DOCA Comch – New
Table of contents

Introduction

Prerequisites

Changes From Previous Release

Modified

Function name and return type changes

Adding const to getter API functions

Environment

Architecture

Objects

Security Considerations

Initialization Flow

doca_comch_server Initialization Flow

doca_comch_client Initialization Flow

doca_comch_consumer Initialization Flow

Teardown Flow

Disconnecting Specific Connection

Tearing Down DOCA Comch

MsgQ (DPA Communication)

Communication Direction

Configuration Phase

Configurations
Mandatory Configurations

Optional Configurations

Device Support

Buffer Support

Execution Phase

Tasks

Control Channel Send Task

Consumer Post Receive Task

Producer Send Task

Events

Control Channel Receive Event

Connection Status Changed Event (Server Only)

Consumer Event

State Machine

Idle

Starting

Running

Stopping

Alternative Datapath Options

DPA

Consumer Ack

Producer Post Send Immediate Only

Producer DMA Copy
DOCA Comch Samples

Running the Samples

Samples

DOCA Comch Control Path Client/Server

DOCA Comch Data Path Client/Server
This guide provides instructions on building and developing applications that require communication channels between the x86 host and the BlueField Arm cores.

**Introduction**

DOCA Comch provides a communication channel between client applications on the host and servers on the BlueField Arm.

Benefits of using DOCA Comch:

- Security – the communication channel is isolated from the network
- Network independent – the state of the communication channel does not depend on the state and configuration of the network
- Ease of use

DOCA Comch provides two different data path APIs:

- Basic DOCA Comch send/receive for control messages
- High bandwidth, low latency, zero-copy, multi-producer, multi-consumer API

The following table summarizes the differences between the two data path APIs:

<table>
<thead>
<tr>
<th>Features</th>
<th>Basic Send/Receive</th>
<th>Fast Path (using doca_comch_consumer/doca_comch_producer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero-copy</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Takes network bandwidth</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Isolated from network</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Max msg size</td>
<td>Fixed</td>
<td>1GB or more (depends on hardware cap)</td>
</tr>
<tr>
<td>Multi-threaded</td>
<td>Safe for a single thread</td>
<td>Allows creation of consumer/ producers per thread.</td>
</tr>
<tr>
<td>Features</td>
<td>Basic Send/Receive</td>
<td>Fast Path (using doca_comch_consumer/doca_comch_producer)</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>Multi-consumer</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Multi-producer</td>
<td>Yes – allows multiple clients per server</td>
<td>Yes – allow multiple producers/consumers per connection</td>
</tr>
<tr>
<td>Requires doca_mmap and doca_buf</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Prerequisites**

This library follows the architecture of a DOCA Core Context, it is recommended to read the following sections before:

- [DOCA Core Execution Model](#)
- [DOCA Core Device](#)
- [DOCA Core Memory Subsystem](#) (fast path only)

**Changes From Previous Release**

**Modified**

**Function name and return type changes**

- doca_error_t doca_comch_server_get_device_rep(const struct doca_comch_server *comch_server, struct doca_dev_rep **rep)
  - DOCA 2.7 version:
    - doca_error_t doca_comch_server_get_device_repr(const struct doca_comch_server *comch_server, struct doca_dev_rep **repr)
  - doca_comch_server_event_connection_status_changed_register(server, server_connection_status_callback, server_connection_status_callback)
  - DOCA 2.7 version:
- `doca_comch_server_event_connection_register(server, server_connection_status_callback,  
  server_connection_status_callback)`

- `doca_error_t doca_comch_consumer_set_dev_max_num_recv(struct doca_comch_consumer  
  *consumer, uint32_t dev_num_recv)`
  
  - DOCA 2.7 version:
    
    - `doca_comch_consumer_set_dev_num_recv(struct doca_comch_consumer *consumer,  
      uint32_t dev_num_recv)`

- `doca_error_t doca_comch_producer_set_dev_max_num_send(struct doca_comch_producer  
  *producer, uint32_t dev_num_send)`
  
  - DOCA 2.7 version:
    
    - `doca_comch_producer_set_dev_num_send(struct doca_comch_producer *producer,  
      uint32_t dev_num_send)`

- `doca_error_t doca_comch_consumer_completion_get_max_num_consumers(const struct  
  doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)`
  
  - DOCA 2.7 version:
    
