STORAGE DEVELOPER CONFERENCE

SDC 21

BY Developers FOR Developers

Virtual Conference
September 28-29, 2021

QEMU NVMe Emulation

What's New

Presented by

Klaus Jensen	Samsung Electronics (SSDR, Denmark)
Padmakar Kalghatgi	Samsung Electronics (SSIR, India)
Naveen Nagar	Samsung Electronics (SSIR, India)

Disclaimer

This presentation and/or accompanying oral statements by Samsung representatives collectively, the “Presentation” is intended to provide information concerning the SSD and memory industry and Samsung Electronics Co., Ltd. and certain affiliates (collectively, “Samsung”).While Samsung strives to provide information that is accurate and up-to-date, this Presentation may nonetheless contain inaccuracies or omissions. As a consequence, Samsung does not in any way guarantee the accuracy or completeness of the information provided in this Presentation.

This Presentation may include forward-looking statements, including, but not limited to, statements about any matter that is not a historical fact; statements regarding Samsung’s intentions, beliefs or current expectations concerning, among other things, market prospects, technological developments, growth, strategies, and the industry in which Samsung operates; and statements regarding products or features that are still in development. By their nature, forward-looking statements involve risks and uncertainties, because they relate to events and depend on circumstances that may or may not occur in the future. Samsung cautions you that forward looking statements are not guarantees of future performance and that the actual developments of Samsung, the market, or industry in which Samsung operates may differ materially from those made or suggested by the forward-looking statements in this Presentation. In addition, even if such forward-looking statements are shown to be accurate, those developments may not be indicative of developments in future periods.

Outline

- Emulated NVMe Introduction
- New Features, including
  - CMB/PMR enhancements
  - Namespace Management and Endurance Group support
  - End-to-End Data Protection
  - Namespace Sharing and Reservations
  - NVMe-MI support

- Rounding up

Emulated NVMe Introduction

Emulated NVMe

- There are several emulated and virtual NVMe controllers available
  - QEMU Emulated PCIe-based NVMe controller *(the topic of this talk)*
  - SPDK virtual controller *(initially intended for fabrics, but check out the exciting work on vfio-user by the SPDK team and Nutanix)*
  - Linux Kernel `nvmet` *(virtual fabrics target)*

Emulated NVMe

- Feature and compliance wise, the QEMU emulated NVMe controller is the most complete
  - Not intended to provide high performance, **compliance**, **correctness** and *(to a lesser degree)*, **feature completeness**, are the primary concerns
  - As Keith *(maintainer and creator of the device)* put it…
    - *“Performance improvements are welcome as long as we don’t harm protocol correctness or maintainability”*

A note on “Feature Completeness”

- To fake or not to fake a feature
  - We try not to bloat the device; rely on system support whenever possible
  - Some features are useful for compliance testing, but may not be appropriate for upstream QEMU
    - If the feature requires extensive “faking” (i.e. Write Uncorrectable) and is of limited use to driver and application developers, it will not be merged and maintained upstream
    - Notable exceptions
      - Zoned Namespace (zones are faked, but the value of the feature is very clear)
      - End-to-End Data Protection (compute-heavy, but value is high)
- Some features mentioned in this talk are not upstream, but available as patches on `github.com/OpenMPDK/qemu-nvme-compliance`

New Features

DULBE Support

- A logical block that has never been written to, or which has been deallocated, is called a “deallocated or unwritten logical block”
  - The NVMe **Error Recovery** feature allows host software to detect such blocks when reading (enabled with the Set Features command)

- In NVMe, logical blocks **may** be marked deallocated by
  - Write Zeroes *(always guarantees that zeroes will be read back)*
  - Dataset Management with Deallocate *(**advisory**; the device may take no action what so ever)*

DULBE Support

- The QEMU block layer provides block status information through a set of “block status flags”
  - Reported by the `bdrv_block_status()` function
  - Discards (`pdiscard`) and write zeroes (`pwrite_zeroes`) **may** explicitly mark blocks as zeroed
    - Adds the `BDRV_BLOCK_ZERO` flag

Deallocation through Write Zeroes

- Multiple factors affect **when** blocks are assigned the `BDRV_BLOCK_ZERO` flag as the result of **write zeroes** (`pwrite_zeroes`)
  - The underlying file system logical block size
  - The block device format (`raw`, `qcow2`, etc.)
    - `raw` relies on “hole punching”
    - `qcow2` manages allocation status in metadata
  - The block device `discard` parameter

		Size of `pwrite_zeroes()`		
Format	**Discard**	**512B**	**4KiB**	**64KiB**
qcow2	`ignore`	n	n	y
qcow2	`unmap`	n	n	y
raw	`ignore`	n	y	y
raw	`unmap`	n	y	y

Deallocation through Write Zeroes

- Multiple factors affect **when** blocks are assigned the `BDRV_BLOCK_ZERO` flag as the result of **write zeroes** (`pwrite_zeroes`)
  - The underlying file system logical block size
  - The block device format (`raw`, `qcow2`, etc.)
    - `raw` relies on “hole punching”
    - `qcow2` manages allocation status in metadata
  - The block device `discard` parameter

Holes can only be “punched” in granularity of the file system logical block size

		Size of `pwrite_zeroes()`		
Format	**Discard**	**512B**	**4KiB**	**64KiB**
qcow2	`ignore`	n	n	y
qcow2	`unmap`	n	n	y
raw	`ignore`	n	y	y
raw	`unmap`	n	y	y

Deallocation through Write Zeroes

- Multiple factors affect **when** blocks are assigned the `BDRV_BLOCK_ZERO` flag as the result of **write zeroes** (`pwrite_zeroes`)
  - The underlying file system logical block size
  - The block device format (`raw`, `qcow2`, etc.)
    - `raw` relies on “hole punching”
    - `qcow2` manages allocation status in metadata
  - The block device `discard` parameter

The `qcow2` format manages allocation in “clusters” of 64 KiB

		Size of `pwrite_zeroes()`		
Format	**Discard**	**512B**	**4KiB**	**64KiB**
qcow2	`ignore`	n	n	y
qcow2	`unmap`	n	n	y
raw	`ignore`	n	y	y
raw	`unmap`	n	y	y

Deallocation through DSM

- The implementation of DSM uses discards *(which **are** advisory)*
- The same factors affect when blocks are marked zeroed by **discards** (`pdiscard`)
  - However, ‘discard=ignore’ completely disables the operation

		Size of `pdiscard()`		
Format	**Discard**	**512B**	**4KiB**	**64KiB**
qcow2	`ignore`	n	n	n
qcow2	`unmap`	n	n	y
raw	`ignore`	n	n	n
raw	`unmap`	n	y	y

DULBE Support

- Thus, the preferable setup for utilizing DULBE support is
  - an image in `raw` format, *and*
  - `logical_block_size` set to the native logical block size of the underlying file system (or block device)

- Depending on the configuration, the device will set appropriate values in Namespace Preferred Deallocate Granularity (**NPDG**) and Alignment (**NPDA**) fields

Namespace Types and Zones

- Merged series to support
  - Namespace Types (TP 4056) – (Niklas)
    - Namespace type specific identify structures (common, NVM and Zoned specific)
  - Zoned Namespaces (TP 4053) – (Dmitry)
    - Emulated support for zones
    - Zone Append

- See talks by **Dmitry** and **Klaus** for details (**SDC ‘20**)

Namespace Sharing

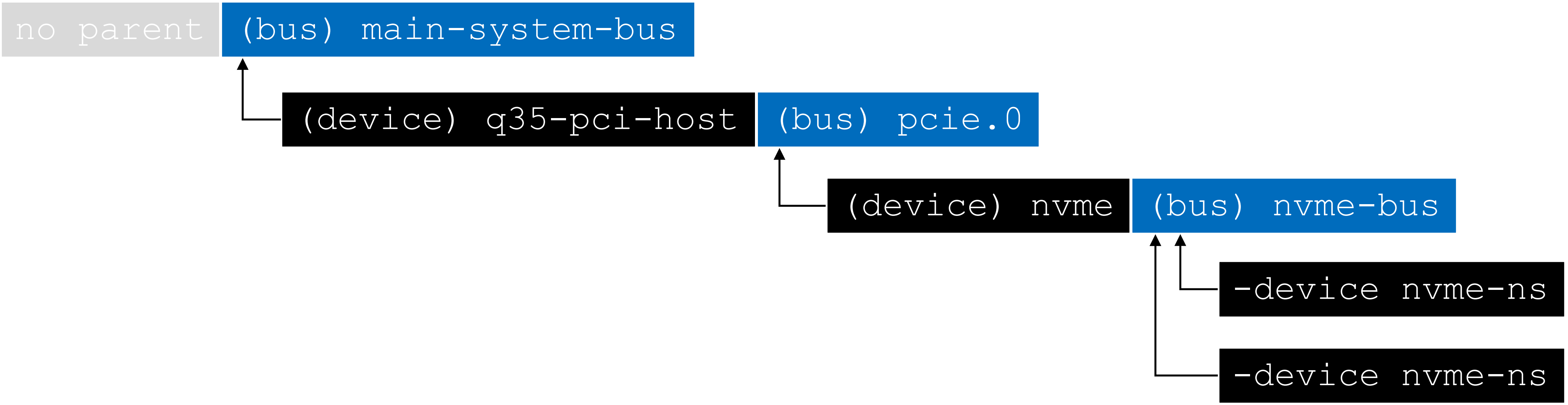
- A **Shared Namespace** refers to a **Namespace** that may be accessed concurrently by two or more **Controllers** within the same **NVM Subsystem**
  - Quite useful for testing advanced drivers and **multi-path I/O**

