



# **CXL\* Type 3 Memory Device Software Guide**

**System Firmware, UEFI and OS Software Implementation  
Guide**

---

***July 2021***

**Revision 1.0**



**Notice: This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.**

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at [intel.com](http://intel.com), or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document. The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit <http://www.intel.com/performance>.

Cost reduction scenarios described are intended as examples of how a given Intel-based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction.

Results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting [www.intel.com/design/literature.htm](http://www.intel.com/design/literature.htm).

Intel, the Intel logo, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

\*Other names and brands may be claimed as the property of others

Copyright © 2021, Intel Corporation. All Rights Reserved.

# Contents

<b>1</b>	<b>Document Overview .....</b>	<b>8</b>
1.1	Document goals .....	8
1.2	Document scope.....	9
1.3	Reference material dependencies for this version of the document.....	9
1.4	Abbreviations.....	10
1.5	Open issues.....	10
<b>2</b>	<b>High-level architecture.....</b>	<b>11</b>
2.1	Conceptual CXL architecture for volatile memory support .....	11
2.2	Conceptual CXL architecture for persistent memory support .....	17
2.3	Basic Linux CXL architecture .....	21
2.4	High-level software component responsibilities .....	23
2.5	High-level System Firmware memory interface overview .....	32
2.6	Memory provisioning.....	34
2.6.1	EFI_MEMORY_MAP.....	36
2.6.2	CEDT CFMWS Example – No XHB interleaving .....	37
2.6.3	CEDT CFMWS Example – x2 XHB interleaving .....	38
2.6.4	CEDT CFMWS Example – x4 XHB interleaving .....	39
2.6.5	CEDT CFMWS Example – x4 XHB interleaving across multiple sockets .....	40
2.7	Managing persistent regions.....	41
2.7.1	Example – Volatile and persistent x2 interleaved regions.....	44
2.7.2	Example – Region interleaved across 2 CXL switches .....	47
2.7.3	Example – Region interleaved across 2 HBs.....	48
2.7.4	Example – Region interleaved across 2 HBs and 4 switches ...	50
2.7.5	Example – 2 regions interleaved across 2 and 4 devices.....	52
2.7.6	Example – Out of order devices within a switch or host .....	53
2.7.7	Example – Out of order devices across HBs (failure case) .....	55
2.7.8	Example – Verifying device position on each HB root port.....	57
2.8	Partitioning and Configuration Sequences .....	60
2.8.1	Volatile and Persistent Memory Partitioning .....	60
2.8.2	Persistent memory region configuration .....	62
2.8.3	System Firmware enumeration .....	63
2.8.4	OS and UEFI driver enumeration.....	64
2.9	Asynchronous event handling .....	65
2.10	Dirty shutdown count handling .....	66
2.11	SRAT and HMAT .....	68
2.11.1	SRAT, HMAT and OS NUMA calculation examples.....	70
2.11.2	GetQoSThrottlingGroup _DSM calculation examples.....	76
2.11.3	Link bandwidth calculation .....	77
2.11.4	Link latency calculation.....	77
2.12	Operation ordering restrictions .....	78
2.13	Basic high-level sequences .....	79
2.13.1	System Firmware boot sequence.....	79
2.13.2	UEFI setup and boot sequence.....	81
2.13.3	OS boot sequence.....	82
2.13.4	OS shutdown sequence .....	83

2.13.5	System Firmware shutdown sequence .....	84
2.13.6	OS hot add sequence .....	85
2.13.7	OS managed hot remove sequence .....	86
2.13.8	Verifying ACPI CEDT, CHBS and CFMWS sequence.....	87
2.13.9	Device discovery and mailbox ready sequence .....	88
2.13.10	Media ready sequence.....	89
2.13.11	Verifying region configuration and assigning HPA ranges sequence.....	90
2.13.12	Find CFMWS for region sequence .....	92
2.13.13	Find CFMWS for volatile sequence.....	93
2.13.14	Verify XHB configuration sequence.....	94
2.13.15	Verify HB root port configuration sequence.....	104
2.13.16	Calculate HDM decoder settings sequence.....	109
2.13.17	Device initialization sequence .....	110
2.13.18	Handle event records sequence .....	111
2.13.19	Retrieve poison list sequence .....	112
2.13.20	Handle health information sequence.....	113
2.13.21	FW first event interrupt sequence .....	114
2.13.22	OS first event interrupt sequence .....	114
2.13.23	Invalidating/Flushing CPU caches sequence.....	115
2.13.24	HPA to DPA translation sequence .....	116
2.13.25	DPA to HPA translation sequence .....	119
2.13.26	GPF sequence .....	121

## Figures

Figure 2-1.	Conceptual CXL architecture for volatile memory support.....	12
Figure 2 -	Conceptual CXL architecture for persistent memory support.....	18
Figure 3 -	Basic Linux CXL architecture.....	21
Figure 4 -	Example per CXL Host Bridge fixed memory allocation .....	35
Figure 5 -	CEDT CFMWS Example - No XHB interleaving.....	37
Figure 6 -	CEDT CFMWS Example - x2 XHB interleave.....	38
Figure 7 -	CEDT CFMWS Example - x4 XHB interleave.....	39
Figure 8 -	CEDT CFMWS Example - x4 XHB interleave across multiple sockets .....	40
Figure 9 -	Components for Managing Regions.....	43
Figure 10 -	Example - Volatile and persistent x2 interleaved regions.....	45
Figure 11 -	Example - Region interleaved across 2 switches .....	47
Figure 12 -	Example - Region interleaved across 2 HBs.....	48
Figure 13 -	Example - Region interleaved across 2 HBs and 4 switches .....	51
Figure 14 -	Example - 2 regions interleaved across 2 and 4 devices .....	52
Figure 15 -	Example - Out of order devices within a switch or host bridge ...	54
Figure 16 -	Example - Out of order devices across HBs (failure case) .....	56
Figure 17 -	Example - Valid x2 HB root port device ordering .....	57
Figure 18 -	Example - Invalid x2 HB root port device ordering .....	58
Figure 19 -	Example - Unbalanced region spanning x2 HB root ports.....	59
Figure 20 -	High-level sequence: System Firmware and UEFI driver memory partitioning.....	60
Figure 21 -	High-level sequence: OS memory partitioning .....	61
Figure 22 -	High-level sequence: Persistent memory region configuration....	62

Figure 23 - High-level sequence: System Firmware enumeration.....	63
Figure 24 - High-level sequence: UEFI and OS enumeration .....	64
Figure 25 - CXL event notification architecture.....	65
Figure 26 - CXL dirty shutdown count handling logic.....	67
Figure 27 - SRAT and HMAT Example: Latency calculations for 1 socket system with no memory present at boot .....	70
Figure 28 - SRAT and HMAT Example: Latency calculations for 2 socket system with no memory present at boot .....	71
Figure 29 - SRAT and HMAT Example: Latency calculations for 1 socket system with volatile memory attached at boot .....	72
Figure 30 - SRAT and HMAT Example: Bandwidth calculations for 1 socket system with volatile memory attached at boot .....	73
Figure 31 - SRAT and HMAT Example: Latency calculations for 2 socket system with volatile memory attached at boot .....	74
Figure 32 - SRAT and HMAT Example: Latency calculations for 1 socket system with persistent memory and hot added memory .....	75
Figure 33 - GetQosThrottlingGroup _DSM example.....	76
Figure 34 - High-level sequence: System Firmware boot .....	80
Figure 35 - High-level sequence: UEFI setup and boot .....	81
Figure 36 - High-level sequence: OS boot .....	82
Figure 37 - High-level sequence: OS shutdown .....	83
Figure 38 - High-level sequence: System Firmware shutdown.....	84
Figure 39 - High-level sequence: OS hot add.....	85
Figure 40 - High-level sequence: OS managed hot remove.....	86
Figure 41 - High-level sequence: Verifying CEDT, CHBS, CFMWS.....	87
Figure 42 - High-level sequence: Device discovery and mailbox ready.....	88
Figure 43 - High-level sequence: Media ready .....	89
Figure 44 - High-level sequence: Verifying region configuration .....	91
Figure 45 - High-level sequence: Finding CFMWS for region.....	92
Figure 46 - High-level sequence: Finding CFMWS for volatile .....	93
Figure 47 - High-level sequence: Verify XHB configuration .....	94
Figure 48 - Example valid x2 XHB configuration execution steps .....	95
Figure 49 - Example invalid x2 XHB configuration .....	96
Figure 50 - Example valid x2 XHB configuration .....	97
Figure 51 - Example valid x4 XHB configuration .....	99
Figure 52 - Example valid x4 XHB configuration .....	100
Figure 53 - Example invalid x4 XHB configuration .....	101
Figure 54 - Example valid x8 XHB configuration .....	103
Figure 55 - High-level sequence: Verify HB root port configuration .....	104
Figure 56 - Example valid region spanning 2 HB root ports .....	105
Figure 57 - Example invalid region spanning 2 HB root ports .....	106
Figure 58 - Example valid region spanning 4 HB root ports .....	107
Figure 59 - Example invalid region spanning 4 HB root ports .....	107
Figure 60 - Example valid region spanning 4 HB root ports on a x2 XHB....	108
Figure 61 - Example invalid region spanning 4 HB root ports on a x2 XHB .	108
Figure 62 - High-level sequence: Calculate HDM decoder settings.....	109
Figure 63 - High-level sequence: Device initialization.....	110
Figure 64 - High-level sequence: Handle event records.....	111

Figure 65 - High-level sequence: Retrieve poison list.....	112
Figure 66 - High-level sequence: Handle health information .....	113
Figure 67 - High-level sequence: FW first event interrupt.....	114
Figure 68 - High-level sequence: OS first event interrupt .....	114
Figure 69 - High-level sequence: Invalidating/flushing CPU caches .....	115
Figure 70 - High-level sequence: HPA to DPA translation.....	116
Figure 71 - High-level sequence: DPA to HPA translation.....	119
Figure 72 - High-level sequence: GPF.....	121

## Tables

Table 2-1. High-level software component responsibilities – System Boot....	23
Table 2 - High-level software component responsibilities - System Shutdown and GPF .....	28
Table 3 - High-level software component responsibilities - Hot Add.....	29
Table 4 - High-level software component responsibilities - Managed Hot Remove .....	31
Table 5 - System firmware memory interface summary .....	32
Table 6 - SRAT and HMAT content .....	68

## Revision History

---

Revision Number	Description	Date
1.0	<ul style="list-style-type: none"><li>Initial release of the document.</li></ul>	June 2021

# 1 Document Overview

---

## 1.1 Document goals

This document is focused on CXL\* 2.0 Type Memory Expander devices and the responsibilities of the software ecosystem to manage, configure, and enumerate these devices. However, much of the content could apply to CXL 1.1 devices that implement the Device Command Interface.

This is considered an informative document and is not meant to prescribe explicit software requirements but to demonstrate how the interfaces provided by CXL, ACPI, UEFI standards and the ECNs

While the document identifies separate volatile and persistent memory architectural components and the flows differentiate volatile versus persistent memory capacity steps, the intent is there is a single CXL type 3 common memory driver that handles both types of devices.

The specific goals of this document include:

- Clearly delineate System Firmware, UEFI and OS driver responsibilities for supporting CXL
- Define informative behaviors for System Firmware, UEFI and OS Drivers for supporting CXL
- Demonstrate a set of high-level sequences for System Firmware, UEFI and OS Drivers to follow for supporting CXL

Non-goals of the document include:

- Specific System Firmware, UEFI and OS driver implementation
- Specific System Firmware, UEFI and OS driver policy
- Exhaustive low-level architecture and sequences
- PCIe details

Target Audience includes:

- CXL software architects and authors
- CXL memory device architects and implementers
- System engineers wanting a deeper understanding of the CXL system software responsibilities, interfaces, and software sequences



## 1.2 Document scope

- CXL 2.0 Memory Devices (Type 3) focused. CXL 1.1 Memory devices implementing the Device Command Interface can also utilize this content.
- Provide enough information for CXL Memory device driver writers to create a high-level architecture/design
- Provide enough information for CXL Memory device driver writers to create high-level requirements and test plans
- The following are considered out of scope for the document at this time:
  - Hot adding of CXL Host Bridges
  - Hot add of devices in FW first: Requires platform to pass knowledge of the VDN MEFN vector to utilize when OS hot-adds the device
  - More than one level of CXL switch is not comprehended in the flows

## 1.3 Reference material dependencies for this version of the document

- PCI Express\* (PCIe\*) 5.1 Specification
- CXL 2.0 Specification
  - + CEDT CFMWS ECN
  - + Mailbox Ready Time ECN
- ACPI 6.4
  - + FADT PERSISTENT\_CPU\_CACHES ECN
  - + ACPI0017 ECN (Code First)
  - + Generic Port ECN (Code First)
- UEFI 2.9
- SNIA Persistent Memory Programming Model
- This document: Add link
- Pmem.io
  - DSC White Paper
  - PMDK

## 1.4 Abbreviations

Abbreviations used in this document that may not be found in the CXL, PCIe, ACPI or UEFI specifications:

- **CFMWS** – ACPI CEDT CXL Fixed Memory Window Structure
- **DPA** – CXL Memory Device Physical Address
- **HB** – CXL Host Bridge. See XHB.
- **HPA** – Host Physical Address
- **Interleave set** – Collection of DPA ranges from one or more devices that make up a single HPA range. See Region.
- **LSA** – Label Storage Area
- **OS Drivers** – The collection of OS kernel components required to implement CXL
- **PMEM** – Persistent memory
- **Region** – CXL term for a memory interleave set. See Interleave set.
- **UEFI Drivers** – UEFI CXL Bus and Memory device drivers
- **XHB** – Cross CXL Host Bridge interleave set. An interleave set the spans multiple Host Bridges. May or may not cross sockets

## 1.5 Open issues

- This CXL SW Implementation Guide
  - Add XHB and switch position validation sequences in the creating regions flow
  - Add FW Activation flow: OS should reload CEL after a FW activation

## 2 *High-level architecture*

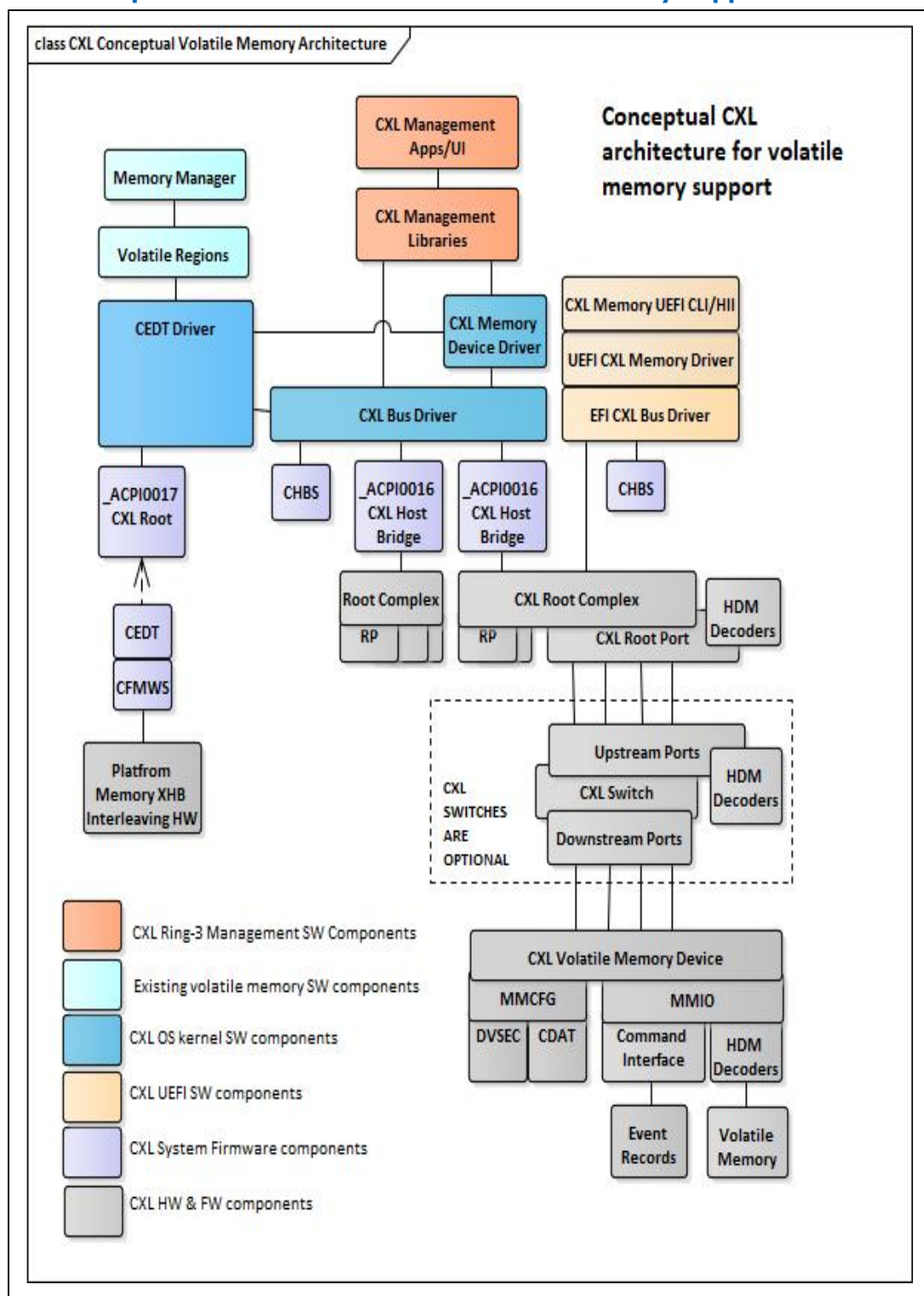
---

The following sections outline the high-level architectural components, their responsibilities, and a basic review of System Firmware interfaces.

### 2.1 **Conceptual CXL architecture for volatile memory support**

The section describes the conceptual CXL HW and SW components required for general CXL support with volatile CXL memory device. The architecture shown is not based on a specific OS implementation. See the [Basic Linux\\* CXL architecture](#) section for more details on the Linux CXL memory architecture.

**Figure 2-1. Conceptual CXL architecture for volatile memory support**



### **CXL Volatile Memory Device (Type 3)**

CXL 1.1 or 2.0 Memory devices may be connected to the CXL Root Port or a CXL switch downstream port by one or more Flexbus lanes. The device is mapped into MMCFG and MMIO regions using standard PCIe mechanisms. The Type 3 specific Memory Device Command MMIO mailbox interface is used to manage and configure the device.

**HDM Decoders** - The MMIO mapped HDM decoders are setup by the System Firmware for known CXL volatile capacity, by the UEFI and OS driver for known CXL persistent capacity, and by the OS for hot added CXL volatile and persistent memory capacity. These registers determine what Host Physical Address (HPA) range will be mapped to the Device Physical Address (DPA) range exposed by the device. HDM decoders are found in all up-stream CXL switch ports as well as in each CXL Root Complex. These HDM decoders will also need to be programmed to account for all the downstream device HDM decoder programming. HDM decoders in the CXL Root Complex determine which root port is the target of the memory transaction. Similarly, HDM decoders in an Upstream Port determine which Downstream Port is the target.

**CDAT** - Standardized device registers to report latency and BW information. System Firmware utilizes this information to build HMAT tables at platform boot time for volatile CXL memory devices. At OS boot and hot-add time, the CDAT information is utilized by the OS in combination with the SRAT and HMAT to build a complete BW and Latency picture for the persistent memory devices.

**Command Interface** - The CXL Device Command Interface utilized by System Firmware, UEFI and OS drivers to:

- Configure and manage the device
- Retrieve and store persistent region and namespace configuration information
- Configure, retrieve, and clear asynchronous device runtime alerts
- The command interface surfaces the following to the System Firmware, UEFI and OS drivers:

#### **Device Capabilities, Capacity, and Partition Information**

**Event Log** - Each CXL Memory device is required to support space for at least one event record in each of the Informational, Warning, Failure, or Fatal Event Severity queues. Asynchronous notification of new entries in the list is done using standard PCIe MSI/MSIx (OS First) or VDM message interrupts (FW First).

**Health Information** - The device must maintain consistent health information including Life Used, Device Temperature, and persistent Dirty Shutdown Count.

### **CXL Switch**

Provides expansion of CXL Root Ports into many downstream switch ports allowing more CXL memory devices to be connected in a scalable way. Each switch has a set of HDM decoders that govern the upstream switch ports decoding of the HPA. The System Firmware, UEFI and OS drivers are responsible for programming these HDM decoders to cover all the devices and switches connected downstream.

### **CXL Root Port**

CXL HW connection to the CXL memory device or CXL switch port via one or more Flexbus lanes. Equivalent to a PCIe Root Port.

## **CXL Root Complex**

Platform specific CXL Root Ports And the equivalent to a PCIe Root Complex. Bus number assignments are the responsibility of the System Firmware and exposed through the CXL Host Bridge ACPI namespace device.

## **ACPI0016 CXL Host Bridge Object**

Virtual SW entity implemented by the System Firmware under the \_SB (System Bus) of the ACPI tree and consumed by the OS. Each ACPI0016 object represents a single CXL Root Complex. Since the root of the CXL tree (the CXL Root Complex) is platform specific and is not presented through a PCI BAR, the System Firmware is responsible for generating an object to represent the collection of CXL Root Ports that represent a CXL Host Bridge. Each HB is represented by a unique ACPI0016 object under the top of the ACPI /SB device tree. There are a number of ACPI methods attached to this object that the System Firmware will implement on behalf of the OS. This is not an exhaustive list of ACPI methods the CXL Host Bridge device is expected to support. Assume all standard PCI Host Bridge driver methods for finding and configuring HW will apply:

- \_CRS – Same as PCIe Host Bridge method
- \_OSC - Determine FW First/OS First responsibility. Follows standard PCIe Host Bridge functionality with CXL additions. See the CXL 2.0 spec \_OSC section.
- \_REG – Same as PCIe Host Bridge method
- \_BBN – Same as PCIe Host Bridge method
- \_SEG – Same as PCIe Host Bridge method
- \_CBR – Retrieve pointer to the CXL Host Bridge register block which provides access to the HDM decoders. If the platform supports hot add of CXL Host Bridges the OS can utilize this method to find the register block after the addition of the new HB. CXL Host Bridge hot add is considered out of scope for this document. CXL Host Bridges present at boot will not have a \_CBR method and the CEDT CHBS must be utilized.
- \_PXM – Same as PCIe Host Bridge method

## **CXL Bus Driver**

A CXL enlightened version of a standard PCIE bus driver that consumes all ACPI0016 CXL Host Bridge ACPI device instances and initiates CXL bus enumeration, understands the relationship between CXL Host Bridges, CXL Switches, and CXL end-point devices, and loads device specific CXL memory device driver instances for each supported end point it enumerates.

## **CXL Memory Device Driver**

Each physical CXL memory device surfaced by the bus driver is consumed by a separate instance of the CXL Memory device driver. This driver consumes the Command Interface and typically exports those features through OS specific IOCTLS to allow the OS In-band Management Stack components to manage the device.

## **ACPI0017 CXL Root Object**

Virtual SW entity implemented by the System Firmware under \_SB of the ACPI tree that represents the presence of the CXL Early Discovery Table

(CEDT). The following methods are attached to this device:

Get QoS Throttling Group DSM – Retrieve the QTG the device should be programmed to:

### **CEDT**

CXL Early Discovery Table – System Firmware provides this ACPI table that UEFI and OS drivers utilize to retrieve pointers to all of the CXL Host Bridge register blocks (CHBS) and a set of fixed memory windows (CFMWS) for each CXL Host Bridge present at platform boot time. The pointer to the register block allows the System Firmware, UEFI and OS drivers to program HDM decoders for the CXL Host Bridges. While the ACPI0017 object will signal the presence of the CEDT, this table is not dependent on the ACPI0017 object since it must be available in early boot, before the ACPI device tree has been created. This is a static table created by the System Firmware at platform boot time.

### **CHBS**

ACPI CXL Host Bridge Structure – Each CXL Host Bridge instance will have a corresponding CHBS which identifies what version the CXL Host Bridge supports and a pointer to the CXL Root Complex register block that is needed for programming the CXL Root Complex's HDM decoder.

### **CFMWS**

CXL Fixed Memory Window Structure – A new structure type in CEDT that describe all of the platform allocated and programmed HPA based windows where the System Firmware, UEFI and OS drivers can map CXL memory.

### **CEDT Driver**

The ACPI0017 CEDT and CFMWS will be consumed by a bus driver that concatenates the HDM decoders for each CXL Host Bridge in to one or more regions (or interleave sets) that the bus driver surfaces to the existing PMEM/SCM stacks. Since interleave sets in CXL may span multiple CXL Host Bridges, this driver handles XHB interleaving, and presents other drivers in the stack with a single consistent set of regions, whether they are contained in a single CXL Host Bridge or span multiple CXL Host Bridges.

### **Volatile Region**

Each volatile region represents an HPA range that utilizes a set collection of CXL Memory devices with volatile capacity.

### **Memory Manager**

Volatile regions are typically consumed by the OS memory manager which controls allocation and deallocation of physically memory on behalf of other Ring 0 and Ring 3 SW components.



### **In-band OS Management Stack**

**CXL Management Apps/UI** – CXL Management applications and user interfaces utilizing CXL Management Libraries.

**CXL Management Libraries** – Management Libraries covering the standardized CXL interfaces.

CXL based in-band management libraries and UI components that will utilize OS implementation specific IOCTLs and pass-through interfaces surfaced by the CXL Root Port Bus Driver, CXL Memory Device Driver, and the PMEM or SCM Region Driver instances.

### **UEFI Management Stack**

**CXL Memory EFI Driver**

**CXL Bus EFI Driver**

**CXL Memory EFI CLI/HII**

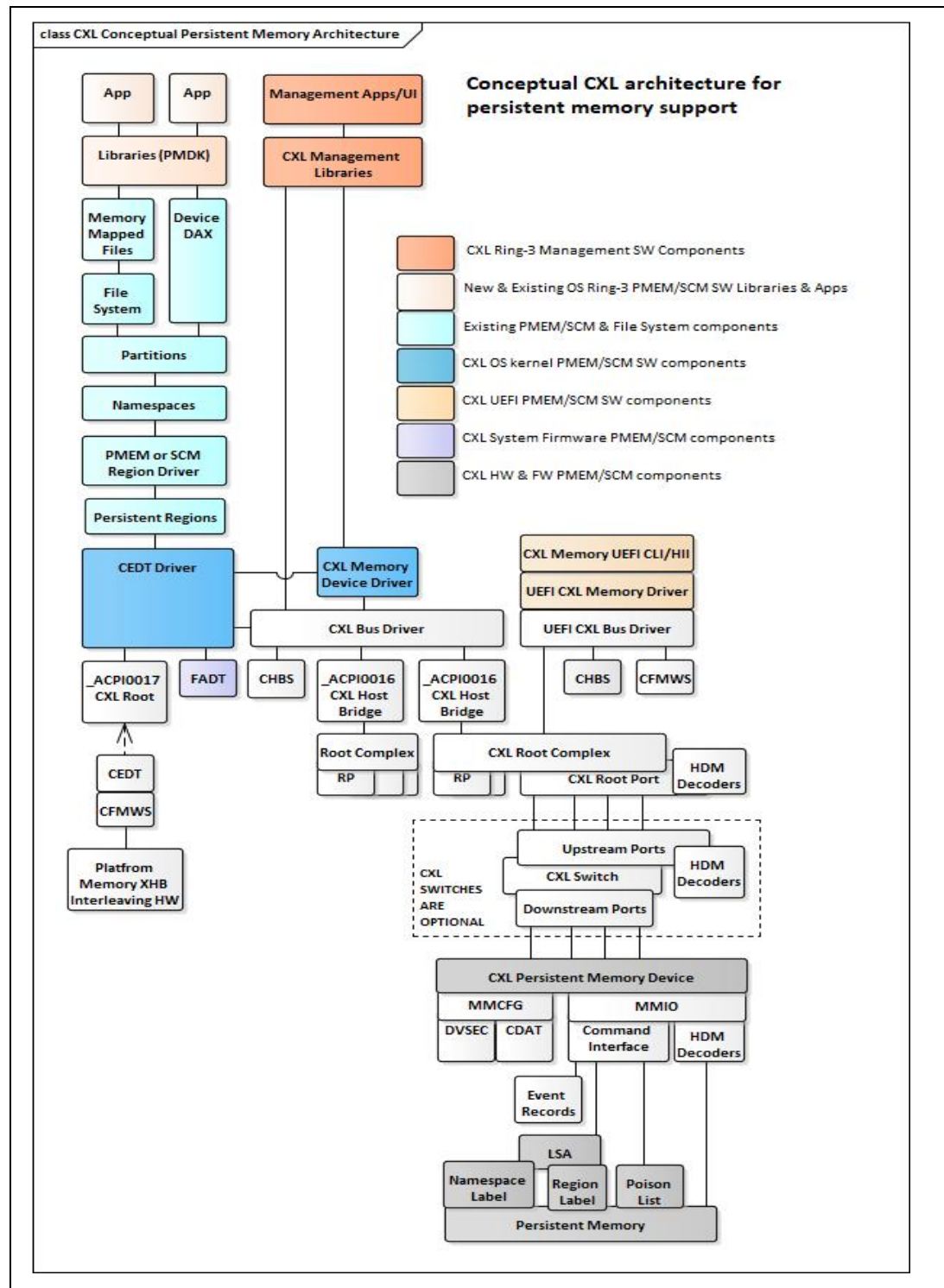
UEFI pre-boot environment CXL bus driver that surfaces EFI\_BLOCK\_IO protocol (utilizing byte addressable persistent memory) for consumption by the OS boot loader. This also provides management interfaces for the UEFI based management stack. This may be implemented using a pre-boot CXL Bus driver and CXL Memory device driver.