    - `doca_comch_consumer_completion_get_max_num_consumers(struct  
      doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)`

- `doca_error_t doca_comch_consumer_completion_get_max_num_consumers(const struct  
  doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)`
  
  - DOCA 2.7 version:
    
    - `doca_comch_consumer_completion_get_max_num_consumers(struct  
      doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_consumers)`

- `doca_error_t doca_comch_consumer_completion_get_max_num_recv(const struct  
  doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_recv)`
  
  - DOCA 2.7 version:
    
    - `doca_comch_consumer_completion_get_max_num_recv(struct  
      doca_comch_consumer_completion *consumer_comp, uint32_t *max_num_recv)`

**Adding const to getter API functions**
• `doca_comch_consumer_task_post_recv_get_buf(const struct doca_comch_consumer_task_post_recv *task)`
  
  ◦ **DOCA 2.7 version:**

    ■ `doca_comch_consumer_task_post_recv_get_buf(struct
doca_comch_consumer_task_post_recv *task)`

• `doca_comch_consumer_task_post_recv_get_producer_id(const struct
doca_comch_consumer_task_post_recv *task)`

  ◦ **DOCA 2.7 version:**

    ■ `doca_comch_consumer_task_post_recv_get_producer_id(struct
doca_comch_consumer_task_post_recv *task)`

• `const uint8_t *doca_comch_consumer_task_post_recv_get_imm_data(const struct
doca_comch_consumer_task_post_recv *task)`

  ◦ **DOCA 2.7 version:**

    ■ `uint8_t *doca_comch_consumer_task_post_recv_get_imm_data(struct
doca_comch_consumer_task_post_recv *task)`

• `doca_comch_consumer_task_post_recv_get_imm_data_len(const struct
doca_comch_consumer_task_post_recv *task)`

  ◦ **DOCA 2.7 version:**

    ■ `doca_comch_consumer_task_post_recv_get_imm_data_len(struct
doca_comch_consumer_task_post_recv *task)`

• `doca_comch_producer_task_send_get_buf(const struct doca_comch_producer_task_send *task)`

  ◦ **DOCA 2.7 version:**

    ■ `doca_comch_producer_task_send_get_buf(struct doca_comch_producer_task_send *task)`

• `doca_comch_producer_task_send_get_consumer_id(const struct doca_comch_producer_task_send *task)`

  ◦ **DOCA 2.7 version:**

    ■ `doca_comch_producer_task_send_get_consumer_id(struct
doca_comch_producer_task_send *task)`

• `doca_comch_producer_task_send_get_imm_data(const struct doca_comch_producer_task_send *task)`
- DOCA 2.7 version:
  - `doa_comch_producer_task_send_get_imm_data(struct doa_comch_producer_task_send *task)`
  - `doa_comch_producer_task_send_get_imm_data_len(const struct doa_comch_producer_task_send *task)`

- DOCA 2.7 version:
  - `doa_comch_producer_task_send_get_imm_data_len(struct doa_comch_producer_task_send *task)`

**Environment**

DOCA Comch based applications can run either on the host machine or on the NVIDIA BlueField Arm.

Sending messages between the host and BlueField Arm can only be run with a BlueField configured with a mode as described in [NVIDIA BlueField Modes of Operation](#).

For basic DOCA Comch send and receive, the following configuration is required:

- `doa_comch_server` context must run on the BlueField Arm cores
- `doa_comch_client` context must run on the host machine

**Note**

Producer and consumer objects can run on both the host and BlueField Arm cores. However, there must be a valid client/server connection already established on the channel.

**Architecture**
DOCA Comch is comprised of four DOCA Core Contexts. All DOCA Comch contexts leverage DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

A `doca_comch_server` context runs on the BlueField Arm and listens for incoming connections from the host side. Such host side connections are initiated by a `doca_comch_client` context.

Servers can receive connections from multiple clients in parallel, however, a client can only connect with one server. An established 1-to-1 connection between a client and a server is represented by a `doca_comch_connection`.

Once an established connection exists between a client and a server, the `doca_comch_producer` and `doca_comch_consumer` contexts can be used to run fast path channels.

