



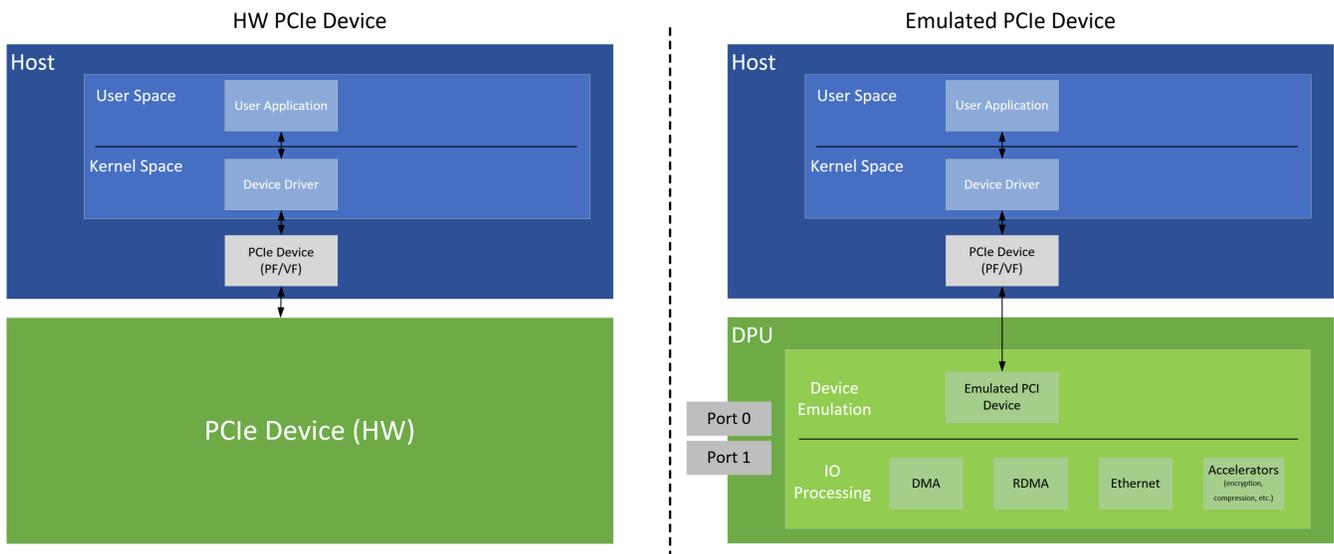
DOCA Device Emulation

Table of contents

DOCA DevEmu PCI	4
DOCA DevEmu PCI Generic	18
DOCA DevEmu Virtio	67
DOCA DevEmu Virtio-FS	73

Introduction

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) provide the ability to emulate a PCIe device. The DOCA Device Emulation subsystem provides a low-level software API for users to develop PCIe devices and their controllers. These APIs include discovery, configuration, hot plugging/unplugging, management, and IO path handling. In simpler terms, the libraries enable the user to implement a hardware PCIe function using software, such that the host is not aware that the PCIe function is emulated, and all interactions from the host are routed to software on the BlueField instead of actual hardware.



The diagram shows the potential for device emulation to replace a regular PCIe function of some PCIe device.

- On the left is a conventional setup where the host is connected to a PCIe device (e.g., NVMe SSD). On the host, user applications interact with the kernel driver of that device, using some software interface, and the driver communicates with the hardware/firmware of the device.
- On the right is a setup where the PCIe device is replaced with a BlueField with an application using DOCA Device Emulation. The application can use the DOCA DevEmu PCI library to control the device, and intercept any IOs written by the host to the PCIe device. Additionally, the application can use other DOCA libraries to perform IO processing (e.g., copying data from host memory using DMA, sending RDMA/Ethernet traffic) and other acceleration libraries for encryption, compression, etc.

Known Limitations

- This library is supported at alpha level; backward compatibility is not guaranteed
- VFs are not currently supported
- Some limitations apply when creating a generic emulated function, for more details refer to [DOCA DevEmu PCI Generic Limitations](#).
- Consult your NVIDIA representative for limitations on the emulated device's behavior

DOCA DevEmu PCI

Note

This library is supported at alpha level; backward compatibility is not guaranteed.

Introduction

DOCA DevEmu PCI is part of the DOCA Device Emulation subsystem. It provides low-level software APIs that allow management of an emulated PCIe device using the emulation capability of NVIDIA® BlueField® networking platforms.

It is a common layer for all PCIe emulation modules, such as DOCA DevEmu PCIe Generic Emulation, and DOCA DevEmu Virtio subsystem emulation.

Prerequisites

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

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

Generic device emulation is part of DOCA device emulation. It is recommended to read the following guides beforehand:

- [DOCA Device Emulation](#)

Environment

DOCA DevEmu PCI Emulation is supported only on the BlueField target. The BlueField must meet the following requirements

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

Info

Please refer to the [DOCA Backward Compatibility Policy](#).

The library must be run with root privileges.

Perform the following:

1. Configure the BlueField to work in DPU mode as described in [NVIDIA BlueField Modes of Operation](#).
2. Enable the PCIe switch emulation capability needed for hot plugging emulated PCIe devices. This can be done by running the following command on the host or BlueField:

```
host/bf> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s  
PCI_SWITCH_EMULATION_ENABLE=1
```

3. Perform a [BlueField system-level reset](#) for the `mlxconfig` settings to take effect.

To support hot-plug feature, the host must have the following boot parameters:

- Intel CPU:

```
intel_iommu=on iommu=pt pci=realloc
```

- AMD CPU:

```
iommu=pt pci=realloc
```

This can be done using the following steps:

i Info

This process may vary depending on the host OS. Users can find multiple guides online describing this process.

1. Add the boot parameters:

```
host> sudo nano /etc/default/grub
Find the variable
GRUB_CMDLINE_LINUX_DEFAULT="<existing-params>"
Add the params at the end
GRUB_CMDLINE_LINUX_DEFAULT="<existing-params> intel_iommu=on iommu=pt
pci=realloc"
```

2. Update configuration.

- For Ubuntu:

```
host> update-grub
```

- For RHEL:

```
host> grub2-mkconfig -o /boot/grub2/grub.cfg
```

3. Perform warm boot.
4. Confirm that the parameters are in effect:

```
host> cat /proc/cmdline  
<existing-params> intel_iommu=on iommu=pt pci=realloc
```

Architecture

The DOCA DevEmu PCI library provides 2 main software abstractions, the PCIe type, and the PCIe device. The PCIe type represents the configurations of the emulated device, while the PCIe device represents an instance of an emulated device. Furthermore, any PCIe device instance must be associated with a single PCIe type, while PCIe type can be associated with many PCIe devices.

Pre Defined PCI Type vs. Generic PCI Type

A PCIe type object can be acquired in 2 different ways:

- Acquire a pre-defined type, using emulation libraries of existing protocols such as [DOCA DevEmu Virtio FS](#) library
- Create from scratch using the [DOCA DevEmu Generic](#) library

In case of pre-defined type, the configurability of the type is limited.

PCIe Type Name

As part of the DOCA PCIe emulation, every type has a name assigned to it. This property is not part of the PCIe specification, but rather it is a mechanism in DOCA that uniquely identifies the PCIe type.

There cannot be 2 different PCIe types with the same name, even across different processes, unless the type in the second process is configured in identical manner to the first one. Furthermore, attempting to configure the second type with same name but with slight configuration difference will fail.

Create Emulated Device

After configuring the desired DOCA Devemu PCIe type, it is possible to create an emulated device based on the configured type using

`doca_devemu_pci_dev_create_rep`. This sequential process ensures that the DOCA DevEmu PCIe device is created with the specified parameters and configuration defined by the PCIe type object. Furthermore, it is possible to destroy the emulated device using `doca_devemu_pci_dev_destroy_rep`.

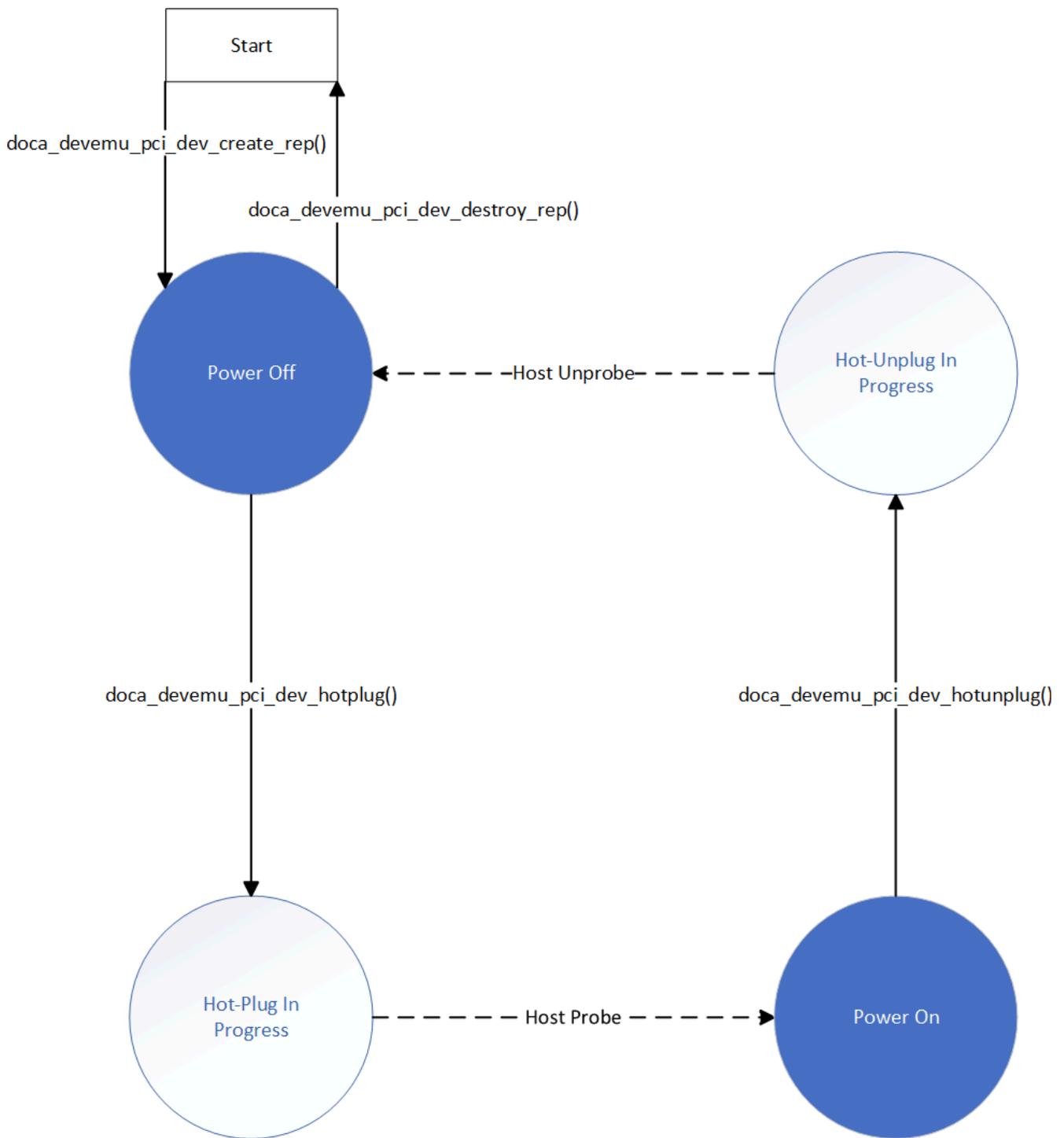
The created device representor starts in "power_off" state and is not visible to the host until hot-plug sequence is issued by the user, see [Hot-plug Emulated Device](#). The device can then be destroyed only while in "power_off" state.

Info

The created emulated device may outlive the application that created it, see [Objects Lifecycle and Persistency](#).

Hot-plug Emulated Device

Hot-plugging refers to the process of emulating the physical attachment of a PCIe device to the host PCIe subsystem after the system has been powered on and initialized. Note that some operating systems require additional settings to enable the process of hot-plugging a PCIe device. For supported systems, this feature proves particularly advantageous for systems that need to remain operational at all times while expanding their hardware resources, such as additional storage and networking capabilities. DOCA DevEmu PCI provides software APIs that allow users to emulate this process in an asynchronous manner.



When creating a PCIe device object, if it starts in "power off" state, then the device is not yet visible to the host. It is possible then, from the BlueField, to hot-plug the device. This starts an async process of the device getting hot-plugged towards the host. Once the process completes, the emulated device transitions to "power on" and becomes visible to the host. Usually at this stage, the emulated device receives its BDF address. The hot-unplug process works in similar async manner.

Using DOCA API, the BlueField Arm can register to any changes to the hot-plug state of each emulated device using

```
doca_devemu_pci_dev_event_hotplug_state_change_register.
```

Emulated Device Discovery

The emulated device is represented as a `doca_devinfo_rep`. It is possible to iterate through all the emulated devices as explained in [DOCA Core Representor Discovery](#).