## 2.2 Conceptual CXL architecture for persistent memory support

This section describes the additional new and updated CXL HW and SW components required to support CXL persistent memory devices. The architecture shown is not based on a specific OS implementation. See the [Basic Linux\\* CXL architecture](#) section for more details on the Linux CXL memory architecture.

Figure 2 - Conceptual CXL architecture for persistent memory support



## **CXL Persistent Memory Device (Type 3)**

**Command Interface** - The CXL Device Command Interface utilized by System Firmware, UEFI and OS drivers to expose additional persistent memory features:

**LSA** - The CXL Memory device is responsible for surfacing a persistent Label Storage Area that the UEFI and OS drivers utilize to read and write Region (interleave set) configuration information and Namespace configuration information. This is required to re-assemble the persistent memory region configuration correctly.

**Region Label** – Persistent configuration information that describes to the UEFI and OS drivers the persistent memory region, the UUID of all devices involved, the amount of persistent capacity each contributes to the region, and the position each device contributes to the region. With persistent memory, the ordering of the device in the region must always be preserved to re-assemble the data from the region correctly.

**Namespace Label** – Persistent configuration information that describes to the UEFI and OS drivers how each persistent memory region is subdivided into namespaces. There are several types of namespaces including BTT based block storage emulation.

**Poison List** - The device is required to maintain a persistent poison list so the UEFI and OS drivers can quickly determine what areas of the media contain invalid data and must be avoided or corrected.

### **FADT**

Fixed ACPI Description Table – Existing ACPI table that is utilized to report fixed attributes of the platform at platform boot time. For CXL, the new PERSISTENT\_CPU\_CACHES attribute is utilized by the platform to report if the CPU caches are considered persistent and by the OS to set application flushing policy.

### **Persistent Region**

Each persistent region represents an HPA range that utilizes a set collection of CXL Memory devices with persistent capacity configured in a specific order. The configuration of each region is described in the region labels stored in the Label Storage Area (LSA) which is exposed through the Command Interface.

### **PMEM or SCM Region Driver**

Each instance of a region (interleave set) will be consumed by a separate instance of the PMEM or SCM (Storage Class Memory) Driver. This is probably significant re-use of the existing OS kernel NVDIMM components for this.

### **Namespaces**

Each region can be subdivided into volumes referred to as Namespaces. The configuration of each Namespace is described in the namespace labels stored in the Label Storage Area (LSA) which is exposed through the Command Interface. Persistent memory namespaces are described in detail in the UEFI specification.

**Partitions**

Each Namespace is typically sub-divided into partitions by the OS.

**File Systems**

Existing file system drivers subdivide each partition into one or more files and supply standard file API and file protection for the user.

**Memory Mapped Files**

Regions are subdivided into Namespaces which are subdivided into partitions and finally subdivided into memory mapped files by the file system. This is one standard mechanism for applications to access persistent memory directly with the security and convenience of a file.

**Device DAX**

A simplified direct pipeline between the application and the persistent memory namespace that bypasses the filesystem and memory mapped file usage.

**Libraries (PMDK)**

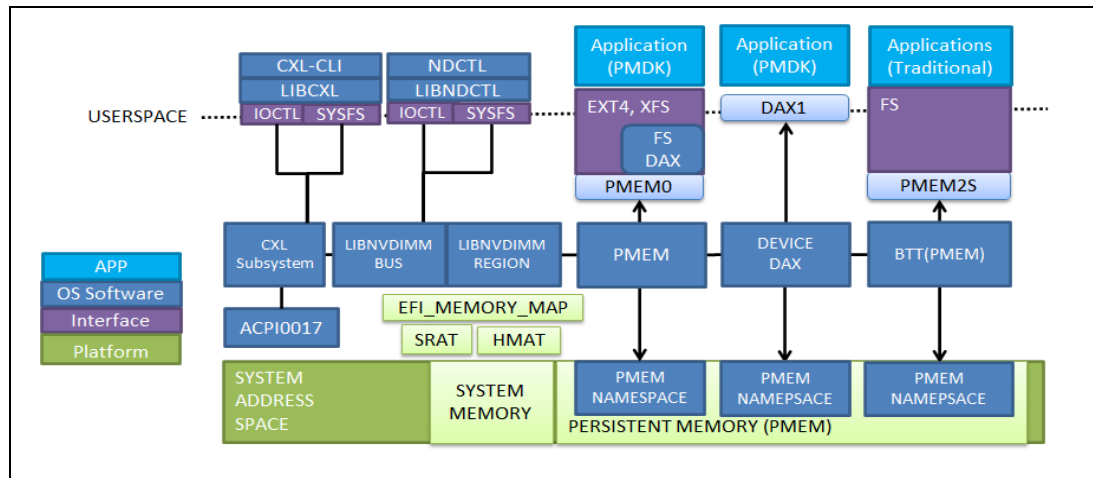
Most persistent memory aware applications make use of Ring3 Libraries like PMDK to simplify the persistent memory programming model. These libraries typically make use of Memory Mapped Files or direct Device DAX to access the persistent memory. There will be additions to these libraries to surface new CXL features.

**Applications**

## 2.3 Basic Linux\* CXL architecture

This section describes the basic Linux CXL architecture, the main SW components, and a brief description about each. CXL for memory devices utilizes the existing Linux NVDIMM architecture, replacing the NFIT based interfaces with ACPI0017 CEDT based CXL interfaces, and adding a CXL subsystem to handle the new CXL specific functionality.

**Figure 3 - Basic Linux CXL architecture**



### CXL Subsystem

New kernel component that utilizes the information provided through the ACPI0017 CEDT. Provides CXL specific services for the NVDIMM BUS component. Provides CXL specific IOCTL and SYSFS interfaces for management of CXL devices

### LIBNVDIMM BUS

Existing LIBNVDIMM BUS component. Provides generic NVDIMM bus related functionality including namespace management. Enumerates memory device endpoints for the LIBNVDIMM REGION component

### LIBNVDIMM REGION

Existing LIBNVDIMM REGION component. Consumes endpoint memory devices produced by the LIBNVDIMM BUS, consumes the region labels on each device (from the LSA), organizes the region labels in to interleave sets, validates the regions, and publishes regions to PMEM, DEVICE DAX and BTT components.

### PMEM

Kernel component that represents a single instance of a region



## **DEVICE DAX**

A simplified direct pipeline between the application and the persistent memory namespace that bypasses the filesystem and memory mapped file usage.

## **FS DAX**

Direct memory access for memory mapped file systems.

## **BTT**

Component that consumes the devices Block Translation Table and implements a block storage protocol on top of persistent memory.

## **PMEM NAMESPACE**

Each region can be subdivided into volumes referred to as Namespaces. The configuration of each Namespace is described in the namespace labels stored in the Label Storage Area (LSA) which is exposed through the Command Interface.

## **CXL-CLI**

Linux CXL memory management Ring3 CLI.

## **LIBCXL**

Linux CXL memory management Ring3 library.

## **NDCTL**

Linux NVDIMM memory management Ring3 CLI utilized with CXL.

## **LIBNDCTL**

Linux NVDIMM memory management Ring3 library.

## 2.4 High-level software component responsibilities

In its most basic form, the delineation of SW component responsibilities is:

- The System Firmware is responsible for enumerating and configuring volatile CXL memory capacity that is present at boot.
- The OS components are responsible for enumerating and configuring all topologies not covered by the previous System Firmware.
- The UEFI driver is optionally responsible for enumerating and configuring persistent memory devices that are in the boot path.

The following tables describes these high-level roles and responsibilities for major SW components in greater detail. Most of these responsibilities are outlined in more details in the following sections of this document.

**Table 2-1. High-level software component responsibilities – System Boot**

Function	System Firmware CXL Responsibilities	UEFI CXL Bus and Memory Driver Responsibilities	OS CXL CEDT, Bus and Memory Driver Responsibilities
<b>CXL hierarchy enumeration</b>	Enumerate complete CXL hierarchy for volatile and persistent capacity	Optionally: Enumerate CXL hierarchy for persistent capacity in the boot path only	Enumerate complete CXL hierarchy for volatile and persistent capacity
<b>MMIO BAR Configuration</b>	<ul style="list-style-type: none"> <li>• Configure BARs as needed to enumerate the CXL hierarchy it is responsible for</li> </ul>	<ul style="list-style-type: none"> <li>• Optionally: Configure BARs as needed to enumerate the CXL hierarchy</li> </ul>	Optionally: Configure BARs as needed to enumerate the CXL hierarchy
<b>Creating persistent memory region labels</b>	N/A – System Firmware does not create persistent memory region labels	When no region labels exist in the device's LSA: <ul style="list-style-type: none"> <li>• Partition the device volatile and persistent boundary according to the device's capabilities and admin policy. If the OS and device support re-partitioning without a reboot (SetPartitionInfo w Immediate flag), UEFI and OS should assume the CDAT may have changed.</li> <li>• Check the available System Firmware programmed CFMWSs available and only allow configuring of persistent memory regions that match the available windows and HB interleave granularity and ways</li> <li>• Write the region labels following the CXL spec</li> </ul>	
<b>Consuming persistent memory region labels</b>	N/A – System Firmware does not consume persistent memory region labels	<ul style="list-style-type: none"> <li>• Read the region labels from each memory device, verify requested configuration</li> <li>• Programming the device for the region label defined</li> </ul>	

Function	System Firmware CXL Responsibilities	UEFI CXL Bus and Memory Driver Responsibilities	OS CXL CEDT, Bus and Memory Driver Responsibilities
<b>Programming CXL HDM decoders for configured regions</b>	<p>Program platform for platform specific volatile CXL capacity:</p> <ul style="list-style-type: none"> <li>For device HDM decoders, program device HDM decoder global control register to enable HDM use, disabling DVSEC decoder use</li> <li>Program HDM decoders in the memory device, upstream switch ports, and CXL Host Bridges to decode volatile memory.</li> <li>Utilize the CDAT DSMAS memory ranges returned by the memory device, and program HDMs aligned to those ranges</li> <li>For immutable volatile configurations, set fixed device configuration indicator in CFMWS and utilize Lock on Commit to prevent others from changing these HDMs</li> <li>Program all platform HW for each fixed memory region that the platform supports and surface fixed windows through ACPI CXL Fixed Memory Window Structures (CFMWS) for HB and XHB interleaving. It is assumed the windows reported represent the platforms best performance configuration.</li> <li>Implement QoS Throttling Group DSM object under to the ACPI0017 device and map device latency and bandwidth to supported platform CFMWS QTG</li> </ul>	<p>For all volatile and memory devices not configured by the System Firmware:</p> <ul style="list-style-type: none"> <li>Track which volatile capacity has already been assigned an HPA range by the System Firmware by checking the devices HDM decoders</li> <li>Utilize the CDAT DSMAS memory ranges returned by the memory device, the QoS Throttling Group (QTG) from the platform, and the QTG from the CFMWS and program HDMs aligned to the DSMAS ranges.</li> <li>Place device in to fixed memory window and use HPA range for programming HDM decoders while avoiding HPA ranges already populated by the System Firmware.</li> <li>For device HDM decoders, program device HDM decoder global control register to enable HDM use, disabling DVSEC decoder use</li> <li>Program HDM decoders in the memory device upstream switch ports, and CXL Host Bridges to decode volatile and persistent memory</li> </ul>	



Function	System Firmware CXL Responsibilities	UEFI CXL Bus and Memory Driver Responsibilities	OS CXL CEDT, Bus and Memory Driver Responsibilities
<b>Event interrupt enabling</b>	FW First: For platform specific volatile CXL capacity: <ul style="list-style-type: none"> <li>Program memory device event log interrupt steering for FW first MEFN vector based on platform options through _OSC, retain ownership of CXL device memory error reporting</li> </ul>	N/A	<ul style="list-style-type: none"> <li>For all volatile and memory devices not configured by the System Firmware: Determine OS First/FW First using _OSC</li> <li>OS First: Program memory device event log interrupt steering for OS First MSI/MSIx</li> <li>FW First: Do not program the device event log interrupts</li> </ul>
<b>Event interrupt handling</b>	FW First: <ul style="list-style-type: none"> <li>Use host specific means to determine which CXL device(s) generated MEFN</li> <li>Check device event log source to determine what logs need harvesting</li> <li>Harvest appropriate event logs, consume event log content, generate WHEA/ELOG CPER entries for the CXL Event Records, notify OS via correctable (SCI) or uncorrectable interrupt (example: NMI)</li> <li>Clear event records</li> <li>Repeat until all logs retrieved and cleared</li> </ul>	N/A	OS First: <ul style="list-style-type: none"> <li>Use OS specific means to locate the device needing attention</li> <li>Check if the device generated MSI/MSI-X because of memory events</li> <li>Check device event log source to determine what logs need harvesting</li> <li>Harvest appropriate event logs, consume event log content</li> <li>Clear event records</li> <li>Repeat until all logs retrieved and cleared</li> </ul> FW First: <ul style="list-style-type: none"> <li>Handle WHEA SCI and NMI interrupts, consume CPER CXL Event Records.</li> </ul>

Function	System Firmware CXL Responsibilities	UEFI CXL Bus and Memory Driver Responsibilities	OS CXL CEDT, Bus and Memory Driver Responsibilities
<b>ACPI0016 ACPI object</b>	<ul style="list-style-type: none"> <li>Create ACPI0016 ACPI objects for each CXL Host Bridge in ACPI namespace</li> <li>Implement standard PCIe Host Bridge methods associated with CXL Host Bridges and CXL_OSC</li> </ul>	N/A	<ul style="list-style-type: none"> <li>Consume ACPI0016 objects and utilize standard PCIe Host Bridge methods for enumeration and configuration</li> </ul>
<b>ACPI0017 ACPI object</b>	<ul style="list-style-type: none"> <li>Create single ACPI0017 ACPI object for the platform</li> <li>Publish CEDT, CHBS, CFMWS</li> <li>Implement Get QTG DSM associated with ACPI0017</li> </ul>	N/A	<ul style="list-style-type: none"> <li>Consume ACPI0017 object and consume CEDT, CHBS, CFMWS</li> <li>Utilize the Get QTG DSM when determining best CFMWS to utilize when programming HDMs during enumeration</li> </ul>
<b>EFI_MEMORY_MAP</b>	<ul style="list-style-type: none"> <li>Create EFI_MEMORY_DESCRIPTORs for all volatile CXL memory present at boot</li> <li>No descriptors for persistent memory ranges as the System Firmware does not program those HDMs</li> <li>No descriptors for hot plug memory ranges as the System Firmware does not program those HDMs</li> </ul>	N/A	<ul style="list-style-type: none"> <li>Consume EFI_MEMORY_MAP as needed for volatile capacity for legacy functionality</li> </ul>
<b>SRAT</b>	<ul style="list-style-type: none"> <li>Create proximity domains for CPUs, attached CXL Host Bridges using Generic Port Affinity Type, and all CXL volatile memory devices</li> <li>No SRAT entries for intermediate switches</li> <li>Build Memory Affinity Structures for each volatile proximity domain with the SRAT Enable flag set.</li> </ul>	N/A	<ul style="list-style-type: none"> <li>Consume SRAT as needed for volatile capacity for legacy functionality</li> </ul>
<b>HMAT and CDAT</b>	<p>For memory devices containing volatile capacity:</p> <ul style="list-style-type: none"> <li>Parse device and switch CDAT and create HMAT entries for CPU and volatile memory proximity domains found in the SRAT</li> </ul>	N/A	<p>For all persistent capacity:</p> <p>Utilize memory device CDAT, switch CDATs, and Generic Port entries to calculate total BW and Latency for the path from the CXL Host Bridge to each device</p>

Function	System Firmware CXL Responsibilities	UEFI CXL Bus and Memory Driver Responsibilities	OS CXL CEDT, Bus and Memory Driver Responsibilities
<b>Security</b>		For memory devices containing persistent capacity: <ul style="list-style-type: none"> <li>• Unlock device before the memory is accessed</li> </ul>	For memory devices containing persistent capacity: <ul style="list-style-type: none"> <li>• Unlock device before the memory is accessed</li> </ul>
<b>Managing device health</b>	For memory devices containing volatile capacity: <ul style="list-style-type: none"> <li>• Check health status before configuring or utilizing memory device</li> </ul>	For memory devices containing persistent capacity: <ul style="list-style-type: none"> <li>• Check health status before configuring or utilizing memory device</li> <li>• SetShutdownState with state=DIRTY before mapping the capacity by programming HDM decoders</li> </ul>	For memory devices containing unconfigured volatile or persistent capacity: <ul style="list-style-type: none"> <li>• Check health status before configuring or utilizing memory device</li> <li>• Retrieve DSC from previous boot and make available to kernel or application components and if DSC has incremented, optionally prevent device access or verify critical data is intact before allowing device to be accessed</li> <li>• SetShutdownState with state=DIRTY before mapping the capacity by programming HDM decoders</li> </ul>

**Table 2 - High-level software component responsibilities - System Shutdown and GPF**

Function	System Firmware CXL Responsibilities	UEFI CXL Bus and Memory Driver Responsibilities	OS CXL CEDT, Bus and Memory Driver Responsibilities
<b>Preparing for device shutdown</b>	<p>If System Firmware OS shutdown handler present: Check each device</p> <p>GetShutdownState == CLEAN and if DIRTY:</p> <ul style="list-style-type: none"> <li>If CPU caches are part of the platform persistence domain: Flush all CPU caches, all Type 2 devices</li> <li>SetShutdownState state=CLEAN after flushing data to the device</li> </ul>	<ul style="list-style-type: none"> <li>N/A</li> </ul>	<ul style="list-style-type: none"> <li>Quiesce memory accesses to the device</li> <li>If FADT.PERSISTENT_CPU_CACHES = 10b: Optionally flush all CPU caches, all Type 2 devices</li> <li>If HDM is not locked: Un-program device HDMs to prevent any other writes</li> <li>SetShutdownState state=CLEAN after flushing data to the device</li> </ul>
<b>Global Persistent Flush (GPF) Initialization</b>	<ul style="list-style-type: none"> <li>Enable GPF on the platform</li> <li>Determine if CPU caches are persistent and export FADT.PERSISTENT_CPU_CACHES ACPI interface</li> </ul>		<ul style="list-style-type: none"> <li>During CXL enumeration: Calculates and configures CXL GPF Timeouts for switches and host bridges the volatile and persistent capacity is connected to</li> </ul>

**Table 3 - High-level software component responsibilities - Hot Add**

Function	System Firmware CXL Responsibilities	UEFI CXL Bus and Memory Driver Responsibilities	OS CXL CEDT, Bus and Memory Driver Responsibilities
Managing regions	<ul style="list-style-type: none"> <li>Program all platform HW for each fixed memory region that the platform supports for hot-plug of volatile or persistent memory and surface fixed windows through ACPI CXL Fixed Memory Window Structures (CFMWS) for HB and XHB interleaving.</li> </ul>	Not Supported	<p>Upon hot-add event: Hot added volatile and persistent memory devices:</p> <ul style="list-style-type: none"> <li>Program HDM decoders for memory device based on assigned HPA range, HDM decoders for upstream switch ports, HDM decoders for CXL Host Bridges</li> <li>Reprogram GPF timeouts and other values that depend on the number of device present, for CXL switches and Host Bridges</li> <li>Utilize the CDAT DSMAS memory ranges supported by the memory device, the QoS Throttling Group (QTG) from the platform, and the QTG from the CFMWS and program HDMs aligned to the DSMAS ranges.</li> <li>Utilize the Get QTG DSM when determining best CFMWS to utilize when programming HDMs</li> </ul>
Event interrupt steering	N/A		<p>Hot added volatile and persistent memory devices:</p> <ul style="list-style-type: none"> <li>OS First: Program memory device event log interrupt steering for OS First MSI/MSIx based on _OSC</li> <li>FW First: Skip programming memory device event log interrupts</li> </ul>
HMAT and CDAT	N/A - _HMA method not required since OS will handle natively		<p>Hot added volatile and persistent memory devices:</p> <ul style="list-style-type: none"> <li>Utilize memory device CDAT, switch CDATs, and CXL Host Bridge HMAT information to calculate total BW and Latency for the path from the CXL Host Bridge to the new device</li> </ul>

Function	System Firmware CXL Responsibilities	UEFI CXL Bus and Memory Driver Responsibilities	OS CXL CEDT, Bus and Memory Driver Responsibilities
SRAT	<ul style="list-style-type: none"> <li>Indicate hot pluggable proximity domains with Memory Affinity Structure HotPluggable indicator</li> </ul>	Not Supported	
GPF	N/A		Hot added volatile and persistent memory devices: <ul style="list-style-type: none"> <li>Read existing switch and host bridge GPF TO values, calculate updated values for persistent memory, and configure CXL GPF Timeouts for switches, and host bridges the persistent memory is connected to</li> </ul>

**Table 4 - High-level software component responsibilities - Managed Hot Remove**

Function	System Firmware CXL Responsibilities	UEFI CXL Bus and Memory Driver Responsibilities	OS CXL CEDT, Bus and Memory Driver Responsibilities
<b>Preparing for device removal</b>	N/A	Not Supported	<p>Removal of volatile or persistent capacity that was previously mapped:</p> <ul style="list-style-type: none"> <li>• If memory pages cannot be vacated, or device removal cannot be supported, don't allow removal</li> <li>• Quiesce all memory accesses</li> <li>• Offline/unmap the memory range so the device memory becomes inaccessible</li> <li>• Flush all CPU caches (what GPF would have done)</li> <li>• If HDM decoders are not locked: Un-program the HDM Decoders for the device being removed</li> <li>• SetShutdownState state=CLEAN</li> </ul>
<b>Managing regions</b>		<ul style="list-style-type: none"> <li>•</li> </ul>	<p>Managed Hot Remove of volatile capacity that was previously mapped:</p> <ul style="list-style-type: none"> <li>• Only allow removal of volatile capacity whose HDM decoders were not previously locked by the System Firmware. System Firmware should only lock HDM decoders for immutable configurations that cannot support hot add or managed hot remove without a system reboot.</li> </ul> <p>Managed hot removed of volatile or persistent capacity that was previously mapped:</p> <ul style="list-style-type: none"> <li>• Determine what HPA range is being removed, program HDM decoders for affected switch ports, HDM decoders for CXL Host Bridges to remove assigned HPA range</li> <li>• Reprogram GPF timeouts and other values that depend on the number of device present, for CXL switches and Host Bridges</li> </ul>

## 2.5 High-level System Firmware memory interface overview

The following table summarizes the legacy memory and CXL related tables the System Firmware produces for consumption by the OS or UEFI drivers. See the SRAT and HMAT section of this document for more details.

**Table 5 - System firmware memory interface summary**

System Firmware Interface	Description	Responsibility for CXL volatile memory capacity	Responsibility for CXL persistent memory capacity	Responsibility for CXL hot-added memory capacity	CXL UEFI and OS driver implications
<b>EFI MEMORY MAP</b>	ACPI/UEFI HPA based memory descriptors describing physically present memory ranges the System Firmware has configured	YES Type EfiConventionalMemory set, NonVolatile clear	NO	NO	Must consume EFI Memory Map



System Firmware Interface	Description	Responsibility for CXL volatile memory capacity	Responsibility for CXL persistent memory capacity	Responsibility for CXL hot-added memory capacity	CXL UEFI and OS driver implications
<b>CEDT CFMWS</b>	ACPI table describing all CXL HPA ranges programmed by the System Firmware at boot time	YES	YES	YES	Must consume CEDT CFMWS
<b>SRAT</b>	ACPI table of Proximity Domains and associated memory ranges	<p>YES</p> <p>-CPU proximity domain for each CPU like today</p> <p>-CXL Host Bridge Generic port proximity domain for each CXL Host Bridge</p> <p>-CXL volatile memory proximity domain for each volatile region configured by System Firmware</p> <p>NO SRAT entries for CXL volatile memory not configured by the System Firmware</p>	<p>YES</p> <p>-CPU proximity domain for each CPU like today</p> <p>-CXL Host Bridge Generic port proximity domain for each host bridge</p> <p>NO SRAT entries for CXL persistent memory</p>	<p>YES</p> <p>-Volatile Memory proximity domains will indicate Hot Pluggable but the size field will represent boot time capacity</p> <p>NO SRAT entries for CXL persistent memory</p>	<p>Without persistent memory information in the SRAT and HMAT, must calculate NUMA distances for persistent memory devices using the Generic Port information in the SRAT combined with the memory device and intermediate switch CDAT.</p> <p>See <a href="#">SRAT and HMAT</a></p>

System Firmware Interface	Description	Responsibility for CXL volatile memory capacity	Responsibility for CXL persistent memory capacity	Responsibility for CXL hot-added memory capacity	CXL UEFI and OS driver implications
<b>HMAT</b>	ACPI table that describes a matrix of BW and Latency performance characteristics between each Initiator (CPU or GI) proximity domain and each Memory proximity domain	YES, covers all of the Initiator and Target proximity domains that are listed in the SRAT	NO	N/A	Section for details

## 2.6 Memory provisioning

- The proposed memory allocation scheme outlined here utilizes a fixed, platform defined set of HPA memory windows that are fixed at platform boot time based on the supported configurations and features for that platform.
- System Firmware fixed Host Physical Address (HPA) based memory windows and restrictions for using each window, are described in the CEDT CXL Fixed Memory Window Structure (CFMWS) to the UEFI and OS drivers.
- CFWMS entries are produced by System Firmware and consumed by the OS.
  - The System Firmware will configure fixed memory windows for use with volatile and persistent memory.
  - The System Firmware will allocate volatile capacity from the volatile memory windows and program HDM decoders for the volatile capacity.
  - The OS will allocate persistent capacity from the persistent memory windows and program HDM decoders for persistent capacity based on region labels read from the LSA.
  - The OS will also allocate volatile and persistent capacity from the same windows for hot adding new volatile/persistent memory or managed hot removing existing volatile/persistent memory.
  - The OS must check programmed CXL memory HDM decoders during enumeration and understand what devices are already utilizing portions of the fixed window HPA range.
- By utilizing this simple fixed mechanism, runtime OS interactions to request the System Firmware to set up or tear down resources during OS boot time enumeration, runtime Hot Add and Managed Hot Remove events, are eliminated.

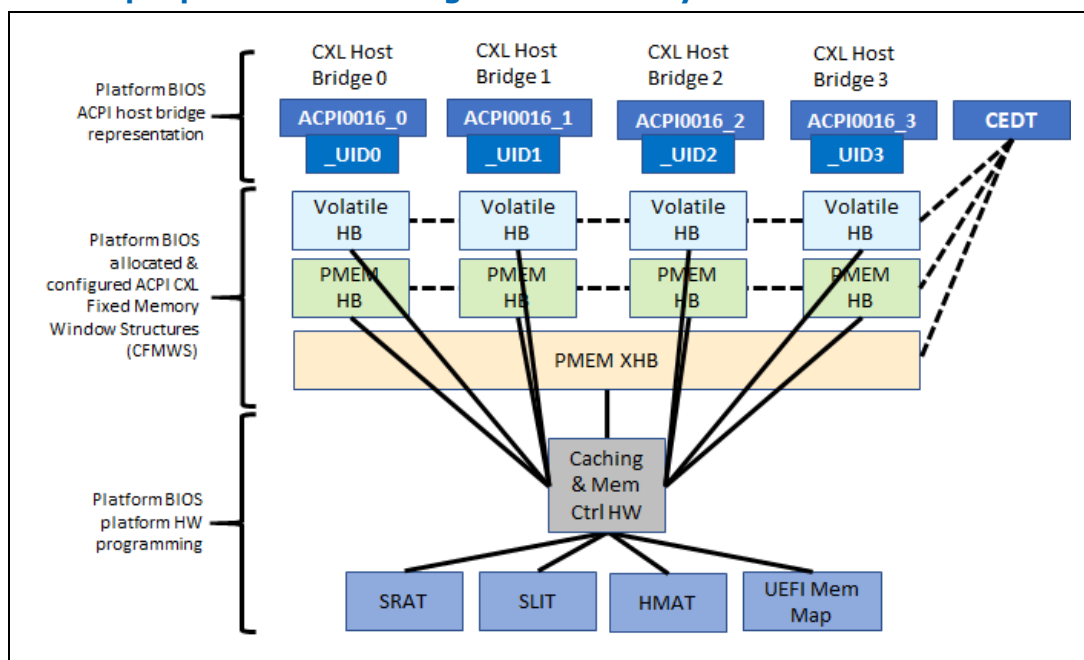
The following requirements limit what the System Firmware may surface to the UEFI and OS drivers:

- The architecture would allow mixing of any combination of the Window Restrictions. It is the responsibility of the System Firmware to only surface windows with Window Restrictions that the platform supports. If the platform HW does not allow T2 and T3 CXL memory devices to utilize the same HPA range, the System Firmware cannot report both T2 and T3 Window Restrictions in a single window.
- The System Firmware may surface windows that provide the UEFI and OS drivers multiple options for configuring a given region. It is UEFI and OS driver policy specific as to which possible window is utilized for configuring the region.
- There cannot be any overlap in the HPA ranges described in any of the CFMWS instances.