The following diagram provides examples of the contexts use:
<table>
<thead>
<tr>
<th>Description</th>
<th>Location</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>doca_comch_server</code></td>
<td>Allows applications on the BlueField Arm cores to listen on a specific server name and accept new incoming connection from the host</td>
<td>BlueField Arm only</td>
</tr>
<tr>
<td><code>doca_comch_client</code></td>
<td>Allows client applications to connect to a specific server name on the BlueField Arm cores</td>
<td>Host only</td>
</tr>
<tr>
<td><code>doca_comch_connection</code></td>
<td>A connection handle created on the client side or the server side when a new connection is established. This handle is used to send/receive messages or to create <code>doca_comch_consumers</code> and <code>doca_comch_producers</code>.</td>
<td>BlueField Arm and host</td>
</tr>
<tr>
<td><code>doca_comch_producer</code></td>
<td>A handle for a FIFO-like send queue that provides a zero-copy API to send messages to a specific <code>doca_comch_consumer</code> on the same <code>doca_comch_connection</code>. Multiple <code>doca_comch_producers</code> can be created per <code>doca_comch_connection</code>.</td>
<td>BlueField Arm and host</td>
</tr>
<tr>
<td><code>doca_comch_consumer</code></td>
<td>A handle for a FIFO-like receive queue that provides a zero-copy API to receive messages from a <code>doca_comch_producer</code>.</td>
<td>BlueField Arm and host</td>
</tr>
</tbody>
</table>

**Security Considerations**

- DOCA Comch guarantees:
  - The client is connected to the server by providing the exact server name on the client side
  - Only clients on the PF/VF/SF represented by the `doca_dev_rep` provided upon server creation can connect to the server
- The connection requests and data path are isolated from the network

- DOCA Comch does not provide security at the application level:
  - It is up to the user to implement application-level security and verify the identity of the client application
  - A server handles applications from a single PF/VF/SF. If a server application detects a compromised client application, the server app should consider all clients (from that PF/VF/SF) compromised.

**Initialization Flow**

**doca_comch_server Initialization Flow**

1. A `doca_comch_server` is created on a specific `doca_dev` and a specific `doca_dev_rep`.

2. A `doca_comch_server` must have a unique name per `doca_dev/doca_dev_rep` (i.e., two servers on the same `doca_dev` and `doca_dev_rep` cannot have the same name).

3. Once `doca_ctx_start()` is called, the `doca_comch_server` can start receiving new connection requests.

4. For the `doca_comch_server` to process new connection requests and messages, the user must periodically call `doca_pe_progress()`.

5. When a new connection request arrives, `doca_comch_server` calls the connection request handler function and passes a `doca_comch_connection` object.

The server can now send and receive messages on the connection represented by `doca_comch_connection`.

**doca_comch_client Initialization Flow**

1. A `doca_comch_client` is created on a specific `doca_dev` is targeting a specific `doca_comch_server`.
2. Once `doca_ctx_start()` is called, `doca_comch_client` asynchronously tries to connect to the server.

3. To establish the connection and receive messages, the user must periodically call `doca_pe_progress()`.

4. When the connection is established, `doca_comch_client` calls the state change callback indicating state change to "RUNNING".

The client can now send a receive messages.

The following diagram describes the initialization of a basic client/server connection on DOCA Comch:
**doca_comch_consumer Initialization Flow**

1. A `doca_comch_consumer` is created on a specific `doca_comch_connection`.

2. `doca_pe_progress()` must be periodically called on the client/server PE to allow registration of the consumer.
3. After the `doca_comch_consumer` moves to "RUNNING" state:

1. `doca_comch_consumer` notifies its existence to the peer (invoking a new consumer event).

2. The application can start posting receive tasks.

3. A `doca_comch_producer` on the peer side can start sending messages to that consumer.

The initialization flow is described in the following diagram:

---

**Teardown Flow**

The teardown flow must be executed in the following order, otherwise errors may occur.
**Disconnecting Specific Connection**

The proper disconnection process for a specific connection consists of the following steps:

1. Stop all consumers and producers linked to the connection.

2. Server/client:

   1. For server, a connection can be disconnected using `doca_comch_server_disconnect()`. If there are any active producers/consumers linked to the connection, the disconnect would fail. A disconnection notifies the client and initiates teardown on that side too.

   2. For client, since there is only one connection at any given time, the connection can be disconnected by calling `doca_ctx_stop()`. If there are any active producers/consumers, the command would fail. Stopping the client context notifies the server of the disconnection and causes a disconnection of the connection on it.