There are 2 ways of filtering the list of emulated devices:

- Get all emulated devices – use `DOCA_DEVINFO_REP_FILTER_EMULATED` as the filter argument in `doca_devinfo_rep_create_list`
- Get all emulated devices that belong to a certain type – `doca_devemu_pci_type_create_rep_list`

Objects Lifecycle and Persistency

This section creates distinction between firmware resources and software resources:

- Firmware resources persist until the next power cycle, and can be accessible from different processes on the BlueField Arm. Such resources are not cleared once the application exits.
- Software resources are representations of firmware resources, and are only relevant for the same thread

Using this terminology, it is possible to describe the objects as follows:

- The PCIe type object `doca_devemu_pci_type` represents a PCIe type firmware resource. The resource persists if any of the following apply:
 - There is at least 1 process holding reference to the PCIe type
 - There is at least 1 PCIe device firmware resource belonging to this type

- The emulated device representor, `doca_devinfo_rep`, represents an emulated PCIe function firmware resource:
 - `doca_devemu_pci_dev_create_rep` can be used to create such firmware resource
 - To destroy the firmware resource, `doca_devemu_pci_dev_destroy_rep` can be used
 - For static functions, the representor resource persists until configured otherwise in NVCONFIG
 - To find existing PCIe device firmware resources, use `doca_devemu_pci_type_create_rep_list`

Function Level Reset

The created emulated devices support PCIe function level reset (FLR).

Using DOCA API, the BlueField Arm can register to FLR event using `doca_devemu_pci_dev_event_flr_register`. Once the driver requests FLR, this event is triggered, calling the user provided callback.

Once FLR is detected, it is expected for the BlueField Arm to do the following:

- Destroy all resources related to the PCIe device. For information on such resources, refer to the guide of concrete PCIe type (generic/virtiofs).
- Stop the PCIe device
- Start the PCIe device again

Device Support

DOCA PCIe Device emulation requires a device to operate. For picking a device, see [DOCA Core Device Discovery](#).

The device emulation library is only supported for BlueField-3.

As device capabilities may change in the future (see [Capability Checking](#)), it is recommended that users choose a device using the following method:

- `doca_devemu_pci_cap_type_is_hotplug_supported` – for create and hot-plug support
- `doca_devemu_pci_cap_type_is_mgmt_supported` – for device discovery only

PCIe Device

Configuration Phase

To start using the DOCA DevEmu PCI Device, users 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 retrieval of events.

Configurations

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

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

Mandatory Configurations

All mandatory configurations are provided during the creation of the PCIe device.

These configurations are as follows:

- A DOCA DevEmu PCIe type object
- A DOCA Device Representor, representing an emulated function with the same type as the provided PCIe object type
- A DOCA Progress Engine object

Optional Configurations

These configurations are optional. If not set, then a default value is used:

- Registering to events as described in the "[Events](#)" section. By default, the user does not receive events.

Execution Phase

This section describes execution on CPU using [DOCA Core Progress Engine](#).

Events

The DOCA DevEmu PCI device exposes asynchronous events to notify about sudden changes according to DOCA Core architecture.

Common events are described in [DOCA Core Event](#).

Hotplug State Change

The hotplug state change event allows users to receive notifications whenever the hotplug state of the emulated device changes. See section "[Hot-plug Emulated Device](#)".

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	<code>doca_devemu_pci_dev_event_hotplug_state_change_register</code>	<code>doca_devemu_pci_cap_type_is_hotplug_supported</code>

Event Trigger Condition

The event is triggered anytime an asynchronous transition happens as follows:

- `DOCA_DEVEMU_PCI_HP_STATE_PLUG_IN_PROGRESS` → `DOCA_DEVEMU_PCI_HP_STATE_POWER_ON`
- `DOCA_DEVEMU_PCI_HP_STATE_UNPLUG_IN_PROGRESS` → `DOCA_DEVEMU_PCI_HP_STATE_POWER_OFF`

- `DOCA_DEVEMU_PCI_HP_STATE_POWER_ON` → `DOCA_DEVEMU_PCI_HP_STATE_UNPLUG_IN_PROGRESS` (when initiated by the host)

Any transition initiated by user is not triggered (e.g., calling hotplug to transition from `POWER_OFF` to `PLUG_IN_PROGRESS`).

The following APIs can be used to initiate hotplug or hot-unplug transition processes:

- `doca_devemu_pci_dev_hotplug`
- `doca_devemu_pci_dev_hotunplug`

Event Output

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

Additionally, the internal cached hotplug state is updated and can be fetched using `doca_devemu_pci_dev_get_hotplug_state`.

Event Handling

Once the event is triggered, it means that the hotplug state has changed. The application is expected to do the following:

- Retrieve the new hotplug state using `doca_devemu_pci_dev_get_hotplug_state`

Function Level Reset

The FLR event allows users to receive notifications whenever the host initiates an FLR flow. See section "[Function Level Reset](#)".

Event Configuration

Description	API to Set the Configuration
Register to the event	<code>doca_devemu_pci_dev_event_flr_register</code>

Event Trigger Condition

The event is triggered anytime the host driver initiates an FLR flow. See section "[Function Level Reset](#)".

Event Output

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

Additionally, the internal cached FLR indicator is updated and can be fetched using `doca_devemu_pci_dev_is_flr`.

Event Handling

Once the event is triggered, it means that the host driver has initiated the FLR flow.

The user must handle the FLR flow by doing the following:

1. Flush all the outstanding requests back to the associated resource
2. Release all the PCIe device resources dynamically created after device start
3. Stop the PCIe device – `doca_ctx_stop`
4. Start the PCIe device again – `doca_ctx_start`
 - Call `doca_pe_progress` repeatedly until the PCIe device transitions to "running" state

For more information on starting the PCIe device again, refer to section "[State Machine](#)".

State Machine

The DOCA DevEmu PCI device object follows the context state machine as described in [DOCA Core Context State Machine](#).

The following section describes how to transition to any 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 section "[Configurations](#)"
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all resources have been destroyed
Stopping	Call progress until all resources have been destroyed

Starting

In this state, it is expected that application:

- Calls progress to allow transition to next state
- Keeps context in this state until FLR flow is complete

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after receiving FLR event (i.e., while FLR is in progress)

Running

In this state, it is expected that application:

- Calls progress to receive events
- Creates/destroys PCIe device resources

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration
Starting	Call progress until FLR flow is completed

Stopping

In this state, it is expected that application:

- Destroys all emulated device resources as described in section "[Function Level Reset](#)".

Allowed operations:

- Destroying PCIe device resources

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call stop without freeing emulated device resources

DOCA DevEmu PCI Generic

Note

This library is supported at alpha level; backward compatibility is not guaranteed.

This guide provides instructions on building and developing applications that require emulation of a generic PCIe device.

Introduction

DOCA DevEmu PCI Generic is part of the DOCA Device Emulation subsystem. It provides low-level software APIs that allow creation of a custom PCIe device using the emulation capability of NVIDIA® BlueField®.

For example, it enables emulating an NVMe device by creating a generic emulated device, configuring its capabilities and BAR to be compliant with the NVMe spec, and operating it from the DPU as necessary.

Prerequisites

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

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

Generic device emulation is part of DOCA PCIe device emulation. It is recommended to read the following guides beforehand:

- [DOCA Device Emulation](#)
- [DOCA DevEmu PCI](#)

Environment

DOCA DevEmu PCI Generic Emulation is supported only on the BlueField target. The BlueField must meet the following requirements:

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

Info

Please refer to the [DOCA Backward Compatibility Policy](#).

Library must be run with root privileges.

Please refer to [DOCA DevEmu PCI Environment](#), for further necessary configurations.

Architecture

DOCA DevEmu PCI Generic allows the creation of a generic PCI type. The PCI Type is part of the DOCA DevEmu PCI library. It is the component responsible for configuring the capabilities and bar layout of emulated devices.

The PCI Type can be considered as the template for creating emulated devices. Such that the user first configures a type, and then they can use it to create multiple emulated devices that have the same configuration.

For a more concrete example, consider that you would like to emulate an NVMe device, then you would create a type and configure its capabilities and BAR to be compliant with the NVMe spec, after that you can use the same type, to generate multiple NVMe emulated devices.

PCIe Configuration Space

The PCIe configuration space is 256 bytes long and has a header that is 64 bytes long. Each field can be referred to as a register (e.g., device ID).

Every PCIe device is required to implement the PCIe configuration space as defined in the PCIe specification.

The host can then read and/or write to registers in the PCIe configuration space. This allows the PCIe driver and the BIOS to interact with the device and perform the required setup.

It is possible to configure registers in the PCIe configuration space header as shown in the following diagram:

	31-24	23-16	15-8	7-0
■ Deprecated ■ Configurable ■ Not configurable	Device ID		Vendor ID	
	Status		Command	
	Class Code			Revision ID
	BIST	Header Type	Latency Timer	Cache Line Size
	BAR0			
	BAR1			
	BAR2			
	BAR3			
	BAR4			
	BAR5			
	Cardbus CIS Pointer			
	Subsystem ID		Subsystem Vendor ID	
	Expansion ROM Base Address			
	Reserved			Capabilities Pointer
	Reserved			
	Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line

i Info

0x0 is the only supported header type (general device).

The following registers are read-only, and they are used to identify the device:

Register Name	Description	Example
Class Code	Defines the functionality of the device Can be further split into 3 values {class : subclass: prog IF}	0x020000 Class: 0x02 (Network Controller) Subclass: 0x00 (Ethernet Controller) Prog IF: 0x00 (N/A)
Revision ID	Unique identifier of the device revision Vendor allocates ID by itself	0x01 (Rev 01)

Register Name	Description	Example
Vendor ID	Unique identifier of the chipset vendor Vendor allocates ID from the PCI-SIG	0x15b3 Nvidia
Device ID	Unique identifier of the chipset Vendor allocates ID by itself	0xa2dc BlueField-3 integrated ConnectX-7 network controller
Subsystem Vendor ID	Unique identifier of the card vendor Vendor allocates ID from the PCI-SIG	0x15b3 Nvidia
Subsystem ID	Unique identifier of the card Vendor allocates ID by itself	0x0051

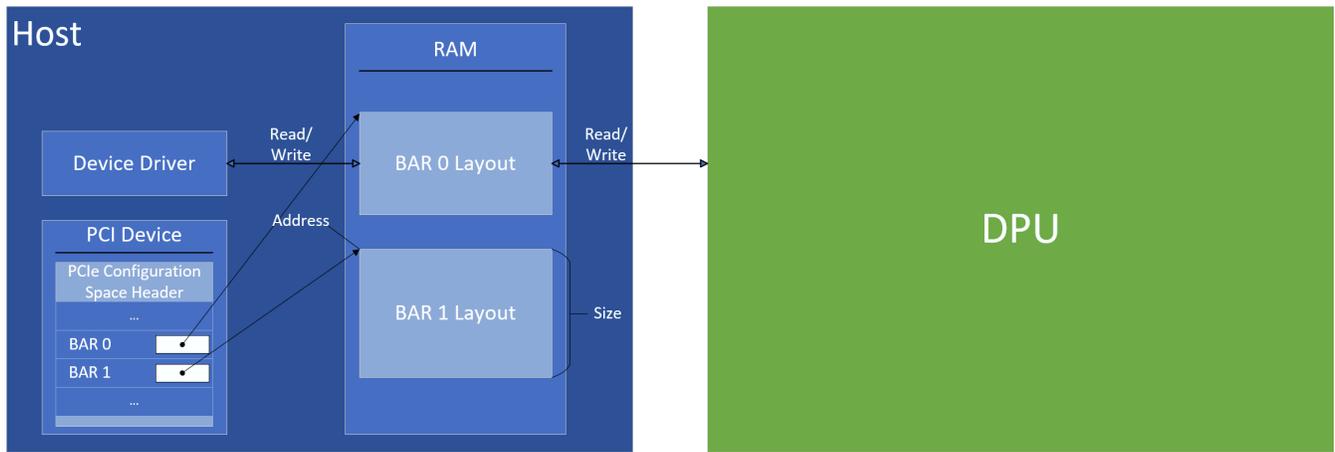
BAR

While the PCIe configuration space can be used to interact with the PCIe device, it is not enough to implement the functionality that is targeted by the device. Rather, it is only relevant for the PCIe layer.

To enable protocol-specific functionality, the device configures additional memory regions referred to as base address registers (BARs) that can be used by the host to interact with the device. Different from the PCIe configuration space, BARs are defined by the device and interactions with them is device-specific. For example, the PCIe driver interacts with an NVMe device's PCIe configuration space according to the PCIe spec, while the NVMe driver interacts with the BAR regions according to the NVMe spec.

Any read/write requests on the BAR are typically routed to the hardware, but in case of an emulated device, the requests are routed to the software.

The DOCA DevEmu PCI type library provides APIs that allow software to pick the mechanism used for routing the requests to software, while taking into consideration common design patterns utilized in existing devices.



Each PCIe device can have up to 6 BARs with varying properties. During the PCIe bus enumeration process, the PCIe device must be able to advertise information about the layout of each BAR. Based on the advertised information, the BIOS/OS then allocates a memory region for each BAR and assigns the address to the relevant BAR in the PCIe configuration space header. The driver can then use the assigned memory address to perform reads/writes to the BAR.