- Requires adding the concept of an **NVM Subsystem**
  - Implemented as a "**bus-less**" and "**un-rooted**" `-device`
    - Added `subsys` link parameter on controller device to plumb it to a subsystem

Aside – QEMU Model Relationships

- QEMU offers two standard ways of “wiring up” device models
  - Explicitly through a special named “link” property
    - Unidirectional “reference-style” relationship
  - Implicitly through “device busses” *(explicitly through a “bus” parameter)*
    - Bidirectional relationship – devices know their parent bus and busses know its children
    - Strict tree – devices can create **one or more** busses but a device can be attached **to one**, **and only one**, parent bus
    - A lot of additional functionality
      - Automatic address assignment
      - Automatic clean-up

QEMU Emulated NVMe Devices Wiring

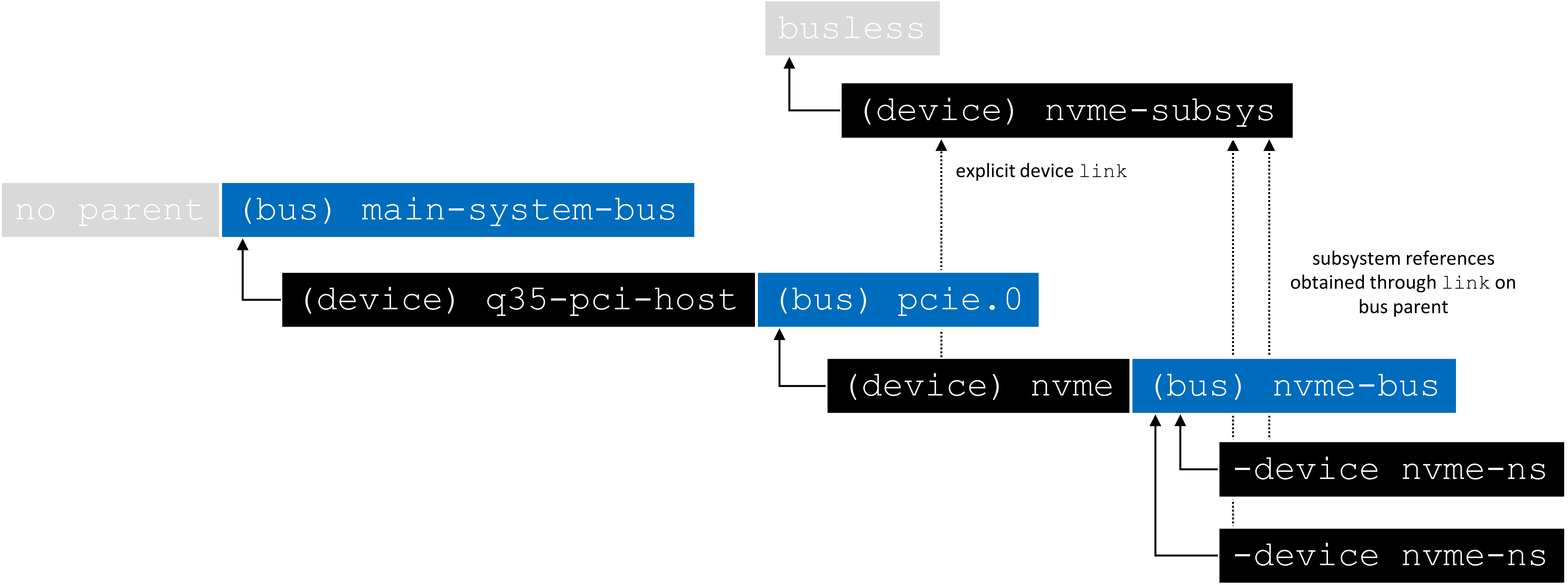


Automatic device “unrealization”

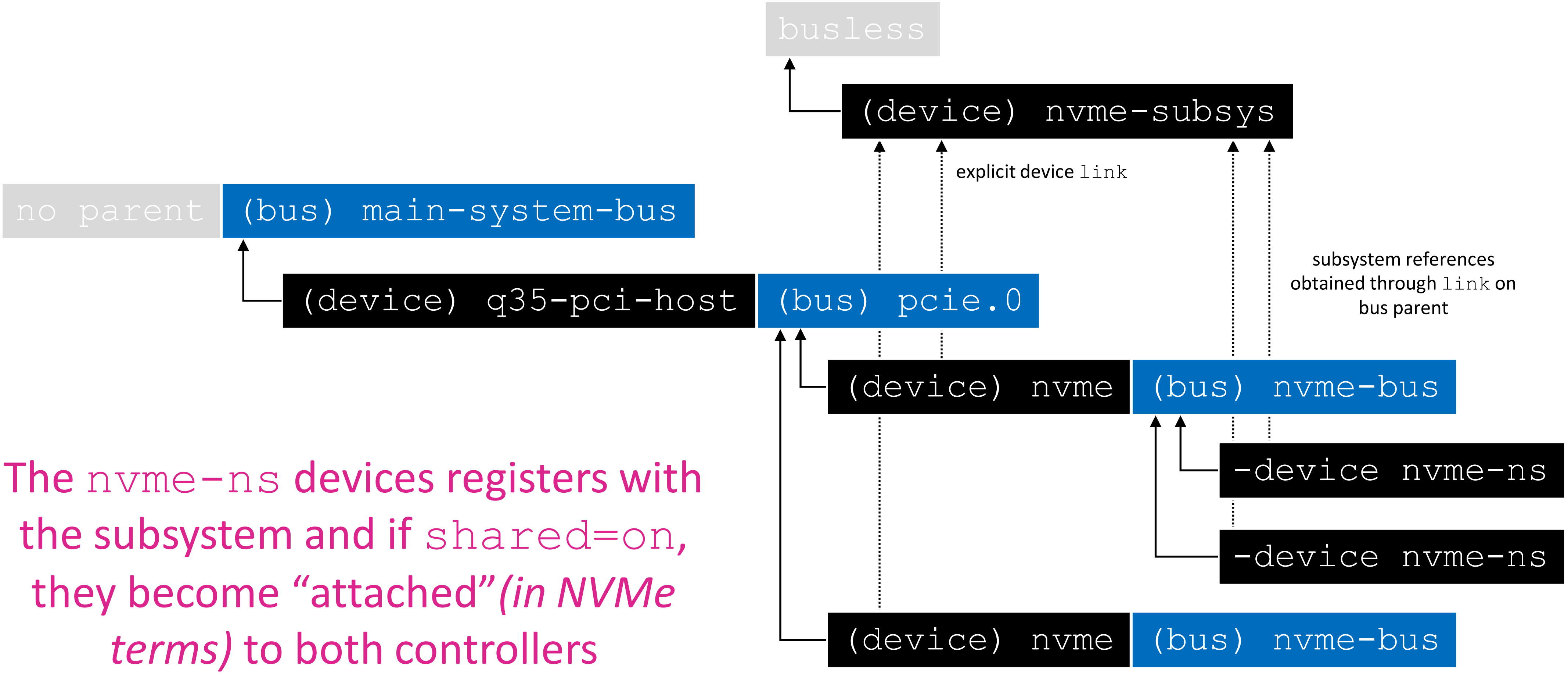
- If a device is removed, all devices on child busses are recursively unrealized
  - This design made a lot of sense when multiple namespace support was merged *(waaaay back in 2019)*

  - And this *should* be a good thing…
  - … but if we add subsystems and shared namespace functionality to the mix…
  - … not so much