**Tearing Down DOCA Comch**

The proper teardown for a DOCA Comch context consists of the following:

1. Stop all consumers and producers linked to the context.

2. Call `doca_ctx_stop()`. If there are any active connections, they would all be disconnected. If there are any active consumers/producers, the command would fail. Disconnecting/stopping the context informs all active peers of the disconnection, and causes teardown (on clients) or disconnection (on server). Calling `doca_ctx_stop()` successfully moves the context to "stopping" state.

3. After moving to stopping state, `doca_pe_progress()` must be called until the context moves to idle state.

**MsgQ (DPA Communication)**
DOCA Comch MsgQ leverages the existing consumer/producer model to allow communication between host/BlueField and DPA.

Since communication between the host/BlueField and DPA is local, there is no need to create a server, client, or connection. Instead the user can create a MsgQ and use it to create producers and consumers directly.

When creating a consumer/producer using the MsgQ, it becomes possible to use them in the DPA application as well as the CPU application:

- The CPU application can utilize existing consumer/producer APIs for communication
- The DPA application has a different set of APIs that are usable within a DPA application
**Communication Direction**

Every instance of a MsgQ can only support a single communication direction as follows:

- **Communication from host/BlueField to DPA**
  
  - This direction may be specified using `doca_comch_msgq_set_dpa_consumer`
  
  - Consumers created from this MsgQ are referred to as DPA consumers, while producers are CPU producers

- **Communication from DPA to host/BlueField**
  
  - This direction may be specified using `doca_comch_msgq_set_dpa_producer`
  
  - Consumers created from this MsgQ are referred to as CPU consumers, while producers are DPA producers

To support bidirectional communication in an application, the user has to create 2 MsgQ instances, as shown in the above diagram.

**Configuration Phase**

To start using the library, users must 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.

**Configurations**

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

To find out if a certain configuration is supported, or what the min/max value for it is, refer to [Device Support](#).

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

- For a basic send/receive client or server:
  - A send task callback
  - A receive event callback
  - A device with appropriate support must be provided on creation
  - A valid server name must be provided on creation (for clients this is the server to connect to)
  - A connection event callback (server only)

- For fast path producer or consumer:
  - A device with appropriate support must be provided on creation
  - An established client to server connection must be provided on creation
  - A `doca_mmap` with PCIe read/write permissions of where data should be received must be provided on creation (consumer only)
  - A post receive task callback (consumer only)
  - A send task callback (producer only)
  - A new consumer callback (triggered upon creation/destruction of a remove consumer)

- For MsgQ fast path producer or consumer:
  - A started MsgQ must be provided on creation
  - A DPA instance must be provided (DPA consumer/producer only)
  - A DPA consumer completion context must be connected (DPA consumer only)
  - A DPA completion context must be attached (DPA producer only)
- A post receive task callback (CPU consumer only)
- The number of receive operations (DPA consumer only)
- A send task callback (CPU producer only)
- The number of send operations (DPA producer only)

**Optional Configurations**

The following configurations are optional, if they are not set then a default value will be used:

For basic send/receive client:

- `doca_comch_(server | client)_set_max_msg_size` – set the maximum size of message that can be sent. If set, it must be matching between server and client.
- `doca_comch_(server | client)_set_recv_queue_size` – set the size of the queue to receive new messages on

For fast path consumers:

- `doca_comch_consumer_set_imm_data_len` – set the length of immediate data that a consumer can receive.

**Device Support**

DOCA Comch requires a device to operate. For instructions on picking a device, see [DOCA Core Device Discovery](#).

As device capabilities are subject to change (see [DOCA Core Device Support](#)), it is recommended to select a device using the following methods:

- For basic client and server:
  - `doca_comch_cap_server_is_supported`
- `doca_comch_cap_client_is_supported`

- For extended fast path functionality:
  - `doca_comch_producer_cap_is_supported`
  - `doca_comch_consumer_cap_is_supported`

Some devices can allow different capabilities as follows:

- The maximum length server name
- The maximum message size
- The maximum receive queue length
- The maximum clients that can connect to a server
- The maximum number of send tasks or post receive tasks
- The maximum buffer length for fast path
- The maximum immediate data supported by a fast path consumer

**Buffer Support**

Basic send and receive between a client and server does not use DOCA buffers and so has no restrictions on buffer type.

- For producers, supplied buffers need only be from a local mmap
- For consumers, post receive buffers are required to be from a PCIe export mmap

⚠️ **Note**

Chained buffers are not supported in DOCA Comch.
Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine. For additional execution environments, refer to section "Alternative Datapath Options".