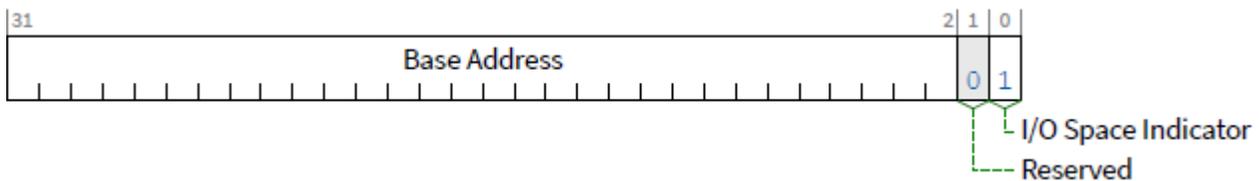
BAR Layout

The PCIe device must be able to provide information with regards to each BAR's layout.

The layout can be split into 2 types, each with their own properties as detailed in the following subsections.

I/O Mapped

According to the PCIe specification, the following represents the I/O mapped BAR:



Additionally, the BAR register is responsible for advertising the requested size during enumeration.

i Info

The size must be a power of 2.

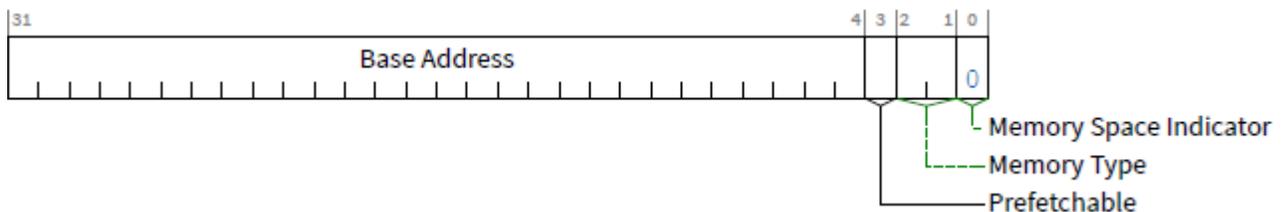
Users can use the following API to set a BAR as I/O mapped:

```
doca_devemu_pci_type_set_io_bar_conf(struct doca_devemu_pci_type
*pci_type, uint8_t id, uint8_t log_sz)
```

- `id` – the BAR ID
- `log_sz` – the log of the BAR size

Memory Mapped

According to the PCIe specification, the following represents the memory mapped BAR:



Additionally, the BAR register is responsible for advertising the requested size during enumeration.

i Info

The size must be a power of 2.

The memory mapped BAR allows a 64-bit address to be assigned. To achieve this, users must specify the bar Memory Type as 64-bit, and then set the next BAR's (BAR ID + 1) size to be 0.

Setting the pre-fetchable bit indicates that reads to the BAR have no side-effects.

Users can use the following API to set a BAR as memory mapped:

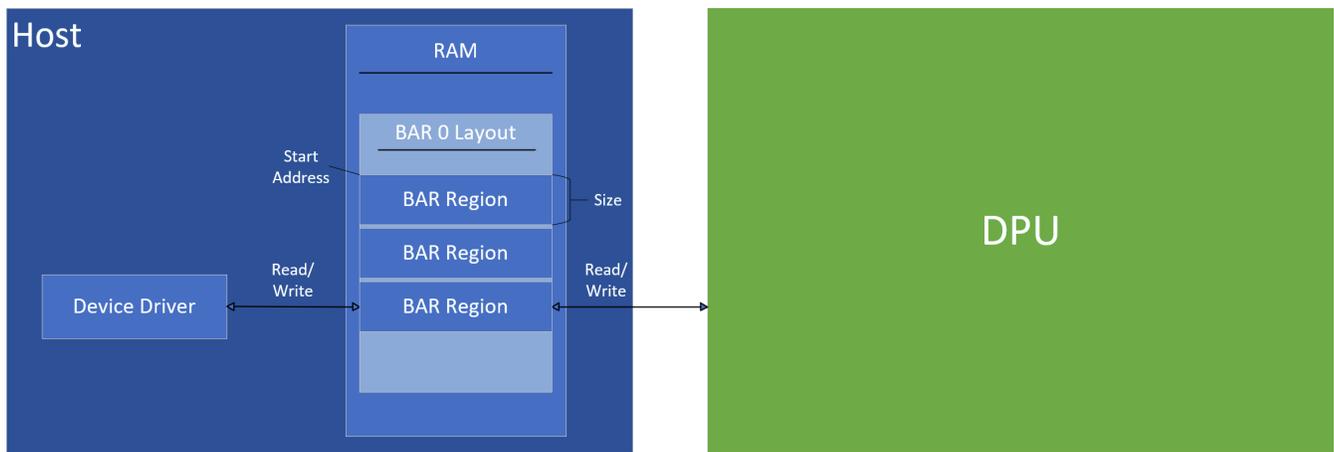
```
doca_devemu_pci_type_set_memory_bar_conf(struct  
doca_devemu_pci_type *pci_type, uint8_t id, uint8_t log_sz, enum  
doca_devemu_pci_bar_mem_type memory_type, uint8_t prefetchable)
```

- `id` – the BAR ID
- `log_sz` – the log of the BAR size. If set to 0, then the size is considered as 0 (instead of 1).
- `memory_type` – specifies the memory type of the BAR. If set to 64-bit, then the next BAR must have `log_sz` set to 0.
- `prefetchable` – indicates whether the BAR memory is pre-fetchable or not (a value of 1 or 0 respectively)

BAR Regions

BAR regions refer to memory regions that make up a BAR layout. This is not something that is part of the PCIe specification, rather it is a DOCA concept that allows the user to customize behavior of the BAR when interacted with by the host.

The BAR region defines the behavior when the host performs a read/write to an address within the BAR, such that every address falls in some memory region as defined by the user.



Common Configuration

All BAR regions have these configurations in common:

- `id` – the BAR ID that the region is part of
- `start_addr` – the start address of the region within the BAR layout relative to the BAR. 0 indicates the start of the BAR layout.
- `size` – the size of the BAR region

Currently, there are 4 BAR region types, defining different behavior:

- Stateful
- DB by offset
- DB by data
- MSIX table
- MSIX PBA

Generic Control Path (Stateful BAR Region)

Stateful region can be used as a shared memory, such that the contents are maintained in firmware. A read from the driver returns the latest value, while a write updates the value

and triggers an event to software running on the DPU.

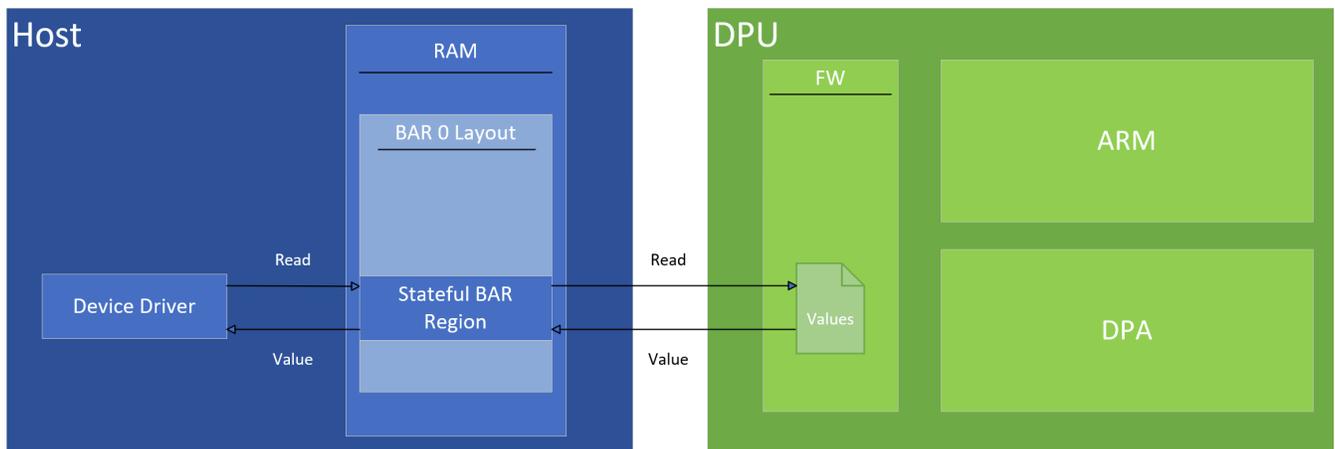
This can be useful for communication between the driver and the device, during the control path (e.g., exposing capabilities, initialization).

i Info

Some limitations apply, please see [Limitations](#) section

Driver Read

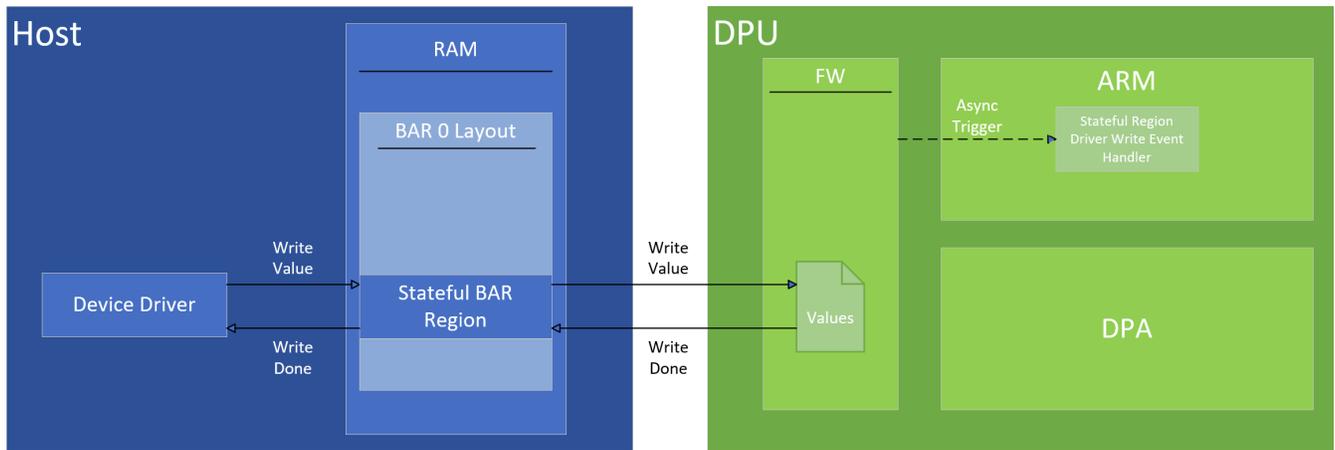
A read from the driver returns the latest value written to the region, whether written by the host or by the driver itself.



Driver Write

A write from the driver updates the value at the written address and notifies software running on the Arm that a write has occurred. The notification on the Arm arrives as an asynchronous event (see

```
doca_devemu_pci_dev_event_bar_stateful_region_driver_write).
```



i Info

The event that arrives to Arm software is asynchronous such that it may arrive after the driver has completed the write.

DPU Read

The DPU can read the values of the stateful region using `doca_devemu_pci_dev_query_bar_stateful_region_values`. This returns the latest snapshot of the stateful region values. It can be particularly useful to find what was written by the driver after the "stateful region driver write event" occurs.

DPU Write

The DPU can write the values of the stateful region using `doca_devemu_pci_dev_modify_bar_stateful_region_values`. This updates the values such that subsequent reads from the driver or the DPU returns these values.

Default Values

The DPU is able to set default values to the stateful region. Default values come in 2 layers:

- Type default values – these values are set for all devices that have the same type. This can be set only if no device currently exists.
- Device default values – these values are set for a specific device and take affect on the next FLR cycle or the next hotplug of the device

A read of the stateful region follows the following hierarchy:

1. Return the latest value as written by the host or driver (whichever was done last).
2. Return the device default values.
3. Return the type default values.
4. Return 0.

No Defaults



Type Default



Device Default

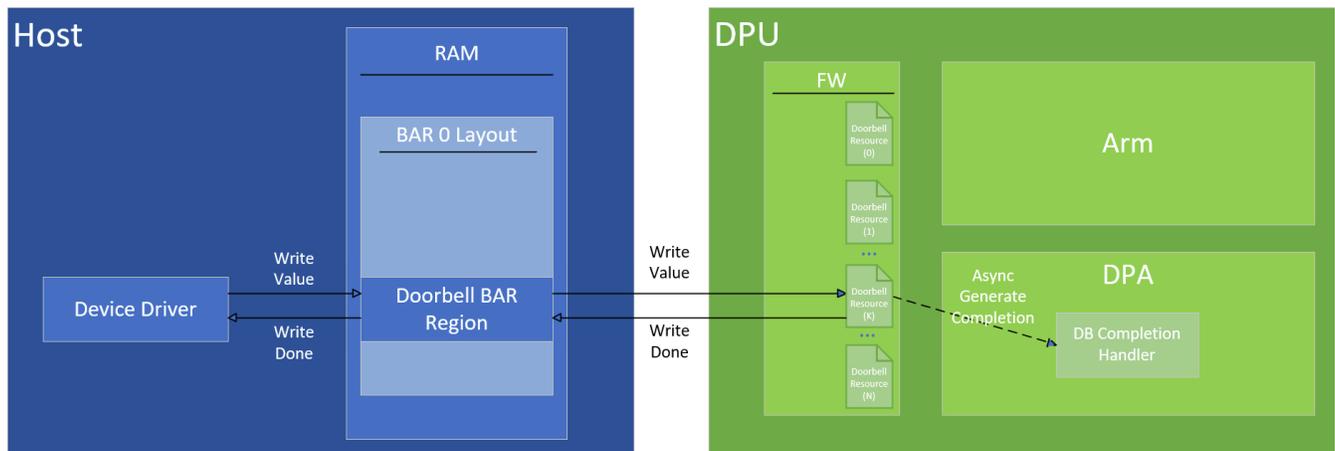


Zeroes

Generic Data Path (DB BAR Region)

Doorbell (DB) regions can be used to implement a consumer-producer queue between the driver and the DPU, such that a write from the driver would trigger an event on the DPU through DPA, allowing it to fetch the written value. This can be useful for communication between the driver and the device, during the data path allowing IO processing.

