum</td>
<td>doca_eth_txq_set_l3_chksum_offload() doca_eth_txq_set_l4_chksum_offload() Disabled by default.</td>
<td>doca_eth_txq_cap_is_l3_chksum_offload_supported() doca_eth_txq_cap_is_l4_chksum_offload_supported()</td>
</tr>
</tbody>
</table>

Task Input

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks number</td>
<td>Number of send tasks &quot;behind&quot; the task batch</td>
<td>This number equals the number of packets to send</td>
</tr>
<tr>
<td>Batch user data</td>
<td>User data associated for the task batch</td>
<td>-</td>
</tr>
<tr>
<td>Packets array</td>
<td>Pointer to an array of buffers pointing at the packets to send per task</td>
<td>The sent packet is the memory in the data segment of each buffer</td>
</tr>
</tbody>
</table>
### Task Output

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>User data array</td>
<td>Pointer to an array of user data per task</td>
<td>–</td>
</tr>
</tbody>
</table>

### Task Completion Success

A task batch is complete if all the send tasks finished successfully and all the packets entered the device's TX hardware. All packets in the "Packet array" are now in the ownership of the user.

### Task Completion Failure

If a task batch fails, then one (or more) of the tasks associated with the task batch failed. The user can look at "Status array" to see which task/packet caused the failure.

Also, the following behavior is expected:

- The context enters stopping state
- The packet's `doca_buf` objects are not modified

### Task Limitations

In addition to all the Send Task Limitations:

- Task batch completion occurs only when all the tasks are completed (no partial completion)
LSO Send Task Batch

This is an extended task batch for LSO Send Task which allows batched sending of LSO packets from a doca_dev.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task batch</td>
<td>Calling doca_eth_txq_task_batch_lso_send_set_conf()</td>
<td>doca_eth_txq_cap_is_type_supported() checking support for Regular Send mode</td>
</tr>
<tr>
<td>Number of task batches</td>
<td>num_task_batches in doca_eth_txq_task_batch_lso_send_set_conf()</td>
<td>-</td>
</tr>
<tr>
<td>Max number of tasks per task batch</td>
<td>num_task_batches in doca_eth_txq_task_batch_lso_send_set_conf()</td>
<td>-</td>
</tr>
<tr>
<td>Max send buffer list length</td>
<td>doca_eth_txq_set_max_send_buf_list_len() (default value is 1)</td>
<td>doca_eth_txq_cap_get_max_send_buf_list_len()</td>
</tr>
<tr>
<td>L3/L4 offload checksum</td>
<td>doca_eth_txq_set_l3_chksum_offload() doca_eth_txq_set_l4_chksum_offload()</td>
<td>doca_eth_txq_cap_is_l3_chksum_offload_supported() doca_eth_txq_cap_is_l4_chksum_offload_supported()</td>
</tr>
<tr>
<td></td>
<td>Disabled by default.</td>
<td></td>
</tr>
<tr>
<td>MSS</td>
<td>doca_eth_txq_set_mss() (default value is 1500)</td>
<td>-</td>
</tr>
<tr>
<td>Max LSO headers size</td>
<td>doca_eth_txq_set_max_lso_header_size() (default value is 74)</td>
<td>doca_eth_txq_cap_get_max_lso_header_size()</td>
</tr>
</tbody>
</table>

Task Input
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks number</td>
<td>Number of send tasks &quot;behind&quot; the task batch</td>
<td>This number equals the number of packets to send</td>
</tr>
<tr>
<td>Batch user data</td>
<td>User data associated for the task batch</td>
<td>-</td>
</tr>
<tr>
<td>Packets payload array</td>
<td>Pointer to an array of buffers pointing at the &quot;large&quot; packet's payload to send per task</td>
<td>The sent packet payload is the memory in the data segment of each buffer</td>
</tr>
<tr>
<td>Packets headers array</td>
<td>Pointer to an array of gather lists, each of which when combined assembles a &quot;large&quot; packet's headers to send per task</td>
<td>See struct doca_gather_list</td>
</tr>
<tr>
<td>User data array</td>
<td>Pointer to an array of user data per task</td>
<td>-</td>
</tr>
</tbody>
</table>

**Task Output**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status array</td>
<td>Pointer to an array of status per task of the finished task batch</td>
</tr>
</tbody>
</table>

**Task Completion Success**

A task batch is complete if all the LSO send tasks finished successfully and all the packets entered the device's TX hardware. All packet payload in "Packets payload array" and packet headers in "Packets headers array" are now in the ownership of the user.