Tasks

DOCA Comch exposes asynchronous tasks that leverage the BlueField hardware according to DOCA Core architecture.

Control Channel Send Task

This task allows the sending of messages between connected client and server objects.

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>Number of tasks</td>
<td>doca_comch_server_task_send_set_conf</td>
<td>doca_comch_cap_get_max_send_tasks</td>
</tr>
<tr>
<td></td>
<td>doca_comch_client_task_send_set_conf</td>
<td></td>
</tr>
<tr>
<td>Maximal message size</td>
<td>doca_comch_server_set_max_msg_size</td>
<td>doca_comch_server_get_max_msg_size</td>
</tr>
<tr>
<td></td>
<td>doca_comch_client_set_max_msg_size</td>
<td>doca_comch_client_get_max_msg_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>Peer</td>
<td>Established client/server connection</td>
<td>-</td>
</tr>
<tr>
<td>Message</td>
<td>Data string to send to remote client/server</td>
<td>The is no requirement for the message to be in DOCA mmap registered memory</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>------</td>
<td>--------------------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>Length</td>
<td>Number of bytes in the message</td>
<td>Must not exceed configured max size</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in DOCA Core Task.

**Task Completion Success**

After the task completes successfully:

- The message is delivered to the connections remote client/server
- A receive event is triggered on the remote side

**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The message is not delivered to the remote side

**Task Limitations**

- The operation is not atomic
- Once the task has been submitted, then the message should not be updated
- Other limitations are described in DOCA Core Task
Consumer Post Receive Task

This task allows consumer objects to publish buffers which are available for remote producers to write to.

Note

A Post Receive task may have a NULL buffer if it only wishes to receive immediate data.

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>doca_comch_consumer_task_post_recv_set_conf</td>
<td>doca_comch_consumer_cap_is_supported</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_comch_consumer_task_post_recv_set_conf</td>
<td>doca_comch_consumer_cap_get_max_num_tasks</td>
</tr>
<tr>
<td>Maximal buffer size</td>
<td>–</td>
<td>doca_comch_consumer_cap_get_max_buf_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>Buffer</td>
<td>Buffer that the consumer can receive data on</td>
<td>Data is appended to the tail of the buffer</td>
</tr>
</tbody>
</table>

Info
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Buffers doca_mmap must have DOCA_ACCESS_FLAG_PCI_READ_WRITE flag set.</td>
</tr>
</tbody>
</table>

**Task Output**

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

**Task Completion Success**

The task only completes once a producer has written to the advertised buffer (or immediate data, or both), not when the post receive has completed.

Upon successful completion, the buffer contains the data written by the producer and its length is updated appropriately.

**Task Completion Failure**

Task failure occurs if a buffer has not been successfully posted to receive data.

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- Producers are not aware of the buffer so would not write to it

**Task Limitations**

- The operation is not atomic
Once the task has been submitted, the buffer should not be read/written to

Buffer must come from memory with PCIe read/write access

Chained buffer lists are not supported

MsgQ consumer does not support providing `doca_buf`, and can only receive immediate data

Other limitations are described in DOCA Core Task

### Producer Send Task

This task allows producer objects to copy buffers for use by remote consumers.

### 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><code>doca_comch_producer_task_send_set_conf</code></td>
<td><code>doca_comch_producer_cap_is_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_comch_producer_task_send_set_conf</code></td>
<td><code>doca_comch_producer_cap_get_max_num_tasks</code></td>
</tr>
<tr>
<td>Maximal buffer Size</td>
<td>-</td>
<td><code>doca_comch_producer_cap_get_max_buf_size</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>Buffer</td>
<td>Buffer that should be copied to a consumer</td>
<td>Only the data residing in the data segment is copied</td>
</tr>
<tr>
<td>Immediate data</td>
<td>Short byte array to add to the post receive completion entry</td>
<td>This is not a zero copy operation but does improve latency for small payloads</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Immediate data length</td>
<td>Length of data immediate data pointed to</td>
<td>Maximum length is determined/set by individual consumers</td>
</tr>
<tr>
<td>Consumer ID</td>
<td>Identifier for the target consumer to write to</td>
<td>Active consumers and their IDs are advertised through consumer events</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in DOCA Core Task.