While DBs are not part of the PCIe specification, it is a widely used mechanism by vendors (e.g., RDMA QP, NVMe SQ, virtio VQ, etc).



The same DB region can be used to manage multiple DBs, such that each DB can be used to implement a queue.

The DPU software can utilize DB resources individually:

- Each DB resource has a unique zero-based index referred to as DB ID
- DB resource can be managed (create/destroy/modify/query) individually
- Each DB resource has a separate notification mechanism. That is, the notification on DPU is triggered for each DB separately.

Driver Write

The DB usually consists of a numeric value (e.g., `uint32_t`) representing the consumer/producer index of the queue.

When the driver writes to the DB region, the related DB resource gets updated with the written value, and a notification is sent to the DPU.

When driver writes to the DB BAR region it must adhere to the following:

- The size of the write must match the size of the DB value (e.g., `uint32_t`)
- The offset within the region must be aligned to the DB stride size or the DB size

The flow would look something as the following:

- Driver performs a write of the DB value at some offset within the DB BAR region
- DPU calculates the DB ID that the write is intended for. Depending on the region type:
 - DB by offset – DPU calculates the DB ID based on the write offset relative to the DB BAR region
 - DB by data – DPU parses the written DB value and extracts the DB ID from it
- DPU updates the DB resource with the matching DB ID to the value written by the driver
- DPU sends a notification to the DPA application, informing it that the value of DB with DB ID has been updated by the driver

Driver Read

The driver should not attempt to read from the DB region. Doing so results in anomalous behavior.

BlueField Write

The BlueField can update the value of each DB resource individually using `doca_devemu_pci_db_modify_value`. This produces similar side effects as though the driver updated the value using a write to the DB region.

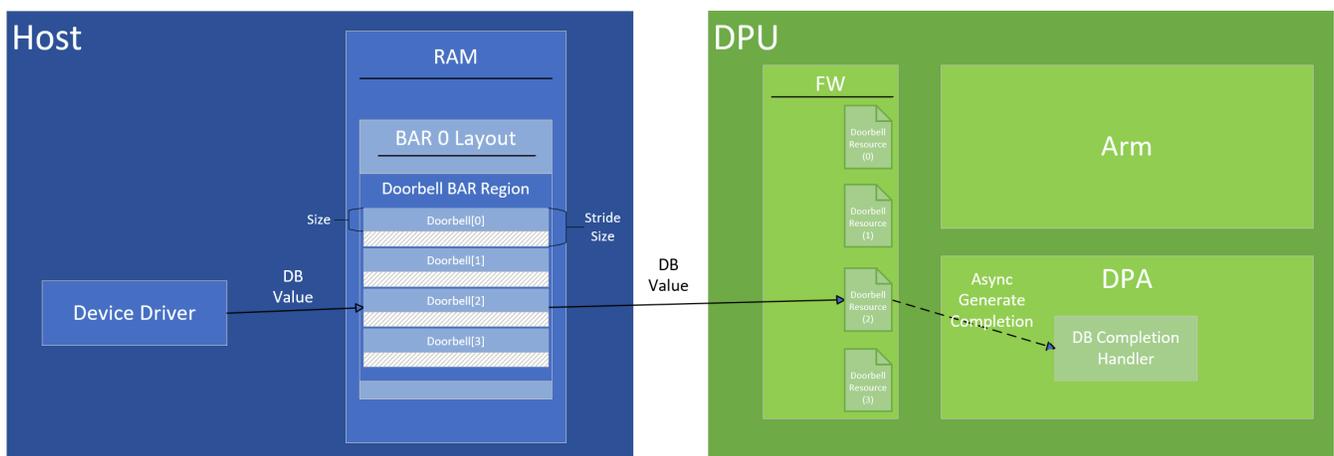
BlueField Read

The BlueField can read the value of each DB resource individually using one of the following methods:

- Read the value from the BlueField Arm using `doca_devemu_pci_db_query_value`
- Read the value from the DPA using `doca_dpa_dev_devemu_pci_db_get_value`

The first option is a time consuming operation and is only recommended for the control path. In the data path, it is recommended to use the second option only.

DB by Offset



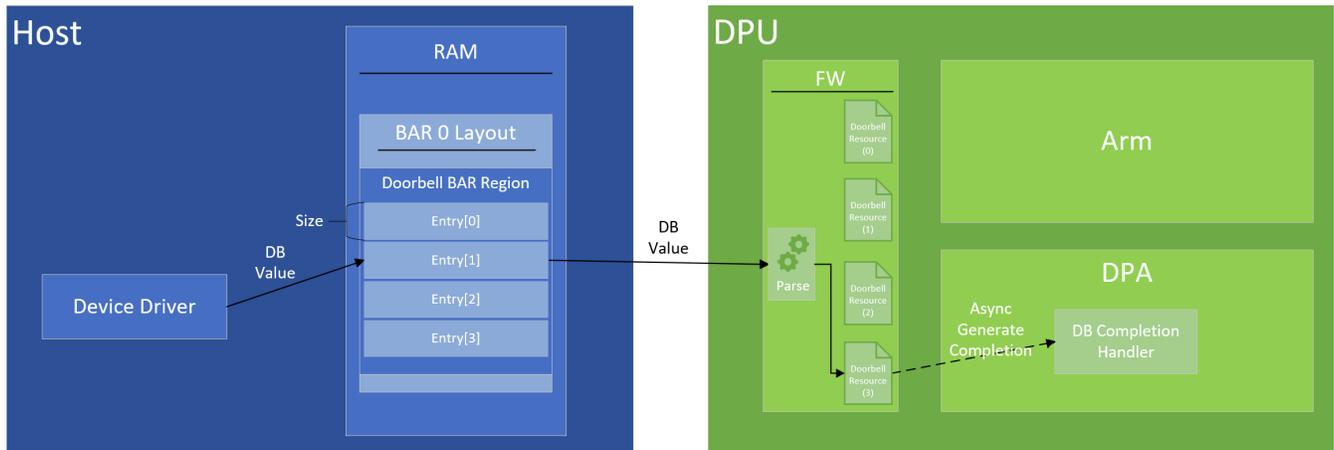
The API `doca_devemu_pci_type_set_bar_db_region_by_offset_conf` can be used to set up DB by offset region. When the driver writes a DB value using this region, the DPU receives a notification for the relevant DB resource, based on the write offset, such that the DB ID is calculated as follows: $db_id = \text{write_offset} / db_stride_size$.

Warning

The area that is part of the stride but not part of the doorbell, should not be used for any read/write operation, doing so will result in

undefined anomalous.

DB by Data



The API `doca_devemu_pci_type_set_bar_db_region_by_data_conf` can be used to set up DB by data region. When the driver writes a DB value using this region, the DPU receives a notification for the relevant DB resource based on the written DB value, such that there is no relation between the write offset and the DB triggered. This DB region assumes that the DB ID is embedded within the DB value written by the driver. When setting up this region, the user must specify where the Most Significant Byte (MSB) and Least Significant Byte (LSB) of the DB ID are embedded in the DB value.

The DPU follows these steps to extract the DB ID from the DB value:

- Driver writes the DB value
- BlueField extracts the bytes between MSB and LSB
- DPU compares MSB index with LSB index
 - If MSB index greater than LSB index: The extracted value is interpreted as Little Endian
 - If LSB index greater than MSB index: The extracted value is interpreted as Big Endian

Example:

DB size is 4 bytes, LSB is 1, and MSB is 3.

- Driver writes value `0xCCDDEEFF` to DB region at index 0 in Little Endian
 - The value is written to memory as follows: `[0]=FF [1]=EE [2]=DD [3]=CC`
- The relevant bytes, are the following: `[1]=EE [2]=DD [3]=CC`
- Since MSB (3) is greater than LSB (1), the value is interpreted as Little Endian:
`db_id = 0xCCDDEE`

MSI-X Capability (MSI-X BAR Region)

Message signaled interrupts extended (MSI-X) is commonly used by PCIe devices to send interrupts over the PCIe bus to the host driver. DOCA APIs allow users to expose the MSI-X capability as per the PCIe specification, and to later use it to send interrupts to the host driver.

To configure it, users must provide the following:

- The number of MSI-X vectors which can be done using `doca_devemu_pci_type_set_num_msix`
- Define an [MSI-X table](#)
- Define an [MSI-X PBA](#)

MSI-X Table BAR Region

As per the PCIe specification, to expose the MSI-X capability, the device must designate a memory region within its BAR as an MSI-X table region. In DOCA, this can be done using `doca_devemu_pci_type_set_bar_msix_table_region_conf`.

MSI-X PBA BAR Region

As per the PCIe specification, to expose the MSI-X capability, the device must designate a memory region within its BAR as an MSI-X pending bit array (PBA) region. In DOCA, this

can be done using `doca_devemu_pci_type_set_bar_msix_pba_region_conf`.

Raising MSI-X From DPU

It is possible to raise an MSI-X for each vector individually. This can be done only using the DPA API `doca_dpa_dev_devemu_pci_msix_raise`.

DMA Memory

Some operations require accessing memory which is set up by the host driver. DOCA's device emulation APIs allow users to access such I/O memory using the DOCA mmap (see [DOCA Core Memory Subsystem](#)).

After starting the PCIe device, it is possible to acquire an mmap that references the host memory using `doca_devemu_pci_mmap_create`. After creating this mmap, it is possible to configure it by providing:

- Access permissions
- Host memory range
- DOCA devices that can access the memory

The mmap can then be used to create buffers that reference memory on the host. The buffers' addresses would not be locally accessible (i.e., CPU cannot dereference the address), instead the addresses would be I/O addresses as defined by the host driver.

The buffers created from the mmap can then be used with other DOCA libraries and accept a `doca_buf` as an input. This includes:

- [DOCA DMA](#)
- [DOCA RDMA](#)
- [DOCA Ethernet](#)
- [DOCA AES-GCM](#)

Function Level Reset

FLR can be handled as described in [DOCA DevEmu PCI FLR](#). Additionally, users must ensure that the following resources are destroyed before stopping the PCIe device:

- Doorbells created using `doca_devemu_pci_db_create_on_dpa`
- MSI-X vectors created using `doca_devemu_pci_msix_create_on_dpa`
- Memory maps created using `doca_devemu_pci_mmap_create`

Limitations

Based on explanation in "[Driver Write](#)", user can assume that DOCA DevEmu PCI Generic supports creating emulated PCI devices with the limitation that when a driver writes to a register, the value is immediately available for subsequent reads from the same register. However, this immediate availability does not ensure that any required internal actions triggered by the write have been completed. It is recommended to rely on specific different register values to confirm completion of the write action. For instance, when implementing a write-to-clear operation, e.g. writing 1 to register A to clear register B, it is advisable to poll register B until it indicates the desired state. This approach ensures that the write action has been successfully executed. If a device specification requires certain actions to be completed before exposing written values for subsequent reads, such a device cannot be emulated using the DOCA DevEmu PCI generic framework.

Device Support

DOCA PCI Device emulation requires a device to operate. For information on picking a device, see [DOCA DevEmu PCI Device Support](#).

Some devices can allow different capabilities as follows:

- The maximum number of emulated devices
- The maximum number of different PCIe types

- The maximum number of BARs
- The maximum BAR size
- The maximum number of doorbells
- The maximum number of MSI-X vectors
- For each BAR region type there are capabilities for:
 - Whether the region is supported
 - The maximum number of regions with this type
 - The start address alignment of the region
 - The size alignment of the region
 - The min/max size of the region

Tip

As the list of capabilities can be long, it is recommended to use the [NVIDIA DOCA Capabilities Print Tool](#) to get an overview of all the available capabilities.