Adding an NVM Subsystem

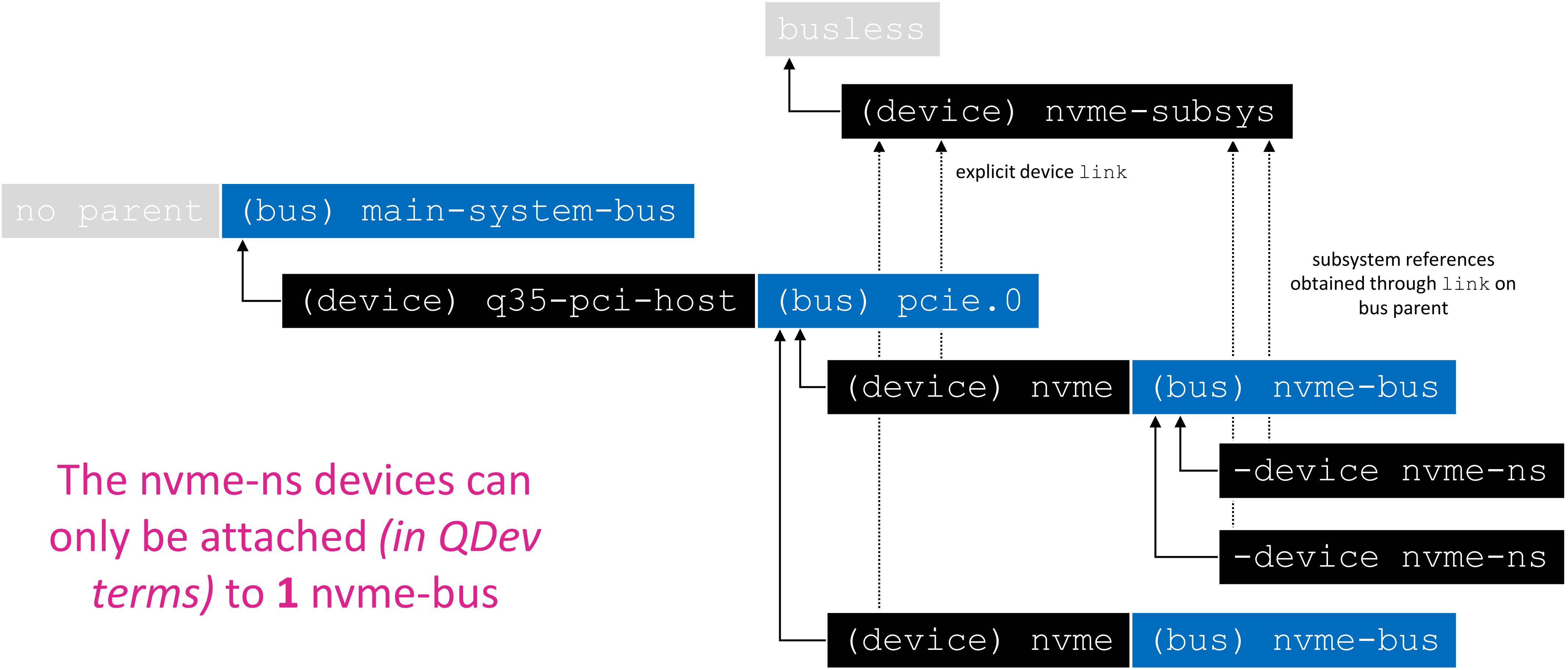


Adding another NVMe Controller



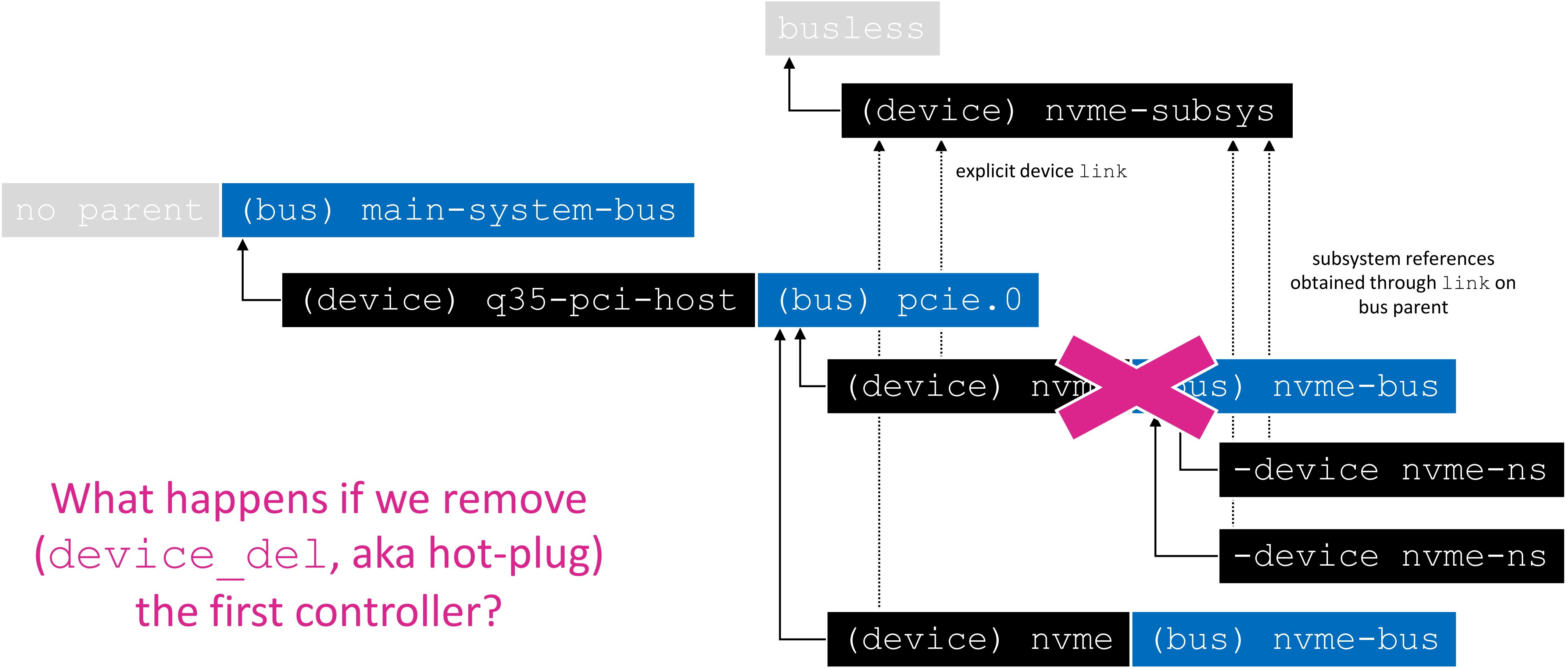
The `nvme-ns` devices registers with the subsystem and if `shared=on`, they become "attached" *(in NVMe terms)* to both controllers

Adding another NVMe Controller



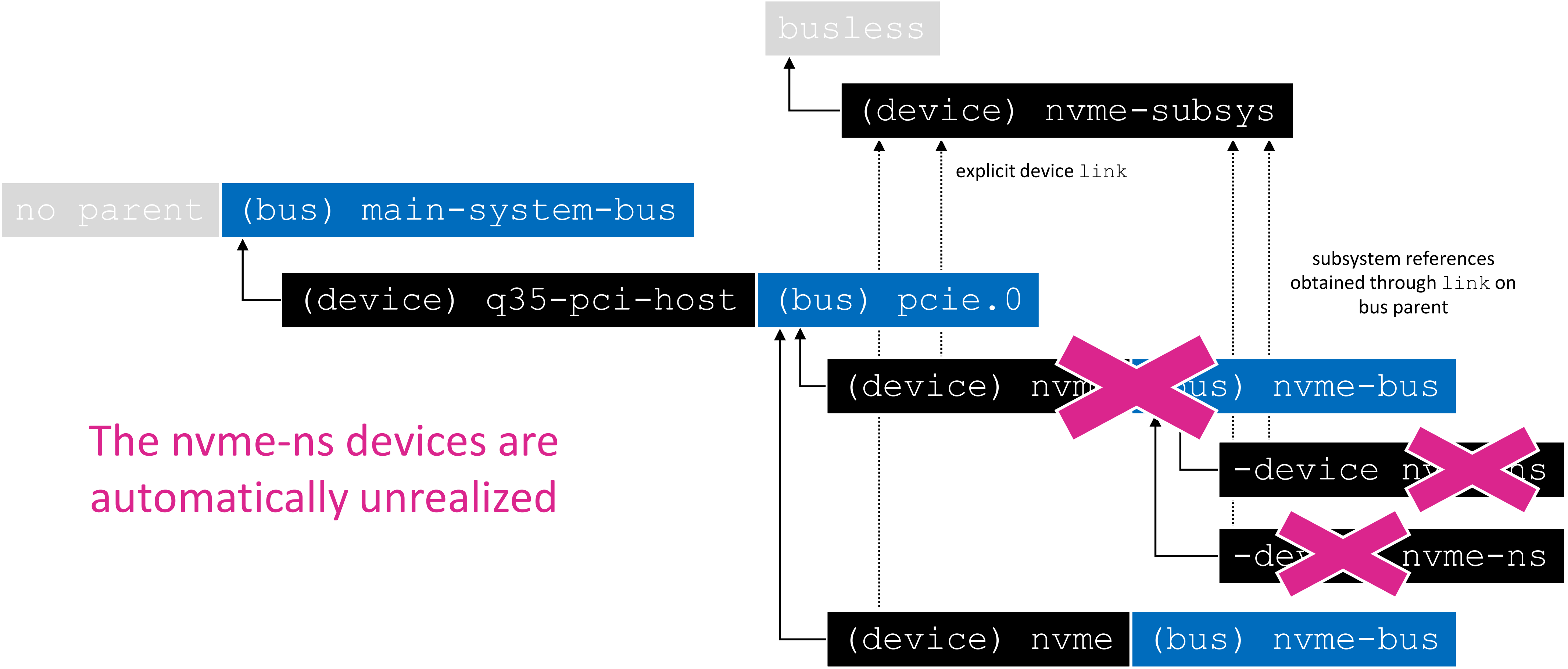
The nvme-ns devices can only be attached *(in QDev terms)* to **1** nvme-bus