**Task Completion Success**

After the task is completed successfully:

- The data is copied form the buffer to the next free buffer posted by the given consumer
- Consumers process buffers from a given consumer in the order they are sent

**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination *doca_buf* objects are not modified
- The destination memory may be modified

**Task Limitations**

- The operation is not atomic
• Once the task has been submitted, the buffer should not be read/written to

• The buffer length should not be greater than consumer post receive buffers (an invalid value is returned otherwise)

• MsgQ producer does not support providing `doca_buf`, and can only send immediate data

• All limitations described in DOCA Core Task

Events

DOCA Comch exposes asynchronous events to notify about changes that happen out of the blue, according to the DOCA Core architecture. See DOCA Core Event.

Common events as described in DOCA Core Event.

Control Channel Receive Event

This event triggers whenever a remote client/server has sent a message to the local client/server object.

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_comch_server_event_msg_recv_register</code>&lt;br&gt;<code>doca_comch_client_event_msg_recv_register</code></td>
<td>-</td>
</tr>
</tbody>
</table>

Event Trigger Condition

The event is triggered when a remote message is received on any currently active connection associated with the client or server.
Event Output

Upon event detection, the registered callback is triggered, passing the following parameters:

- A pointer to the message data
- The length in bytes of the message
- The active connection on which the message was received

Info

The data is only valid in the context of the callback.

Connection Status Changed Event (Server Only)

This event provides asynchronous updates on the state of any connections associated with a server.

Note

A client object can only connect to a single server, so its connection state can be tracked through its `doca_ctx` state and the generic `doca_ctx_set_state_changed_cb` function.

Event Configuration
### Event Trigger Condition

The event is triggered when a new connection is either established or a current connection disconnected on a server.

### Event Output

Separate callbacks are registered for connection or disconnection events with the appropriate one triggered based on the specific event.

Both callbacks contain a Boolean indicating if the connection or disconnection was successful.

### Consumer Event

This event indicates that a new consumer object has been created or an existing consumer object has been destroyed.

### Event Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
</table>
| Register to the event | doca_comch_server_event_consumer_register  
doca_comch_client_event_consumer_register | –                    |

### Event Trigger Condition


The event is triggered whenever a new consumer is created or a current consumer destroyed on the remote side of an established DOCA Comch connection.

**Event Output**

The event hits a separate callback for either the creation or destruction of a consumer.

Callback parameters include:

- The established DOCA Comch connection on which the consumer is connected (on the remote side)
- The ID of the consumer (a unique value per Comch connection)

**State Machine**

The DOCA Comch library follows the Context state machine 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 the 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>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all tasks have been freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all tasks are completed and freed</td>
</tr>
</tbody>
</table>

**Starting**

In this state it is expected that the application will:

- Call progress to allow transition to next state (e.g., when a connection attempt completes)

Allowed operations:

- Call progress

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>

**Running**

In this state, it is expected that the application:

- Allocates and submit tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocate a previously configured task
• Submit an allocated task

• 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>
<tr>
<td>Starting</td>
<td>Call progress until context state transitions</td>
</tr>
</tbody>
</table>

**Stopping**

In this state, it is expected that the application will:

• Free any completed tasks

Allowed operations:

• Allocate previously configured task

• Submit an allocated task

• 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>Running</td>
<td>Call progress and fatal error occurs</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop without freeing all tasks</td>
</tr>
</tbody>
</table>

**Alternative Datapath Options**

DOCA Comch can be run on as part of DPA data path, using the **MsgQ**.
DPA

Using the `MsgQ` it is possible to create consumer/producer on the DPA. They follow the definition described in DOCA Core DPA.

Since these objects can be used in DPA, they have DPA APIs that can be used to perform the data path operations expanded on in the following subsections.

**Consumer Ack**

The `doca_dpa_dev_comch_consumer_ack` API prepares the DPA consumer to receive a number of immediate messages from CPU producers.

**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>Queue Size</td>
<td><code>doca_comch_consumer_set_dev_max_num_recv</code></td>
<td>–</td>
</tr>
</tbody>
</table>

**Input**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Messages</td>
<td>A number describing how many additional immediate messages this consumer can receive</td>
<td>Must not exceed the queue size</td>
</tr>
</tbody>
</table>

**Completion**

Whenever a message is received from the CPU producer a completion element is generated and can be polled using `doca_dpa_dev_comch_consumer_get_completion`.