Run the tool as root user as follows:

```
$ sudo /opt/mellanox/doca/tools/doca_caps -p <pci-
address> -b devemu_pci
Example output: PCI: 0000:03:00.0
    devemu_pci
        max_hotplug_devices
15
        max_pci_types
2
        type_log_min_bar_size
12
```

30	type_log_max_bar_size
11	type_max_num_msix
64	type_max_num_db
1	type_log_min_db_size
2	type_log_max_db_size
2	type_log_min_db_stride_size
12	type_log_max_db_stride_size
2	type_max_bars
12	bar_max_bar_regions
12	type_max_bar_regions
supported	bar_db_region_identify_by_offset
supported	bar_db_region_identify_by_data
4096	bar_db_region_block_size
16	bar_db_region_max_num_region_blocks
2	type_max_bar_db_regions
2	bar_max_bar_db_regions
4096	bar_db_region_start_addr_alignment
64	bar_stateful_region_block_size

```

    bar_stateful_region_max_num_region_blocks
4
    type_max_bar_stateful_regions
1
    bar_max_bar_stateful_regions
1
    bar_stateful_region_start_addr_alignment
64
    bar_msix_table_region_block_size
4096
bar_msix_table_region_max_num_region_blocks    1
    type_max_bar_msix_table_regions
1
    bar_max_bar_msix_table_regions
1
bar_msix_table_region_start_addr_alignment
4096
    bar_msix_pba_region_block_size
4096
    bar_msix_pba_region_max_num_region_blocks
1
    type_max_bar_msix_pba_regions
1
    bar_max_bar_msix_pba_regions
1
    bar_msix_pba_region_start_addr_alignment
4096
    bar_is_32_bit_supported
unsupported
    bar_is_1_mb_supported
unsupported
    bar_is_64_bit_supported
supported

```

```
pci_type_hotplug
supported
pci_type_mgmt
supported
```

PCI Type

Configurations

This section describes the configurations of the DOCA DevEmu PCI Type object, that can be provided before start.

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

Mandatory Configurations

The following are mandatory configurations and must be provided before starting the PCI type:

- A DOCA device that is an emulation manager or hotplug manager. See [Device Support](#).

Optional Configurations

The following configurations are optional:

- The PCIe device ID
- The PCIe vendor ID
- The PCIe subsystem ID
- The PCIe subsystem vendor ID
- The PCIe revision ID

- The PCIe class code
- The number of MSI-X vectors for MSI-X capability
- One or more memory mapped BARs
- One or more I/O mapped BARs
- One or more DB region
- An MSI-X table and PBA regions
- One or more stateful regions

Info

If these configurations are not set then a default value is used.

PCI Device

Configuration Phase

This section describes additional configuration options, on top of the ones already described in [DOCA DevEmu PCI Device Configuration Phase](#).

Configurations

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

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

Optional Configurations

The following configurations are optional:

- Setting the stateful regions' default values – If not set, then the type default values are used. See [stateful region default values](#) for more.

Execution Phase

This section describes additional events, on top of the ones already described in [DOCA DevEmu PCI Device Events](#).

Events

DOCA DevEmu PCI Device exposes asynchronous events to notify about changes that happen suddenly according to the DOCA Core architecture.

Common events are described in [DOCA Core Event](#).

BAR Stateful Region Driver Write

The stateful region driver write event allows you to receive notifications whenever the host driver writes to the stateful BAR region. See section "[Driver Write](#)" for more information.

Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	<pre>doca_devemu_pci_dev_event_bar_stateful_region_driver_write_register</pre>	<pre>doca_devemu_pci_cap_type_get_max_bar_stateful_regions</pre>

If there are multiple stateful regions for the same device, then registration is done separately for each region. The details provided on registration (i.e., `bar_id` and start address) must match a region previously configured for PCIe type.

Trigger Condition

The event is triggered anytime the host driver writes to the stateful region. See section "[Driver Write](#)" for more information.

Output

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

Additionally, the event callback receives an event object of type

```
struct doca_devemu_pci_dev_event_bar_stateful_region_driver_write
```

which can be used to retrieve:

- The DOCA DevEmu PCI Device representing the emulated device that triggered the event –
`doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_pc`
- The ID of the BAR containing the stateful region –
`doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_ba`
- The start address of the stateful region –
`doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_ba`

Event Handling

Once the event is triggered, it means that the host driver has written to someplace in the region.

The user must perform either of the following:

- Query the new values of the stateful region –
`doca_devemu_pci_dev_query_bar_stateful_region_values`
- Modify the values of the stateful region –
`doca_devemu_pci_dev_modify_bar_stateful_region_values`

It is possible also to do both. However, it is important that the memory areas that the host wrote to are either queried or overwritten with a modify operation.

Note

Failure to do so results in a recurring event. For example, if the host wrote to the first half of the region, but BlueField Arm only queries the second half of the region after receiving the event. Then the library retriggers the event, assuming that the user did not handle the event.

PCI Device DB

After the PCIe device has been created, it can be used to create DB objects, each DB object represents a DB resources identified by a DB ID. See [Generic Data Path \(DB BAR Region\)](#).

When creating the DB, the DB ID must be provided, this can hold different meaning for [DB by offset](#) and [DB by data](#). The DB object can then be used to get a notification to the DPA once a driver write occurs, and to fetch the latest value using the DPA.

Configuration

The flow for creating and configuring a DB should be as follows:

1. Create the DB object:

```
arm> doca_devemu_pci_db_create_on_dpa
```

2. (Optional) Query the DB value:

```
arm> doca_devemu_pci_db_query_value
```

3. (Optional) Modify the DB value:

```
arm> doca_devemu_pci_db_modify_value
```

4. Get the DB DPA handle for referencing the DB from the DPA:

```
arm> doca_devemu_pci_db_get_dpa_handle
```

5. Bind the DB to the DB completion context using the handle from the previous step:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_bind_db
```

 **Warning**

It is important to perform this step before the next one. Otherwise, the DB completion context will start receiving completions for an unbound DB.

6. Start the DB to start receiving completions on DPA:

```
arm> doca_devemu_pci_db_start
```

 **Info**

Once DB is started, a completion is immediately generated on the DPA.

Similarly the flow for destroying a DB would look as follows:

1. Stop the DB to stop receiving completions:

```
arm> doca_devemu_pci_db_stop
```

Info

This step ensures that no additional completions will arrive for this DB

2. Acknowledge all completions related to this DB:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_ack
```

Info

This step ensures that existing completions have been processed.

3. Unbind the DB from the DB completion context:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_unbind_db
```

 **Warning**

Make sure to not perform this step more than once.

4. Destroy the DB object:

```
arm> doca_devemu_pci_db_destroy
```

Fetching DBs on DPA

To fetch DBs on DPA, a DB completion context can be used. The DB completion context serves the following purposes:

- Notifying a DPA thread that a DB value has been updated (wakes up thread)
- Providing information about which DB has been updated

The following flow shows how to use the same DB completion context to get notified whenever any of the DBs are updated, and to find which DBs were actually updated, and finally to get the DBs' values:

1. Get DB completion element:

```
doca_dpa_dev_devemu_pci_get_db_completion
```

2. Get DB from completion:

```
doca_dpa_dev_devemu_pci_db_completion_element_get_db_propertie
```

3. Store the DB (e.g., in an array).
4. Repeat steps 1-3 until there are no more completions.
5. Acknowledge the number of received completions:

```
doca_dpa_dev_devemu_pci_db_completion_ack
```

6. Request notification on DPA for the next completion:

```
doca_dpa_dev_devemu_pci_db_completion_request_notification
```

7. Go over the DBs stored in step 3 and for each DB:

1. Request a notification for the next time the host driver writes to this DB:

```
doca_dpa_dev_devemu_pci_db_request_notification
```

2. Get the most recent value of the DB:

```
doca_dpa_dev_devemu_pci_db_get_value
```

Query/Modify DB from Arm

It is possible to query the DB value of a particular DB using `doca_devemu_pci_db_query_value` on the Arm. Similarly, it is possible to modify the DB value using `doca_devemu_pci_db_modify_value`. When modifying the DB value, the side effects of such modification is the same as if the host driver updated the DB value.

Tip

Querying and modifying operations from the Arm are time consuming and should be used in the control path only. Fetching DBs on DPA is the recommended approach for retrieval of DB values in the data path.

PCIe Device MSI-X Vector

After the PCIe device has been created, it can be used to create MSI-X objects. Each MSI-X object represents an MSI-X vector identified by the vector index.

The MSI-X object can be used to send a notification to the host driver from the DPA.

Configuration

The MSI-X object can be created using `doca_devemu_pci_msix_create_on_dpa`. An MSI-X vector index must be provided during creation, this is a value in the range $[0, \text{num_msix})$, such that `num_msix` is the value previously set using `doca_devemu_pci_type_set_num_msix`.

Once the MSI-X object is created, `doca_devemu_pci_msix_get_dpa_handle` can be used to get a DPA handle for use within the DPA.

Raising MSI-X

The MSI-X object can be used on the DPA to raise an MSI-X vector using `doca_dpa_dev_devemu_pci_msix_raise`.

DOCA DevEmu Generic Samples

This section describes DOCA DevEmu Generic samples.

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

- List details about emulated devices with same generic type
- Create and hot-plug/hot-unplug an emulated device with a generic type
- Handle Host driver write using stateful region
- Handle Host driver write using DB region
- Raise MSI-X to the Host driver
- Perform DMA operation to copy memory buffer between the Host driver and the DPU Arm

Structure

All the samples utilize the same generic PCI type. The configurations of the type reside in `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`

The structure for some samples is as follows:

- `/opt/mellanox/doca/samples/doca_devemu/<sample_directory>`
 1. `dpu`
 1. `host`
 2. `device`
 2. `host`

Samples following this structure will have two binaries: `dpu` (1) and `host` (2), the former should be run on the BlueField and represents the controller of the emulated device, while the latter should be run on the host and represents the host driver.

For simplicity, the host (2) side is based on the VFIO driver, allowing development of a driver in user-space.

Within the `dpu` (a) directory, there is a `host` (a) and `device` (b) directories. `host` in this case refers to the BlueField Arm processor, while `device` refers to the DPA processor. Both directories are compiled into a single binary.

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:

```
cd /opt/mellanox/doca/samples/doca_devemu/<sample_name>[/dpu  
or /host]  
meson /tmp/build  
ninja -C /tmp/build
```

Info

The binary `doca_<sample_name>[_dpu or _host]` is created under `/tmp/build/`.

3. Sample (e.g., `doca_devemu_pci_device_db`) usage:

1. BlueField side (`doca_devemu_pci_device_db_dpu`):

```
Usage: doca_devemu_pci_device_db_dpu [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

Program Flags:
  -p, --pci-addr          The DOCA device PCI
address. Format: XXXX:XX:XX.X or XX:XX.X
  -u, --vuid             DOCA Devemu emulated
device VUID. Sample will use this device to handle
Doorbells from Host
  -r, --region-index     The index of the DB
region as defined in devemu_pci_type_config.h. Integer
  -i, --db-id           The DB ID of the DB.
Sample will listen on DBs related to this DB ID. Integer
```

2. Host side (`doca_devemu_pci_device_db_host`):

```

Usage: doca_devemu_pci_device_db_host [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

Program Flags:
  -p, --pci-addr          PCI address of the
emulated device. Format: XXXX:XX:XX.X
  -g, --vfio-group        VFIO group ID of the
device. Integer
  -r, --region-index      The index of the DB
region as defined in devemu_pci_type_config.h. Integer
  -d, --db-index          The index of the
Doorbell to write to. The sample will write at byte
offset (db-index * db-stride)
  -w, --db-value          A 4B value to write
to the Doorbell. Will be written in Big Endian

```

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

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

Additional sample setup:

- The BlueField samples require the emulated device to be already hot-plugged:
 - Such samples expect the VUID of the hot-plugged device (`-u, --vuid`)
 - The `list` sample can be used to find if any hot-plugged devices exist and what their VUID is
 - The `hot-plug` sample can be used to hot plug a device if no such device already exists
- The host samples require the emulated device to be already hot-plugged and that the device is bound to the VFIO driver:
 - The samples expect 2 parameters `-p` (`--pci-addr`) and `-g` (`--vfio-group`) of the emulated device as seen by the host
 - The [PCI Device List](#) sample can be used from the BlueField to find the PCIe address of the emulated device on the host
 - Once the PCIe address is found, the host can use the script `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py` to bind the VFIO driver

```
$ sudo python3  
/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bin  
<pcie-address-of-emulated-dev>
```

- The script is a python3 script which expects the PCIe address of the emulated device as a positional argument (e.g., `0000:3e:00.0`)
- The script outputs the VFIO group ID

- The script must be used only once after the device is hot-plugged towards the host for the first time
- The hot-unplug sample requires that the device be unbound from the VFIO driver:
 - Use the script located at `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py` from the host to unbind the VFIO driver as follows:

```
$ sudo python3
/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bin
<pcie-address-of-emulated-dev> --unbind
```

- This python3 script expects the PCIe address of the emulated device as a positional argument (e.g., `0000:3e:00.0`) along with the `--unbind` argument

Samples

PCI Device List

This sample illustrates how to list all emulated devices that have the generic type configured in