Removing an NVMe Controller



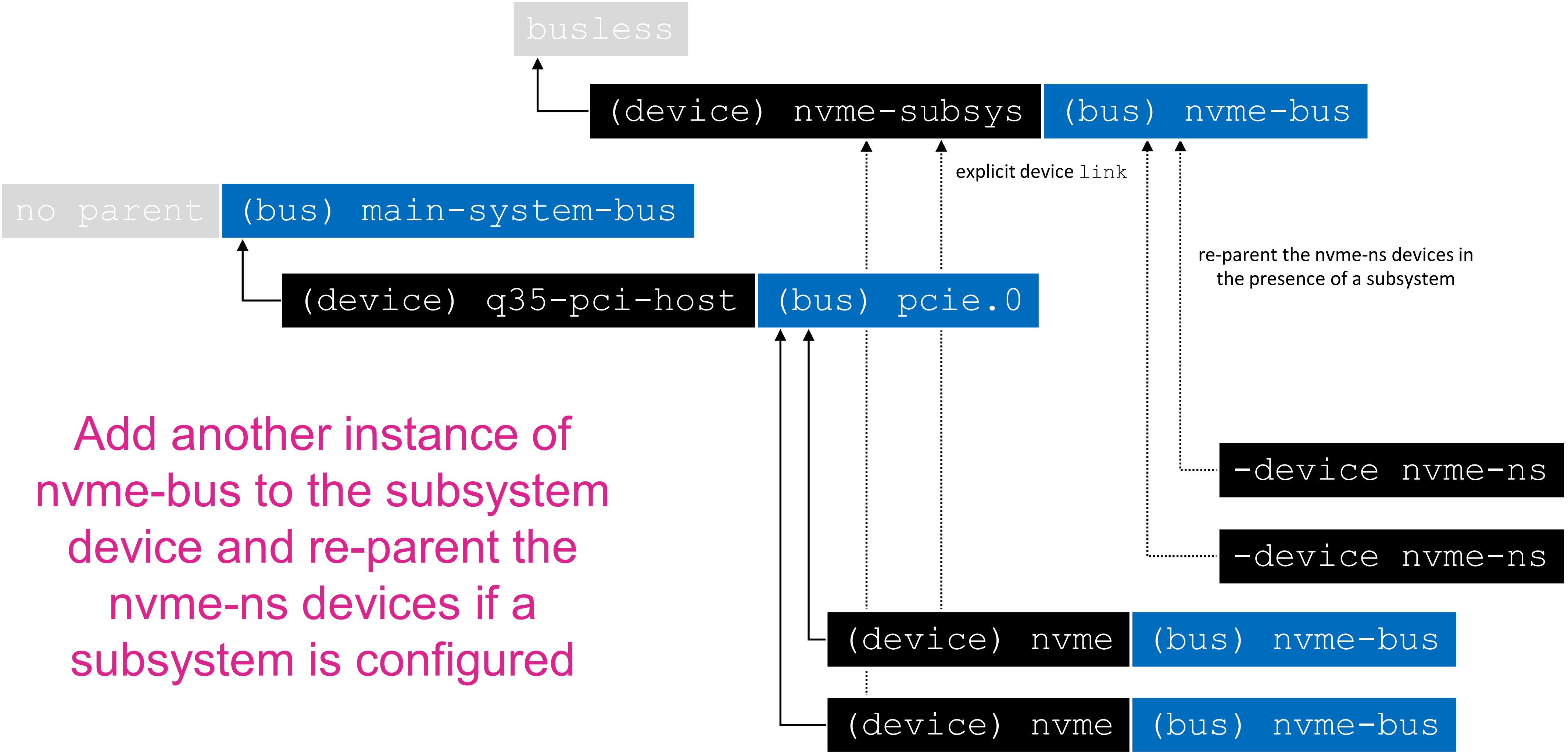
What happens if we remove (device_del, aka hot-plug) the first controller?

Removing an NVMe Controller



The nvme-ns devices are automatically unrealized

Fixing the mess



Add another instance of nvme-bus to the subsystem device and re-parent the nvme-ns devices if a subsystem is configured

Rethining the model

- ***What if**…* Subsystems and Namespaces were **not** modeled as *devices*
  - Neither subsystems or namespaces expose virtual hardware (e.g. memory, IRQs,…) to the guest
  - Fundamentally, they are just *concepts* in a device model that happen to benefit design-wise from being independent devices

- User creatable objects (`-object`) might be more appropriate

The hw/nvme “devpocalypse”

- RFC patch series posted mid September
  - 13 files changed, 2612 insertions(+), 1164 deletions(-)
  - Ouch…
- Major refactoring of the `hw/nvme` subsystem
  - Introduces NVM Subsystems and Namespaces as user creatable objects

Goodbye **implicit bus’ing** – Hello **explicit topology**!

The hw/nvme “devpocalypse”

- Series goals
  - Introduces a new experimental controller device (`x-nvme-ctrl`)
  - Introduces new experimental user creatable objects
    - `x-nvme-ns-{nvm,zoned}`
    - `x-nvme-subsystem`
  - Exploits QEMU Object Model inheritance
    - The `x-nvme-ns` abstract object provides the base implementation of NVMe namespace types
    - The `x-nvme-ns-zoned` derives from the `x-nvme-ns-nvm` object
  - Retains backwards compatibility by keeping the existing devices around
    - Uses the new object code internally, **no code duplication**
    - Deprecate the subsystem and namespace devices as the experimental objects stabilizes

The hw/nvme “devpocalypse”

- Perks of introducing brand new models
  - Easy clean-up of some confusing device parameters
    - Controller
      - `msix_qsize → max-irq-vectors`
      - `aer_max_queued → max-aer-retention`
      - Remove `num_queues`, `use-intel-id`
    - Namespace
      - Fix Simple Copy related parameters that should have been defined in bytes and not LBAs
      - Remove the `eui64-default` compatibility parameter
      - `detached`, `shared → attach-to`
    - Subsystem
      - `nqn → subnqn`

The hw/nvme “devpocalypse”

- Set up a subsystem

```
-object x-nvme-subsystem,id=subsys-1
```

- Adding controllers

```
-device x-nvme-ctrl,id=ctrl-1,subsys=subsys-1
-device x-nvme-ctrl,id=ctrl-2,subsys=subsys-1
-device x-nvme-ctrl,id=ctrl-3,subsys=subsys-1
```

- Adding namespaces and attach to specific controllers

```
-object x-nvme-ns-nvm,id=ns-nvm-1,subsys=subsys1, \
    attached-to=ctrl-1, \
    attached-to=ctrl-3
-object x-nvme-ns-zoned,id=ns-zoned-1,subsys=subsys1, \
    attached-to=all
```

STORAGE DEVELOPER CONFERENCE

SDC 21

BY Developers FOR Developers

Virtual Conference
September 28-29, 2021

CMB/PMR Enhancements

CMB/PMR Enhancements

- NVMe 1.4 introduces major enhancements with regards to CMB/PMR.
- Host driver needs to explicitly configure the CMB/PMR BAR Addresses in CMBMSC/PMRMSC fields.
- This helps in avoiding DMA misrouting from the Guest OS.
- QEMU has implemented the above enhancements meant for NVMe 1.4.
- QEMU emulates the CMB/PMR region by allocating a memory region on the Host OS.

CMB/PMR Enhancements

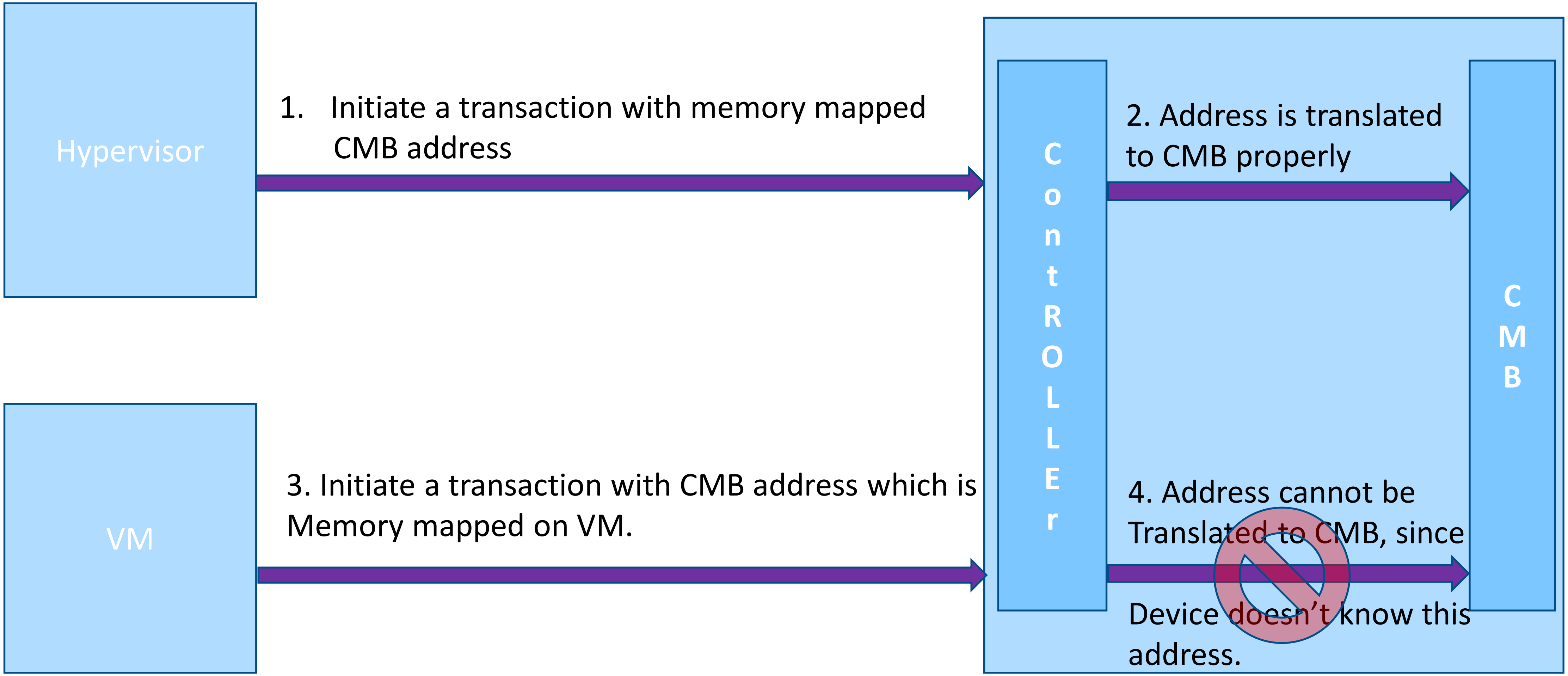
- QEMU has handled invalid use of CMB in case of certain commands when host tries to write data into CMB.