Using the generated completion, it is possible to get the following outputs:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immedia te</td>
<td>A pointer to the immediate message</td>
<td>The message lifetime is the same as the completion element lifetime. That is, once the completion is</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Message</td>
<td>that the CPU producer sent</td>
<td>acked using <code>doca_dpa_dev_comch_consumer_completion_ack</code>, the pointer is no longer valid. To retain the message past the completion lifetime, the user must copy the contents of the message.</td>
</tr>
<tr>
<td>Immediate Message Length</td>
<td>The length in bytes of the immediate message that the CPU producer sent</td>
<td></td>
</tr>
<tr>
<td>Producer ID</td>
<td>The ID of the CPU producer that sent the message</td>
<td>User can find the IDs of each producer by using <code>doca_comch_producer_get_id</code>.</td>
</tr>
</tbody>
</table>

**Limitations**

- The maximal immediate message size is 32 bytes

**Producer Post Send Immediate Only**

The `doca_dpa_dev_comch_producer_post_send_imm_only` API sends an immediate message to the CPU consumer. Once the message arrives at the CPU consumer side, the CPU consumer receive task completes.

The CPU producer must have posted a receive task prior to this. The user can verify if the consumer can receive the message using `doca_dpa_dev_comchProducer_is_consumer_empty`. Note, however, that this may add overhead.

**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>Queue Size</td>
<td><code>doca_comch_producer_set_dev_max_num_send</code></td>
<td>–</td>
</tr>
</tbody>
</table>
Input

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Message</td>
<td>Short byte array to be sent to the CPU consumer</td>
<td>This is not a zero copy operation but does improve latency for small payloads</td>
</tr>
<tr>
<td>Immediate Message Length</td>
<td>Length of the message the immediate message points to</td>
<td>The maximum length is 32 bytes</td>
</tr>
<tr>
<td>Consumer ID</td>
<td>Identifier for the target CPU consumer to write to</td>
<td>User can find the IDs of each consumer by using <code>doca_comch_consumer_get_id</code></td>
</tr>
</tbody>
</table>
| Completion Requested        | Flag indicating whether to generate a completion once the send is completed                    | This refers to the DPA producer completion which is separate from the completion the CPU consumer receives
|                             |                                                                                               | - 0 – no completion                                                                          |
|                             |                                                                                               | - 1 – otherwise                                                                             |

Completion

Once the message arrives to the CPU consumer, a completion element is generated, indicating that the send is complete (this is separate from the completion the CPU consumer receives) and can be polled using `doca_dpa_dev_get_completion`.

Using the generated completion, it is possible to get the following outputs:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer User Data</td>
<td>Producer user data provided during configuration of the producer</td>
<td>User data previously set using <code>doca_ctx_set_user_data</code> when configuring this producer. User data which is returned belongs to the DPA producer this completion has been generated for, and can be used to identify the specific producer.</td>
</tr>
</tbody>
</table>

Limitations
• The maximal immediate message size is 32 bytes

**Producer DMA Copy**

The `doca_dpa_dev_comch_producer_dma_copy` API performs a DMA copy operation and, once the copy operation is done, sends an immediate message to the CPU consumer. Once the message arrives at the CPU consumer side, the CPU consumer receive task completes.

The CPU producer must have posted a receive task prior to this. The user can verify if the consumer can receive the message using `doca_dpa_dev_comch_producer_is_consumer_empty`. Note, however, that this may add overhead.

**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>Queue Size</td>
<td><code>doca_comch_producer_set_dev_num_recv</code></td>
<td>–</td>
</tr>
</tbody>
</table>