The following figure demonstrates an example of the intended architecture with the following example fixed windows:

- Volatile HB – Window for adding interleaved volatile memory that is local to this HB.
- PMEM HB – Window for adding interleaved and non-interleaved persistent memory that is local to this HB.
- PMEM XHB – Window for adding interleaved persistent memory that spans multiple HBs.

**Figure 4 - Example per CXL Host Bridge fixed memory allocation**



See the next examples that outline the intended content of the CFMWS, for various CXL topologies.



### **2.6.1 EFI\_MEMORY\_MAP**

The CEDT CFMWS structures are based on the resources the System Firmware has allocated and programmed with the platform HW. See previous sections for the outline of the EFI\_MEMORY\_MAP content.

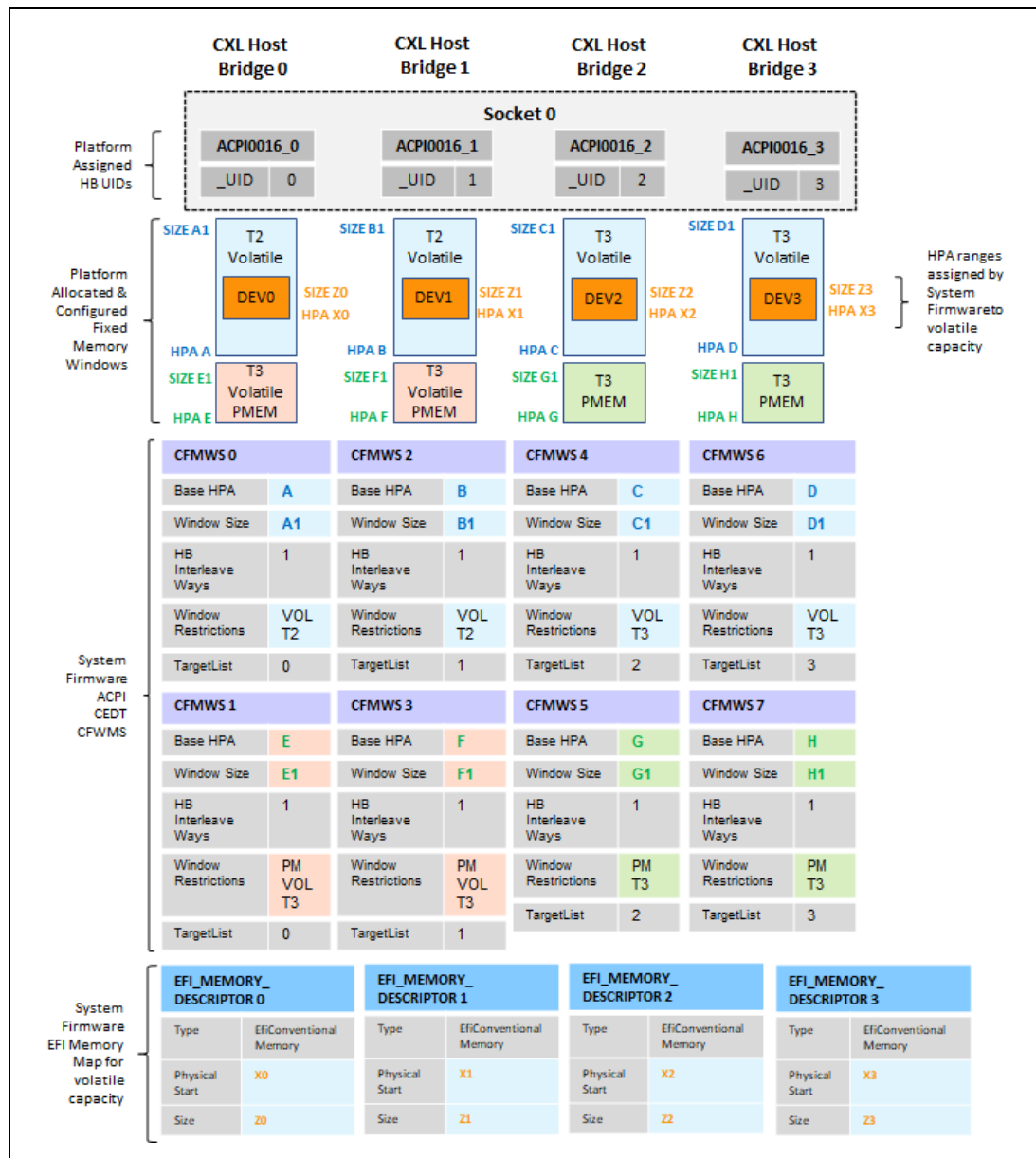
The next examples outline the intended content of both the EFI\_MEMORY\_MAP and CFMWS, for various CXL topologies.

## 2.6.2 CEDT CFMWS Example – No XHB interleaving

The following example demonstrates how the System Firmware might set up the fixed memory windows for each CXL Host Bridge and the resulting CEDT CXL Fixed Memory Window Structure (CFMWS).

In this example there is no XHB interleaving.

**Figure 5 - CEDT CFMWS Example - No XHB interleaving**

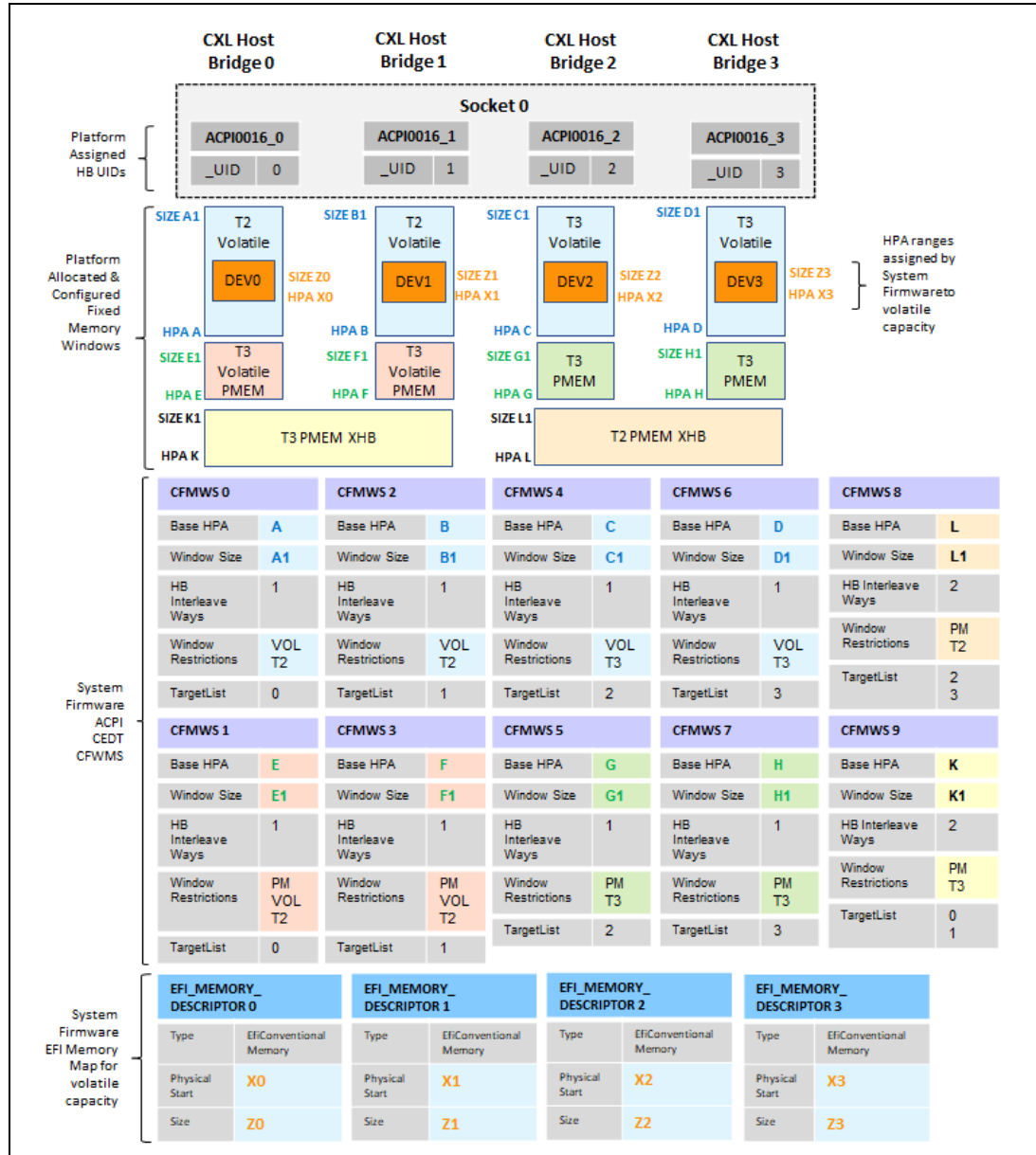


## 2.6.3 CEDT CFMWS Example – x2 XHB interleaving

The following example demonstrates how the System Firmware might set up the fixed memory windows for each CXL Host Bridge and the resulting CEDT CXL Fixed Memory Window Structure (CFMWS).

In this example, the platform supports a x2 Persistent XHB interleave between CXL Host Bridges 0 and 1 and a x2 Persistent XHB interleave between CXL Host Bridges 2 and 3.

**Figure 6 - CEDT CFMWS Example - x2 XHB interleave**

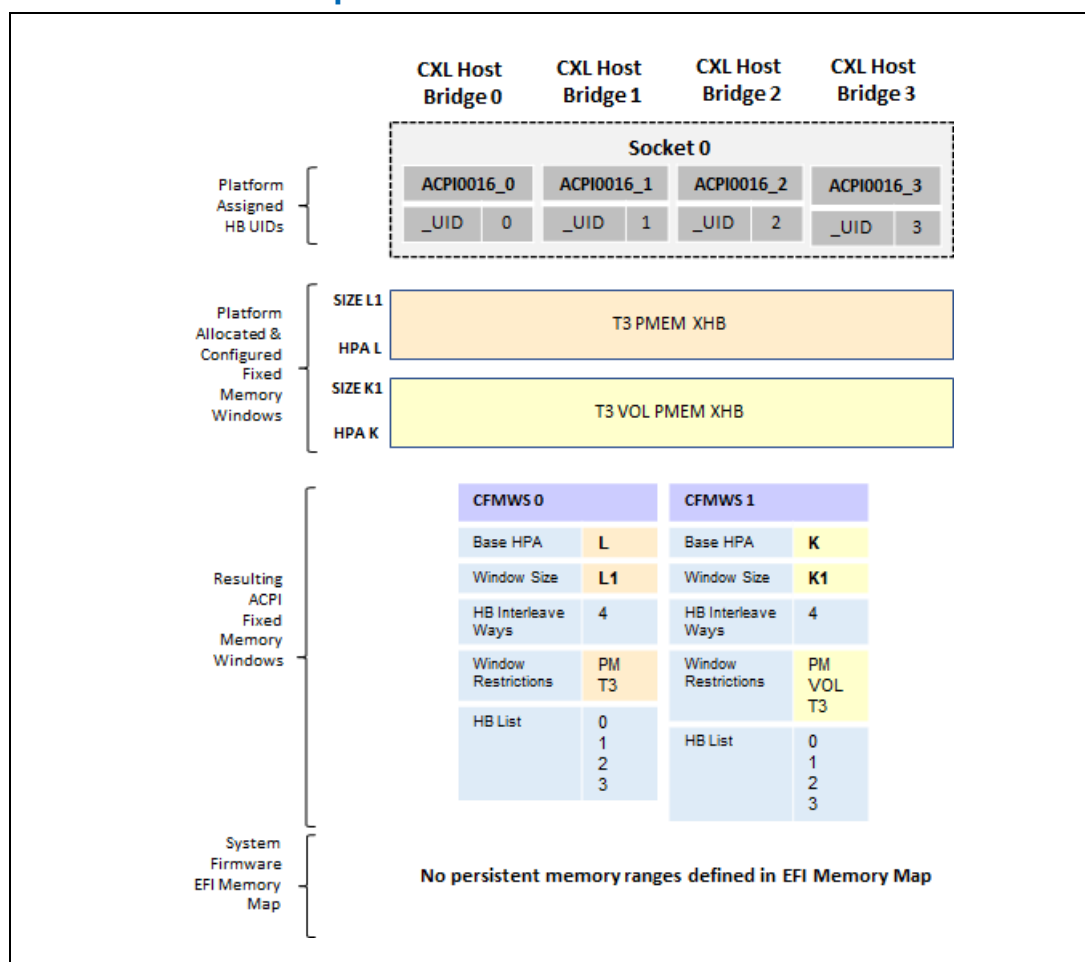


## 2.6.4 CEDT CFMWS Example – x4 XHB interleaving

The following example demonstrates how the System Firmware might set up the fixed memory windows for each CXL Host Bridge and the resulting CEDT CXL Fixed Memory Window Structure (CFMWS).

In this example, the platform supports a x4 Type 3 persistent XHB interleave between CXL Host Bridges 0, 1, 2 and 3 and a x4 Type 3 volatile or persistent XHB interleave between CXL Host Bridges 0, 1, 2 and 3. Note that in this example, the EFI MEMORY MAP contains no persistent memory descriptors, but it is assumed there would be other volatile memory descriptors, not shown.

**Figure 7 - CEDT CFMWS Example - x4 XHB interleave**

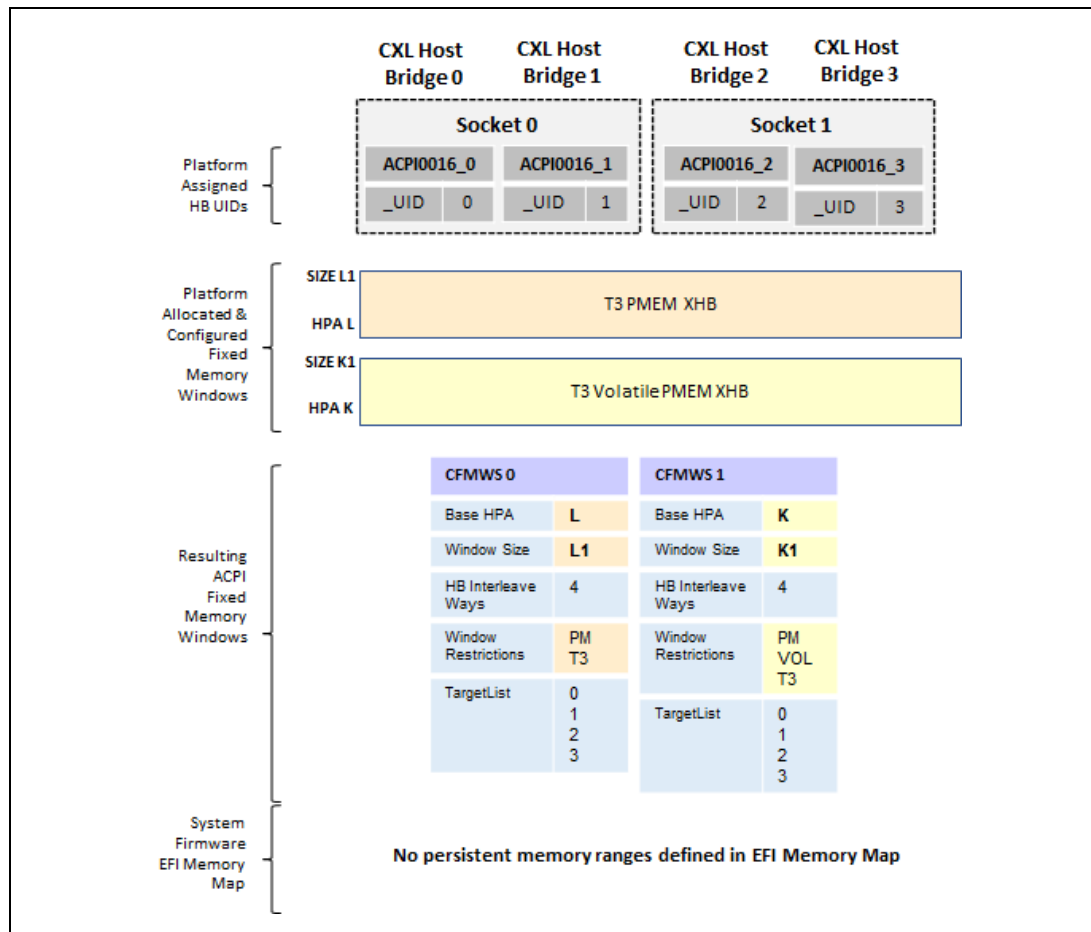


## 2.6.5 CEDT CFMWS Example – x4 XHB interleaving across multiple sockets

The following example demonstrates how the System Firmware might set up the fixed memory windows for each CXL Host Bridge and the resulting CEDT CXL Fixed Memory Window Structure (CFMWS).

In this example, the platform supports a x4 Type 3 persistent XHB interleave can send between CXL Host Bridges 0, 1, 2 and 3 and a x4 Type 3 volatile or persistent XHB interleave between CXL Host Bridges 0, 1, 2 and 3. CXL Host Bridges 0 and 1 are in socket 0 and CXL Host Bridges 3 and 4 are in socket 1. Note that in this example, the EFI MEMORY MAP contains no persistent memory descriptors, but it is assumed there would be other volatile memory descriptors, not shown.

**Figure 8 - CEDT CFMWS Example - x4 XHB interleave across multiple sockets**



Since the CXL Fixed Memory Windows surfaced in the CEDT are based on the CXL Host Bridge \_UIDs, which must be unique across the platform, the resulting windows are identical to the previous example. Thus, the organization of CXL Host Bridges into sockets from a hardware perspective, imposes no



restrictions on which CXL Host Bridges can be utilized in an XHB interleave. The platform is free to choose which CXL Host Bridges it will allow to be interleaved together without having to describe the hardware restrictions to the UEFI and OS drivers.

## 2.7 Managing persistent regions

The next text outlines the responsibilities of the System Firmware, UEFI and OS drivers to manage persistent regions across the CXL fabric.

### HDM decoders

- Contain the Host Physical Address (HPA) Base and Size programmed by the System Firmware, UEFI and OS drivers at CXL hierarchy enumeration time.
- There is no Device Physical Address (DPA) programming in the HDM decoders. The DPA is inferred by the device based on the HDM decoder programming.
- In this architecture the System Firmware should utilize the “lock on Commit” feature to lock programmed HDMs for volatile configurations that utilize any CFMWS with the `CFMWS.WindowRestrictions.FixedDeviceConfiguration` set.
- UEFI and OS drivers should not utilize the “lock on Commit” for the HDM decoders so the UEFI and OS driver are free to re-program HDMs for any device, at any time, governed by the CFMWS Windows Restrictions and the QoS Throttling Group.
- The CXL spec requires the memory device Get Partition Info reported volatile capacity must start with DPA 0. The volatile or persistent capacity also does not need to be described by a single decoder; however, all the volatile capacity must be programmed first. If there is no volatile capacity, the first device decoder can be used for persistent memory ranges.
- These rules force the System Firmware, UEFI and OS drivers to program the lowest HDM decoder with volatile capacity first, and only after all volatile DPA range has been accounted for (either programmed or skipped) can the System Firmware, UEFI and OS drivers program any persistent memory ranges in the remaining HDM decoders.

### System Firmware Region Management Responsibilities

- The System Firmware reports the base address to the CXL Host Bridge MMIO register block (CHBCR) in the CXL Early Discovery Table (CEDT) for each CXL Host Bridge discovered by the System Firmware, at platform boot time.
- The System Firmware creates a separate ACPI0016 device instance in the ACPI device tree for every CXL Host Bridge discovered by the System Firmware, at platform boot time.
- The System Firmware produces the CEDT ACPI table which should contain a CHBS instance for each host bridge present at boot and one or more CFMWS instances to describe the platform resources available to the UEFI

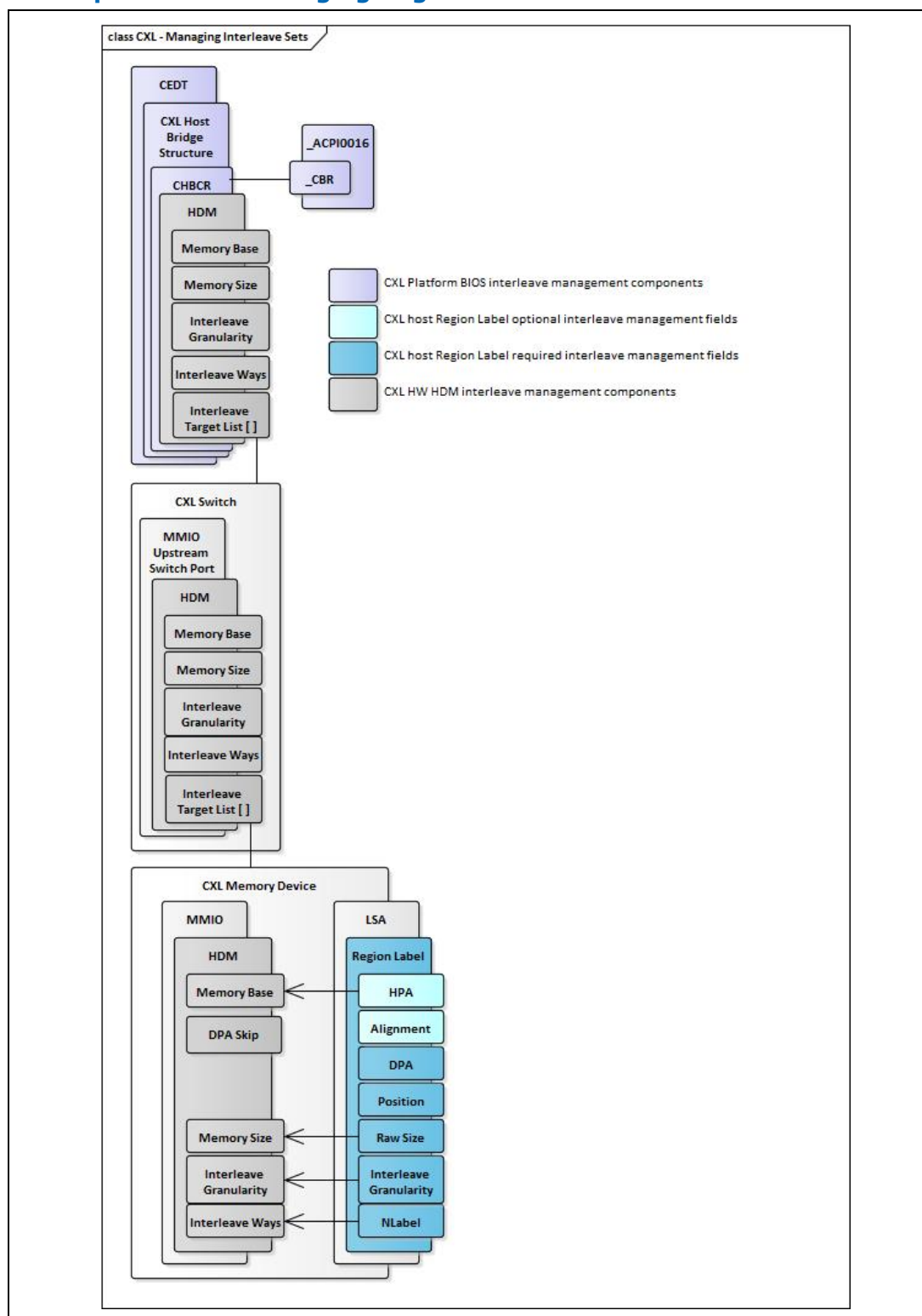
and OS CXL bus driver.

## UEFI and OS Region Management Responsibilities

- UEFI and OS drivers enumerate the entire CXL hierarchy of CXL memory devices and CXL switches following the PCIe/CXL specified bus enumeration sequence.
- For each CXL memory device found UEFI and OS drivers:
  - Retrieve device HDM decoder capabilities.
  - OS drivers only: Program device HDM decoder for volatile memory not configured by System Firmware.
  - Program device HDM decoders for persistent memory. Each region label on the device will require a separate HDM decoder to be programmed on the device. The UEFI driver will only need to program HDMs for persistent memory capacity that is part of the persistent memory boot path.
- For each CXL Switch UEFI and OS drivers:
  - Programs the CXL Switch's Upstream Port matching HDM decoder Memory Base, Memory Size, Interleave Granularity, Interleave Ways based on the summation of all the downstream devices and switches (relative to this Host Bridge) HDM decoder programming.
  - Based on the Pos field in the region label on each device in the region, program the CXL Switch's Upstream Port matching HDM decoder Target List. The order of the CXL Downstream Switch Ports in the target list must match the configured ordering of each device in the region to maintain consistent region data.
- For each CXL Host Bridge the UEFI and OS drivers:
  - Program the CXL Host Bridge matching HDM decoder Memory Base, Memory Size, Interleave Granularity, Interleave Ways based on the summation of all the downstream devices and switches (relative to this HB) HDM decoder programming.
  - Based on the Pos field in the region label on each device in the region, program the CXL Host Bridge's root port matching HDM decoder Target List. The order of the CXL Root Ports in the target list must match the configured ordering of each device in the region to maintain consistent region data.
- For UEFI and OS drivers that update the interleave set/region configuration data as part of managing the interleave sets:
  - In addition to the previous responsibilities, UEFI and OS drivers that update the region labels must follow the power fail safe label update rules as specified in the CXL specification.
  - After updating region configuration information on the device, UEFI and OS drivers are responsible for re-programming the CXL HDM decoders for all effected switch ports and CXL Host Bridges, as outlined previously.

The following figure demonstrates the components involved in managing regions.

**Figure 9 - Components for Managing Regions**

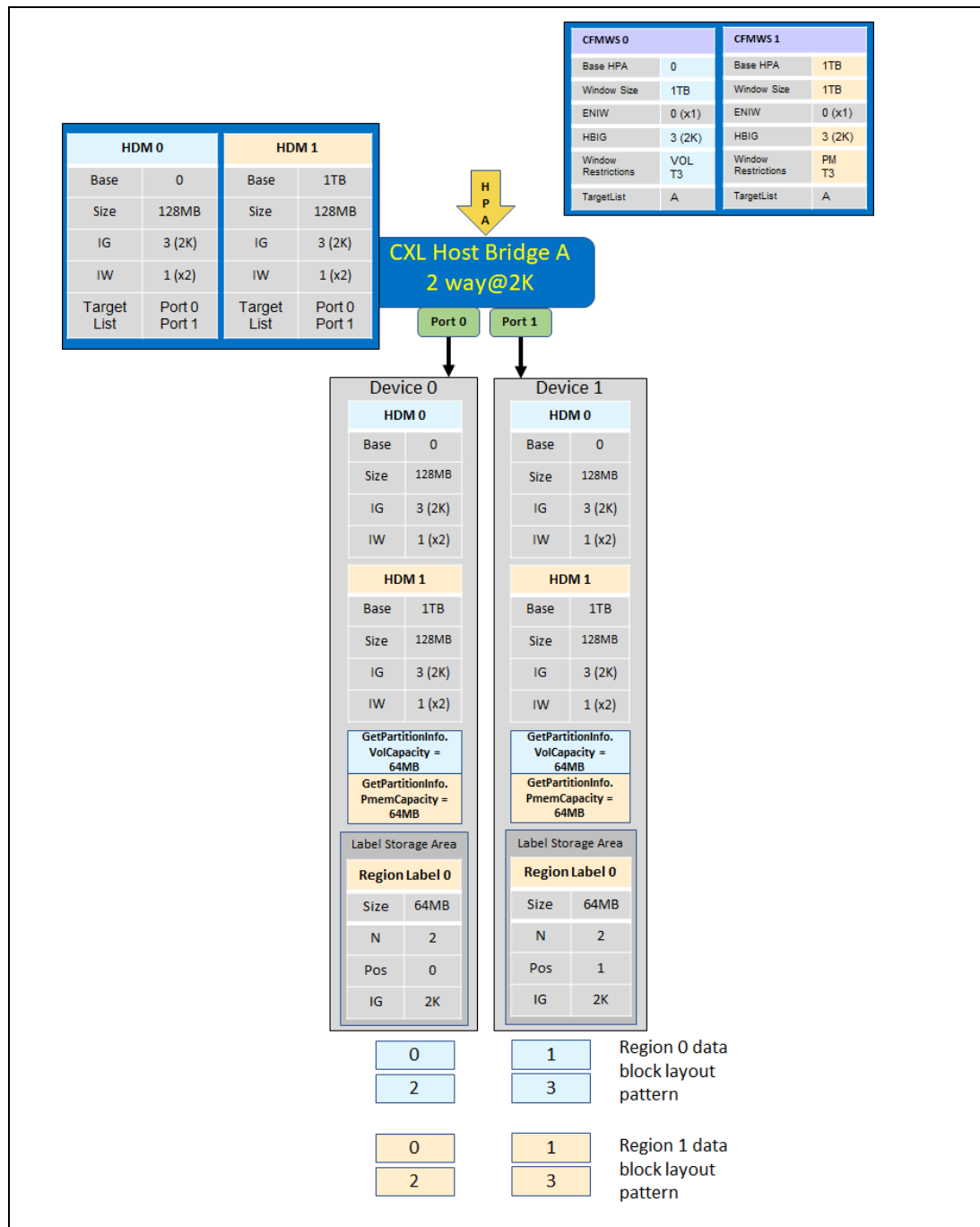


### **2.7.1 Example – Volatile and persistent x2 interleaved regions**

This example demonstrates contents of the region label retrieved from each device's LSA, the System Firmware CFMWS, and how the System Firmware, UEFI and OS drivers would program the HDM decoders in each memory device, each switch, and each HB to implement the desired interleave.