- The Persistent Memory Region (PMR) is an optional region of general purpose PCI Express read/write persistent memory.

- Users can specify as part of launch command for QEMU the required size of CMB buffer and memory backend file for PMR.

..continued

DMA misrouting explained



1. Initiate a transaction with memory mapped CMB address

2. Address is translated to CMB properly

3. Initiate a transaction with CMB address which is Memory mapped on VM.

4. Address cannot be Translated to CMB, since Device doesn't know this address.

..continued

To configure the CMB\PMR address, the Guest must first configure the CRE field (Capabilities Register Enable). After which the QEMU will configure CMBSZ and CMBLOC parameters.

Once the CRE is configured, the Guest can configure the CBA (Controller Base Address) and CMSE (Controller Memory Space Enable).

If the CBA field is valid i.e the address falls within the range specified and CMB’s CBA doesn’t overlap with the PMR and vice versa then then CBAI (Controller Base Address Invalid) field will be set to ‘0’ otherwise it will be set to ‘1’.

STORAGE DEVELOPER CONFERENCE

SDC 21

BY Developers FOR Developers

Virtual Conference
September 28-29, 2021

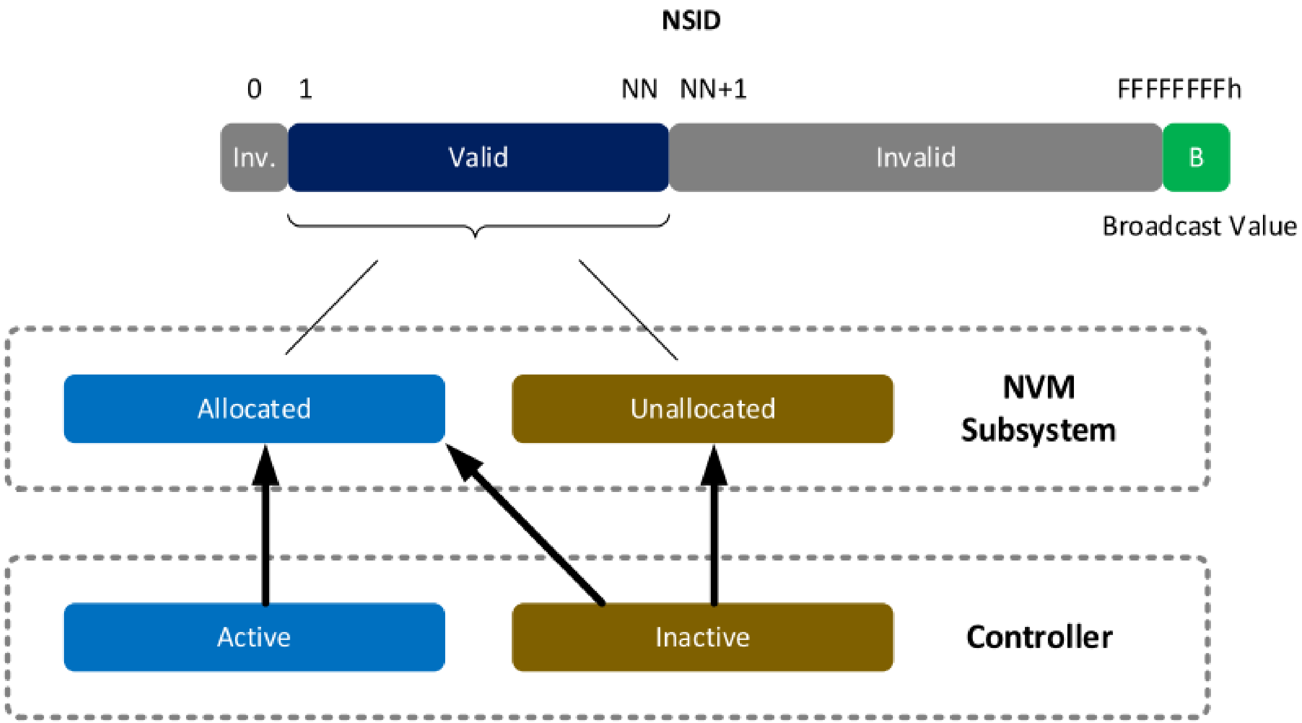
Namespace Management

Understanding NVMe Namespaces

- In NVMe subsystem, a namespace is a quantity of non-volatile memory storage that may be formatted into logical blocks.
- A namespace ID (NSID) is an identifier used by a controller to provide access to a namespace.
- Namespace Management provides an interface for the host to manage the multiple namespaces .
- Supports Namespace creation and deletion.
- Supports Namespace sharing across controllers in Subsystem.

NSID Types

Figure 348: NSID Types



- Fig: NSID Types (Ref: https://nvmexpress.org/)

Namespace Creation

- At run time, namespace can be formatted with the specified attributes
  - FLBAS (LBA data size and Metadata size combinations)
  - Namespace Size (NSZE)
  - End-to-end Data Protection Type Settings (Type 1, or, 2 or 3)
  - Namespace Sharing Capabilities (NMIC)

Design Implementation in QEMU

- Add N number of nvme-ns devices and only allocated namespaces i.e size of the underlying drive is non zero will show up in guest vm.
- A new parameter `tnvmcap` is introduced to ensure that combined capacity of all the created namespace should not exceed `tnvmcap`
- Size of underlying drive for unallocated namespaces is 0.
- Namespace creation will grab an unallocated namespace and initialize it with a certain size that host specifies. We use block truncate to change the block device size from 0 to NSZE provided by host.

Example QEMU invocation

```
qemu-system-x86_64 \
  … \
  -device nvme-subsys,id=subsys0,tnvmcap=8
  -device nvme,id=nvme0,serial=ctrl1,subsys=subsys0 \
  -device nvme,id=nvme1,serial=ctrl2,subsys=subsys0 \
  -drive id=ns1,file=ns1.img \
  -device nvme-ns,nsid=1,bus=nvme0,drive=ns1 \
  -drive id=ns2,file=ns2.img \
  -device nvme-ns,nsid=2,bus=nvme0,drive=ns2\
```

EEDP

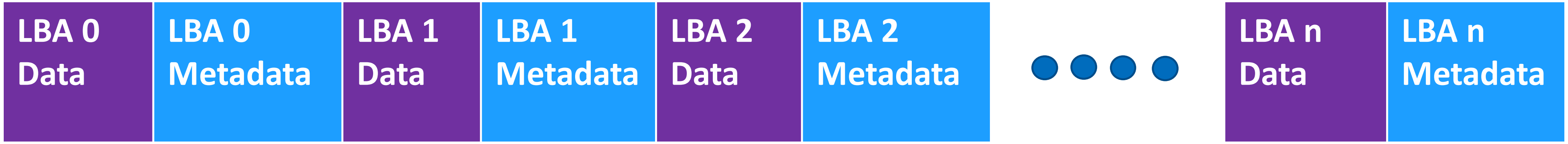
End-to-End Data Protection (EEDP)

NVMe provides support for metadata for the logical blocks. These metadata can be used for data protection of the logical block.

The metadata can be of varying length from 8 to 128 bytes whereas the Protection Information , as per NVMe 1.4 specification, in the metadata can be of only 8 bytes.

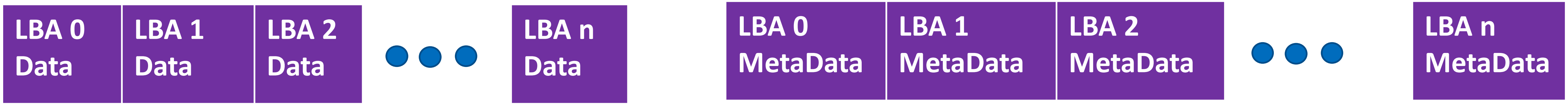
Nvme specification provides two main types of metadata i.e DIF (Data Integrity Field) and Data Integrity Extension (DIX). In the former (DIF), the metadata is contiguous to the logical block whereas in the latter (DIX) it is separate from the logical block.