```
/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h
```

The sample logic includes:

1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`.
2. Creating a list of all emulated devices belonging to this type.
3. Iterating over the emulated devices.
4. Dumping their VUID.
5. Dumping their PCIe address as seen by the host.

6. Releasing the resources.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`
 - `devemu_pci_device_list/`
 - `devemu_pci_device_list_sample.c`
 - `devemu_pci_device_list_main.c`
 - `meson.build`
 - `devemu_pci_common.h`; `devemu_pci_common.c`
 - `devemu_pci_type_config.h`

PCI Device Hot-Plug

This sample illustrates how to create and hot-plug/hot-unplug an emulated device that has the generic type configured in

```
/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h .
```

The sample logic includes:

1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h` .
2. Acquiring the emulated device representor:
 - If the user did not provide VUID as input, then creating and using a new emulated device.
 - If the user provided VUID as an input, then searching for an existing emulated device with a matching VUID and using it.
3. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).
4. Registering to the PCIe device's hot-plug state change event.

5. Initializing hot-plug/hot-unplug of the device:

1. If the user did not provide VUID as input, then initializing hot-plug flow of the device.
2. If the user provided VUID as input, then initializing hot-unplug flow of the device.

6. Using the PE to poll for hot-plug state change event.

7. Waiting until hot-plug state transitions to expected state (power on or power off).

8. Cleaning up resources.

- If hot-unplug was requested, then the emulated device is destroyed as well.
- Otherwise, the emulated device persists.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`
 - `devemu_pci_device_hotplug/`
 - `devemu_pci_device_hotplug_sample.c`
 - `devemu_pci_device_hotplug_main.c`
 - `meson.build`
 - `devemu_pci_common.h`; `devemu_pci_common.c`
 - `devemu_pci_type_config.h`

PCI Device Stateful Region

This sample illustrates how the host driver can write to a stateful region, and how the BlueField Arm can handle the write operation.

This sample consists of a host sample and BlueField sample. It is necessary to follow the [additional sample setup](#) detailed previously.

The BlueField sample logic includes:

1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`.
2. Acquiring the emulated device representor that matches the provided VUID.
3. Creating a PCIe device context to manage the emulated device, and connecting it to a progress engine (PE).
4. For each stateful region configured in `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`, registering to the PCIe device's stateful region write event.
5. Using the PE to poll for driver write to any of the stateful regions.
 - Every time the host driver writes to the stateful region, the handler is invoked and performs the following:
 1. Queries the values of the stateful region that the host wrote to.
 2. Logs the values of the stateful region.
 - The sample polls indefinitely until the user presses [Ctrl+c] to close the sample.
6. Cleaning up resources.

The host sample logic includes:

1. Initializing the VFIO device with a matching PCIe address and VFIO group.
2. Mapping the stateful memory region from the BAR to the process address space.
3. Writing the values provided as input to the beginning of the stateful region.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`
 - `devemu_pci_device_stateful_region/dpu/`
 - `devemu_pci_device_stateful_region_dpu_sample.c`

- `devemu_pci_device_stateful_region_dpu_main.c`
- `meson.build`
- `devemu_pci_device_stateful_region/host/`
 - `devemu_pci_device_stateful_region_host_sample.c`
 - `devemu_pci_device_stateful_region_host_main.c`
 - `meson.build`
- `devemu_pci_common.h`; `devemu_pci_common.c`
- `devemu_pci_host_common.h`; `devemu_pci_host_common.c`
- `devemu_pci_type_config.h`

PCI Device DB

This sample illustrates how the host driver can ring the doorbell and how the BlueField can retrieve the doorbell value. The sample also demonstrates how to handle FLR.

This sample consists of a host sample and BlueField sample. It is necessary to follow the [additional sample setup](#) detailed previously.

The BlueField sample logic includes:

- Host (BlueField Arm) logic:
 1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config`
 2. Initializing DPA resources:
 1. Creating DPA instance, and associating it with the DPA application.
 2. Creating DPA thread and associating it with the DPA DB handler.

3. Creating DB completion context and associating it with the DPA thread.
3. Acquiring the emulated device representor that matches the provided VUID.
4. Creating a PCIe device context to manage the emulated device, and connecting it to progress engine (PE).
5. Registering to the context state changes event.
6. Registering to the PCIe device FLR event.
7. Using the PE to poll for any of the following:
 1. Every time the PCIe device context state transitions to running, the handler performs the following:
 1. Creates a DB object.
 2. Makes RPC to DPA, to initialize the DB object.
 2. Every time the PCIe device context state transitions to stopping, the handler performs the following:
 1. Makes RPC to DPA, to un-initialize the DB object.
 2. Destroys the DB object.
 3. Every time the host driver initializes or destroys the VFIO device, an FLR event is triggered. The FLR handler performs the following:
 1. Destroys DB object.
 2. Stops the PCIe device context.
 3. Starts the PCIe device context again.
 4. The sample polls indefinitely until the user presses [Ctrl+c] to close the sample.



During this time, the DPA may start receiving DBs from the host.

8. Cleaning up resources.

- Device (BlueField DPA) logic:

1. Initializing application RPC:

1. Setting the global context to point to the DB completion context DPA handle.
2. Binding DB to the doorbell completion context.

2. Un-initializing application RPC:

1. Unbinding DB from the doorbell completion context.

3. DB handler:

1. Getting DB completion element from completion context.
2. Getting DB handle from the DB completion element.
3. Acknowledging the DB completion element.
4. Requesting notification from DB completion context.
5. Requesting notification from DB.
6. Getting DB value from DB.

The host sample logic includes:

1. Initializing the VFIO device with its matching PCIe address and VFIO group.
2. Mapping the DB memory region from the BAR to the process address space.
3. Writing the value provided as input to the DB region at the given offset.

References:

- /opt/mellanox/doca/samples/doca_devemu/
 - devemu_pci_device_db/dpu/
 - host/
 - devemu_pci_device_db_dpu_sample.c
 - device/
 - devemu_pci_device_db_dpu_kernels_dev.c
 - devemu_pci_device_db_dpu_main.c
 - meson.build
 - devemu_pci_device_db/host/
 - devemu_pci_device_db_host_sample.c
 - devemu_pci_device_db_host_main.c
 - meson.build
 - devemu_pci_common.h; devemu_pci_common.c
 - devemu_pci_host_common.h; devemu_pci_host_common.c
 - devemu_pci_type_config.h

PCI Device MSI-X

This sample illustrates how BlueField can raise an MSI-X vector, sending a signal towards the host, and shows how the host can retrieve this signal.

This sample consists of a host sample and a BlueField sample. It is necessary to follow the [additional sample setup](#) detailed previously.

The BlueField sample logic includes:

- Host (BlueField Arm) logic:
 1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config`.
 2. Initializing DPA resources:
 1. Creating a DPA instance and associating it with the DPA application.
 2. Creating a DPA thread and associating it with the DPA DB handler.
 3. Acquiring the emulated device representor that matches the provided VUID.
 4. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).
 5. Creating an MSI-X vector and acquiring its DPA handle.
 6. Sending an RPC to the DPA to raise the MSI-X vector.
 7. Cleaning up resources.
- Device (BlueField DPA) logic:
 1. Raising the MSI-X RPC by using the MSI-X vector handle.

The host sample logic includes:

1. Initializing the VFIO device with the matching PCIe address and VFIO group.
2. Mapping each MSI-X vector to a different FD.
3. Reading events from the FDs in a loop.
 1. Once the DPU raises MSI-X, the FD matching the MSI-X vector returns an event which is then printed to the screen.
 2. The sample polls the FDs indefinitely until the user presses [Ctrl+c] to close the sample.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`

- `devemu_pci_device_msix/dpu/`
 - `host/`
 - `devemu_pci_device_msix_dpu_sample.c`
 - `device/`
 - `devemu_pci_device_msix_dpu_kernels_dev.c`
 - `devemu_pci_device_msix_dpu_main.c`
 - `meson.build`
- `devemu_pci_device_msix/host/`
 - `devemu_pci_device_msix_host_sample.c`
 - `devemu_pci_device_msix_host_main.c`
 - `meson.build`
- `devemu_pci_common.h`; `devemu_pci_common.c`
- `devemu_pci_host_common.h`; `devemu_pci_host_common.c`
- `devemu_pci_type_config.h`

PCI Device DMA

This sample illustrates how the host driver can set up memory for DMA, then the DPU can use that memory to copy a string from the BlueField to the host and from the host to the BlueField.

This sample consists of a host sample and a BlueField sample. It is necessary to follow the [additional sample setup](#) detailed previously.

The BlueField sample logic includes:

1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`.
2. Acquiring the emulated device representor that matches the provided VUID.
3. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).
4. Creating a DMA context to use for copying memory across the host and BlueField.
5. Setting up an mmap representing the host driver memory buffer.
6. Setting up an mmap representing a local memory buffer.
7. Use the DMA context to copy memory from host to BlueField.
8. Use the DMA context to copy memory from BlueField to host.
9. Cleaning up resources.

The host sample logic includes:

1. Initializing the VFIO device with the matching PCIe address and VFIO group.
2. Allocating memory buffer.
3. Mapping the memory buffer to I/O memory. The BlueField can now access the memory using the I/O address through DMA.
4. Copying the string provided by user to the memory buffer.
5. Waiting for the BlueField to write to the memory buffer.
6. Un-mapping the memory buffer.
7. Cleaning up resources.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`
 - `devemu_pci_device_dma/dpu/`
 - `devemu_pci_device_dma_dpu_sample.c`

- `devemu_pci_device_dma_dpu_main.c`
- `meson.build`
- `devemu_pci_device_dma/host/`
 - `devemu_pci_device_dma_host_sample.c`
 - `devemu_pci_device_dma_host_main.c`
 - `meson.build`

DOCA DevEmu Virtio

Note

This library is supported at alpha level; backward compatibility is not guaranteed.

Introduction

DOCA DevEmu Virtio, which is part of the DOCA Device Emulation subsystem, introduces low-level software APIs that provide building blocks for developing and manipulating virtio devices using the device emulation capability of NVIDIA® BlueField®. This subsystem incorporates a core library that handles a common logic for various types of virtio devices, such as virtio-FS. One of its key responsibilities is managing the standard "device reset" procedure outlined in the virtio specification. This core library serves as a foundation for implementing shared functionalities across different virtio device types, ensuring consistency and efficiency in device operations and behaviors.

DOCA provides support for emulating virtio devices over the PCIe bus. The PCIe transport is commonly used for virtio devices. Configuration, discovery, and features related to PCIe (such as MSI-X and PCIe device hot plug/unplug) are managed through the DOCA DevEmu PCI APIs. This modular design enables each layer within the DOCA Device Emulation subsystem to manage its own business logic and facilitates seamless integration with the other layers, ensuring independent functionality and operation throughout the system.

This subsystem also includes device-specific libraries for various virtio device types (e.g., a library for a virtio-FS device).

From the host's perspective, there is no difference between para-virtual, DOCA-emulated, and actual hardware devices. The host uses the same virtio device drivers to operate the device under all circumstances.

Prerequisites

Virtio device emulation is part of the DOCA Device Emulation subsystem. It is, therefore, recommended to read the following guides beforehand:

- [DOCA Device Emulation](#)
- [DOCA DevEmu PCI](#)

Environment

DOCA DevEmu Virtio is supported on the BlueField target only.

The BlueField must meet the following requirements

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

Info

Please refer to the [DOCA Backward Compatibility Policy](#).

Library must be run with root privileges.

Architecture

The DOCA DevEmu Virtio core library provides the following software abstractions:

- Virtio type – extends the PCIe type, represents common/default virtio configurations of emulated virtio devices
- Virtio device – extends the PCIe device, represents an instance of an emulated virtio device
- Virtio IO context – represents a progress context which is responsible for processing virtio descriptors and their associated virtio queues

DOCA DevEmu Virtio library does not provide APIs to configure the entire BAR layout of the virtio device as this configuration is done internally. However, the library offers APIs to

configure some of the registers within the common configuration structure (see [Virtio Device](#)).

Virtio Common Configuration

According to the virtio specification, the common PCIe configuration structure layout is as follows:

```
struct virtio_pci_common_cfg {
    /* About the whole device. */
    le32 device_feature_select; /* read-write */
    le32 device_feature; /* read-only for driver */
    le32 driver_feature_select; /* read-write */
    le32 driver_feature; /* read-write */
    le16 config_msix_vector; /* read-write */
    le16 num_queues; /* read-only for driver */
    u8 device_status; /* read-write */
    u8 config_generation; /* read-only for driver */