In this example, 2 devices are interleaved together with volatile capacity and the same 2 devices are interleaved together with persistent capacity and each device contributes 64MB of volatile and persistent capacity, for simplicity.

**Figure 10 - Example - Volatile and persistent x2 interleaved regions**



Here is example Linux SYSFS hierarchy demonstrating how this configuration would be interpreted by the CXL Subsystem component in the Linux architecture.

Note: This is provisional/draft output and will be updated with more accurate text in another release.

```
/sys/bus/cxl/devices/root0
├── address_space0
│   ├── devtype
│   ├── end
│   ├── start
│   ├── supports_ram
│   ├── supports_type3
│   └── uevent
├── address_space1
│   ├── devtype
│   ├── end
│   ├── start
│   ├── supports_pmem
│   ├── supports_type3
│   └── uevent
├── ...
├── devtype
├── dport0 -> ../LNXXSYSTM:00/LNXXSYBUS:00/ACPI0016:00
├── port1
│   ├── decoder1.0
│   │   ├── devtype
│   │   ├── end
│   │   ├── locked
│   │   ├── start
│   │   ├── subsystem -> ../../../../bus/cxl
│   │   ├── target_list
│   │   ├── target_type
│   │   └── uevent
│   ├── devtype
│   ├── dport0 -> ../../pci0000:34/0000:34:00.0
│   ├── subsystem -> ../../../../bus/cxl
│   ├── uevent
│   ├── uport -> ../../LNXXSYSTM:00/LNXXSYBUS:00/ACPI0016:00
│   ├── subsystem -> ../../../../bus/cxl
│   ├── uevent
│   └── uport -> ../../platform/ACPI0017:00
└── ...
```

// CFMWS0  
// cxl\_address\_space  
// 1TB  
// 0  
  
// CFMWS1  
// cxl\_address\_space  
// 1TB  
// 1TB  
  
// cxl\_root  
// Host Bridge A  
// Port 0  
// HDM 0  
  
// cxl\_port  
// Port 0 Address  
  
// HB assignment  
  
// HB parent

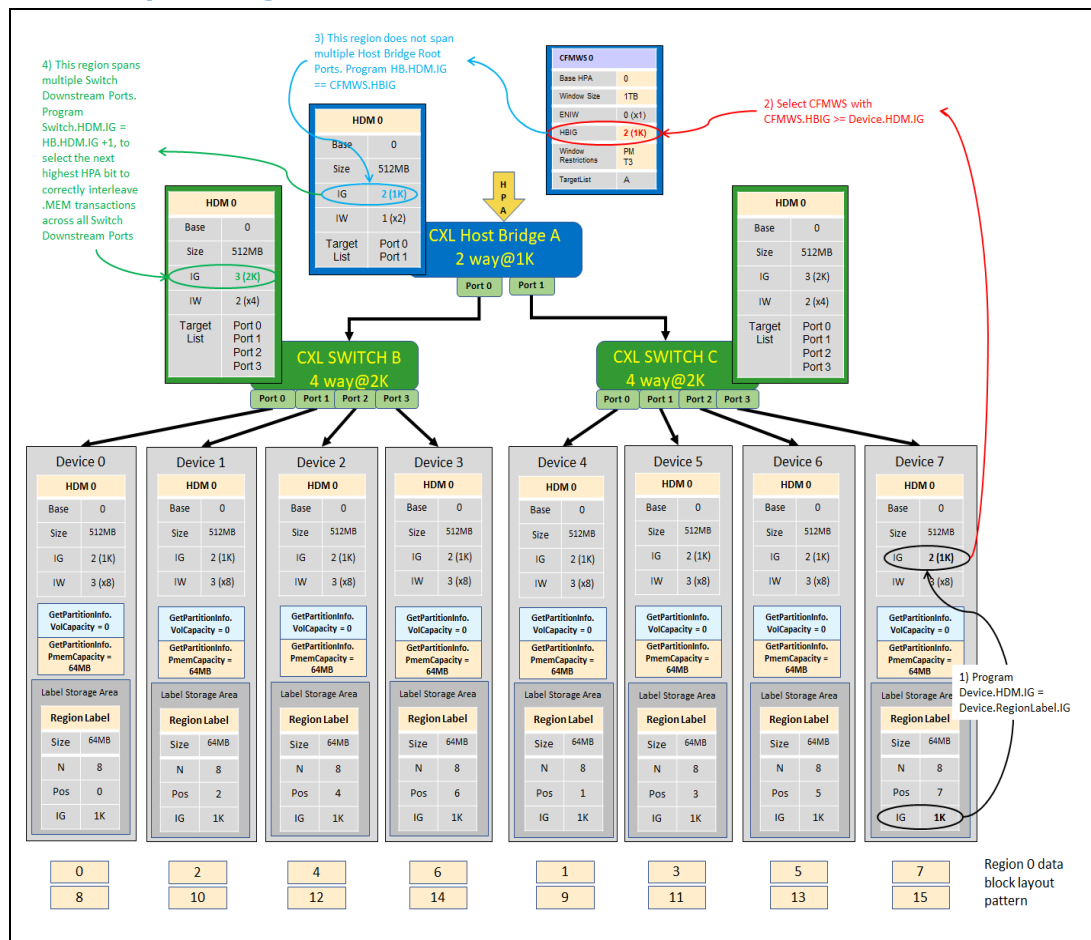
## 2.7.2 Example – Region interleaved across 2 CXL switches

This example demonstrates contents of the region label retrieved from each device's LSA, the System Firmware CFMWS, and how the System Firmware, UEFI and OS drivers would program the HDM decoders in each memory device, each switch, and each HB to implement the desired interleave.

In this example, all 8 devices are interleaved together into a single region and each device contributes 64MB of persistent capacity, for simplicity.

Note how the Switch interleave granularity is programmed to 2K and the Host Bridge interleave granularity is programmed to 1K, so that each switch utilizes HPA[12:11] and the Host Bridge utilizes HPA[10] to select the target port. This allows proper distribution of the HPA in a round robin fashion across all the ports in each HostBridge.HDM.InterleaveTargetList[ ] and Switch.HDM.InterleaveTargetList[ ] the region is associated with. This leads to good performance since maximum distribution of memory requests across all HBs and switches is achieved, as shown in the resulting region data block layout pattern at the bottom of the figure.

**Figure 11 - Example - Region interleaved across 2 switches**



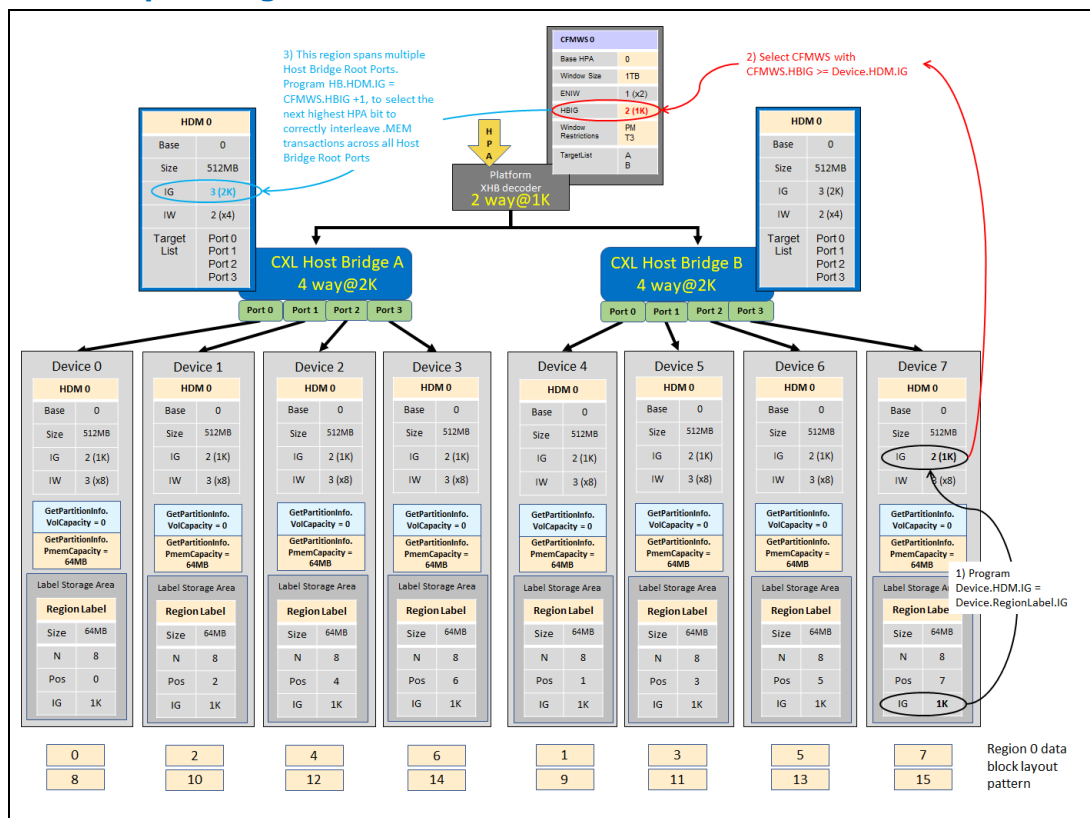
## 2.7.3 Example – Region interleaved across 2 HBs

This example demonstrates contents of the region label retrieved from each device's LSA, the System Firmware CFMWS, and how the System Firmware, UEFI and OS drivers would program the HDM decoders in each memory device, and each HB to implement the desired interleave.

In this example, all 8 devices are interleaved together into a single region that spans multiple CXL Host Bridges and each device contributes 64 MB bytes of persistent capacity, for simplicity.

Note how the Host Bridge interleave granularity is programmed to 2K and the platform XHB interleave granularity is pre-programmed to 1K so that the Host Bridge utilizes HPA[11] to select the root port and the platform XHB interleave granularity utilizes HPA[10] to select the Host Bridge. This allows proper distribution of the HPA in a round robin fashion across all the Host Bridges in the platform XHB CFMWS.InterleaveTargetList[ ] and each root port in the HostBridge.HDM.InterleaveTargetList[ ] the region is associated with. This leads to good performance since maximum distribution of memory requests across all HBs and switches is achieved, as shown in the resulting region data block layout pattern at the bottom of the figure.

**Figure 12 - Example - Region interleaved across 2 HBs**



Here is example Linux SYSFS output demonstrating how this configuration would be interpreted by the CXL Subsystem component in the Linux architecture.



Note: This is provisional/draft output and will be updated with more accurate text in another release.

```

/sys/bus/cxl/devices/root0
├── address_space0
│   ├── devtype
│   │   └── CFMWS0
│   ├── end
│   │   └── cxl_address_space
│   ├── start
│   │   └── 1TB
│   ├── supports_pmem
│   ├── supports_type3
│   └── uevent
└── ...
├── devtype
│   └── cxl_root
├── dport0 -> ../LNXYSTYM:00/LNXYBUS:00/ACPI0016:00
│   └── Host Bridge A
├── port1
│   ├── decoder1.0
│   │   └── HDM 0
│   ├── devtype
│   │   └── cxl_port
│   ├── dport0 -> ../pci0000:34/0000:34:00.0
│   │   └── Port 0 Address
│   ├── subsystem -> ../bus/cxl
│   ├── uevent
│   └── uport -> ../LNXYSTYM:00/LNXYBUS:00/ACPI0016:00
│       ├── HB assignment
│       ├── Port 1
│       ├── HDM 0
│       ├── cxl_port
│       └── Port 0 Address
├── port2
│   ├── decoder1.0
│   │   └── HDM 0
│   ├── devtype
│   │   └── cxl_port
│   ├── dport0 -> ../pci0000:35/0000:35:00.0
│   │   └── Port 1 Address
│   ├── subsystem -> ../bus/cxl
│   ├── uevent
│   └── uport -> ../LNXYSTYM:00/LNXYBUS:00/ACPI0016:00
│       ├── HB assignment
│       ├── Port 2
│       ├── HDM 0
│       ├── cxl_port
│       └── Port 2 Address
├── port3
│   ├── decoder1.0
│   │   └── HDM 0
│   ├── devtype
│   │   └── cxl_port
│   ├── dport0 -> ../pci0000:36/0000:36:00.0
│   │   └── Port 2 Address
│   ├── subsystem -> ../bus/cxl
│   ├── uevent
│   └── uport -> ../LNXYSTYM:00/LNXYBUS:00/ACPI0016:00
│       ├── HB assignment
│       ├── Port 3
│       ├── HDM 0
│       ├── cxl_port
│       └── Port 2 Address
├── port4
│   ├── decoder1.0
│   │   └── HDM 0
│   ├── devtype
│   │   └── cxl_port
│   ├── dport0 -> ../pci0000:37/0000:37:00.0
│   │   └── Port 2 Address
│   ├── subsystem -> ../bus/cxl
│   ├── uevent
│   └── uport -> ../LNXYSTYM:00/LNXYBUS:00/ACPI0016:00
│       ├── HB assignment
│       ├── Port 3
│       ├── HDM 0
│       ├── cxl_port
│       └── Port 2 Address
├── subsystem -> ../bus/cxl
├── uevent
├── uport -> ../platform/ACPI0017:00
│   └── HB parent
├── devtype
│   └── cxl_root
├── dport5 -> ../LNXYSTYM:00/LNXYBUS:00/ACPI0016:01
│   └── Host Bridge B
├── port6
│   ├── decoder1.0
│   │   └── HDM 0
│   ├── devtype
│   │   └── cxl_port
│   ├── dport0 -> ../pci0000:38/0000:38:00.0
│   │   └── Port 0 Address
│   ├── subsystem -> ../bus/cxl
│   ├── uevent
│   └── uport -> ../LNXYSTYM:00/LNXYBUS:00/ACPI0016:00
│       ├── HB assignment
│       ├── Port 1
│       ├── HDM 0
│       ├── cxl_port
│       └── Port 1 Address
├── port7
│   ├── decoder1.0
│   │   └── HDM 0
│   ├── devtype
│   │   └── cxl_port
│   ├── dport0 -> ../pci0000:39/0000:39:00.0
│   │   └── Port 1 Address
│   ├── subsystem -> ../bus/cxl
│   ├── uevent
│   └── uport -> ../LNXYSTYM:00/LNXYBUS:00/ACPI0016:00
│       ├── HB assignment
│       ├── Port 2
│       ├── HDM 0
│       ├── cxl_port
│       └── Port 2 Address
├── port8
│   ├── decoder1.0
│   │   └── HDM 0
│   ├── devtype
│   │   └── cxl_port
│   ├── dport0 -> ../pci0000:3A/0000:3A:00.0
│   │   └── Port 2 Address
│   ├── subsystem -> ../bus/cxl
│   ├── uevent
│   └── uport -> ../LNXYSTYM:00/LNXYBUS:00/ACPI0016:00
│       ├── HB assignment
│       ├── Port 3
│       ├── HDM 0
│       ├── cxl_port
│       └── Port 2 Address
├── port9
│   ├── decoder1.0
│   │   └── HDM 0
│   ├── devtype
│   │   └── cxl_port
│   ├── dport0 -> ../pci0000:3B/0000:3B:00.0
│   │   └── Port 2 Address
│   ├── subsystem -> ../bus/cxl
│   ├── uevent
│   └── uport -> ../LNXYSTYM:00/LNXYBUS:00/ACPI0016:00
│       ├── HB assignment
│       ├── Port 3
│       ├── HDM 0
│       ├── cxl_port
│       └── Port 2 Address
├── subsystem -> ../bus/cxl
├── uevent
└── uport -> ../platform/ACPI0017:00
    └── HB parent

```

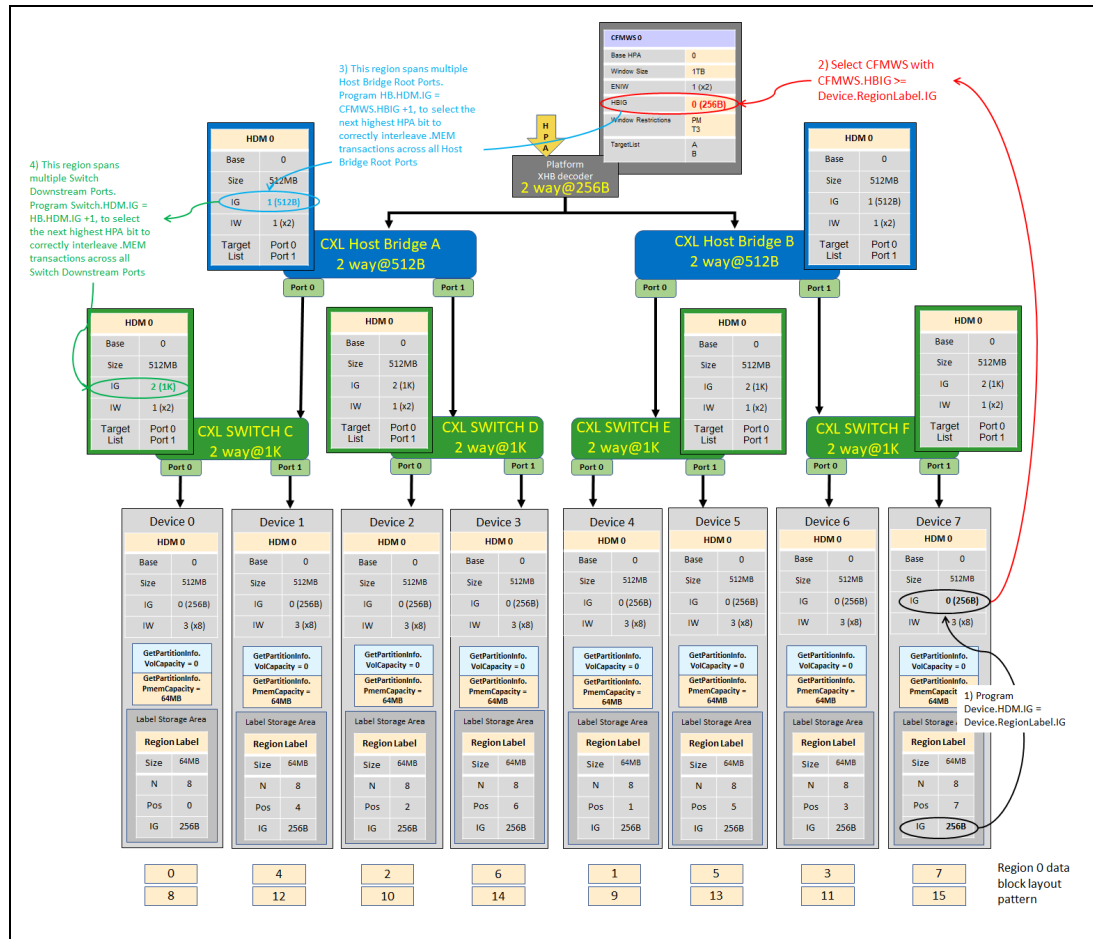
## 2.7.4 Example – Region interleaved across 2 HBs and 4 switches

This example demonstrates contents of the region label retrieved from each device's LSA, the System Firmware CFMWS, and how the System Firmware, UEFI and OS drivers would program the HDM decoders in each memory device, and each HB to implement the desired interleave.

In this example, all 8 devices are interleaved together into a single region that spans multiple CXL Host Bridges and multiple Switches, and each device contributes 64 MB of persistent capacity, for simplicity.

Note how the switch interleave granularity is programmed to 1K, the Host Bridge interleave granularity is programmed to 512B and the platform XHB interleave granularity is programmed to 256B so that each switch utilizes HPA[10] to select the Downstream Port, each Host Bridge utilizes HPA[9] to select the root port and the platform XHB interleave granularity utilizes HPA[8] to select the Host Bridge. This allows proper distribution of the HPA in a round robin fashion across all of the Host Bridges in the platform XHB CFMWS.InterleaveTargetList, each root port in the HostBridge.HDM.InterleaveTargetList, and each switch port in the Switch.HDM.InterleaveTargetlist[ ] the region is associated with. This leads to good performance since maximum distribution of memory requests across all HBs and switches is achieved, as shown in the resulting region data block layout pattern at the bottom of the figure.

**Figure 13 - Example - Region interleaved across 2 HBs and 4 switches**

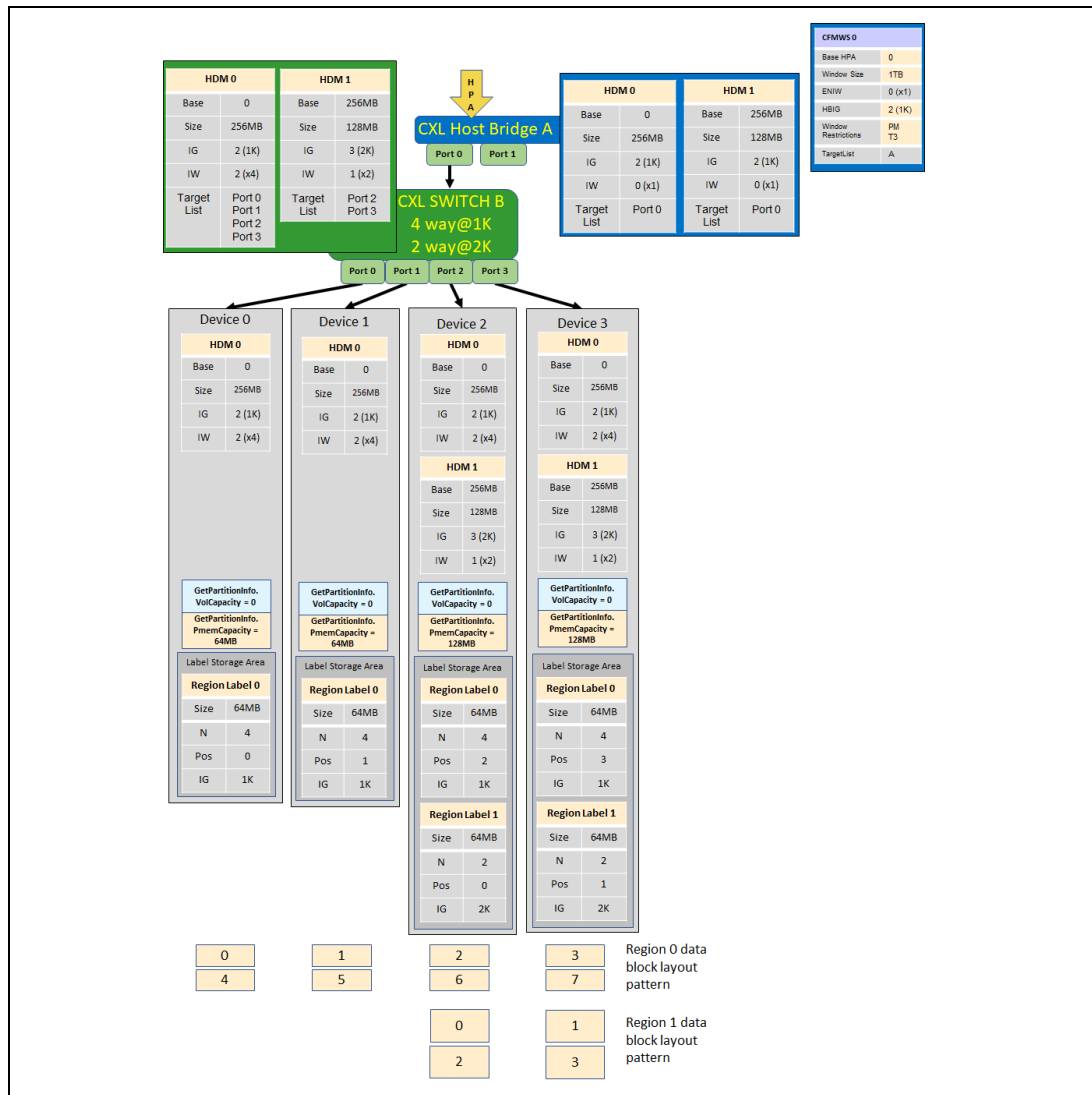


## 2.7.5 Example – 2 regions interleaved across 2 and 4 devices

This example demonstrates contents of the region label retrieved from each device's LSA, the System Firmware CFMWS, and how the System Firmware, UEFI and OS drivers would program the HDM decoders in each memory device, each switch, and each HBs to implement the desired interleave.

In this example, 4 devices are interleaved together into region 0, 2 of the same devices are also interleaved together into region 1, and each device contributes 64 MB bytes of persistent capacity, for simplicity. Note that a second set of region labels are utilized to describe the second interleave and that HDM decoder 1 on the device, switches, and the CXL Host Bridge is utilized to program the second HPA range.

**Figure 14 - Example – 2 regions interleaved across 2 and 4 devices**



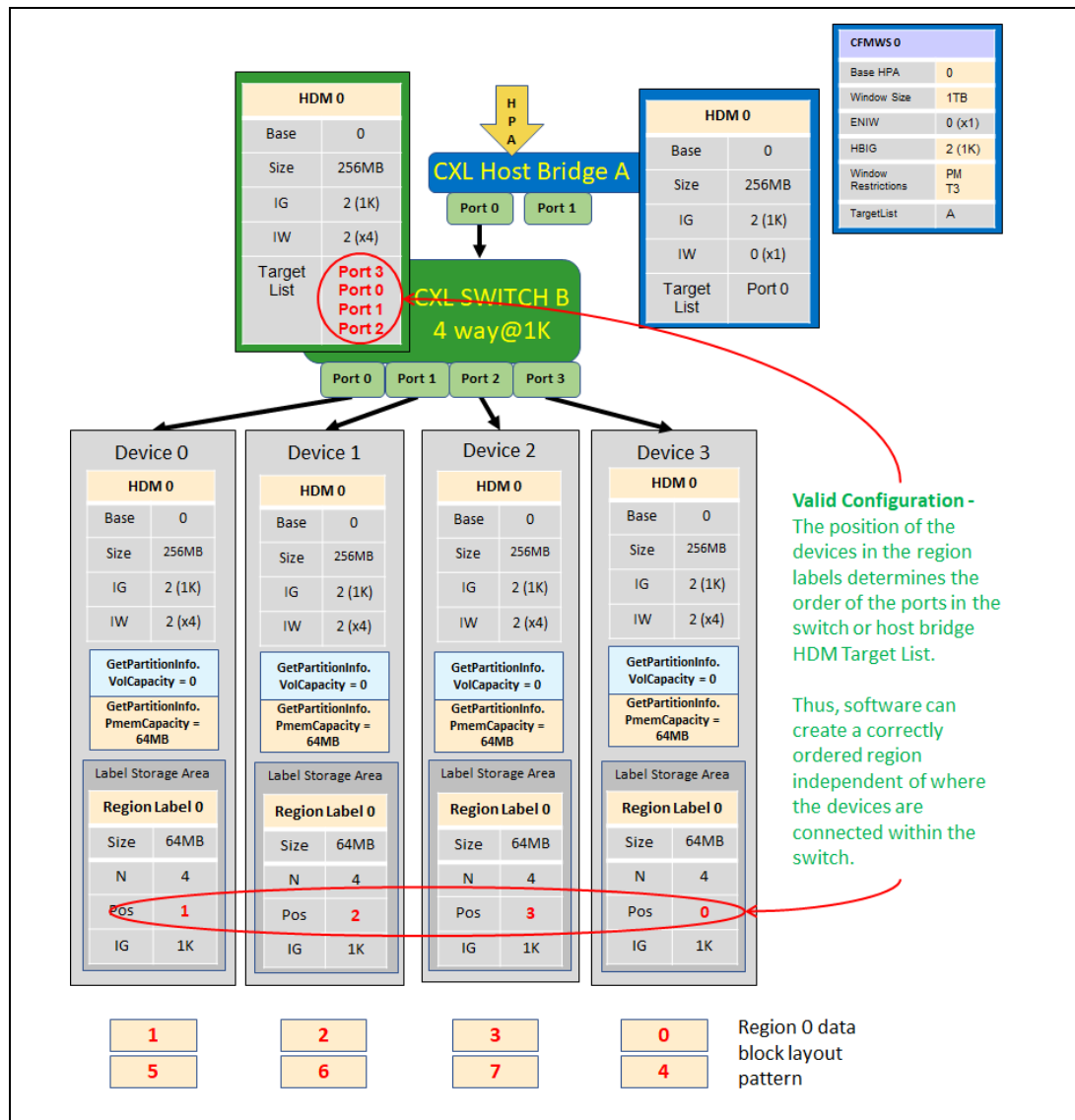
## 2.7.6 Example – Out of order devices within a switch or host

This example demonstrates contents of the region label retrieved from each device's LSA, the System Firmware CFMWS, and how the System Firmware, UEFI and OS drivers would program the HDM decoders in each memory device, each switch, and each HBs to implement the desired interleave.