.. continued



LBA 0 Data	LBA 0 Metadata	LBA 1 Data	LBA 1 Metadata	LBA 2 Data	LBA 2 Metadata	● ● ● ●	LBA n Data	LBA n Metadata

DIF



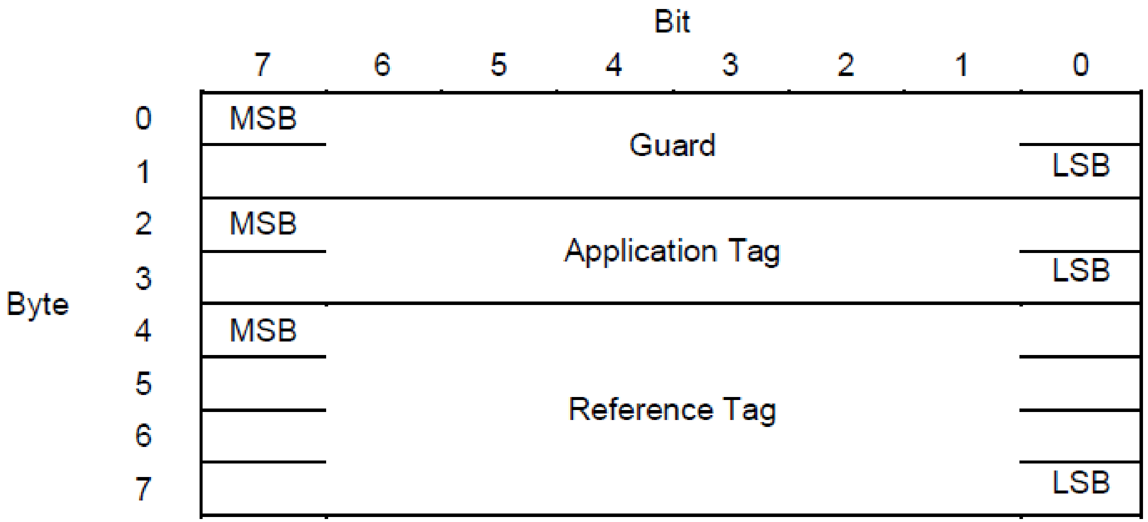
LBA 0 Data	LBA 1 Data	LBA 2 Data	● ● ●	LBA n Data

LBA 0 MetaData	LBA 1 MetaData	LBA 2 MetaData	● ● ●	LBA n MetaData

DIX

.. continued

As mentioned, Protection Information consists of 8 bytes of data of which 2 bytes is the Guard tag, 2 bytes is the Application tag and the rest 4 bytes is the Reference tag. It is as shown below.



Byte	Bit 7	6	5	4	3	2	1	0
0	MSB	Guard						
1								LSB
2	MSB	Application Tag						
3								LSB
4	MSB	Reference Tag						
5								
6								
7								LSB

Guard tag -> CRC calculation over the LBA data

Reference Tag -> Contains the logical block number which the metadata represents to prevent out-of-order data.

.. continued

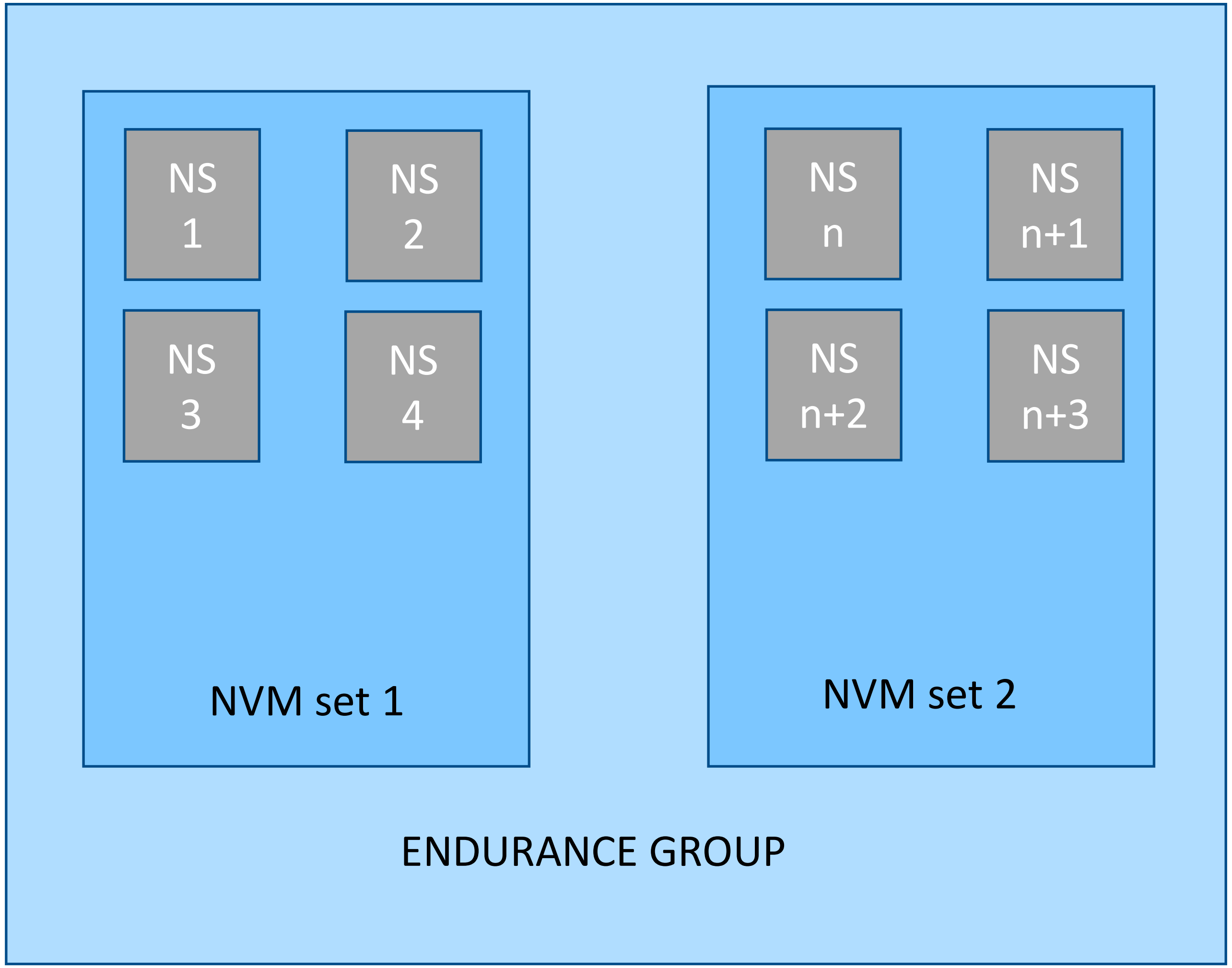
To support DIF/DIX format and to store data, QEMU provides block image files.

Since DIF stores the block data and metadata in contiguous manner one nvme image is sufficient but as DIX supports separate metadata , qemu creates a different block image for the metadata.

QEMU provides EEDP support for Write, Read, Compare, Write Zeros, Copy and Verify Commands

Endurance Groups

Endurance Group



NS 1

NS 2

NS 3

NS 4

NVM set 1

NS n

NS n+1

NS n+2

NS n+3

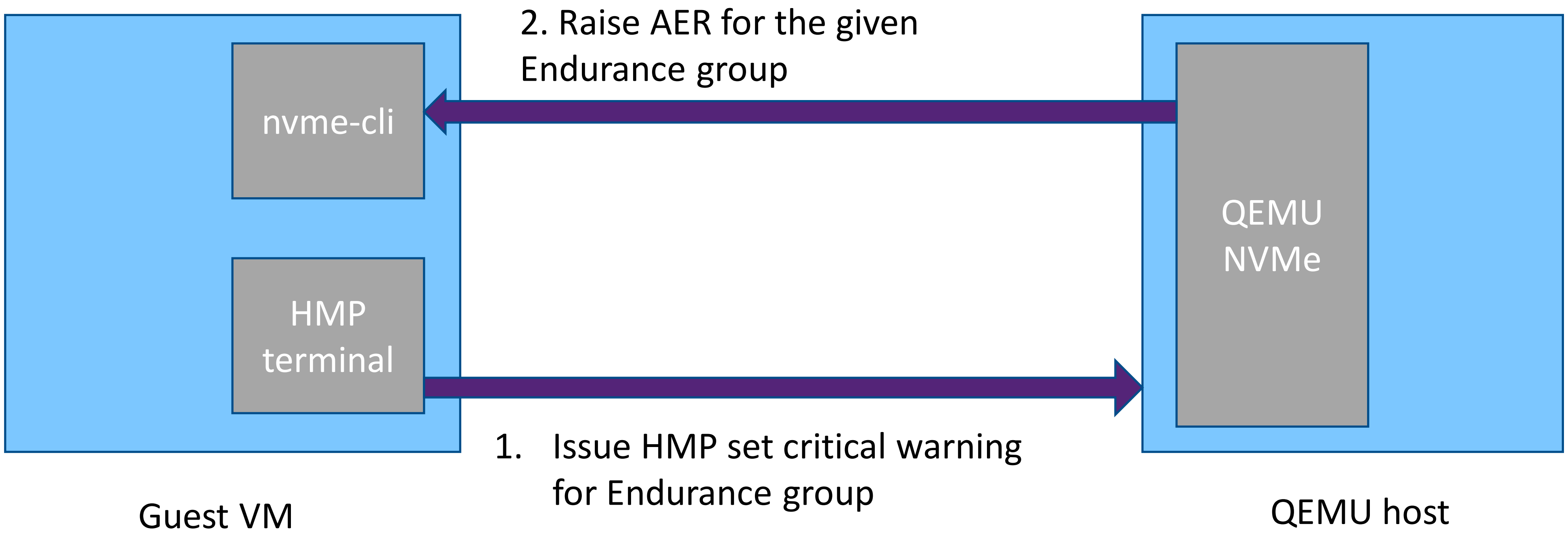
NVM set 2

ENDURANCE GROUP

..continued

- Endurance group contains grouping of one ore more NVM sets, which in turn contains the grouping of one or more Namespaces.
- Endurance group enables separation of the media units. If better managed, this feature provides in improving the life of the SSD.
- This feature provides the user in better management of the storage for eg: by having NAND type tied to a particular Endurance group.
- QEMU has provided the support for Endurance group log page, Endurance group aggregate log page and AER generation in case of critical warning.
- As it is an emulator, QEMU has provided an “HMP” command to trigger critical warning AER for the Endurance Group.