    /* About a specific virtqueue. */
    le16 queue_select; /* read-write */
    le16 queue_size; /* read-write */
    le16 queue_msix_vector; /* read-write */
    le16 queue_enable; /* read-write */
    le16 queue_notify_off; /* read-only for driver */
    le64 queue_desc; /* read-write */
    le64 queue_driver; /* read-write */
    le64 queue_device; /* read-write */
    le16 queue_notify_data; /* read-only for driver */
    le16 queue_reset; /* read-write */
};
```

The DOCA DevEmu Virtio core library provides the ability to configure some of the listed registers using the appropriate setters.

Virtio Type

The virtio type extends the PCIe type and describes the common/default configuration of emulated virtio devices, including the common virtio configuration space registers (such as `num_queues`, `queue_size`, and others).

Virtio type is currently read-only (i.e., only getter APIs are available to retrieve information). The following methods can be used for this purpose:

- `doca_devemu_virtio_type_get_num_queues` – for getting the default initial value of the `num_queues` register for the associated virtio devices
- `doca_devemu_virtio_type_get_queue_size` – for getting the default initial value of the `queue_size` register for the associated virtio devices
- `doca_devemu_virtio_type_get_device_features_63_0` – for getting the default initial values of the `device_feature` bits (0-63) for the associated virtio devices
- `doca_devemu_virtio_type_get_config_generation` – for getting the default initial value of the `config_generation` register for the associated virtio devices

The default virtio type is extended by a virtio device's specific type (e.g., virtio-FS type) and cannot be created on demand.

Virtio Device

The virtio device extends the PCIe device. Before using the DOCA DevEmu Virtio device, it is recommended to read the guidelines of [DOCA DevEmu PCI device](#) and [DOCA Core context configuration phase](#).

The virtio device is extended by a virtio-specific device (e.g., virtio FS device) and cannot be created on demand.

Virtio Device Configurations

The virtio device context can be configured to match the application use case and optimize the utilization of system resources.

Mandatory Configurations

The mandatory configurations are as follows:

- `doca_devemu_virtio_dev_set_num_required_running_virtio_io_ctxs` – to set the number of required running virtio IO contexts to be bound to the virtio device context. The virtio device context does not move to `running` state (according to the [DOCA Core context state machine](#)) before having this amount of running virtio IO contexts bound to it.
- `doca_devemu_virtio_dev_event_reset_register` – to register to the virtio device reset event. This configuration is mandatory

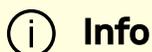
Optional Configurations

The optional configurations are as follows:

- `doca_devemu_virtio_dev_set_device_features_63_0` – to set the values of the `device_feature` bits (0-63). If not set, the default value is taken from the virtio type configuration.
- `doca_devemu_virtio_dev_set_num_queues` – to set the value of the `num_queues` register. If not set, the default value is taken from the virtio type configuration.
- `doca_devemu_virtio_dev_set_queue_size` – to set the value of the `queue_size` register for all virtio queues. If not set, the default value is taken from the virtio type configuration.

Events

DOCA DevEmu Virtio device exposes asynchronous events to notify about sudden changes, according to DOCA Core architecture.



Common events are described in [DOCA DevEmu PCI Device events](#) and in [DOCA Core Event](#).

Reset Event

The reset event allows users to receive notifications whenever the device reset flow is initialized by the device driver. Upon receiving this event, it is guaranteed that no further requests are routed to the user via any associated virtio IO context until the reset flow is completed.

To complete the reset flow the user must:

1. Flush all outstanding requests back to the virtio IO context associated with the request.
2. Perform one of the following:
 - Call `doca_devemu_virtio_dev_reset_complete`.
 - Follow FLR flow:
 1. `doca_ctx_stop` – stop the virtio device with its associated virtio IO contexts and wait until the device and its associated virtio IO contexts transition to `idle` state
 2. `doca_ctx_start` – start the virtio device with its associated virtio IO contexts and wait until the device and its associated virtio IO contexts transition to `running` state

Now, the device and its associated virtio IO contexts should be fully operational again, the device is allowed to route new requests via any associated virtio IO context.

Virtio IO

The virtio IO context extends the [DOCA Core context](#). Before using the DOCA DevEmu Virtio IO, it is recommended to read the guidelines of [DOCA Core context configuration](#)

[phase](#) .

This context is associated with a single DOCA virtio device and is bound to the virtio device context upon start. The virtio IO context is a thread-unsafe object and is progressed by a single [DOCA Core progress engine](#). Usually, users configure a single virtio IO context per BlueField core used by the application service.

The virtio IO context is responsible to route new incoming virtio requests towards the application and to complete handled requests back to the device driver. It can only route requests while in `running` state and when its associated virtio device is also in `running` state .

DOCA DevEmu Virtio-FS

Note

This library is supported at alpha level; backward compatibility is not guaranteed.

Introduction

The DOCA DevEmu Virtio-FS library is part of the DOCA DevEmu Virtio subsystem. It provides low-level software APIs that provide building blocks for developing and manipulating virtio filesystem devices using the device emulation capability of NVIDIA® BlueField® DPUs.

DOCA supports emulating virtio-FS devices over the PCIe bus. The PCIe transport is the common transport used for virtio devices. Configuration, discovery, and features related to PCIe (e.g., MSI-X and PCIe device hot plug/unplug) are managed through the DOCA DevEmu PCI APIs. Configuring common virtio registers and handling generic virtio logic (e.g., virtio device reset flow) is handled by the DOCA Virtio common library. This modular design enables each layer within the DOCA Device Emulation subsystem to manage its own business logic. It facilitates seamless integration with the other layers, ensuring independent functionality and operation throughout the system.

The DOCA Devemu Virtio-FS library efficiently handles virtio descriptors, carrying FUSE requests, sent by the device driver, and translating them into abstract virtio-FS requests which are then routed to the user. This translation process ensures that the underlying device-specific acceleration details are abstracted away, allowing applications to interact with abstracted virtio-FS requests.

Users of this library are responsible for developing a virtio-FS controller, which manages the underlying DOCA Devemu Virtio-FS device alongside an external backend file system which is outside DOCA's scope. The controller application is designed to receive DOCA Virtio-FS requests and process them according to virtio-FS and FUSE specifications, translating FUSE-based commands into the appropriate backend filesystem protocol.

Prerequisites

Virtio-FS device emulation is part of DOCA DevEmu Virtio subsystem. It is, therefore, recommended to read the following guides before proceeding:

- [DOCA Device Emulation](#)
- [DOCA DevEmu PCI](#)
- [DOCA DevEmu Virtio](#)

Environment

DOCA DevEmu Virtio-FS is supported on the BlueField target only. The BlueField must meet the following requirements:

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

Info

Please refer to the [DOCA Backward Compatibility Policy](#).

Note

Library must be run with root privileges.

Perform the following:

1. Configure BlueField to work in DPU mode as described in [NVIDIA BlueField Modes of Operation](#).
2. Enable emulation by running the following on the host or DPU:

```
host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s  
VIRTIO_FS_EMULATION_ENABLE=1
```

3. Configure the number of static virtio-FS physical functions and the number of MSIX for each physical function to expose. This can be done by running the following command on the DPU:

```
host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s  
VIRTIO_FS_EMULATION_NUM_PF=2 VIRTIO_FS_EMULATION_NUM_MSIX=18
```

4. Perform a [BlueField system reboot](#) for the `mlxconfig` settings to take effect.

Note

DOCA does not support hot plugging virtio-FS PF devices into the host PCIe subsystem or SR-IOV for virtio-FS devices.

Architecture

The DOCA DevEmu Virtio-FS library provides the following main software abstractions:

- The virtio-FS type – extends the virtio type; represents common/default virtio-FS configurations of emulated virtio-FS devices
- The virtio-FS device – extends the virtio device; represents an instance of an emulated virtio-FS device
- The virtio-FS IO context – extends the virtio IO context; represents a progress context responsible for processing virtio descriptors, carrying FUSE requests, and their associated virtio queues (e.g., hiprio, request, admin, and notification queues).
- The virtio-FS request

Virtio-FS Feature Bits

According to the virtio specification, a virtio-FS device may report support for `VIRTIO_FS_F_NOTIFICATION` which indicates the ability to handle FUSE notify messages sent via the notification queue.

Note

Currently, DOCA does not support reporting the `VIRTIO_FS_F_NOTIFICATION` feature to the driver.

Virtio-FS Configuration Layout

According to the virtio specification, the virtio-FS configuration structure layout is as follows:

```
struct virtio_fs_config {
    char tag[36];
    le32 num_request_queues;
    le32 notify_buf_size;
};
```

The `tag` and `num_request_queues` fields are always available. The `notify_buf_size` field is only available when `VIRTIO_FS_F_NOTIFICATION` is set.

(i) Note

Currently, there is no support for reporting the `VIRTIO_FS_F_NOTIFICATION` feature to the driver. Therefore, `notify_buf_size` field is not available.

Virtio-FS Type

The virtio-FS type extends the virtio type and describes the common/default configuration of emulated virtio-FS devices, including some of the virtio-FS configuration space registers (e.g., `num_request_queues`).

Currently, the virtio-FS type is read-only (i.e., only getter APIs are available to retrieve information). The following method can be used for this purpose:

- `doca_devemu_vfs_type_get_num_request_queues` – to get the default initial value of the `num_request_queues` register for the associated virtio-FS devices

DOCA supports the default virtio-FS type. To retrieve the default virtio-FS type, users use the following method:

- `doca_devemu_vfs_is_default_vfs_type_supported` – check if the default DOCA Virtio-FS type is supported by the device. If supported:
 - `doca_dev_open` – open supported DOCA device
 - `doca_devemu_vfs_find_default_vfs_type_by_dev` – get the default DOCA Virtio-FS type associated with the device

Virtio-FS Device

The virtio-FS device extends the virtio device. Before using the DOCA DevEmu Virtio-FS device, it is recommended to read the guidelines of [DOCA DevEmu Virtio device](#), [DOCA DevEmu PCI device](#), and [DOCA Core context configuration phase](#).

This section describes how to create, configure, and operate the virtio-FS device.

Virtio-FS Device Configurations

The virtio-FS emulated device might be in several different visibility levels from the host point of view:

- Visible/non-visible to the PCIe subsystem – If the device is visible to the PCIe subsystem, the user is not able to configure PCIe-related parameters (e.g., number of MSI-X vector, `subsystem_id`).
- Visible/non-visible to the virtio subsystem – If the device is visible to the virtio subsystem, the user is not be able to configure virtio-related parameters (e.g., number of queues, `queue_size`).

The flow for creating and configuring a virtio-FS device is as follows:

1. `doca_devemu_vfs_dev_create` – Create a new DOCA DevEmu Virtio-FS device instance.
2. `doca_devemu_vfs_dev_set_tag` – Set a unique tag for the device according to the virtio specification.
3. `doca_devemu_vfs_dev_set_num_request_queues` – Set the number of request queues for the device.
4. `doca_devemu_vfs_dev_set_vfs_req_user_data_size` – Set the user data size of the virtio-FS request. If set, a buffer with this size is allocated for each DOCA DevEmu Virtio-FS on behalf of the user.
5. Configure virtio-related parameters as described in [DOCA Virtio configurations](#).

Note

`doca_devemu_virtio_dev_set_num_queues` should be equal to the number of request queues + 1 (for the `hiprio`

queue) since DOCA does not currently support the virtio-FS notification queue.

6. Configure PCIe-related parameters as described in [DOCA DevEmu PCI configurations](#).
7. `doca_ctx_start` – Start the virtio-FS device context to finalize the configuration phase.
 - The virtio-FS device object follows the DOCA context state machine as described in [DOCA Core context state machine](#)
 - The virtio-FS device context moves to `running` state after the initial number of virtio IO contexts is bound to it and turns to `running` state, as described at [DOCA DevEmu Virtio configurations](#)

At this point, the DOCA Devemu Virtio-FS context is fully operational.

Mandatory Configurations

The following are mandatory configurations:

- `doca_devemu_vfs_dev_set_tag` – set a unique tag for the device

Optional Configurations

The optional configurations are as follows:

- `doca_devemu_vfs_dev_set_num_request_queues` – set the number of request queues for the device. If not set, the default value is taken from the virtio-FS type configuration.
- `doca_devemu_vfs_dev_set_vfs_req_user_data_size` – set the user data size of the virtio-FS request. If not set, user data size defaults to 0.

Virtio-FS Device Events

DOCA DevEmu Virtio-FS device exposes asynchronous events to notify about changes that happen out of the blue, according to the DOCA Core architecture.

Common events are described in [DOCA DevEmu Virtio device events](#), [DOCA DevEmu PCI device events](#) and in [DOCA Core event](#) .

Virtio-FS IO

The virtio-FS IO context extends the Virtio IO Context. To start using the DOCA DevEmu Virtio-FS IO it is recommended to read the guidelines of [DOCA DevEmu Virtio IO](#) and [DOCA Core context configuration phase](#).

This section describes how to create, configure and operate the virtio-FS IO context.

Virtio-FS IO Configurations

The flow for creating and configuring a virtio-FS IO context should be as follows:

1. `doca_devemu_vfs_io_create` – Create a new DOCA DevEmu Virtio-FS IO instance.
2. `doca_devemu_vfs_io_event_vfs_req_notice_register` – Register event handler for incoming virtio-FS requests.
3. `doca_ctx_start` – Start the virtio-FS IO context to finalize the configuration phase. The virtio-FS IO object follows the [DOCA Core context state machine](#). The virtio-FS device context moves to `running` state after the initial number of virtio-FS IO contexts is bound to it and moves to `running` state (as described at [DOCA DevEmu Virtio configurations](#)).

Mandatory Configurations

The following are mandatory configurations:

- `doca_devemu_vfs_io_event_vfs_req_notice_register` – Register event handler for incoming virtio-FS requests is mandatory

Virtio-FS Request

The virtio-FS request object serves as an abstraction for handling requests arriving on virtio-FS queues, including high-priority, request, or notification queues. These requests are initially generated by the device driver through created virtio queues and then routed to the user via a registered event handler, which is set up using `doca_devemu_vfs_io_event_vfs_req_notice_register`, on the associated virtio IO context. This event handler, issued by the DOCA Virtio FS library, ensures that users can receive and process virtio-FS requests effectively within their application. Once the event handler is called, the ownership of the virtio-FS request and the associated request user data move to the user. The request ownership moves back to the associated virtio IO context once it is completed by the user by calling `doca_devemu_vfs_req_complete`.

The following APIs operate a virtio-FS request:

1. `doca_devemu_vfs_req_get_datain` – Get a DOCA buffer representing the data-in of the virtio-FS request. This DOCA buffer represents the host memory for the device-readable part of the request according to the virtio specification.
2. `doca_devemu_vfs_req_get_dataout` – Get a DOCA buffer representing the data-out of the virtio-FS request. This DOCA buffer represents the host memory for the device-writable part of the request according to the virtio specification.
3. `doca_devemu_vfs_req_complete` – Complete the virtio-FS request. The associated virtio-FS IO context completes the request toward the device driver according to the virtio-FS specification.

Discovery

Emulated virtio-FS PCIe functions are represented by a `doca_devinfo_rep`. To find the suitable `doca_devinfo_rep` that is used as the input parameter for `doca_devemu_vfs_dev_create`, users should first discover the existing device representors using the below:

1. `doca_devinfo_create_list` – Get a list of all DOCA devices.
2. `doca_devemu_vfs_is_default_vfs_type_supported` – Check whether the device can manage device associated to virtio-FS type.

3. If supported:

1. `doca_dev_open` – Get an instance of the DOCA device that can be used as virtio-FS emulation manager.
2. `doca_devemu_vfs_find_default_vfs_type_by_dev` – Get the default virtio-FS device type.
3. `doca_devemu_vfs_type_as_pci_type` – Cast virtio-FS type to PCIe type.
4. `doca_devemu_pci_type_rep_list_create` – Create a list of all available representor devices for the virtio-FS type.

4. At this point, the user can choose the preferred representor device, open it using `doca_dev_rep_open`, and proceed with the flow described in section "[Virtio-FS Device Configurations](#)".

Initialization

This section describes the initialization flow of a DOCA DevEmu Virtio-FS device and one or more DOCA DevEmu Virtio-FS IO contexts (4 in this example). In this procedure, the user sets up and prepares the environment before starting to receive control path events (from the virtio-FS device context) and IO requests (from the virtio-FS IO contexts). During initialization, the user should configure various essential components to ensure correct behavior.

The user should perform the following:

1. Choose 4 Arm cores to run the application threads on.
2. Create 4 [DOCA Core progress engine](#) (PE) objects (`pe1`, `pe2`, `pe3`, `pe4`).
3. Find the suitable representor device according to the [Discovery](#) flow or any other method.
4. Create, configure, and start a new virtio-FS device according to the virtio-FS device [configuration flow](#). Assume `pe1` is associated with the virtio-FS device and `doca_devemu_virtio_dev_set_num_required_running_virtio_io_ctxs` is set to 4.
5. Create, configure, and start 4 new virtio-FS IO contexts according to the virtio-FS IO [configuration flow](#). Assume `pe1`, `pe2`, `pe3`, and `pe4` are associated with each

of the 4 virtio-FS IO contexts respectively.

6. At this point, the 4 virtio-FS IO contexts transition to `running` state, followed by the virtio-FS device context transitioning to `running` state.

i Note

During the initialization flow, it is guaranteed that no virtio/PCIe control path or IO path events are generated until the virtio-FS device has transitioned to `running` state.

Teardown

This section describes the teardown flow of DOCA DevEmu Virtio-FS device and one or more DOCA DevEmu Virtio-FS IO contexts (4 in this example). In this procedure, the user cleans all the resources allocated in the initialization flow and all the outstanding events and requests.

The user should perform the following:

1. Start the teardown flow by calling `doca_ctx_stop`. This causes the DOCA Virtio-FS device context to transition to `stopping` state. It is guaranteed that no virtio/PCIe control path events is generated during this state.
2. Call `doca_ctx_stop` for any DOCA Virtio-FS IO context. This causes the DOCA Virtio-FS IO context to transition to `stopping` state. It is guaranteed that no IO path events are generated during this state.
3. Flush all outstanding virtio-FS requests to the associated virtio-FS IO contexts by calling `doca_devemu_vfs_req_complete`. Upon completing all the requests associated with a virtio-FS IO context, the DOCA Virtio-FS IO context transitions to `idle` state.
4. At this point, it is safe to destroy the virtio-FS IO context by calling `doca_devemu_vfs_io_destroy`. Destroying a virtio-FS IO context not in `idle` state will fail.

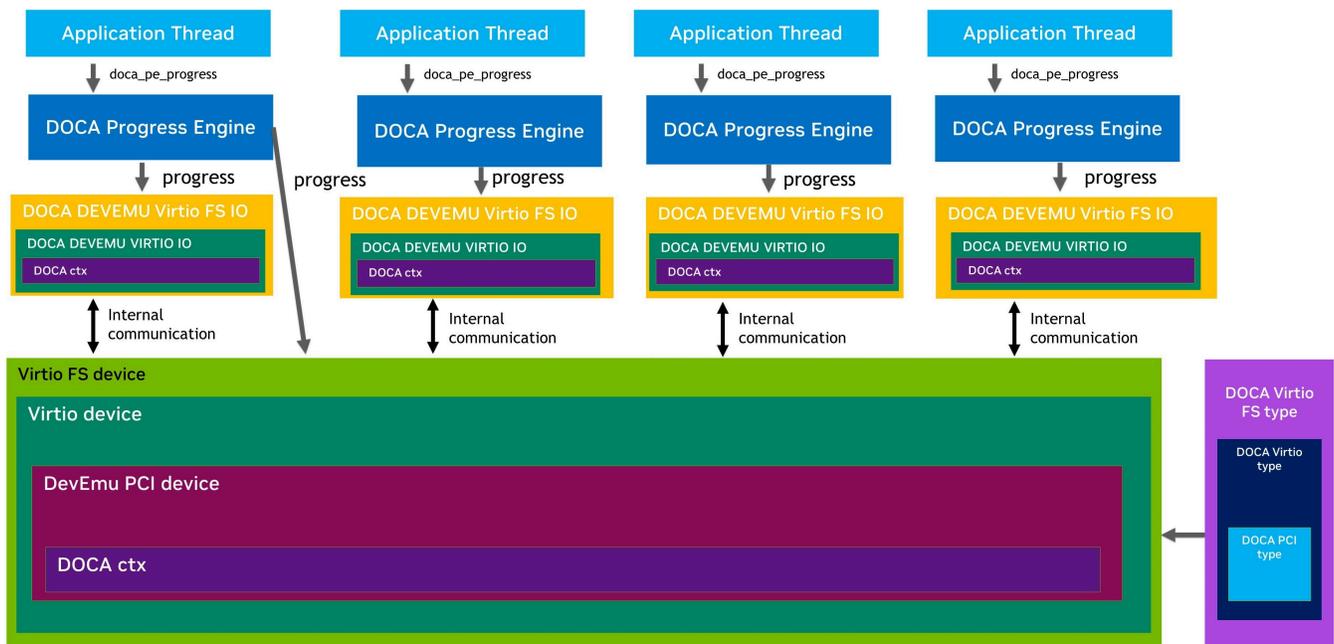
- Once all 4 virtio-FS IO contexts associated with the virtio-FS device transition to `idle` state, the DOCA Virtio-FS device context transitions to `idle` state as well.
- At this point, it is safe to destroy the virtio-FS device context by calling `doca_devemu_vfs_dev_destroy`. Destroying a virtio-FS device context not in `idle` state will fail.

Execution Phase

This section describes execution on BlueField Arm cores using several [DOCA Core PE](#) objects (one per core):

- Choose 4 Arm cores to run the application threads on.
- Create 4 DOCA Core PE objects. The application threads should periodically call `doca_pe_progress` to advance all DOCA contexts associated with the PE.
- Create, configure, and start the DOCA Virtio-FS device.
- Create, configure, and start 4 DOCA Virtio-FS IO contexts.

The progress of DOCA Virtio-FS objects is illustrated by the following diagram:



Control Path

The DOCA Virtio-FS device context extends the DOCA Virtio device context (which extends the DOCA PCIe device context). This means that the DOCA Virtio-FS device control path is comprised by all the object it extends (i.e., DOCA Context, DOCA DevEmu PCI device, and DOCA DevEmu Virtio device).

The following events can be triggered by a virtio-FS device context:

- DOCA context state change events as described in [DOCA Core context state machine](#) and in [DOCA DevEmu PCI state machine](#)
- DOCA DevEmu PCI [FLR flow](#)
- DOCA DevEmu Virtio [reset flow](#)

The DOCA Virtio-FS IO context extends the DOCA Virtio IO context (which extends the DOCA core context). This means that the DOCA Virtio-FS IO context control path is comprised by all the object it extends (i.e., DOCA Context and DOCA DevEmu Virtio IO).

The following events can be triggered by a Virtio-FS IO context:

- DOCA context state change events as described in [DOCA Core context state machine](#)

In addition to the control path events, the DOCA DevEmu Virtio-FS IO context also produces IO path events as described in [IO path](#).

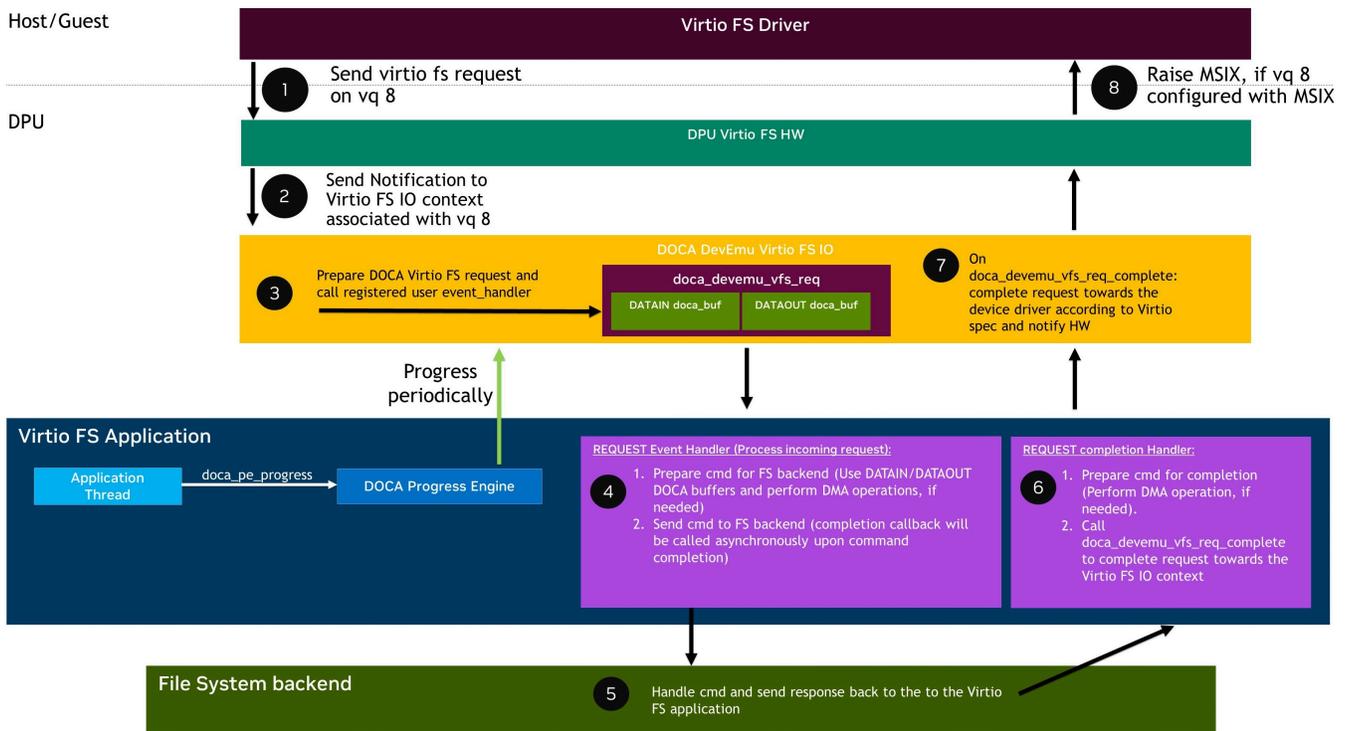
IO Path

This section describes the flow for a single virtio-FS request sent by the device driver until its completion.

It is assumed that the user properly configured an event handler for an incoming virtio-FS request as explained in section "[Virtio-FS IO Configurations](#)".

It is also assumed that the user is familiar with the virtio-FS specification and has the ability to perform DMA operations to/from the host using [DOCA DMA](#) or any other suitable method.

The DOCA virtio-FS flow is illustrated in the following diagram:



Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order

to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

© Copyright 2025, NVIDIA. PDF Generated on 06/05/2025