**Input**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination Mmap</td>
<td>Mmap representing the memory to be used as the destination of the copy operation</td>
<td>This mmap must have <code>LOCAL_READ_WRITE</code> access enabled</td>
</tr>
<tr>
<td>Destination Address</td>
<td>The address to be used as the destination of the copy operation</td>
<td>The address and copy length must be within the range of the destination mmap's memory range</td>
</tr>
<tr>
<td>Source Mmap</td>
<td>Mmap representing the memory to be used as the source of the copy operation</td>
<td>This mmap must have <code>LOCAL_READ</code> access enabled</td>
</tr>
<tr>
<td>Source Address</td>
<td>The address to be used as the source of the copy operation</td>
<td>The address and copy length must be within the range of the source mmap's memory range</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Length</strong></td>
<td>The length of the copy operation</td>
<td>Source and destination addresses must not overlap</td>
</tr>
<tr>
<td><strong>Immediate Message</strong></td>
<td>Short byte array to be sent to the CPU consumer once the copy operation is done</td>
<td>This is not a zero copy operation but does improve latency for small payloads</td>
</tr>
<tr>
<td><strong>Immediate Message Length</strong></td>
<td>Length of the message the immediate message points to</td>
<td>The maximum length is 32 bytes</td>
</tr>
<tr>
<td><strong>Consumer ID</strong></td>
<td>Identifier for the target CPU consumer to write to</td>
<td>User can find the IDs of each consumer using <code>doca_comch_consumer_get_id</code></td>
</tr>
</tbody>
</table>
| **Completion Requested** | Flag indicating whether to generate a completion once the send is completed | This refers to the DPA producer completion which is separate from the completion the CPU consumer receives  
  • 0 – no completion  
  • 1 – otherwise |

**Completion**

Once copy is complete and the message arrives to the CPU consumer, a completion element is generated, indicating that the copy is complete (this is separate from the completion the CPU consumer receives) and can be polled using `doca_dpa_dev_get_completion`.

Using the generated completion, it is possible to get the following outputs:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Producer User Data</strong></td>
<td>Producer user data provided during configuration of the producer</td>
<td>The user data set using <code>doca_ctx_set_user_data</code> when configuring this producer. The user data which is returned belongs to the DPA producer this completion has been generated for, and can be used to identify the specific producer.</td>
</tr>
</tbody>
</table>
Limitations

- The maximal immediate message size is 32 bytes

DOCA Comch Samples

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

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

- Set up a client/server between host and BlueField Arm cores and use it to send text messages
- Configure fast path producers and consumers, and send messages between them

Info

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

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:
The binary doca_<sample_name> is created under /tmp/build/.

3. All DOCA Comch samples accept the same input arguments:

<table>
<thead>
<tr>
<th>Sample</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>doca_comch_ctrl_path_server</td>
<td>-p, --pci-addr</td>
<td>DOCA Comch device PCIe address</td>
</tr>
<tr>
<td>doca_comch_ctrl_path_client</td>
<td>-r, --rep-pci</td>
<td>DOCA Comch device representor PCIe address (required only on BlueField Arm)</td>
</tr>
<tr>
<td>doca_comch_data_path_high_speed_server</td>
<td>-t, --text</td>
<td>Text to be sent to the other side of channel (overwrites default)</td>
</tr>
<tr>
<td>doca_comch_data_path_high_speed_client</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4. For additional information per sample, use the -h option:

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

## Samples

### DOCA Comch Control Path Client/Server

1. **Note**

   `doca_comch_ctrl_path_server` must be run on the BlueField Arm side and started before `doca_comch_ctrl_path_client` is started on the host.
This sample sets up a client server connection between the host and BlueField Arm cores.

The connection is used to pass two messages, the first sent by the client when the connection is established and the second by the server on receipt of the client's message.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the core DOCA structures.
3. Initializing and configuring client/server contexts.
4. Registering tasks and events for sending/receiving messages and tracking connection changes.
5. Allocating and submitting tasks for sending control path messages.
6. Handling event completions for receiving messages.
7. Stopping and destroying client/server objects.

References:

- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_client/comch_ctrl_path_client_main.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_client/comch_ctrl_path_client_sample.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_server/comch_ctrl_path_server_main.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_server/comch_ctrl_path_server_sample.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_common.c
- /opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_common.h

**Note**

**DOCA Comch Data Path Client/Server**
This sample sets up a client server connection between host and BlueField Arm.

The connection is used to create a producer and consumer on both sides and pass a message across the two fastpath connections.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the core DOCA structures.
3. Initializing and configuring client/server contexts.
4. Initializing and configuring producer/consumer contexts on top of an established connection.
5. Submitting post receive tasks for population by producers.
6. Submitting send tasks from producers to write to consumers.
7. Stopping and destroying producer/consumer objects.
8. Stopping and destroying client/server objects.

References:

- /opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_client/comch_data_path_high_speed_client
- /opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_client/comch_data_path_high_speed_client
- /opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_server/comch_data_path_high_speed_server
- /opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_server/comch_data_path_high_speed_server
- /opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_common.c
• /opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_common.h