**Task Completion Failure**

If a task batch fails, then one (or more) of the tasks associated with the task batch failed, and the user can look at the "Status array" to try and figure out which task/packet caused the failure.
Also, the following behavior is expected:

- The context enters stopping state
- The packets payload `doca_buf` objects are not modified
- The packets headers `doca_gather_list` objects are not modified

**Task Limitations**

In addition to all the LSO Send Task Limitations:

- Task batch completion happens only when all the tasks are completed (no partial completion)

**Event Batch**

DOCA ETH exposes asynchronous event batches to notify about changes that happen asynchronously.

**DOCA ETH RXQ**

**Managed Receive Event Batch**

This is an extended event batch for Managed Receive Event which allows receiving packets from a `doca_dev` (without requiring the application to manage the memory the packets are written to).

**Event Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to the event batch</td>
<td>Calling <code>doca_eth_rxq_event_batch</code></td>
<td><code>doca_eth_rxq_cap_is_type_supported</code> checking support for</td>
</tr>
<tr>
<td>Description</td>
<td>API to Set the Configuration</td>
<td>API to Query Support</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Max events number: Equal to the maximum number of completed events per event batch completion</td>
<td>h_managed_recv_register()</td>
<td>Managed Memory Pool Receive mode</td>
</tr>
<tr>
<td></td>
<td>events_number_max in doca_eth_rxq_event_batch()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h_managed_recv_register()</td>
<td></td>
</tr>
<tr>
<td>Min events number: Equal to the minimum number of completed events per event batch completion</td>
<td>events_number_min in doca_eth_rxq_event_batch()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h_managed_recv_register()</td>
<td></td>
</tr>
</tbody>
</table>

**Event Trigger Condition**

The event batch is triggered every time a number of packets (number between "Min events number" and "Max events number") are received.

**Event Batch Success Handler**

The success callback (provided in the event of batch registration) is invoked and the user is expected to perform the following:

1. Identify the number of received packets by `events_number`.
2. Use the `pkt_array` parameter to process the received packets.
3. Use `event_batch_user_data` to get the application context.
4. Query the L3/L4 checksum results of the packets using `l3_ok_array` and `l4_ok_array`.
5. Free the buffers from `pkt_array` (a `doca_buf` object) and return it to the library. This can be done in two ways:
   - Iterating over the buffers in `pkt_array` and freeing them using `doca_buf_dec_refcount()`.
• Freeing all the buffers in pkt_array together (gives better performance) using doca_eth_rxq_event_batch_managed_recv_pkt_array_free().

**Event Batch Failure Handler**

The failure callback (provided in the event batch registration) is invoked, and the following happens:

- The context enters stopping state
- The pkt_array parameter is NULL
- The l3_ok_array parameter is NULL
- The l4_ok_array parameter is NULL
- The event_batch_user_data parameter contains the value provided by the application when registering the event

**DOCA ETH TXQ**

There are no event batches in ETH TXQ at the moment.

**State Machine**

The DOCA ETH library follows the Context state machine as described in [DOCA Core Context State Machine](#).

The following section describes how to move to the state and what is allowed in each state.

**Idle**
In this state it is expected that application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Creating the context</td>
</tr>
<tr>
<td>Running</td>
<td>Calling stop after:</td>
</tr>
<tr>
<td></td>
<td>• All tasks are completed and freed</td>
</tr>
<tr>
<td></td>
<td>• All doca_buf objects returned by Managed Receive Event callback are freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Calling progress until:</td>
</tr>
<tr>
<td></td>
<td>• All tasks are completed and freed</td>
</tr>
<tr>
<td></td>
<td>• All doca_buf objects returned by Managed Receive Event callback are freed</td>
</tr>
</tbody>
</table>

**Starting**

This state cannot be reached.

**Running**
In this state it is expected that application will do the following:

- Allocate and submit tasks
- Call progress to complete tasks and/or receive events

Allowed operations:

- Allocate previously configured task
- Submit a task
- Call `doca_eth_rxq_get_flow_queue_id()` to connect the RX queue to DOCA Flow
- Call stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

**Stopping**

In this state, it is expected that application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks
- Frees `doca_buf` objects returned by Managed Receive Event callback

Allowed operations:

- Call progress

It is possible to reach this state as follows:
## Alternative Datapath Options

In addition to the CPU datapath (mentioned in Execution Phase), DOCA ETH supports running on GPU datapath. This allows applications to release the CPU from datapath management and allow low latency GPU processing of network traffic.

To export the handles, the application should call `doca_ctx_set_datapath_on_gpu()` before `doca_ctx_start()` to program the library to set up a GPU operated context.

To get the GPU context handle, the user should call `doca_rxq_get_gpu_handle()` which returns a pointer to a handle in the GPU memory space.

### Note

The datapath cannot be managed concurrently for the GPU and the CPU.