In this example, 4 devices are interleaved together into region 0 on the same CXL Host Bridge and each device contributes 64 MB bytes of persistent capacity, for simplicity. However, the devices are plugged into the CXL Root Ports of the CXL switch in a random order compared to the ordering specified by the region label Position field found on each device. This could happen if the system the devices were originally plugged into failed and the devices were moved to an identical system for data recovery purposes, but the original ordering was shuffled in the process. Since the CXL switch and CXL Host Bridge HDMs contains a target list that specifies the order the root ports are interleaved, there is no need to reject this configuration. The UEFI and OS drivers program the CXL Switch or CXL Host Bridge HDMs target list to match the position of the device specified in the region label storage in the device's LSA without the need for a configuration change.

As long as all of the devices are plugged in to the same CXL Host Bridge, either directly, or through a CXL switch as shown in the example case here, the Target List in the CXL switch HDM decoder or CXL Host Bridge HDM decoder can be programmed to fix the ordering. See the next examples for additional constraints when verifying device ordering across CXL Host Bridges.

Figure 15 - Example - Out of order devices within a switch or host bridge



### 2.7.7 Example – Out of order devices across HBs (failure case)

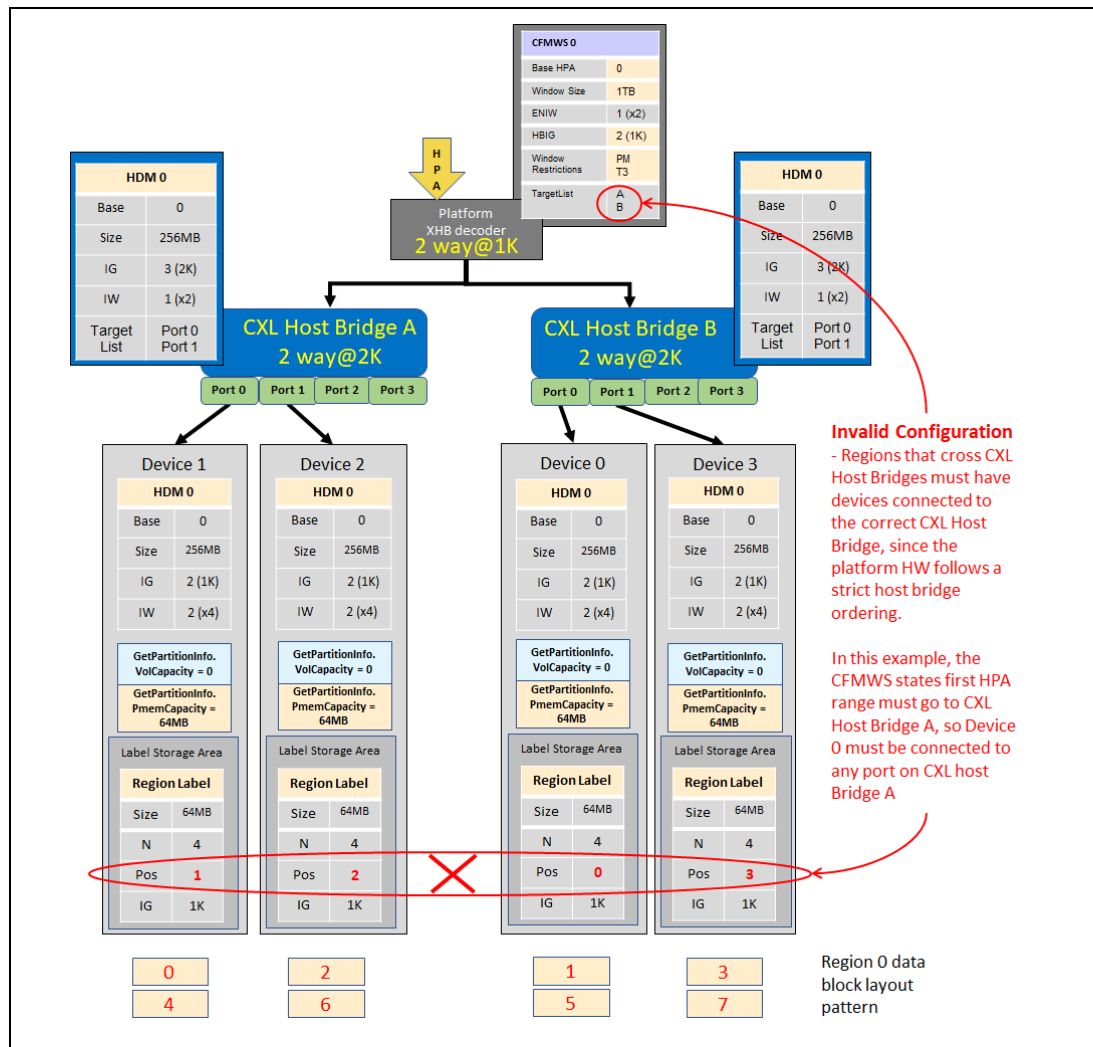
This example demonstrates contents of the region label retrieved from each device's LSA, the System Firmware CFMWS, and how the System Firmware, UEFI and OS drivers would program the HDM decoders in each memory device, each switch, and each HBs to implement the desired interleave.

In this example, 4 devices are interleaved together into region 0 with 2 devices on one host bridge and 2 devices on the next host bridge using an XHB region. Because platform HW may restrict ordering across host bridges and the OS cannot re-configure it, the ordering of the devices across host bridges now matters and adds an additional responsibility when checking for valid regions.

This example shows an illegal configuration because Device 0 and Device 1 must be on CXL Host Bridge A and Device 2 and Device 3 must be on CXL Host Bridge B. The first HPA range will be claimed by CXL Host Bridge A, so the device in POS 0 must be on CXL Host Bridge A. The second HPA range will be claimed by CXL Host Bridge B, so the device in POS 1 must be on CXL Host Bridge B. The third HPA range will be claimed by CXL Host Bridge A, so the device in POS 2 must be on A. The fourth HPA range will be claimed by CXL Host bridge B, so the device in POS 3 must be on B, and so on.

The suggested algorithm for software to check for proper device connection to each host bridge is outlined in the [Verify XHB configuration sequence](#).

**Figure 16 - Example - Out of order devices across HBs (failure case)**





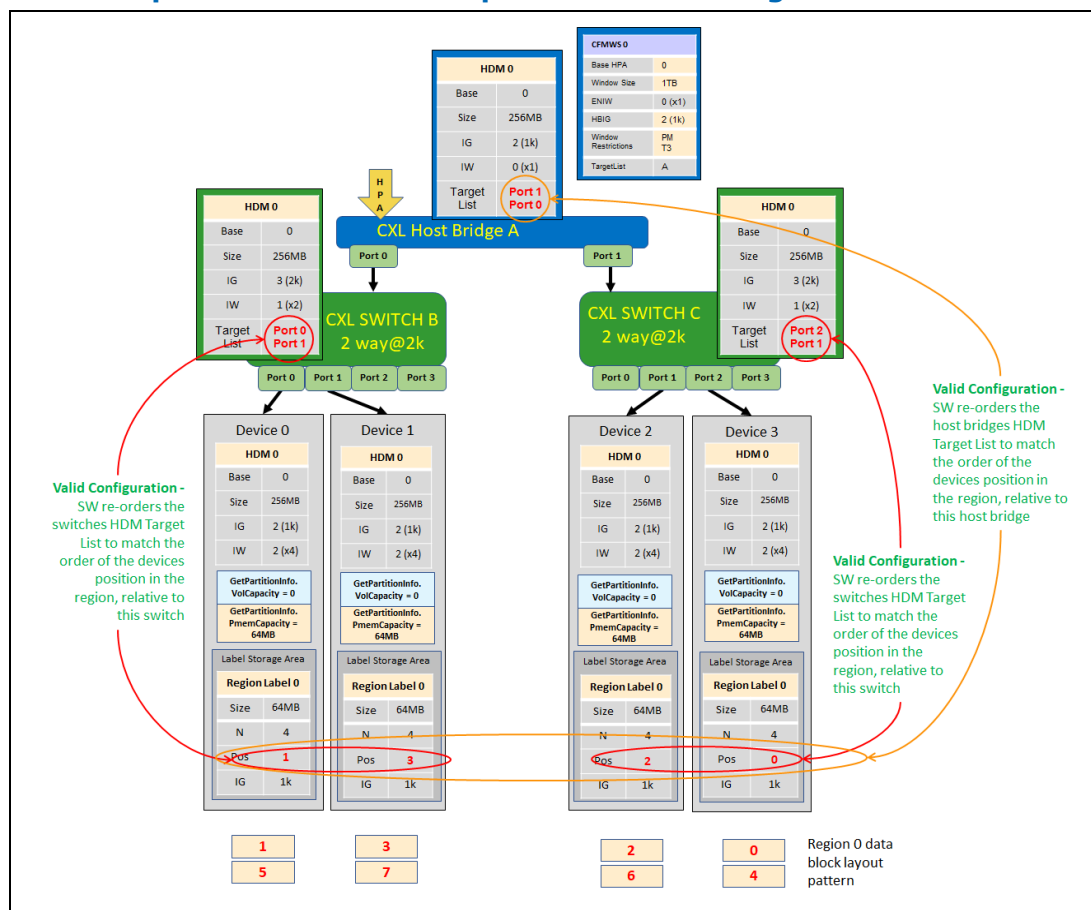
## 2.7.8 Example – Verifying device position on each HB root port

This example demonstrates contents of the region label retrieved from each device's LSA, the System Firmware CFMWS, and how the System Firmware, UEFI and OS drivers would program the HDM decoders in each memory device, each switch, and each HBs to implement the desired interleave.

This example demonstrates the verification required to ensure that each device's position in the region lands on the correct root port of the host bridge. While the ordering of root ports in the HB can be changed by reshuffling the HB.HDM.InterleaveTargetList[], the devices must still be connected in such a way that a valid setting for the HB.HDM.InterleaveTargetList[] array can be computed. In this case the devices are connected in such a way that valid settings exist.

The suggested algorithm for software to check for proper device connection to each root port on the host bridge is outlined in the [Verify HB root port configuration sequence](#).

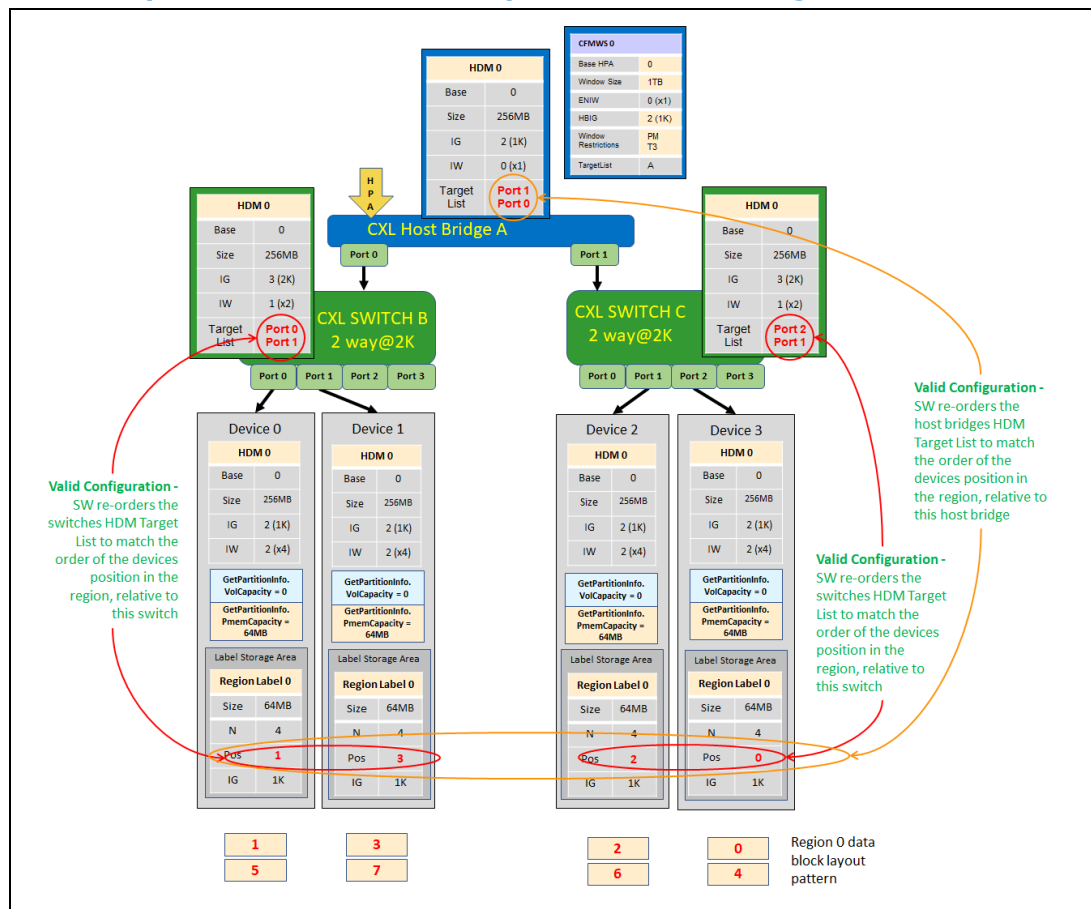
**Figure 17 – Example - Valid x2 HB root port device ordering**



This example demonstrates the verification required to ensure that each device position in the region lands on the correct root port of the host bridge. While the ordering of root ports in the HB can be changed by reshuffling the HB.HDM.InterleaveTargetList[], the devices must still be connected in such a way that a valid setting for the HB.HDM.InterleaveTargetList[] array can be computed. In this case the devices are not connected in such a way that valid settings exist.

The suggested algorithm for software to check for proper device connection to each root port on the host bridge is outlined in the Basic high-level sequences [Verify HB root port configuration sequence](#).

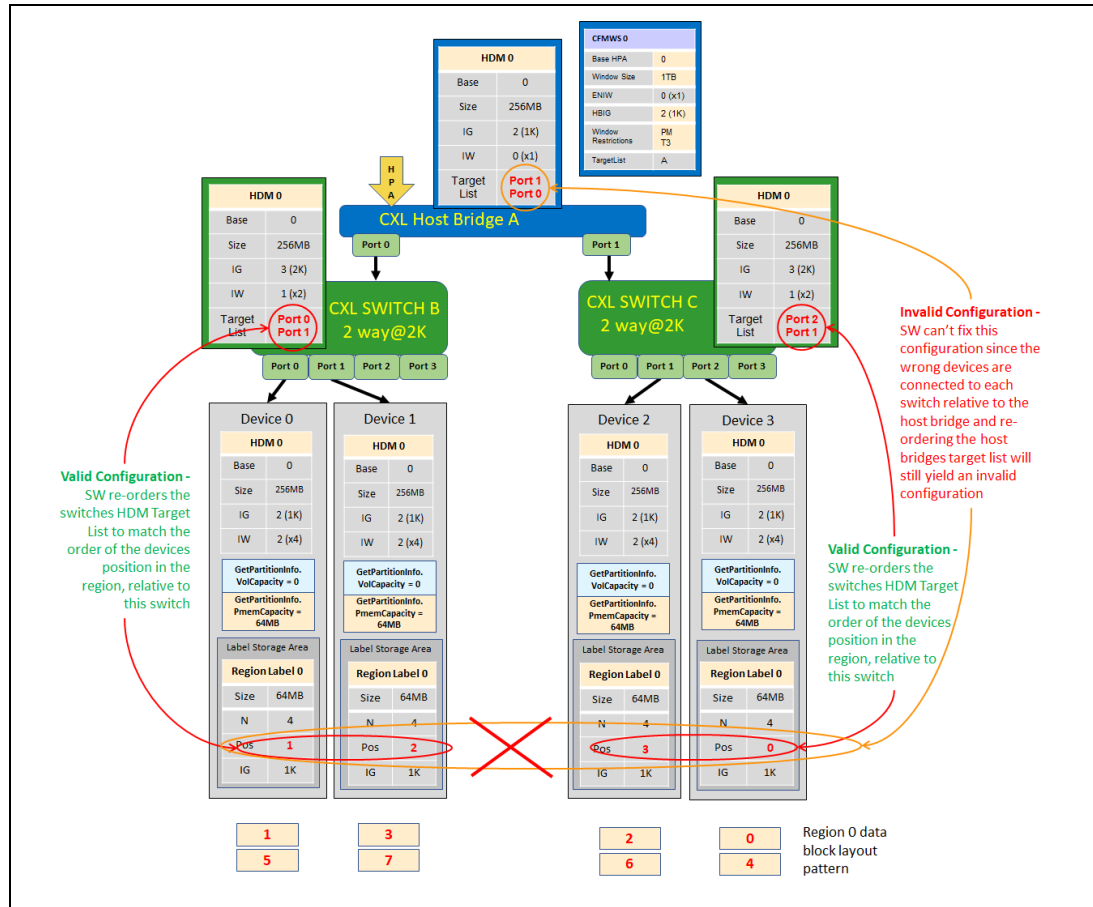
**Figure 18 - Example - Invalid x2 HB root port device ordering**



This example demonstrates the verification required to ensure that each device position in the region lands on the correct root port of the host bridge. While the ordering of root ports in the HB can be changed by reshuffling the HB.HDM.InterleaveTargetList[], the devices must still be connected in such a way that a valid setting for the HB.HDM.InterleaveTargetList[] array can be computed. In this case the devices are not connected in such a way that valid settings exist because the devices are not balanced across all of the root ports that the region spans.

The suggested algorithm for software to check for proper device connection to each root port on the host bridge is outlined in the [Verify HB root port configuration sequence](#).

**Figure 19 - Example - Unbalanced region spanning x2 HB root ports**



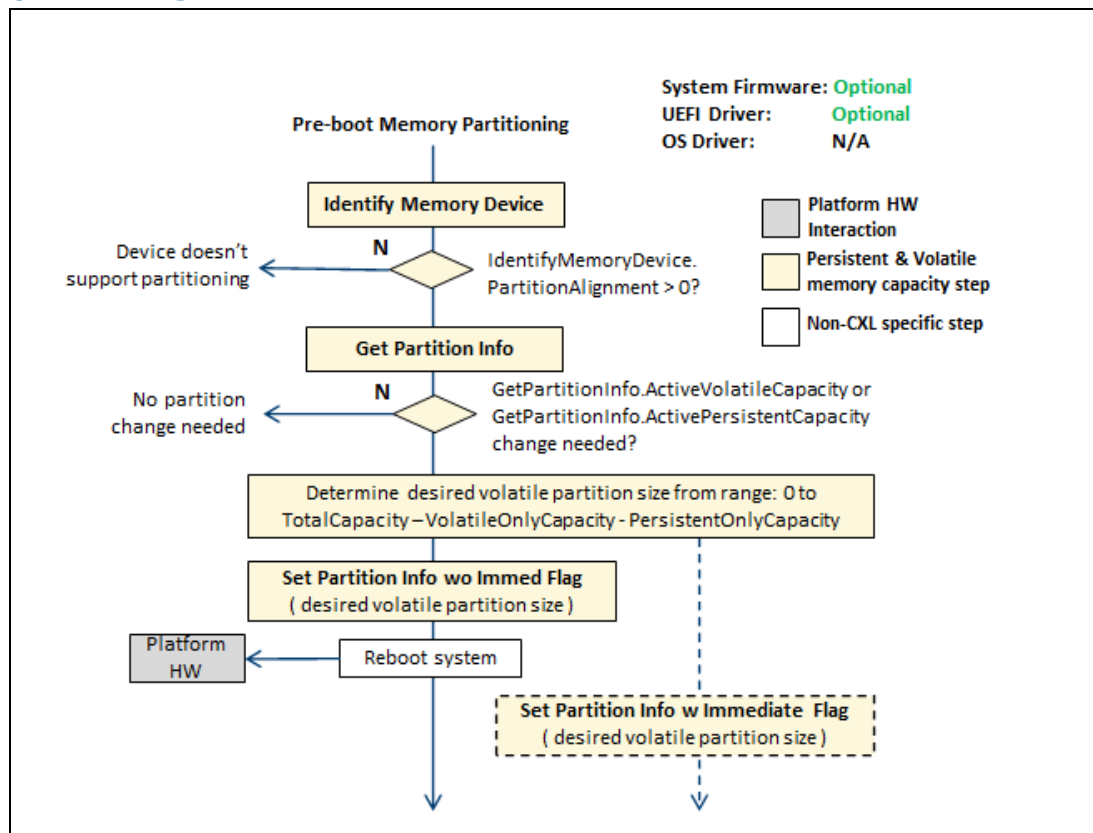
## 2.8 Partitioning and Configuration Sequences

The following sections outline the basic responsibilities and sequences for partitioning the amount of volatile and persistent capacity on the device, configuring the persistent memory interleaving, and enumerating partitions and regions.

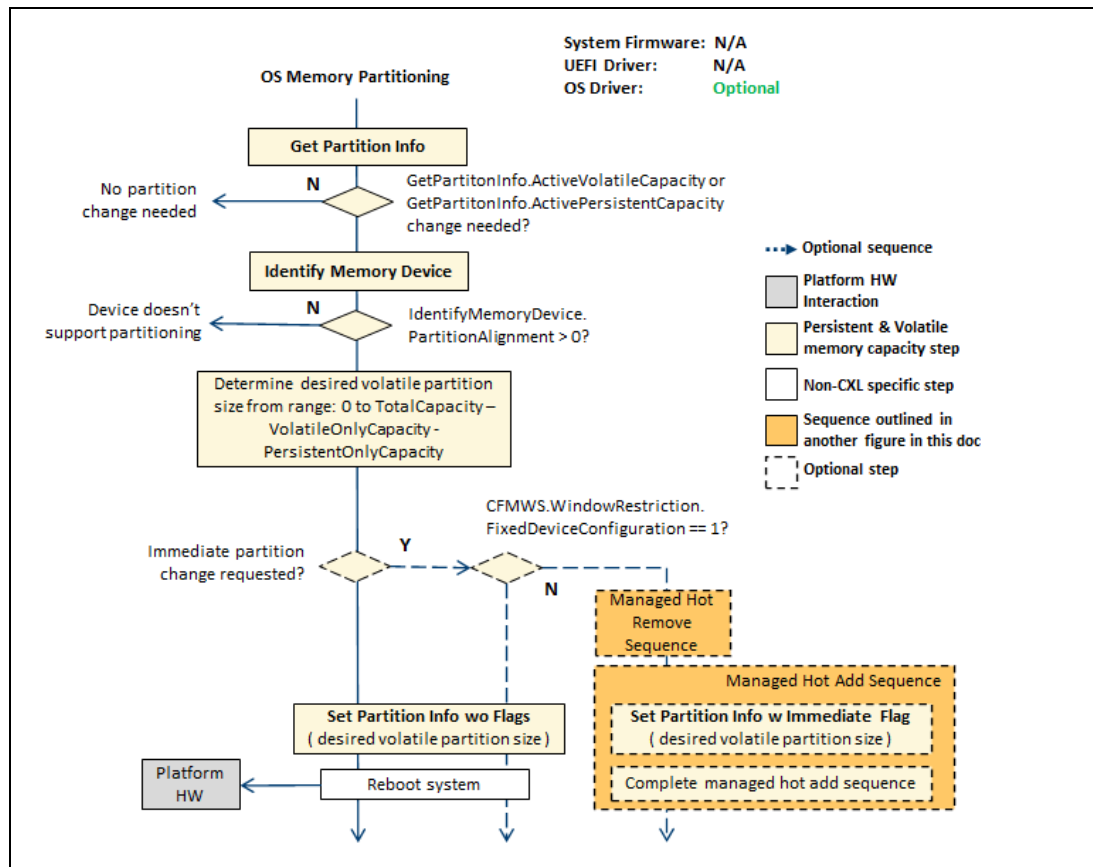
### 2.8.1 Volatile and Persistent Memory Partitioning

The following sequences outline the basic steps in partitioning the amount of volatile and persistent capacity the device will utilize. This sequence can be performed by several entities including System Firmware, UEFI driver, and OS driver. These sequences are only executed when the device partitioning needs to be changed. The OS can optionally utilize the Managed Hot Remove and Hot Add sequences to re-partition memory without rebooting.

**Figure 20 - High-level sequence: System Firmware and UEFI driver memory partitioning**



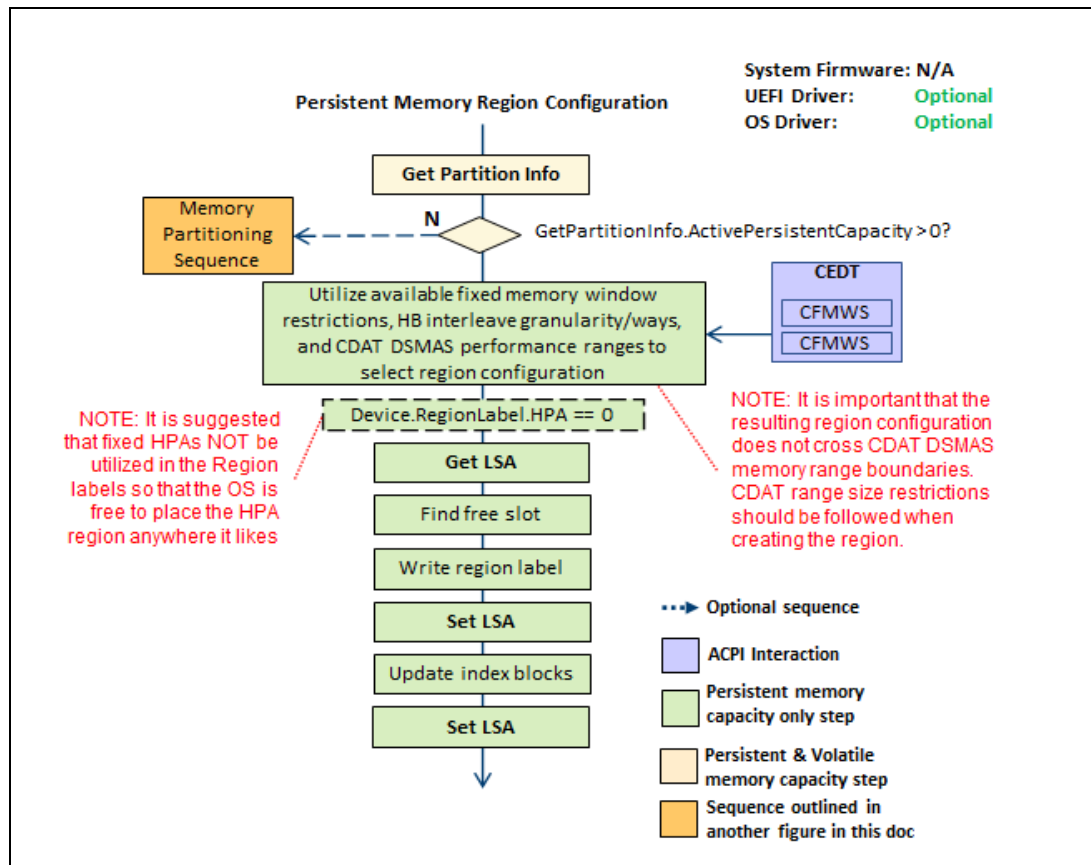
**Figure 21 - High-level sequence: OS memory partitioning**



## 2.8.2 Persistent memory region configuration

The following sequence outlines the basic steps in configuring persistent memory regions to interleave memory across multiple devices. This sequence can be performed by several entities including UEFI driver, and OS driver. This sequence is only executed when the device's persistent memory region configuration needs to be changed and the data does not have to be preserved (example: data is backed up somewhere).