..continued



2. Raise AER for the given Endurance group
nvme-cli
HMP terminal
QEMU NVMe
1. Issue HMP set critical warning for Endurance group
Guest VM
QEMU host

STORAGE DEVELOPER CONFERENCE

SDC 21

BY Developers FOR Developers

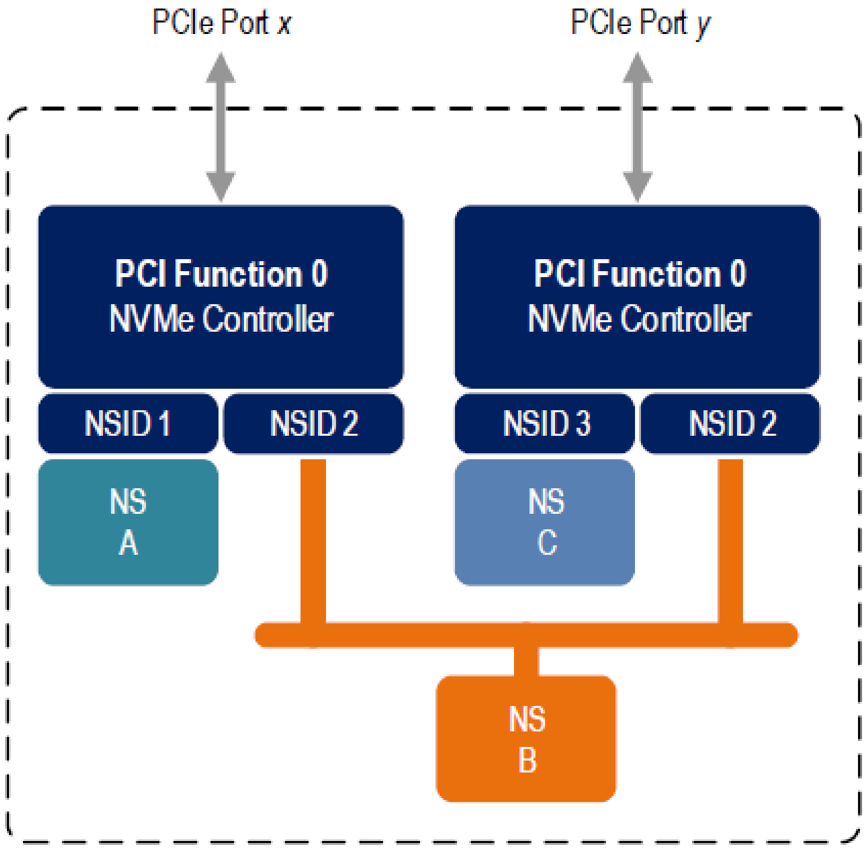
Virtual Conference
September 28-29, 2021

Reservations Overview

Introduction

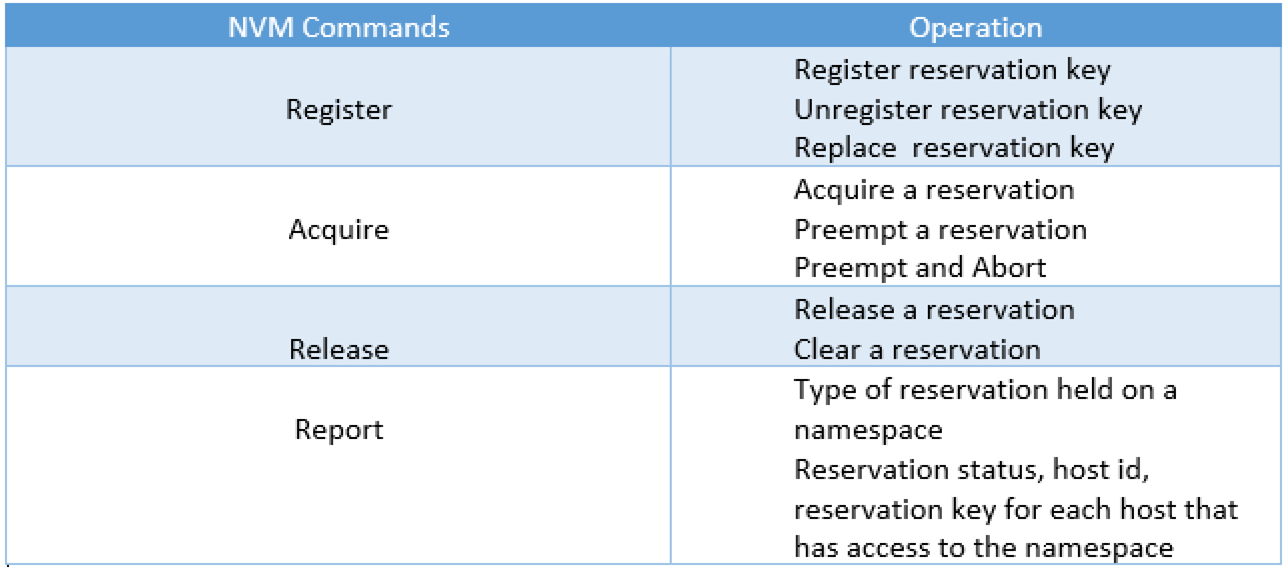
- Reservations provide capabilities that may be utilized by two or more hosts to provide coordinated access to a shared namespace
- Reservations are on a namespace and restrict host access to that namespace

Dual Port Logical View



- Fig: NVM Subsystem with Two Controllers and Two Ports (Ref: https://nvmexpress.org/)

Types Of Reservation Commands



NVM Commands	Operation
Register	Register reservation key Unregister reservation key Replace reservation key
Acquire	Acquire a reservation Preempt a reservation Preempt and Abort
Release	Release a reservation Clear a reservation
Report	Type of reservation held on a namespace Reservation status, host id, reservation key for each host that has access to the namespace

Reservation Register

- The Reservation Register command is used to register, unregister, or replace a reservation key. It uses Command Dword 10 and a Reservation Register data structure in memory

- *nvme resv-register* <device> [-n <nsid>] [-c <crkey>][-k <nrkey>][-r <rrega>][-p <cptpl>][-I]

Reservation Acquire

- The Reservation Acquire command is used to acquire a reservation on a namespace, preempt a reservation held on a namespace, and abort a reservation held on a namespace. It uses Command Dword 10 and a Reservation Acquire data structure in memory.

- *nvme resv-acquire* <device> [-n <nsid>][-c <crkey>][-p <prkey>][-t <rtype>][-a <racqa>][-I]

Reservation Release

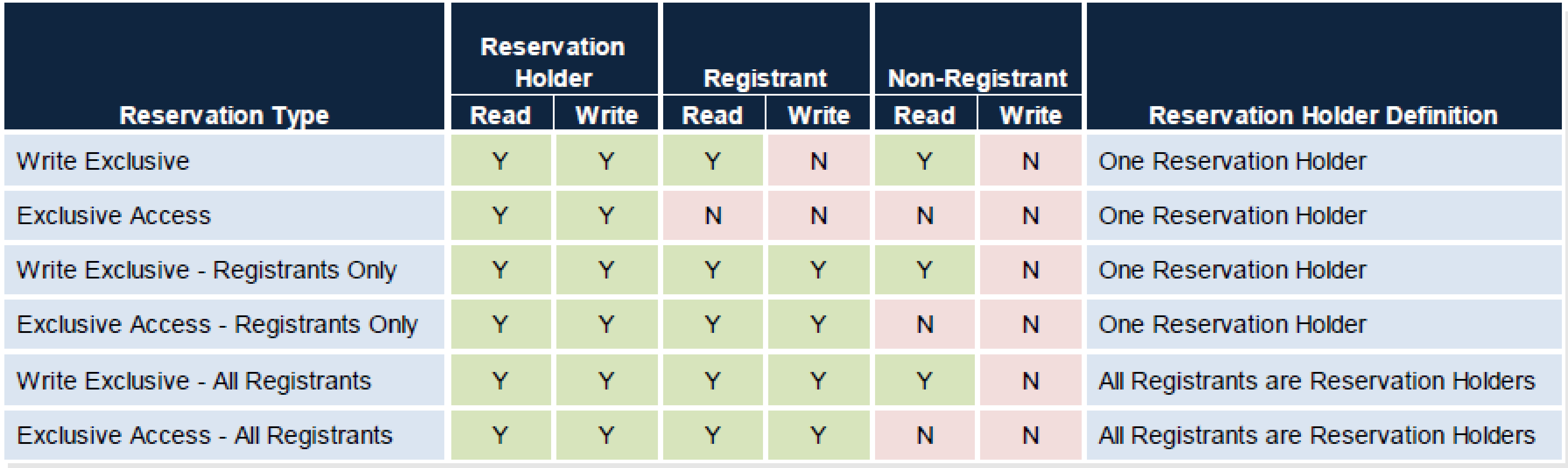
- The Reservation Release command is used to release or clear a reservation held on a namespace. It uses Command Dword 10 and a Reservation Release data structure in memory

- *nvme resv-release* <device> [-n <nsid>][-c <crkey>][-t <rtype>][-a <rrela>][-l]

Reservation Report

- It returns information about current reservations that describes the registration and reservation status of a namespace.
- *nvme resv-report* <device> [-n <nsid>][-d <num-dwords>][-c <cdw11>][-b][-o <fmt>]

Command Behavior In Presence of a Reservation



	Reservation Holder		Registrant		Non-Registrant		
Reservation Type	Read	Write	Read	Write	Read	Write	Reservation Holder Definition
Write Exclusive	Y	Y	Y	N	Y	N	One Reservation Holder
Exclusive Access	Y	Y	N	N	N	N	One Reservation Holder
Write Exclusive - Registrants Only	Y	Y	Y	Y	Y	N	One Reservation Holder
Exclusive Access - Registrants Only	Y	Y	Y	Y	N	N	One Reservation Holder
Write Exclusive - All Registrants	Y	Y	Y	Y	Y	N	All Registrants are Reservation Holders
Exclusive Access - All Registrants	Y	Y	Y	Y	N	N	All Registrants are Reservation Holders