The DOCA ETH context is configured on the CPU and then exported to the GPU:
The following example shows the expected flow for a GPU-managed datapath with packets being scattered to GPU memory (for `doca_eth_rxq`):

1. Create a DOCA GPU device handler.
2. Create `doca_eth_rxq` and configure its parameters.
3. Set the datapath of the context to GPU.
4. Start the context.
5. Get a GPU handle of the context.
For more information regarding the GPU datapath see DOCA GPUNetIO.

**DOCA ETH Samples**

This section describes DOCA ETH samples based on the DOCA ETH library.

The samples illustrate how to use the DOCA ETH API to do the following:

- Send "regular" packets (smaller than MTU) using DOCA ETH TXQ
- Send "large" packets (larger than MTU) using DOCA ETH TXQ
- Receive packets using DOCA ETH RXQ in Regular Receive mode
- Receive packets using DOCA ETH RXQ in Managed Memory Pool Receive mode
Running the Samples

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample (e.g., eth_txq_send_ethernet_frames):

   ```bash
   cd /opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary `eth_txq_send_ethernet_frames` is created under `/tmp/build/`.

3. Sample (e.g., eth_txq_send_ethernet_frames) usage:

   Usage: doca_eth_txq_send_ethernet_frames [DOCA Flags] [Program Flags]

   DOCA Flags:
   - `h`, `--help`         Print a help synopsis
   - `v`, `--version`     Print program version information
   - `l`, `--log-level`   Set the (numeric) log level for the program
   <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>

   --sdk-log-level        Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>

   -j, --json <path>      Parse all command flags from an input json file

All of the DOCA samples described in this section are governed under the BSD-3 software license agreement.
4. For additional information per sample, use the `-h` option:

```
/tmp/build/<sample_name> -h
```

## Samples

**Note**

The following samples are for the CPU datapath. For GPU datapath samples, see [DOCA GPUNetIO](#).

### ETH TXQ Send Ethernet Frames

This sample illustrates how to send a "regular" packet (smaller than MTU) using DOCA ETH TXQ.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with one buffer to the packet's data.
4. Writing the packet's content into the allocated buffer.
5. Allocating elements from DOCA buffer inventory for the buffer.
6. Initializing and configuring DOCA ETH TXQ context.

7. Starting the DOCA ETH TXQ context.

8. Allocating DOCA ETH TXQ send task.

9.Submitting DOCA ETH TXQ send task into progress engine.

10. Retrieving DOCA ETH TXQ send task from the progress engine.

11. Handling the completed task using the provided callback.

12. Stopping the DOCA ETH TXQ context.

13. Destroying DOCA ETH TXQ context.

14. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/eth_txq_send_ethernet_frames_s
- /opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/eth_txq_send_ethernet_frames_r
- /opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/meson.build

**ETH TXQ LSO Send Ethernet Frames**

This sample illustrates how to send a "large" packet (larger than MTU) using DOCA ETH TXQ.

The sample logic includes:

1. Locating DOCA device.

2. Initializing the required DOCA Core structures.

3. Populating DOCA memory map with one buffer to the packet's payload.

4. Writing the packet's payload into the allocated buffer.
5. Allocating elements from DOCA Buffer inventory for the buffer.

6. Allocating DOCA gather list consisting of one node to the packet's headers.

7. Writing the packet's headers into the allocated gather list node.

8. Initializing and configuring DOCA ETH TXQ context.

9. Starting the DOCA ETH TXQ context.

10. Allocating DOCA ETH TXQ LSO send task.

11. Submitting DOCA ETH TXQ LSO send task into progress engine.

12. Retrieving DOCA ETH TXQ LSO send task from the progress engine.

13. Handling the completed task using the provided callback.

14. Stopping the DOCA ETH TXQ context.

15. Destroying DOCA ETH TXQ context.

16. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/eth_txq_lso_send_ethernet_frames
- /opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/eth_txq_lso_send_ethernet_frames
- /opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/meson.build

**ETH TXQ Batch Send Ethernet Frames**

This sample illustrates how to send a batch of "regular" packets (smaller than MTU) using DOCA ETH TXQ.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.

3. Populating DOCA memory map with multiple buffers, each representing a packet's data.

4. Writing the packets' content into the allocated buffers.

5. Allocating elements from DOCA Buffer inventory for the buffers.

6. Initializing and configuring DOCA ETH TXQ context.

7. Starting the DOCA ETH TXQ context.

8. Allocating DOCA ETH TXQ send task batch.

9. Copying all buffers' pointers to task batch's `pkt_arr`.

10. Submitting DOCA ETH TXQ send task batch into the progress engine.

11. Retrieving DOCA ETH TXQ send task batch from the progress engine.

12. Handling the completed task batch using the provided callback.

13. Stopping the DOCA ETH TXQ context.


15. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/eth_txq_batch_send_ether
- /opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/eth_txq_batch_send_ether
- /opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/meson.build

**ETH TXQ Batch LSO Send Ethernet Frames**

This sample illustrates how to send a batch of "large" packets (larger than MTU) using DOCA ETH TXQ.
The sample logic includes:

1. Locating DOCA device.

2. Initializing the required DOCA Core structures.

3. Populating DOCA memory map with multiple buffers, each representing a packet's payload.

4. Writing the packets' payload into the allocated buffers.

5. Allocating elements from DOCA Buffer inventory for the buffers.

6. Allocating DOCA gather lists each consisting of one node for the packet's headers.

7. Writing the packets' headers into the allocated gather list nodes.

8. Initializing and configuring DOCA ETH TXQ context.

9. Starting the DOCA ETH TXQ context.

10. Allocating DOCA ETH TXQ LSO send task.

11. Copying all buffers' pointers to task batch's pkt_payload_arr.

12. Copying all gather lists' pointers to task batch's headers_arr.

13. Submitting DOCA ETH TXQ LSO send task batch into the progress engine.

14. Retrieving DOCA ETH TXQ LSO send task batch from the progress engine.

15. Handling the completed task batch using the provided callback.

16. Stopping the DOCA ETH TXQ context.

17. Destroying DOCA ETH TXQ context.

18. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_txq_batch_lso_send_ethernet_frames/eth_txq_batch_lso_sen
ETX RXQ Regular Receive

This sample illustrates how to receive a packet using DOCA ETH RXQ in Regular Receive mode.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with one buffer to the packet's data.
4. Allocating element from DOCA Buffer inventory for each buffer.
5. Initializing DOCA Flow.
6. Initializing and configuring DOCA ETH RXQ context.
7. Starting the DOCA ETH RXQ context.
8. Starting DOCA Flow.
9. Creating a pipe connecting to DOCA ETH RXQ's RX queue.
10. Allocating DOCA ETH RXQ receive task.
11. Submitting DOCA ETH RXQ receive task into the progress engine.
12. Retrieving DOCA ETH RXQ receive task from the progress engine.
13. Handling the completed task using the provided callback.
15. Stopping the DOCA ETH RXQ context.
16. Destroying DOCA ETH RXQ context.

17. Destroying DOCA Flow.

18. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/eth_rxq_regular_receive_sample.c
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/eth_rxq_regular_receive_main.c
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/meson.build

**ETH RXQ Managed Receive**

This sample illustrates how to receive packets using DOCA ETH RXQ in Managed Memory Pool Receive mode.

The sample logic includes:

1. Locating DOCA device.

2. Initializing the required DOCA Core structures.

3. Calculating the required size of the buffer to receive the packets from DOCA ETH RXQ.

4. Populating DOCA memory map with a packets buffer.

5. Initializing DOCA Flow.

6. Initializing and configuring DOCA ETH RXQ context.

7. Registering DOCA ETH RXQ managed receive event.

8. Starting the DOCA ETH RXQ context.


10. Creating a pipe connecting to DOCA ETH RXQ's RX queue.
11. Retrieving DOCA ETH RXQ managed receive events from the progress engine.

12. Handling the completed events using the provided callback.


14. Stopping the DOCA ETH RXQ context.

15. Destroying DOCA ETH RXQ context.


17. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/eth_rxq_managed_mempool
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/eth_rxqmanaged_mempool
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/meson.build

**ETH RXQ Batch Managed Receive**

This sample illustrates how to receive batches of packets using DOCA ETH RXQ in Managed Memory Pool Receive mode.

The sample logic includes:

1. Locating DOCA device.

2. Initializing the required DOCA Core structures.

3. Calculating the required size of the buffer to receive the packets from DOCA ETH RXQ.

4. Populating DOCA memory map with a packets buffer.

5. Initializing DOCA Flow.

6. Initializing and configuring DOCA ETH RXQ context.
7. Registering DOCA ETH RXQ managed receive event batch.

8. Starting the DOCA ETH RXQ context.


10. Creating a pipe connecting to DOCA ETH RXQ's RX queue.

11. Retrieving DOCA ETH RXQ managed receive event batches from the progress engine.

12. Handling the completed event batches using the provided callback.


14. Stopping the DOCA ETH RXQ context.

15. Destroying DOCA ETH RXQ context.


17. Destroying all DOCA Core structures.

Reference:

- /opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/eth_rxq_batch_managed_mempool_receive
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/eth_rxq_batch_managed_mempool_receive
- /opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/meson.build