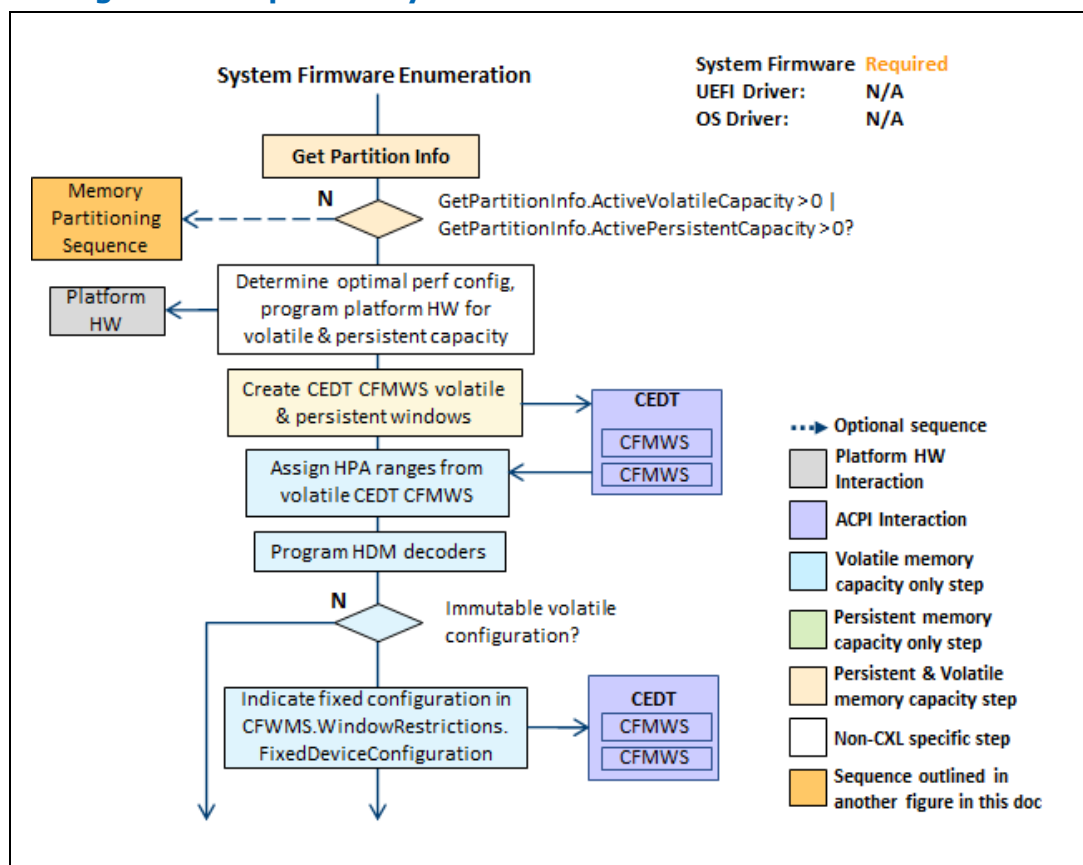
**Figure 22 - High-level sequence: Persistent memory region configuration**



## 2.8.3 System Firmware enumeration

The following sequence outlines the basic steps the System Firmware performs when enumerating memory devices with volatile and persistent capacity. This sequence is performed by the System Firmware on every system boot.

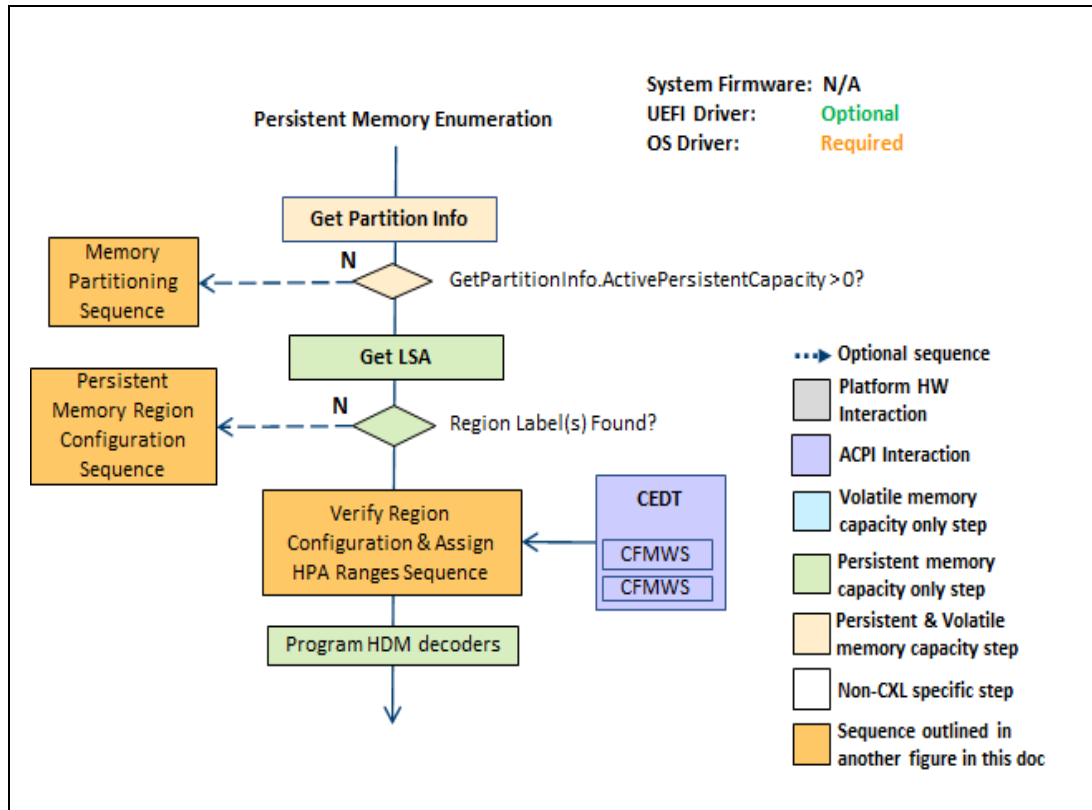
**Figure 23 - High-level sequence: System Firmware enumeration**



## 2.8.4 OS and UEFI driver enumeration

The following sequence outlines the basic steps the OS and UEFI drivers performs when enumerating memory devices with volatile and persistent capacity. This sequence is performed by the UEFI Driver and the OS on every system boot.

**Figure 24 - High-level sequence: UEFI and OS enumeration**

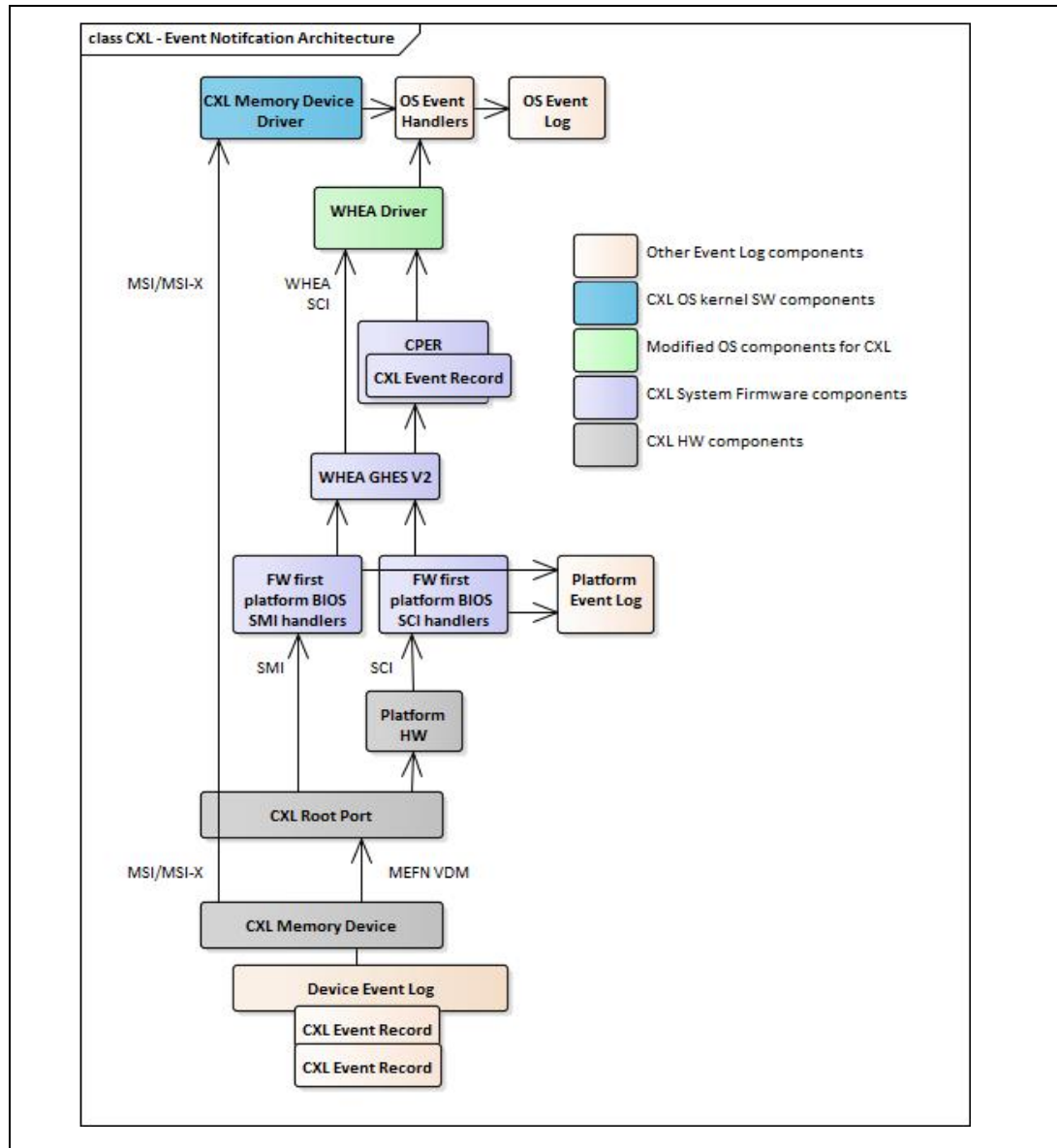




## 2.9 Asynchronous event handling

The following figure outlines the basic architecture of asynchronous event handling and the associated components. There are two memory error notification paths with CXL, the OS first PCIe/CXL MSI/MSIx path that notifies the OS CXL memory device driver directly in the form of the interrupt, or the FW First PCIe/CXL MEFN VDM where the platform HW turns the VDM MEFN message in to a platform firmware interrupt such as SCI or SMI on IA platforms.

**Figure 25 - CXL event notification architecture**



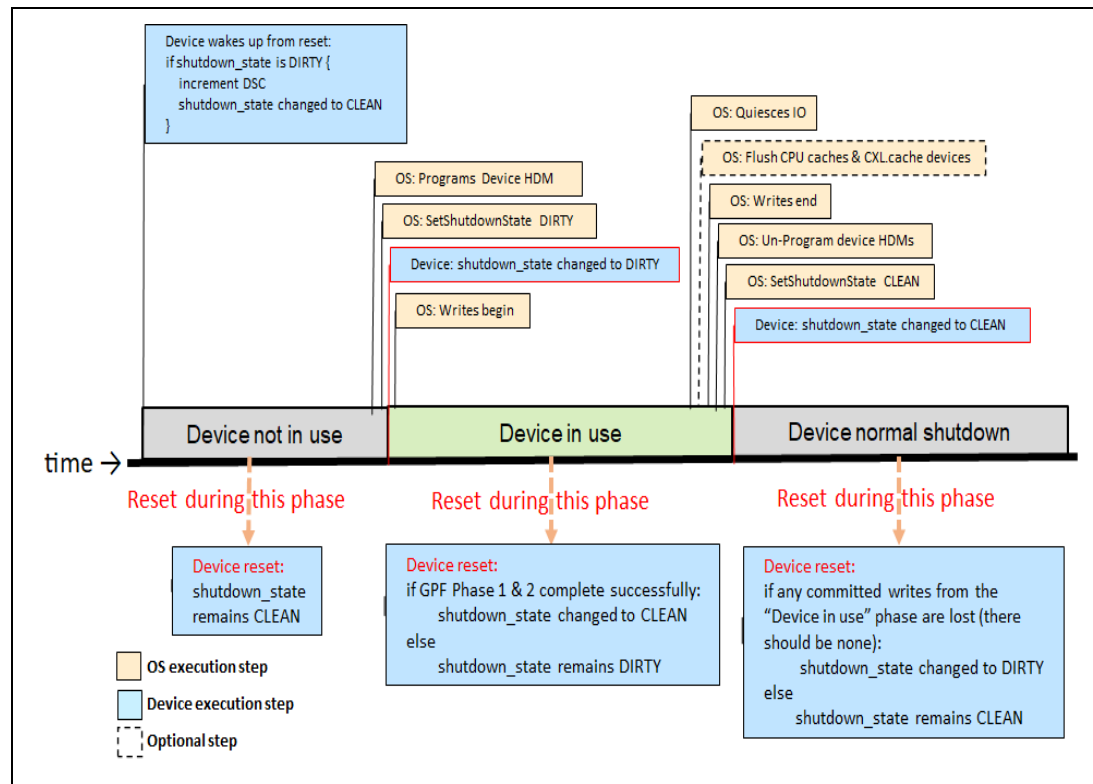
## 2.10 Dirty shutdown count handling

There are some basic rules that influenced the resulting dirty shutdown count handling flow presented here. These rules include:

- If there are writes that the OS previously sent (before marking the device as CLEAN) and they are still sitting in some device buffer that is not persistent, then it is the device's responsibility to maintain the promise that those stores are persistent, and if it cannot, change the state to DIRTY.
- If the device's shutdown state is set to CLEAN and it drops writes that occur after being set to CLEAN, the state can remain CLEAN. It is the OS's responsibility to set the device shutdown state to DIRTY before allowing any writes.
- Assuming the device's HDM decoders are not locked: The OS should un-program the HDM decoders for the memory device after the last write has completed and before setting the SetShutdownState state=CLEAN.
- It is an OS error to send more writes after setting the device's shutdown state to CLEAN and un-programming the HDM decoders prevents writes after the device is clean.
- The device is free to implement the persistent domain using flush-on-power fail volatile buffers and if those flushes fail, the device must change the state to DIRTY, even if software decided it is done using the device and marked it as CLEAN.
- It is the OS's responsibility to ensure CPU caches are properly flushed if caching of persistent data is enabled before setting the device's shutdown state to CLEAN.
- It is the OS's responsibility to quiesce all further CXL.MEM activity to the device and optionally flush CPU caches, before setting the shutdown state to CLEAN.
- In case there are conditions where the OS fails to complete the flushing and does not set the device's shutdown state to CLEAN, the System Firmware should flush CPU caches, before setting the shutdown state to CLEAN.

The following figure outlines the basic System Firmware, OS and Device steps for handling, detecting, and reporting a dirty shutdown, the device phases, and timeline.

**Figure 26 - CXL dirty shutdown count handling logic**



## 2.11 SRAT and HMAT

The following examples outline the architectural assumptions for implementing the SRAT and HMAT with CXL volatile and persistent memory capacity.

The following proximity domains may appear in the System Firmware generated SRAT:

**Table 6 - SRAT and HMAT content**

Proximity Domain	Affinity Type	Affinity Structure	Memory Range
Each CPU socket	0 – Processor	Enabled: SET	Local memory (example: attached via DDR)
Each CXL host bridge	6 – Generic Port	Enabled: SET	N/A
Each volatile memory range reported by CXL type 2 and 3 region	1 – Memory	Enabled: SET HotPluggable: Platform specific NonVolatile: CLEAR	System Firmware is responsible for creating SRAT memory range entries for every portion of the CMFWS it has programmed volatile device HDM decoders from
Each CXL type 1 and 2 accelerator device initiator	5 – Generic Initiator	Enabled: SET	System Firmware is responsible for creating SRAT memory range entries for Type 2 accelerator memory ranges

The following general rules apply to the System Firmware generated HMAT:

- HMAT returns best case latency (unloaded) L, and best-case bandwidth B, between pairs of proximity domains described in the SRAT.
- Since the SRAT does not contain proximity domains for persistent memory capacity, the HMAT will not contain performance information related to persistent memory capacity.
- If a path P is made up of subpaths P1, P2, ..., Pn
  - $L(P) = L(P1) + L(P2) + \dots + L(Pn)$  where L is the latency
  - $B(P) = \text{MIN} [B(P1), B(P2), \dots, B(Pn)]$  where B is the Bandwidth
- Subpaths can be one of two types
  - External Link
    - A. L and B can be computed based on knowledge of the link width, frequency and optionally retimer count. This is described in further detail below.
  - Internal Link
    - A. Internal to a CXL device – Device CDAT returns L and B of each subpath.
    - B. Internal to a CXL switch - Switch CDAT returns L and B of each subpath.
    - C. Internal to the CPU – System Firmware gets L and B from CPU datasheet. OS infers this from the HMAT.

The following general rules apply to the OS generated NUMA paths:

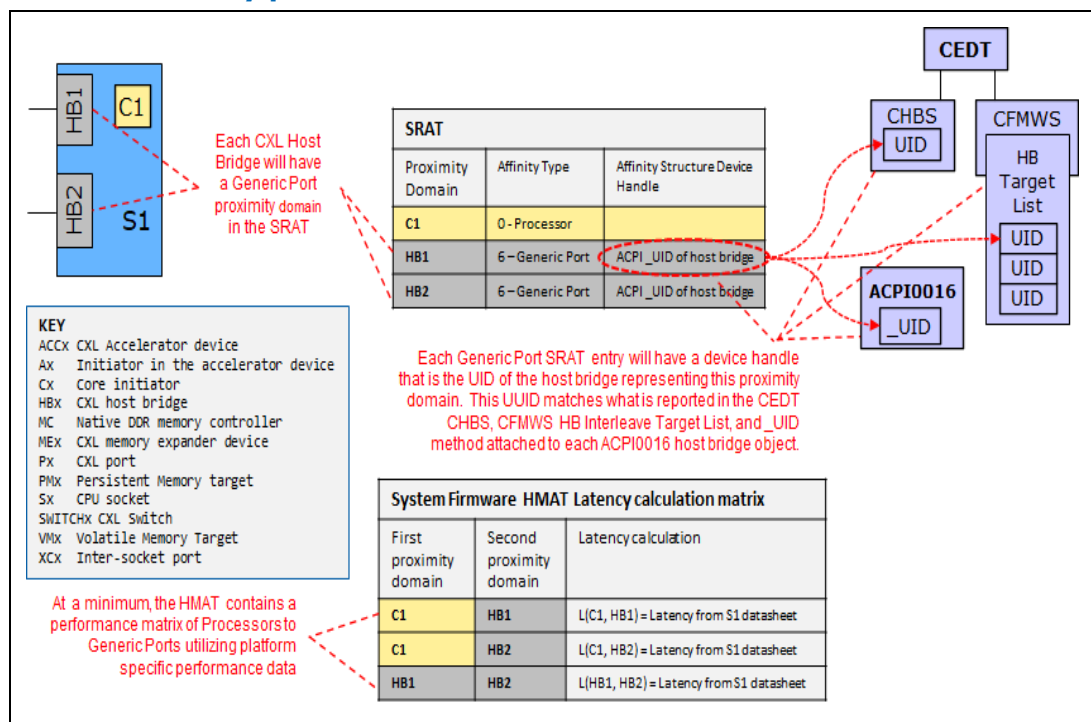
- SRAT does not contain persistent memory proximity domains.
- Therefore, HMAT does not contain persistent memory performance information.
- OS is responsible for calculating the NUMA path performance for all persistent memory enumerated at OS boot.
- OS is responsible for calculating the NUMA path performance for all volatile and persistent memory added at OS runtime through the Hot Add sequence.
- OS utilizes a combination of HMAT information, device and switch CDAT information, and the CXL link status to calculate the performance L and B for each NUMA path.

## 2.11.1 SRAT, HMAT and OS NUMA calculation examples

The following examples outline the previous responsibilities and demonstrate how the L and B are calculated by the System Firmware and the OS.

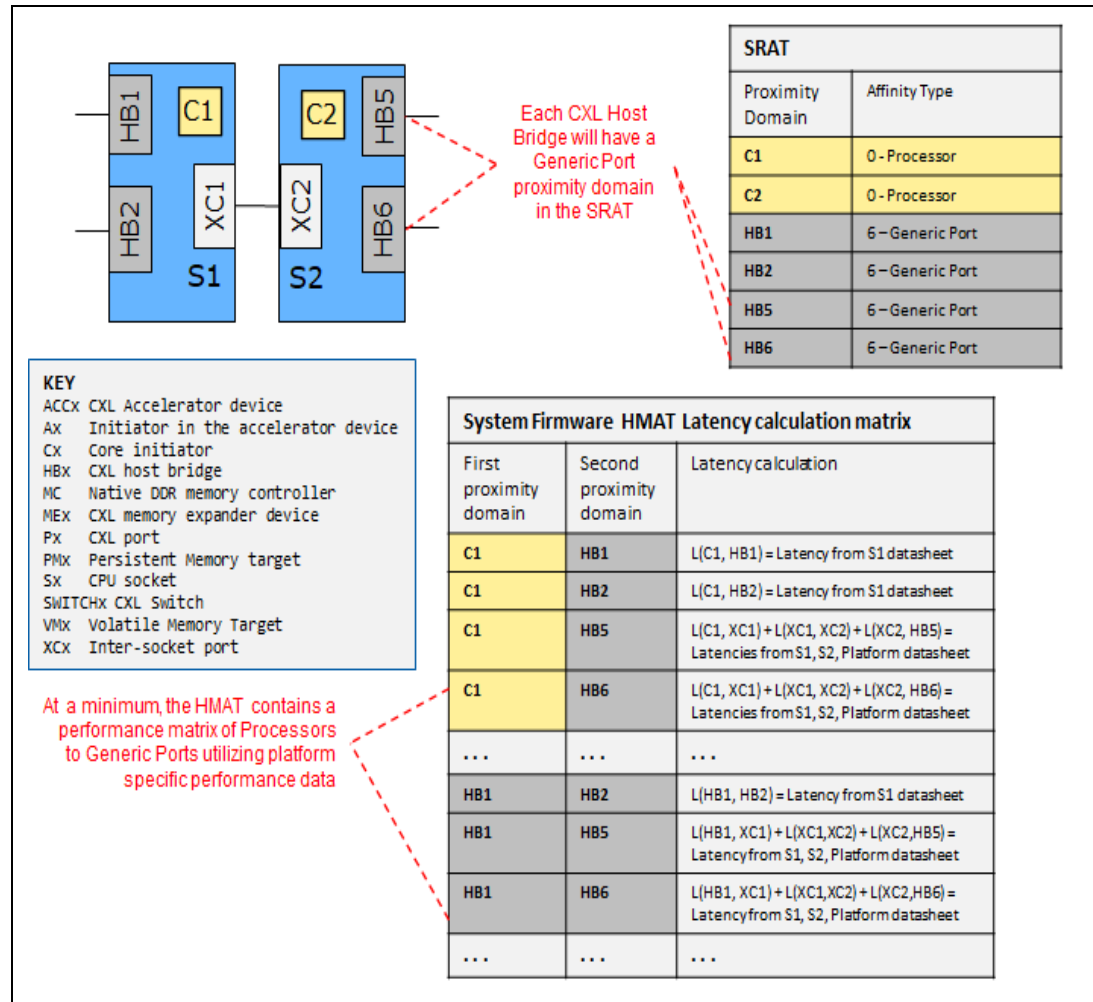
In this example, a single socket system has no memory attached at system boot. The SRAT is limited to proximity domains for CPU and root ports attached to the CPU. Note that each CXL host bridge will have a unique SRAT proximity domain. Also note how the device handle for the Generic Port SRAT entries identifies the UID of the host bridge representing the proximity domain and that matches the UID in the CEDT CHBS, CFWMS HB Interleave Target List, and the UID returned from the ACPI0016 \_UID method.

**Figure 27 - SRAT and HMAT Example: Latency calculations for 1 socket system with no memory present at boot**



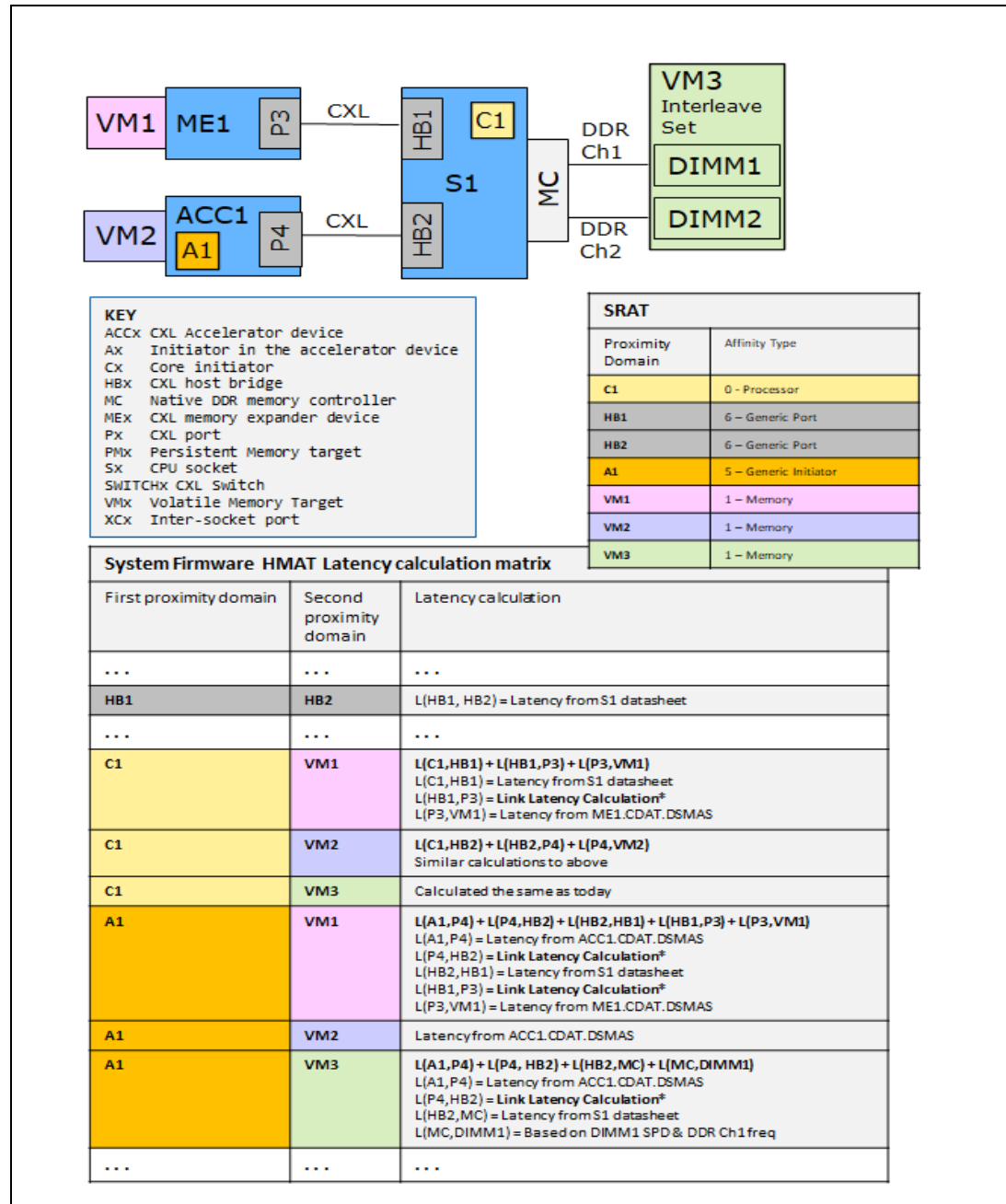
In this example, a two-socket system has no memory attached at system boot. The SRAT is limited to proximity domains for each CPU and the root ports attached to each CPU. Note that each CXL host bridge will have a unique SRAT proximity domain.

**Figure 28 - SRAT and HMAT Example: Latency calculations for 2 socket system with no memory present at boot**



In this example, a one socket system has native DDR attached memory, a CXL Type 2 accelerator device with volatile memory, and a CXL Type 3 memory expander device with volatile memory attached. This shows example Latency calculations the System Firmware performs to build the latency portion of the HMAT.