- Fig: Command Behavior in presence of Reservation (Ref: https://nvmexpress.org/)

Practical Example

- Let 3 Hosts are available in subsystem with host identifiers as HostID_A, HostID_B and HostID_C
- Host A - Register (NSID, Key_A) -> ok
- Host B - Register (NSID, Key_B) -> ok
- Host A - Acquire (NSID, ExclusiveAccessRegistrantsOnly, Key_A) -> ok
- Host C - Acquire (NSID, ExclusiveAccessRegistrantsOnly, Key_C) -> error
- Host A - Write (NSID) ->ok
- Host B - Read or Write (NSID) ->ok
- Host C - Read or Write (NSID) -> error
- Host A - Release (NSID, Key_A) ->ok
- Host C - Write (NSID) ->ok

Implementation in Qemu Device

- Map Data Structure is used inside subsystem

```
typedef struct NvmeReservations {
  int   nsid;
  int   rtype;
  bool  rstatus;
  int   curr_key;
} NvmeReservations;
```

- `NvmeReservations *reservations[max_controllers][max_namespaces];`

STORAGE DEVELOPER CONFERENCE

SDC 21

BY Developers FOR Developers

Virtual Conference
September 28-29, 2021

NVMe-MI in QEMU

QEMU-MI

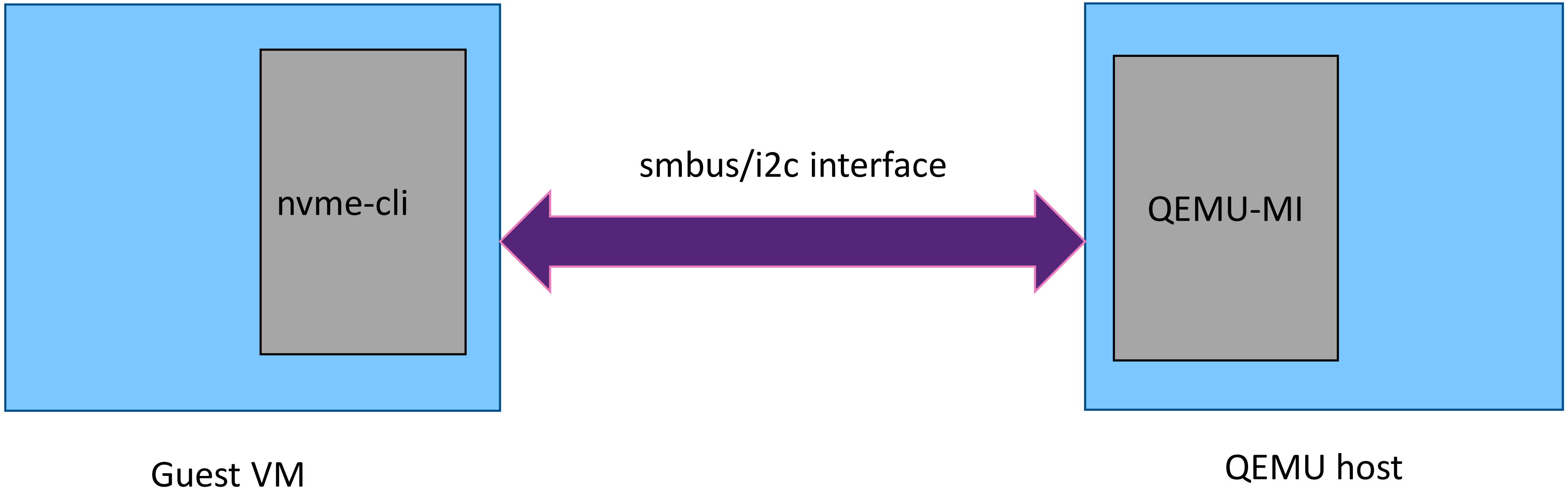
- We have proposed and implemented a new feature to be enabled in QEMU i.e. NVMe-MI.
- At present , QEMU supports NVMe over PCIe transport. However, our implementation is submitted as an RFC for enabling NVMe-MI over i2c in QEMU.
- To enable testing of QEMU-MI interface, we have also implemented the sideband interface in nvme-cli as a plugin.
- This will enable early validation of various modules (TestSuite/Platform) pertaining to MI before the arrival of actual device.

..continued

- Following commands are implemented in QEMU-MI module
  - Read nvme-mi data structure
  - Get config
  - Set config
  - Vpd read/write
  - Identify
  - Get logpage
  - Get features

- Work is in progress to implement rest of the features and commands.

..continued



STORAGE DEVELOPER CONFERENCE
SDC 21
BY Developers FOR Developers

Virtual Conference
September 28-29, 2021

Administrative Controller

Administrative Controller

- NVMe 1.4 specification defines different controller type viz. IO Controller, Admin controller and Discovery controllers.
- IO controller mainly focuses on commands that access logical block data, whereas Administrative controller focusses on commands that provide management capabilities.
- Below figure depicts the different Controller types supported in NVMe.

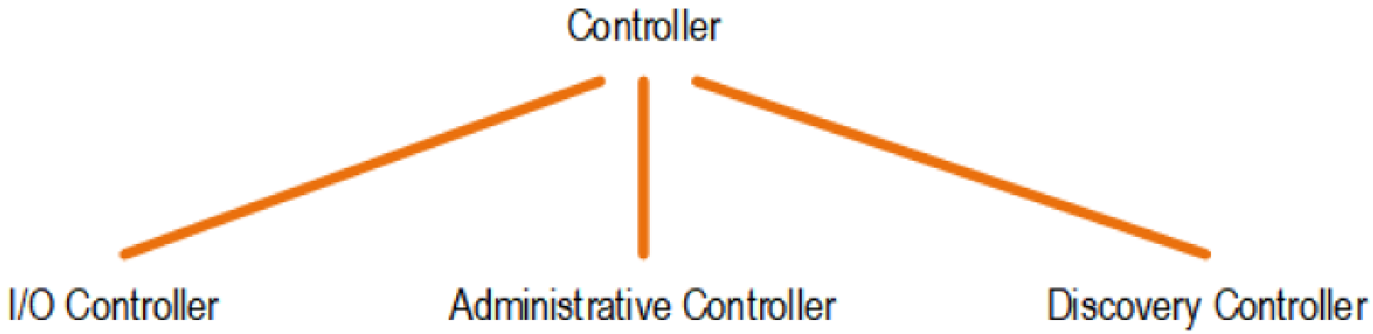


Fig: Controller Types (Ref: https://nvmexpress.org)

.. continued

- QEMU implements Administrative Controller feature by providing the support for Controller Type (CNTRLTYPE) field in Identify Controller to inform the host what kind of Controller it is.

- Also, QEMU prohibits the command like Create I/O SQ/CQ and Delete I/O SQ/CQ which are not meant for Administrative Controller.

- In the launch command, “cntrl_type” parameter can be set to “3” if one wants the controller to be Administrative Controller.

```
-device nvme,serial=<serial>,id=<bus_name>, cntrl_type=3
```

STORAGE DEVELOPER CONFERENCE

SDC 21

BY Developers FOR Developers

Virtual Conference
September 28-29, 2021

Namespace Write Protect

Namespace Write Protection Config

NVMe 1.4 specification provides Namespace Write Protection Config feature using which host can control the write protection state of the Namespace.

There are 4 states viz, No Write Protect, Write Protect, Write Protect until power cycle, Permanent Write Protect.

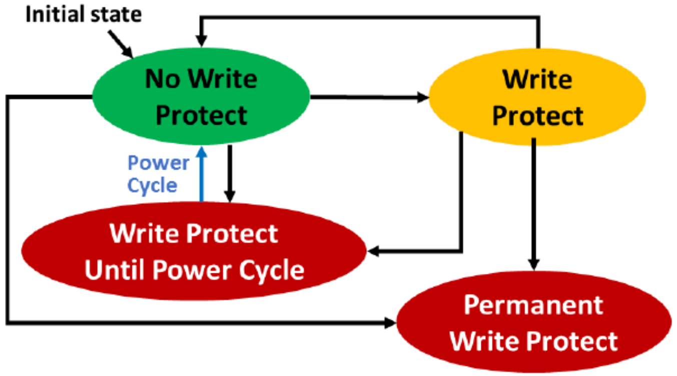


Fig: Namespace Write Protection State model (Ref: https://nvmexpress.org/ NVMe1.4b)

.. continued

- As qemu emulation doesn’t support Power cycle, we have added a QMP command to emulate a power cycle from host and change the Namespace Write Protect state.

- Also, since the state/s in qemu cannot be persistent, we have provided an option in launch command to emulate the “Permanent Write protection state” configuration.

```
-device nvme-ns,drive=<drive_id>,bus=<bus_name>,nsid=<nsid>, subsys=<subsys_id>, perm_wr_protect=<true|false>[optional]
```

Rounding up

- Several new features introduced this year
  - Helps developers test drivers and applications

- The QEMU Emulated NVMe device continues to be an active subsystem of QEMU, with increasing number of contributors
  - Like any open source project, we are always looking for new contributors, reviewers and **people willing to test**!

Thank You!

Please take a moment to rate this session.

Your feedback is important to us.