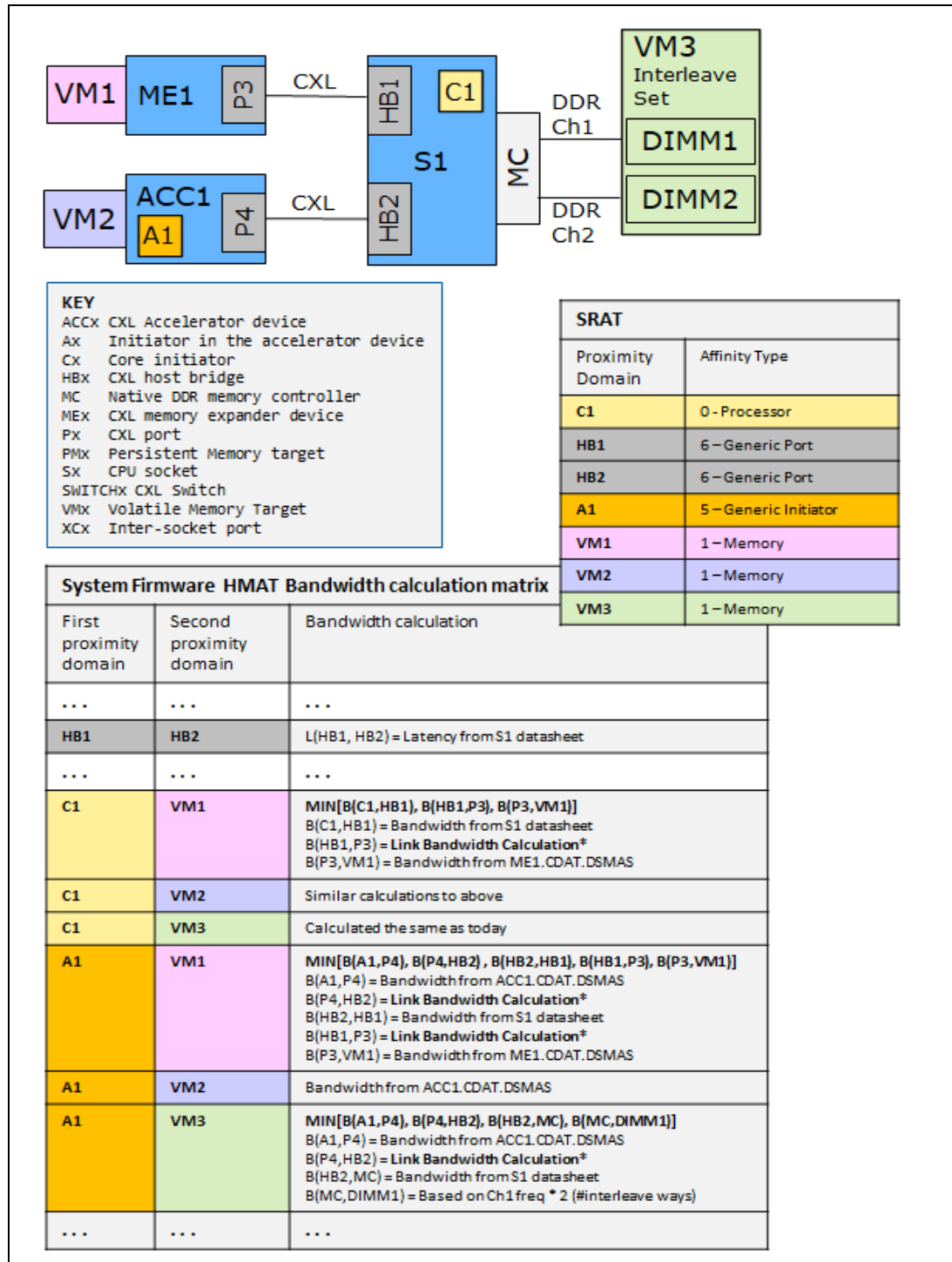
**Figure 29 - SRAT and HMAT Example: Latency calculations for 1 socket system with volatile memory attached at boot**



In this example, the same one socket system as the previous example has native DDR attached memory, a CXL Type 2 accelerator device with volatile memory, and a CXL Type 3 memory expander device with volatile memory attached. This shows example Bandwidth calculations the System Firmware performs to build the bandwidth portion of the HMAT.

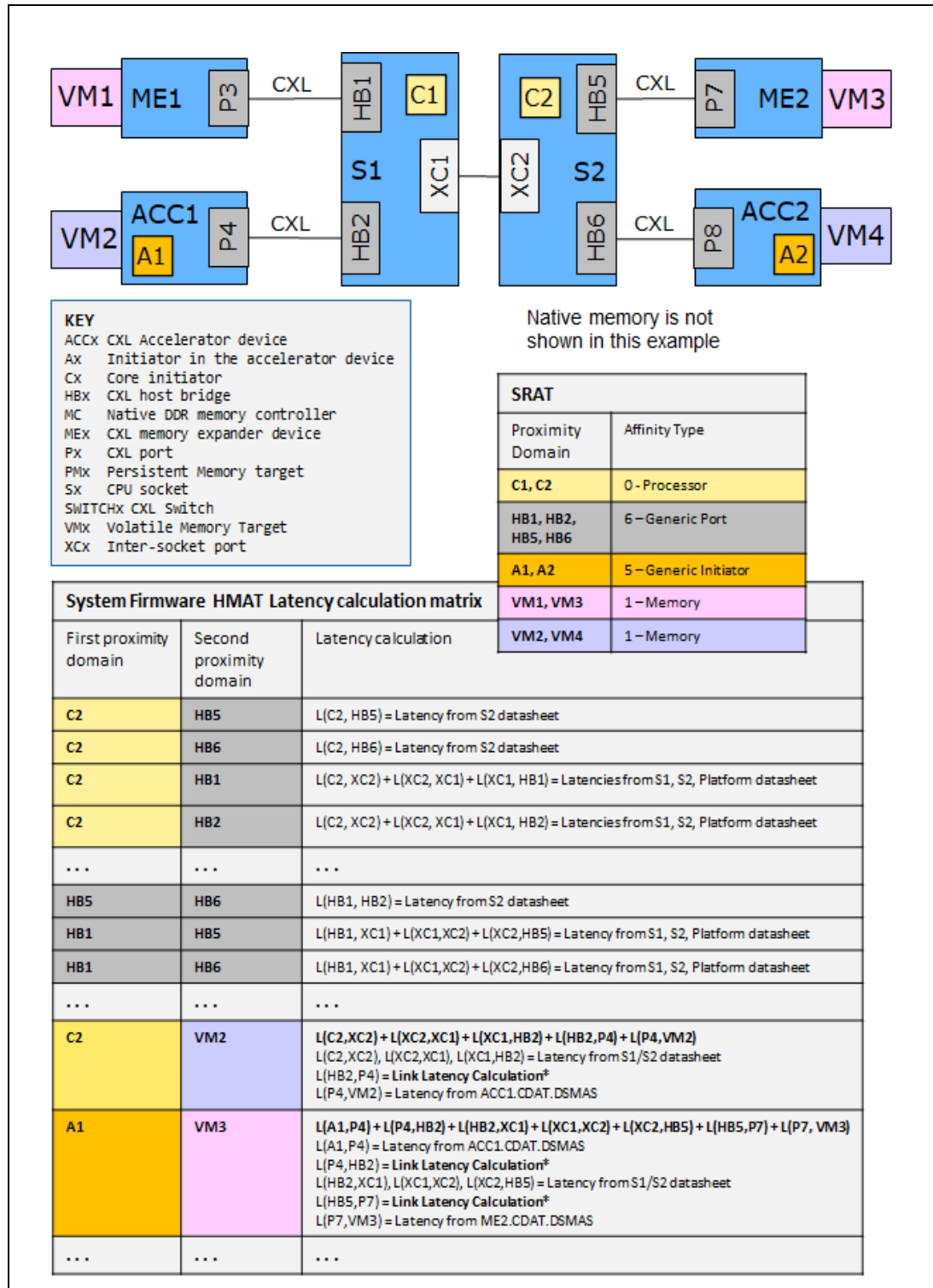


**Figure 30 - SRAT and HMAT Example: Bandwidth calculations for 1 socket system with volatile memory attached at boot**



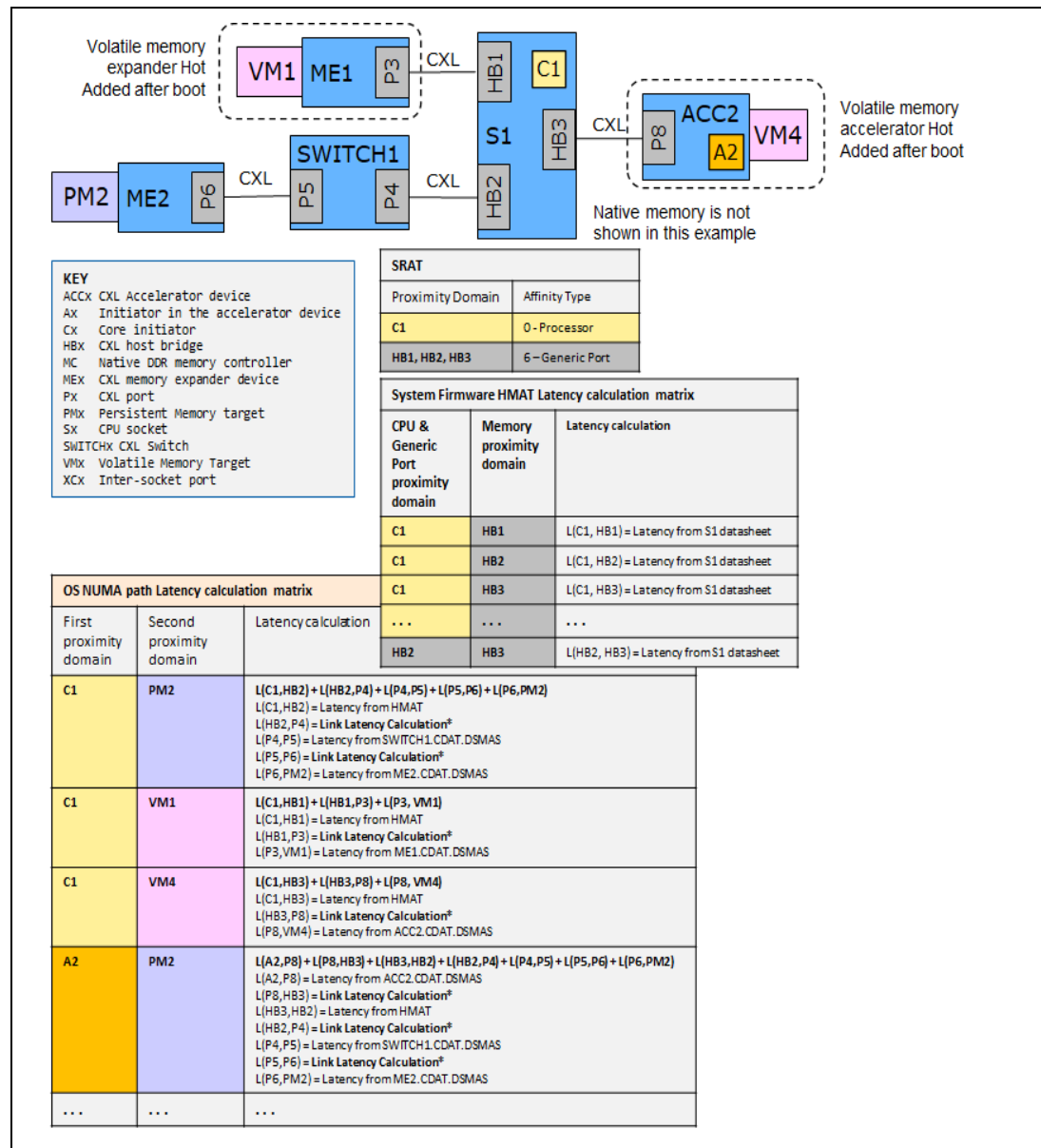
In this example, a two-socket system has two CXL Type 2 accelerator devices with volatile memory, and two CXL Type 3 memory expander devices with volatile memory attached. This shows example Latency calculations the System Firmware performs to build the latency portion of the HMAT.

**Figure 31 - SRAT and HMAT Example: Latency calculations for 2 socket system with volatile memory attached at boot**



In this example, a one socket system has a CXL Type 3 memory expander device with persistent memory attached via a switch, either at OS boot time, or added after OS boot via the Hot Add sequence. The system also has a CXL Type 3 memory expander device with volatile memory, added after OS boot via the Hot Add sequence. This shows example NUMA path Latency calculations the OS performs for each persistent memory device present at OS boot time, and volatile and persistent memory devices added later via the Hot Add sequence. Note that the OS relies on SRAT and HMAT information, the switch and device CDAT information, and PCIe link status to determine the overall latency and bandwidth for these devices.

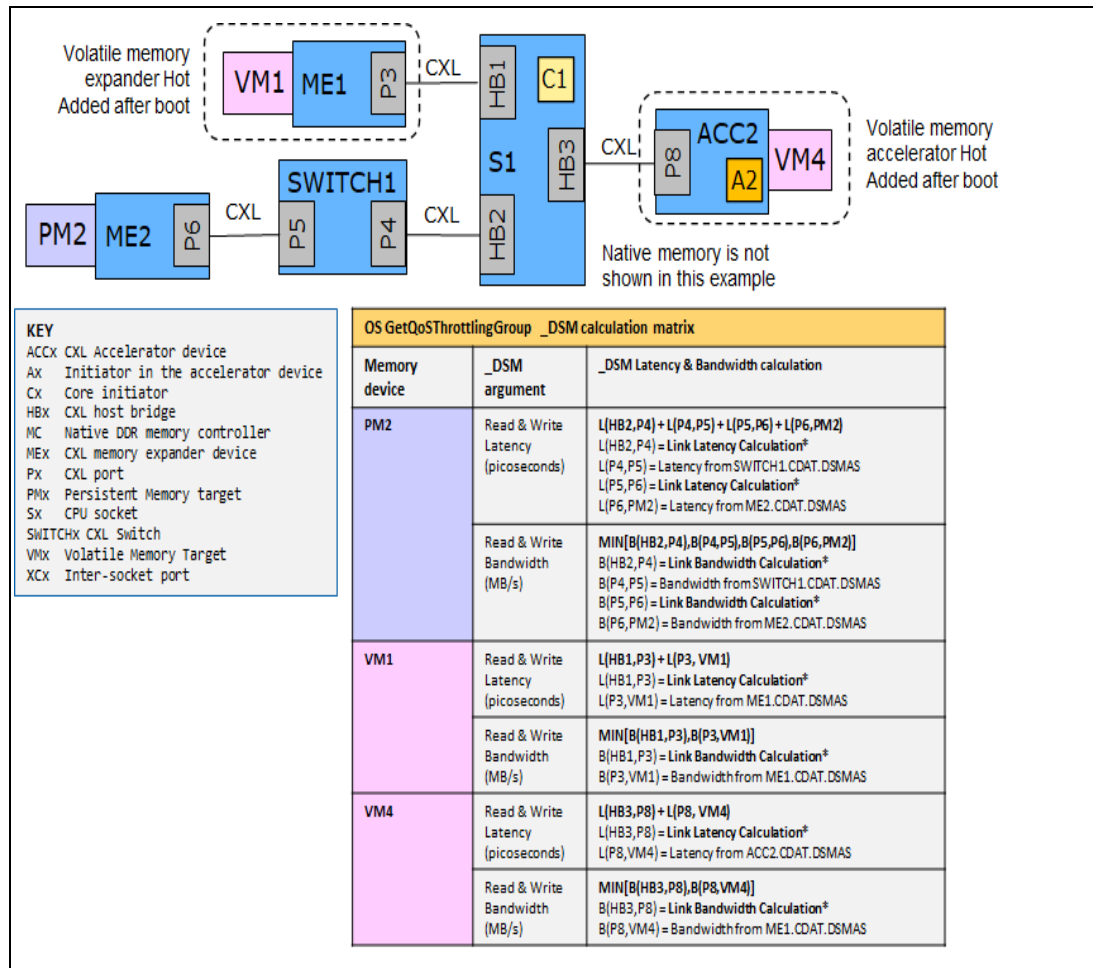
**Figure 32 - SRAT and HMAT Example: Latency calculations for 1 socket system with persistent memory and hot added memory**



## 2.11.2 GetQoSThrottlingGroup \_DSM calculation examples

To execute the GetQoSThrottlingGroup \_DSM, the OS needs to calculate the bandwidth and latency from the CXL host bridge to the device. The calculations required for the \_DSM are identical to the OS NUMA path performance calculations, except the CXL Host Bridge root port to the CPU performance component is not included. The following shows the previous PMEM and Hot Add example and the calculations required to call the GetQoSThrottlingGroup \_DSM.

**Figure 33 - GetQoSThrottlingGroup \_DSM example**



### 2.11.3 Link bandwidth calculation

Here is the basic CXL link bandwidth calculation the System Firmware and OS utilize in the previous examples. This calculation is performed at each CXL Host Bridge root port and, if present, at the downstream ports of the switch as outlined in the previous examples.

LinkOperatingFrequency (GT/s) = Use the current negotiated link speed, PCIeLinkStatus.Speed, to reference the device's supported link speeds bit mask, PCIeLinkCapabilities2.SupportedLinkSpeedsVector

DataRatePerLink (MB/s) = LinkOperatingFrequency (GT/s) / 8  
(bytes/Transfer)

LinkBandwidth (MB/s) = PCIeLinkStatus.NegotiatedLinkWidth x  
DataRatePerLink

### 2.11.4 Link latency calculation

Here is the basic CXL link latency calculation the System Firmware and OS utilize in the previous examples. This calculation is performed at each CXL Host Bridge root port and, if present, at the downstream ports of the switch as outlined in the previous examples. Unless the PCIe link speed or width are severely degraded, all these terms may be considered negligible, relative to the device latencies, by the OS implementation.

LinkPropagationLatency (ps) = This term is assumed to be negligible relative to the other latencies and 0 is used for the rest of the calculation

FlitLatency (ps) = FlitSize (bytes – From CXL spec) / LinkBandwidth  
(MB/s – From Link Bandwidth calculation)

RetimerLatency (ps) = This term is assumed to be negligible relative to the other latencies and 0 is used

LinkLatency (ps) = LinkPropagationLatency + FlitLatency +  
RetimerLatency

## 2.12 Operation ordering restrictions

The following section outlines specific ordering of operations requirements that the UEFI and OS drivers shall adhere to when implementing a CXL memory driver.

- System Firmware, UEFI and OS drivers shall wait for `MemoryDeviceStatusRegister.MailboxInterfaceReady` before executing mailbox commands.
- System Firmware, UEFI and OS drivers shall wait for `MemoryDeviceStatusRegister.MediaStatus==Ready` before executing `.MEM` transactions.
- Persistent capacity required ordering requirements to maintain data consistency:
  - If `GetSecurityState.SecurityState == Locked`, the device shall be unlocked before persistent memory requests utilizing the HDMs begin.
  - Anytime the memory device transitions from security locked to security unlocked, the OS shall invalidate all CPU caches and all Type 2 cache lines that map to persistent memory, before memory requests are started.
  - Shall set `SetShutdownState state=DIRTY` before the first write to pmem via `CXL.memory`.
  - If `FADT.PERSISTENT_CPU_CACHES == 10b`, OS shall flush all CPU caches and all Type 2 cache lines that map to persistent memory, before setting `SetShutdownState state=CLEAN`.
  - OS should un-program the memory device's HDM decoders after the last write to pmem via memory.
  - OS shall set `SetShutdownState state=CLEAN` after writes have completed and HDM decoders un-programmed.
- Persistent capacity optional RAS requirements for best practices:
  - `SetTimestamp` before `SetEventInterruptPolicy` to minimize the device generating logs without timestamps.

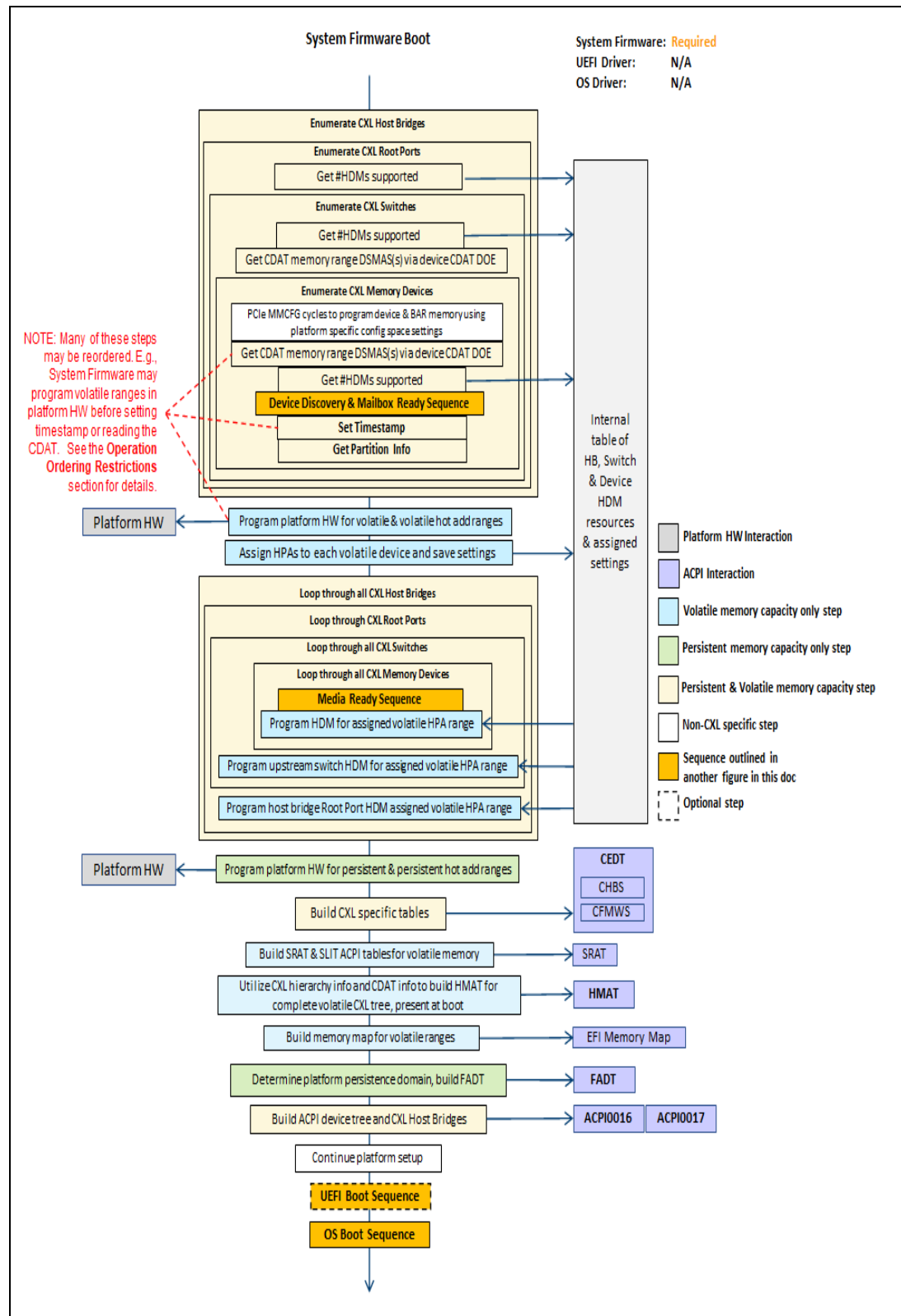
## 2.13 Basic high-level sequences

The following sections outline the basic high-level sequences with major steps outlined in the previous high-level responsibilities table. This only covers high-level steps that are specific to the CXL memory device driver sequences. The rest of the document provides details on these steps.

### 2.13.1 System Firmware boot sequence

The following figure outlines the basic high-level System Firmware boot sequence.

**Figure 34 - High-level sequence: System Firmware boot**

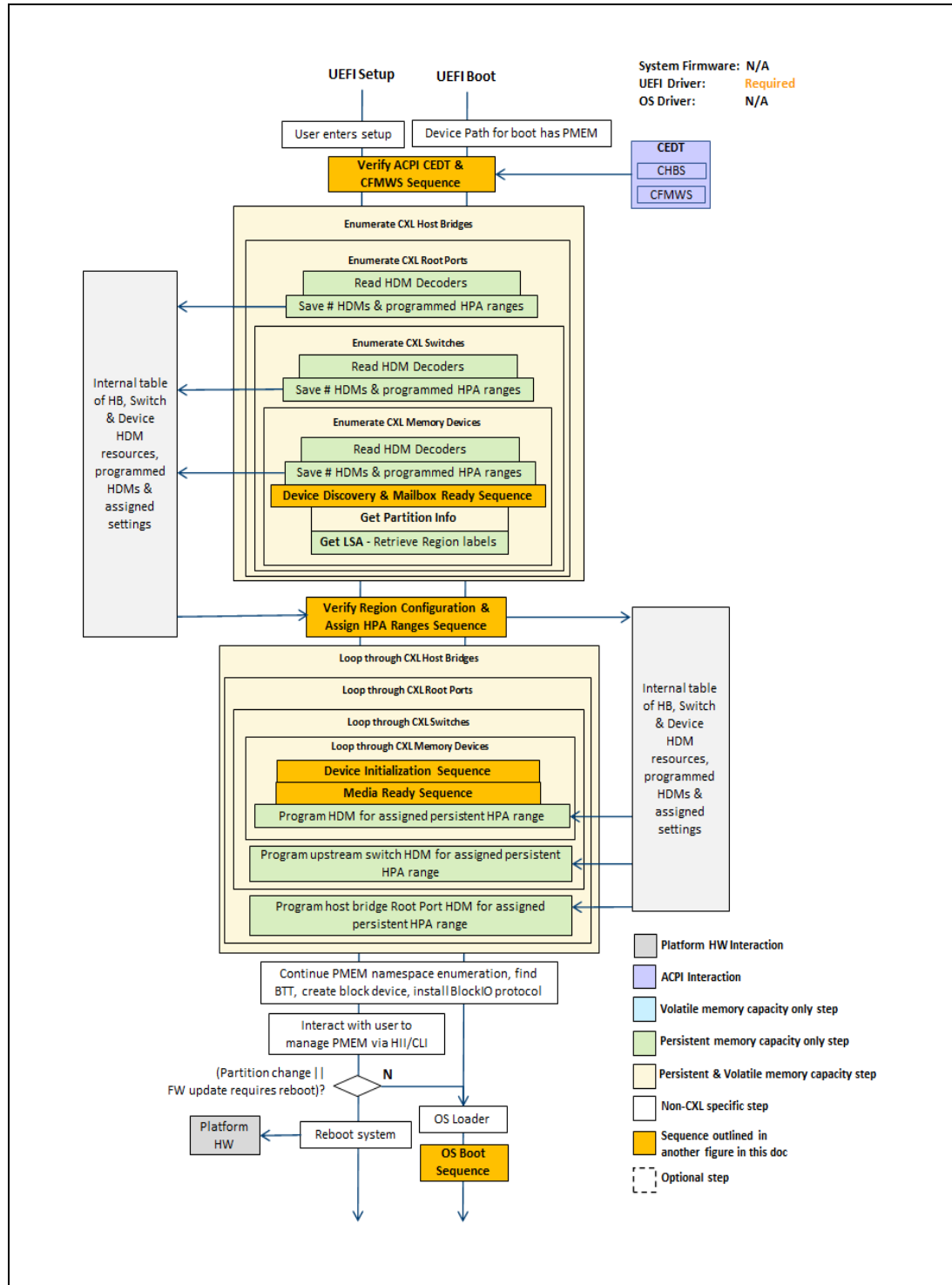




## 2.13.2 UEFI setup and boot sequence

The following figure outlines the basic high-level UEFI setup and boot sequence.

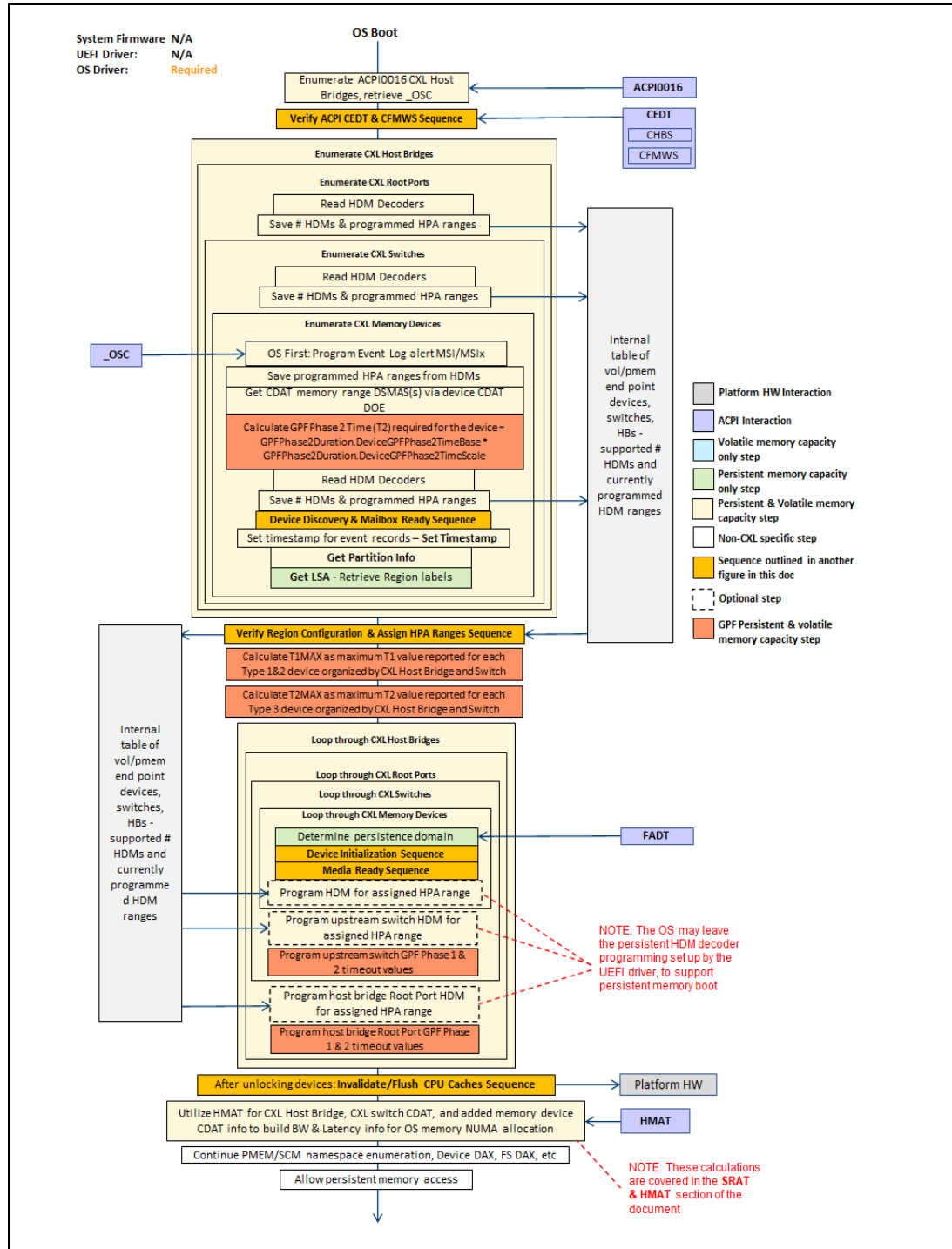
**Figure 35 - High-level sequence: UEFI setup and boot**



## 2.13.3 OS boot sequence

The following figure outlines the basic high-level OS boot sequence.

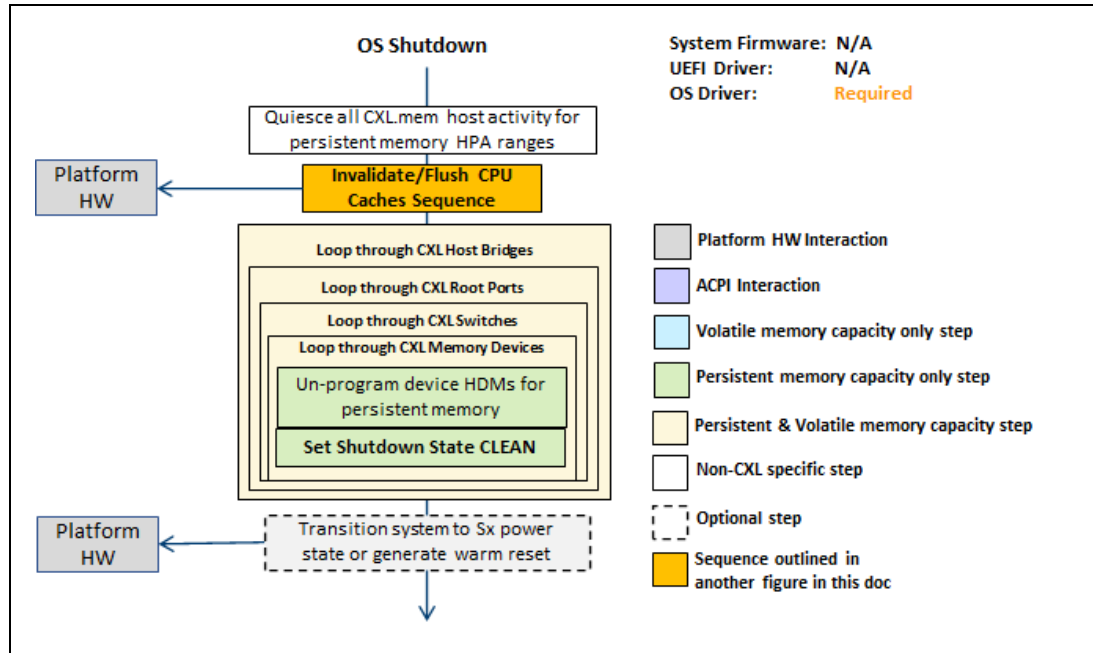
**Figure 36 - High-level sequence: OS boot**



## 2.13.4 OS shutdown sequence

The following figure outlines the basic high-level OS graceful shutdown sequence.

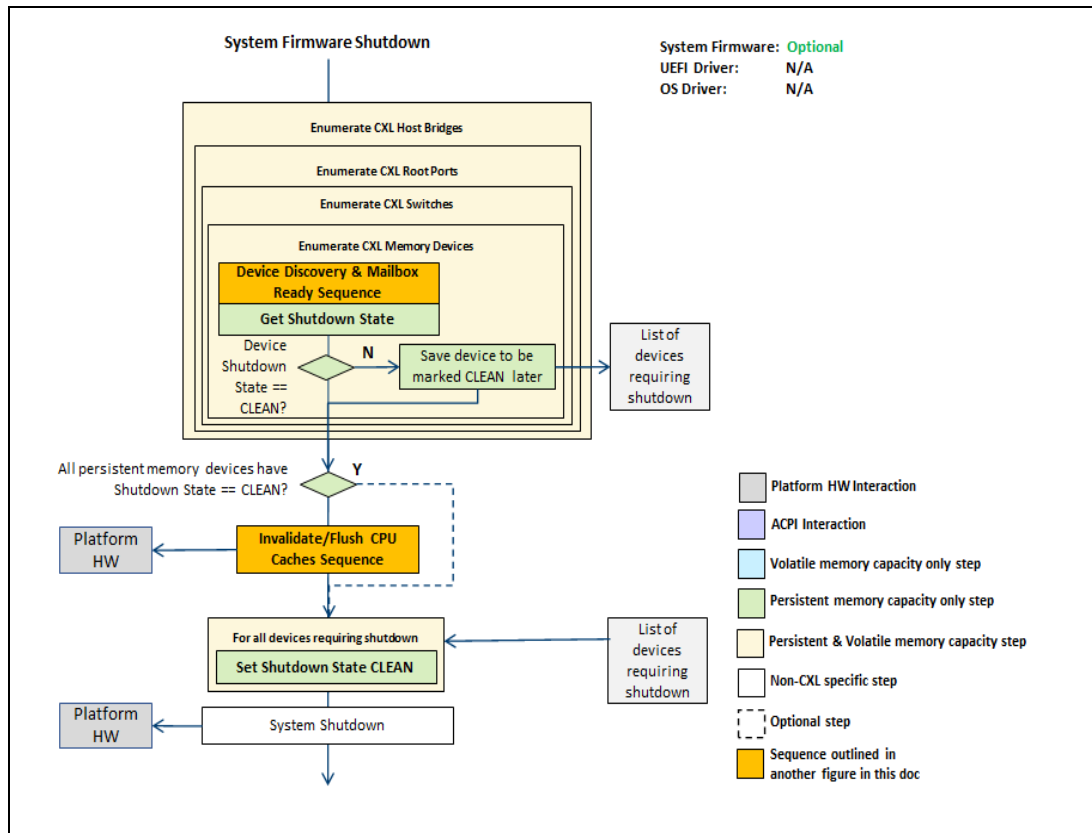
**Figure 37 - High-level sequence: OS shutdown**



## 2.13.5 System Firmware shutdown sequence

The following figure outlines the basic high-level System Firmware graceful shutdown sequence. This flow only applies to platform designs that utilize a System Firmware shutdown handler invoked from the OS on a graceful shutdown.

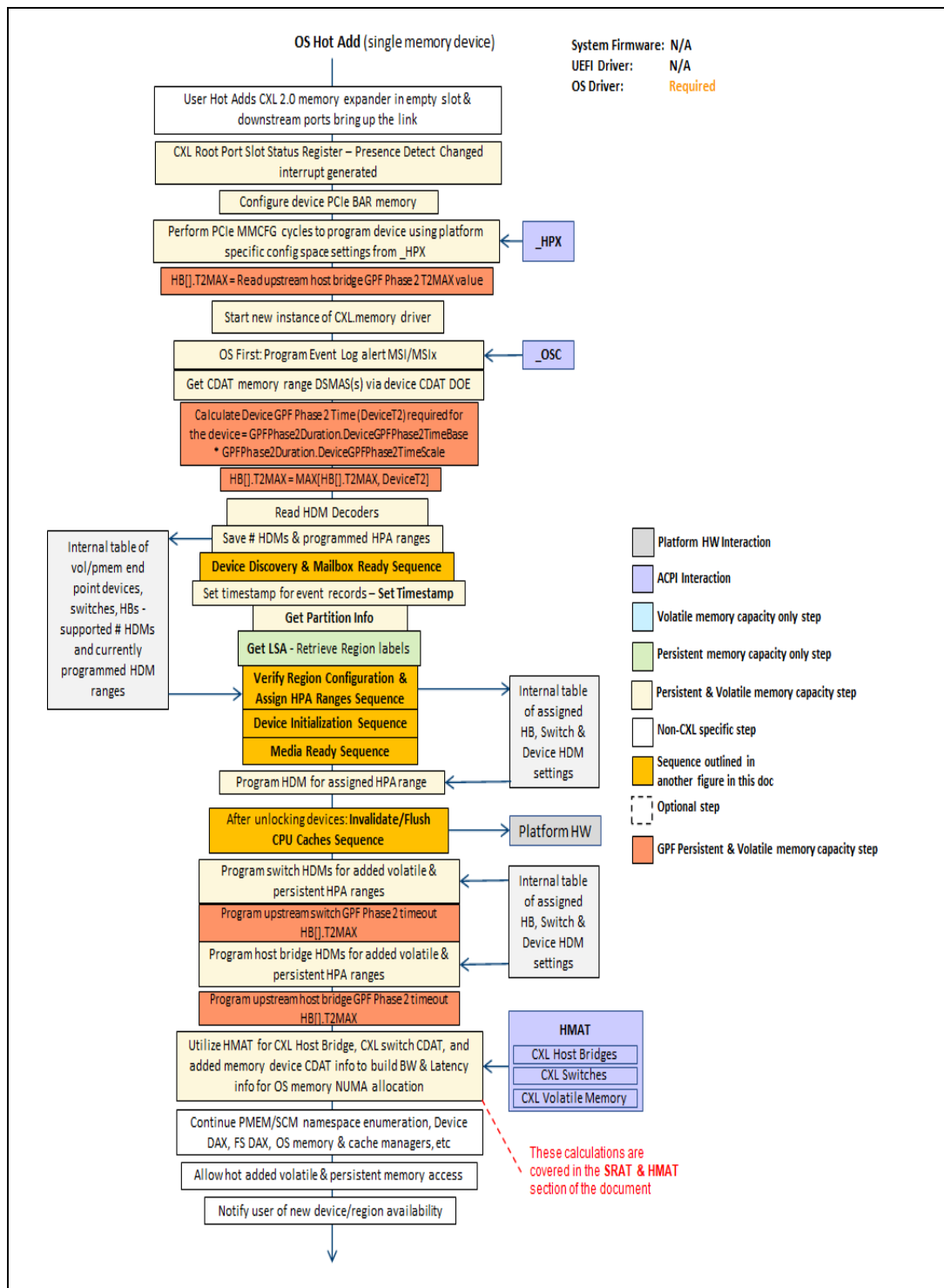
**Figure 38 - High-level sequence: System Firmware shutdown**



## 2.13.6 OS hot add sequence

The following figure outlines the basic high-level OS memory device hot add sequence for a single memory device.

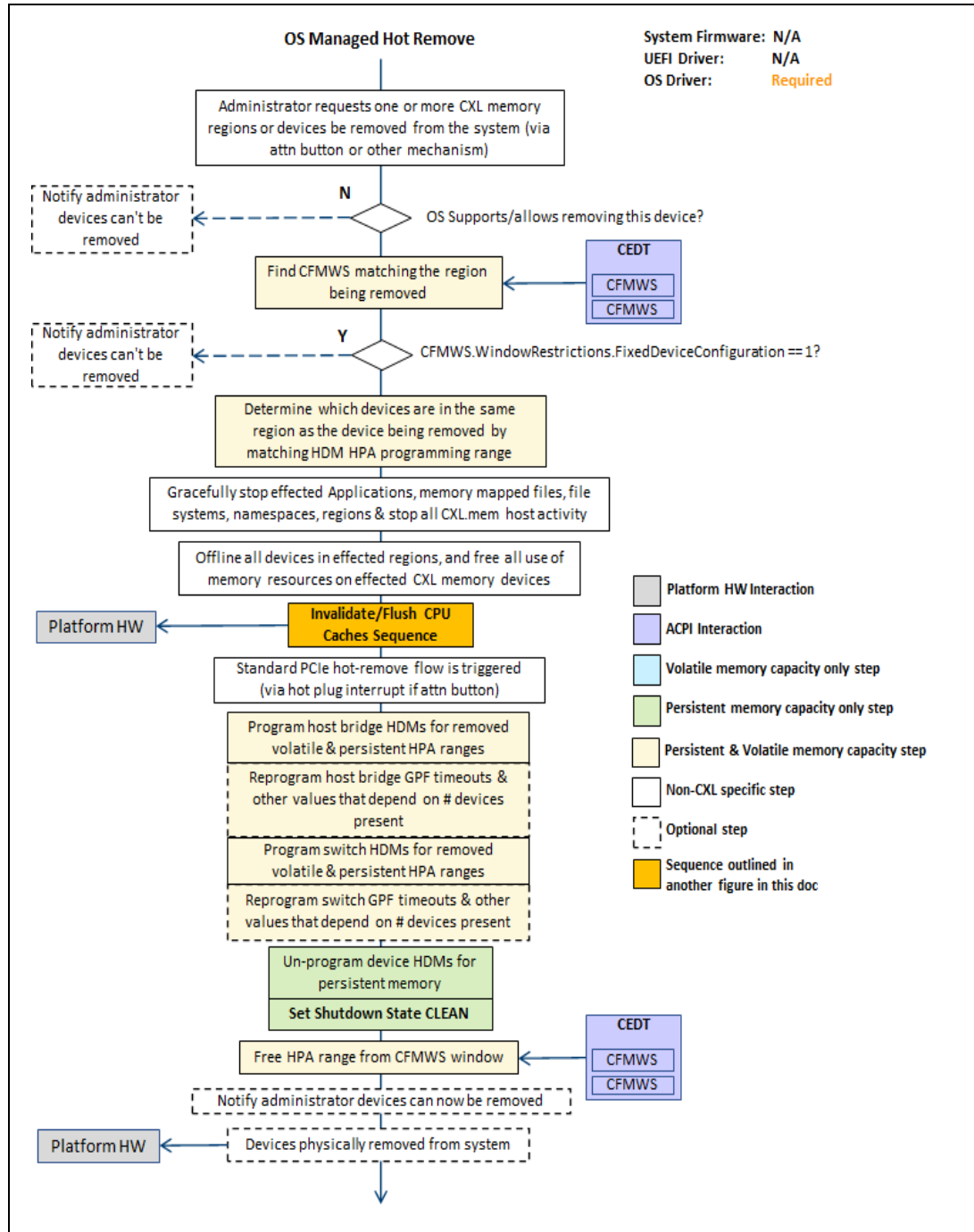
**Figure 39 - High-level sequence: OS hot add**



## 2.13.7 OS managed hot remove sequence

The following figure outlines the basic high-level OS memory managed hot remove sequence.

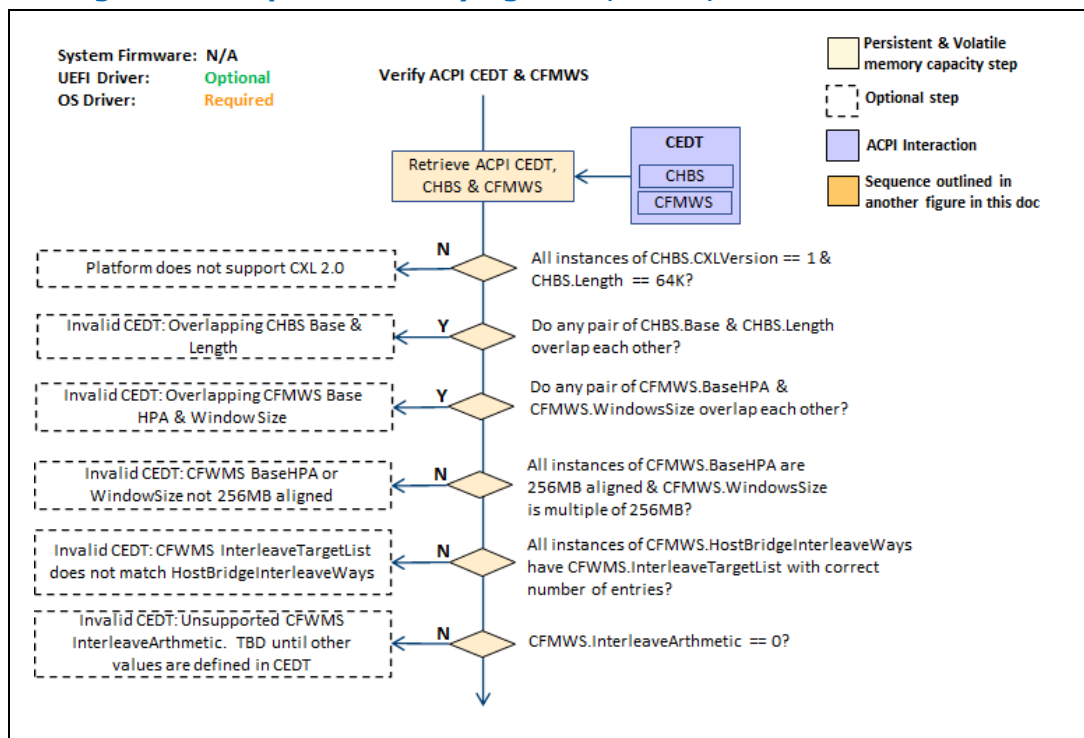
**Figure 40 - High-level sequence: OS managed hot remove**



## 2.13.8 Verifying ACPI CEDT, CHBS and CFMWS sequence

The following sequence outlines the basic steps the OS and UEFI drivers perform when verifying the consistency of the ACPI CEDT, CHBS and CFMWS tables produced by the System Firmware.

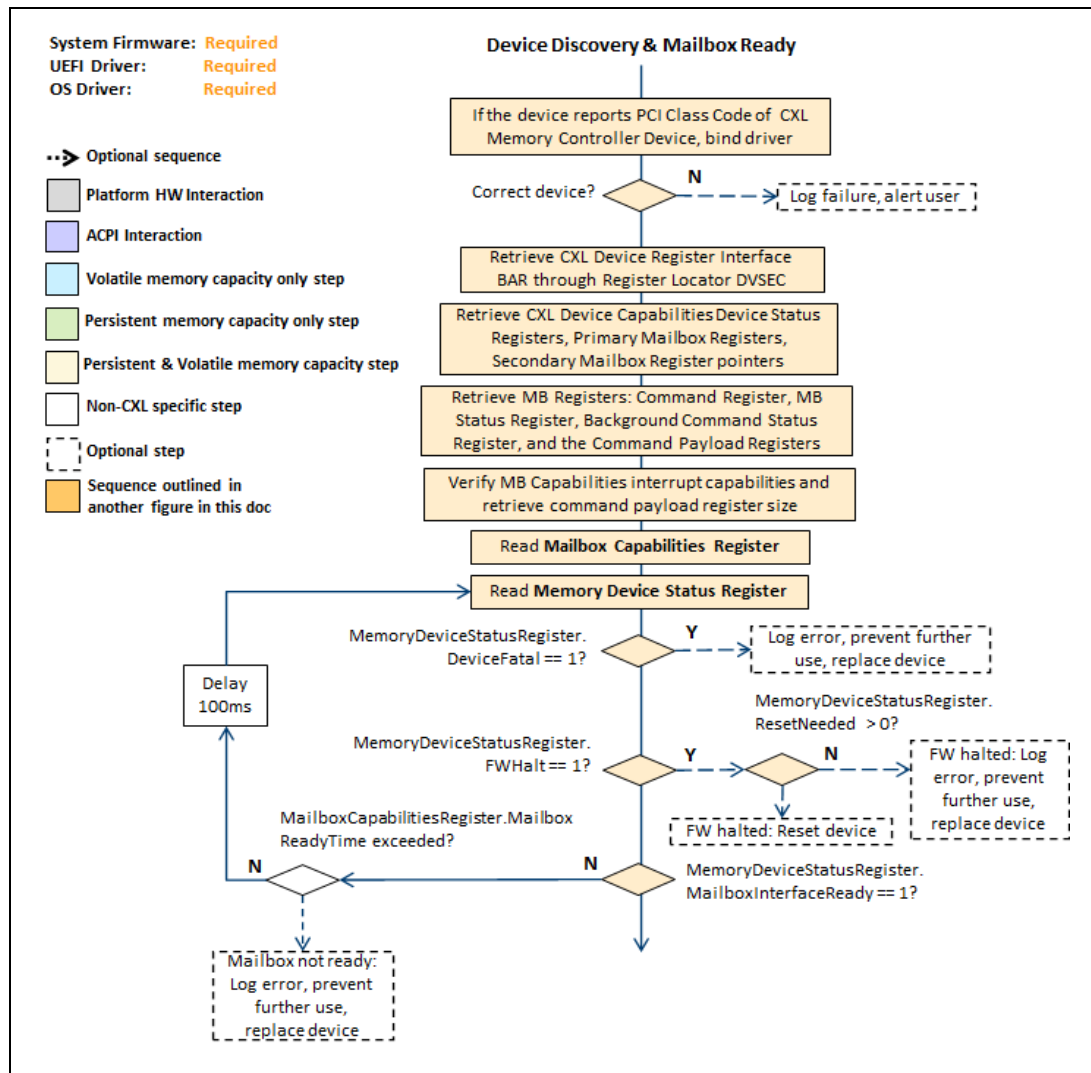
**Figure 41 - High-level sequence: Verifying CEDT, CHBS, CFMWS**



## 2.13.9 Device discovery and mailbox ready sequence

The following figure outlines the basic high-level sequence the System Firmware, UEFI and OS drivers will execute to determine if the CXL end point device found during PCI enumeration is a CXL memory device, discovery of the mailbox interface and polling on mailbox ready status for the device. After this sequence the device's mailbox interface can be utilized.

**Figure 42 - High-level sequence: Device discovery and mailbox ready**

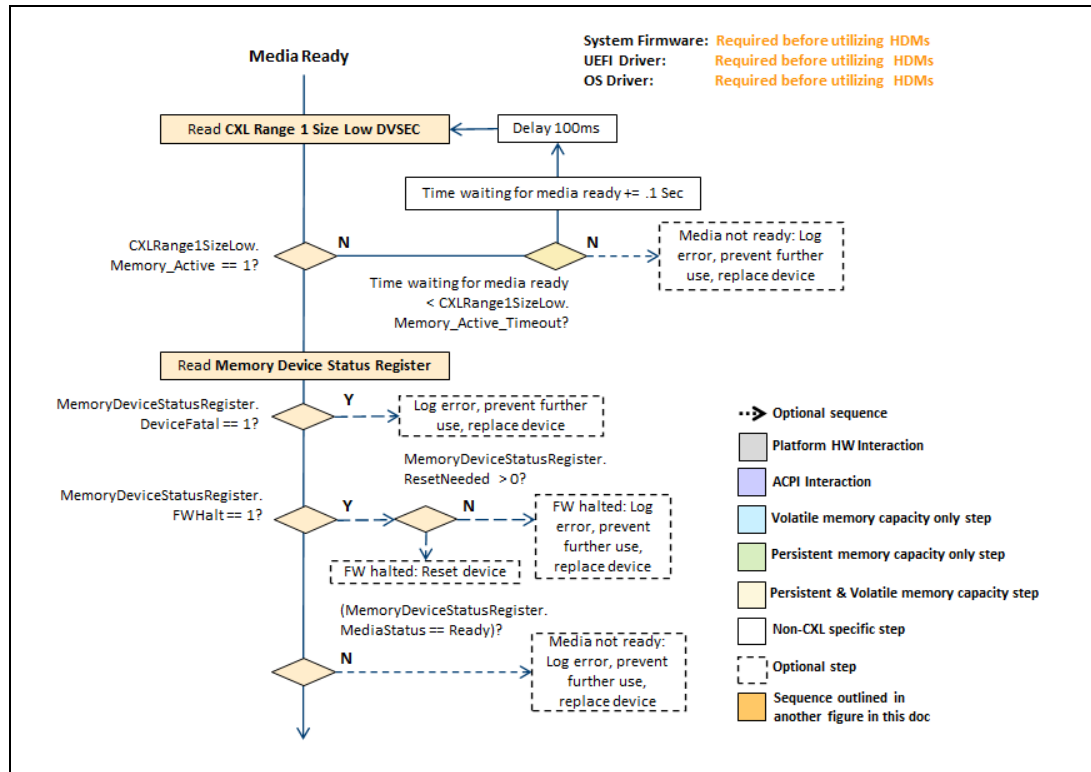




## 2.13.10 Media ready sequence

The following figure outlines the basic high-level sequence the System Firmware, UEFI Driver and OS drivers will execute to determine if the memory device is ready for media accesses using the HDMs. After this sequence the device's media can be utilized using the HDMs.

**Figure 43 - High-level sequence: Media ready**

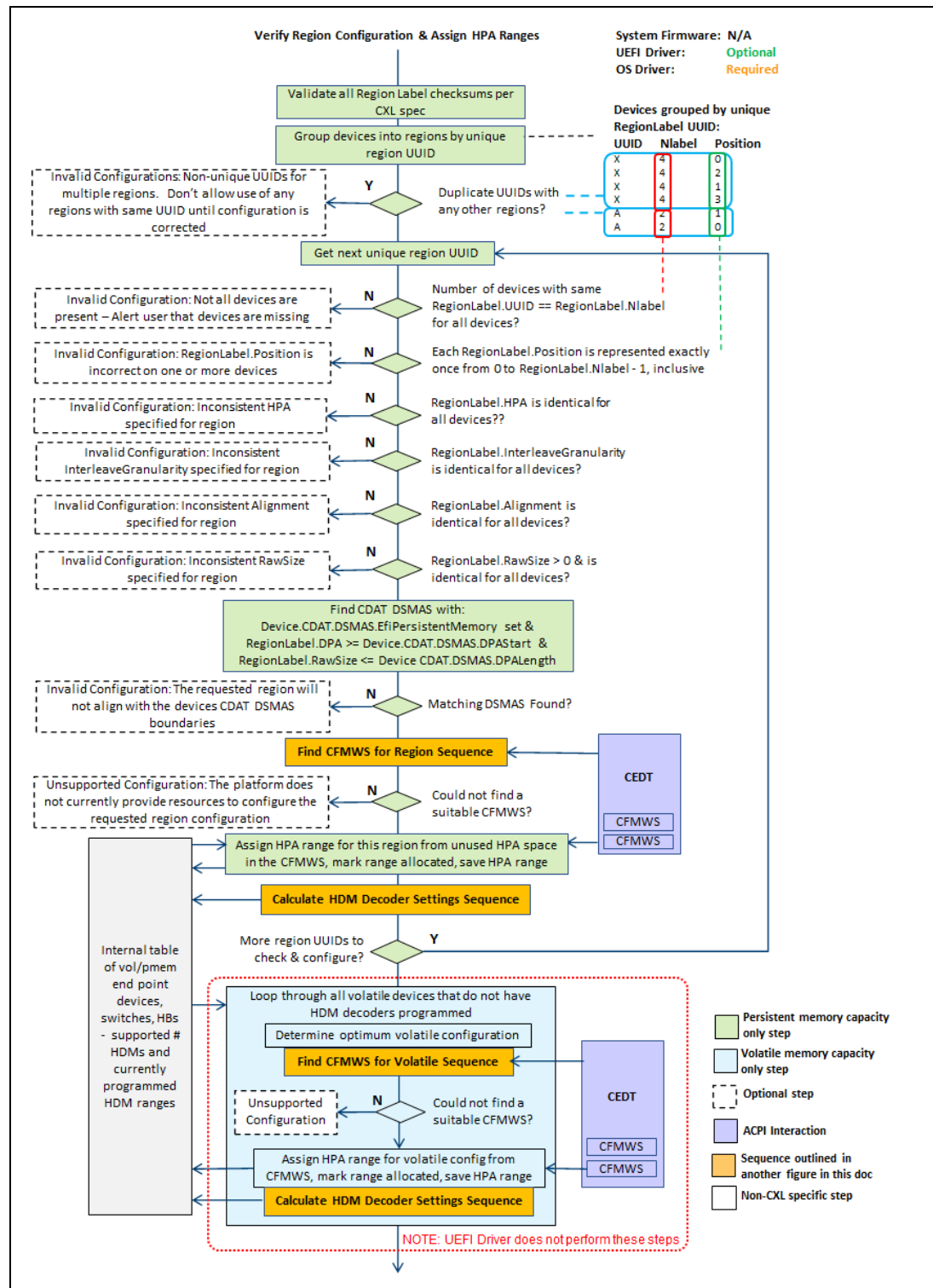


### **2.13.11 Verifying region configuration and assigning HPA ranges sequence**

The following sequence outlines the basic steps the OS and UEFI drivers perform when verifying the persistent memory region configuration and assigning HPA ranges from the CEDT. The sequence is performed anytime a persistent memory device is enumerated by the OS and UEFI drivers on every system boot, and by the OS drivers when handling a hot add of a device.

Note that intermediate CXL switches between CXL memory device and the CXL Host Bridge do not play a part in determining proper region configuration. If the connected devices in the region span multiple CXL Host Bridges, the OS and UEFI drivers must verify proper device ordering across host bridges. If the correct devices are connected to each host bridge, the order the devices are connected within the host bridge does not matter, as shown in the out of order memory region examples.

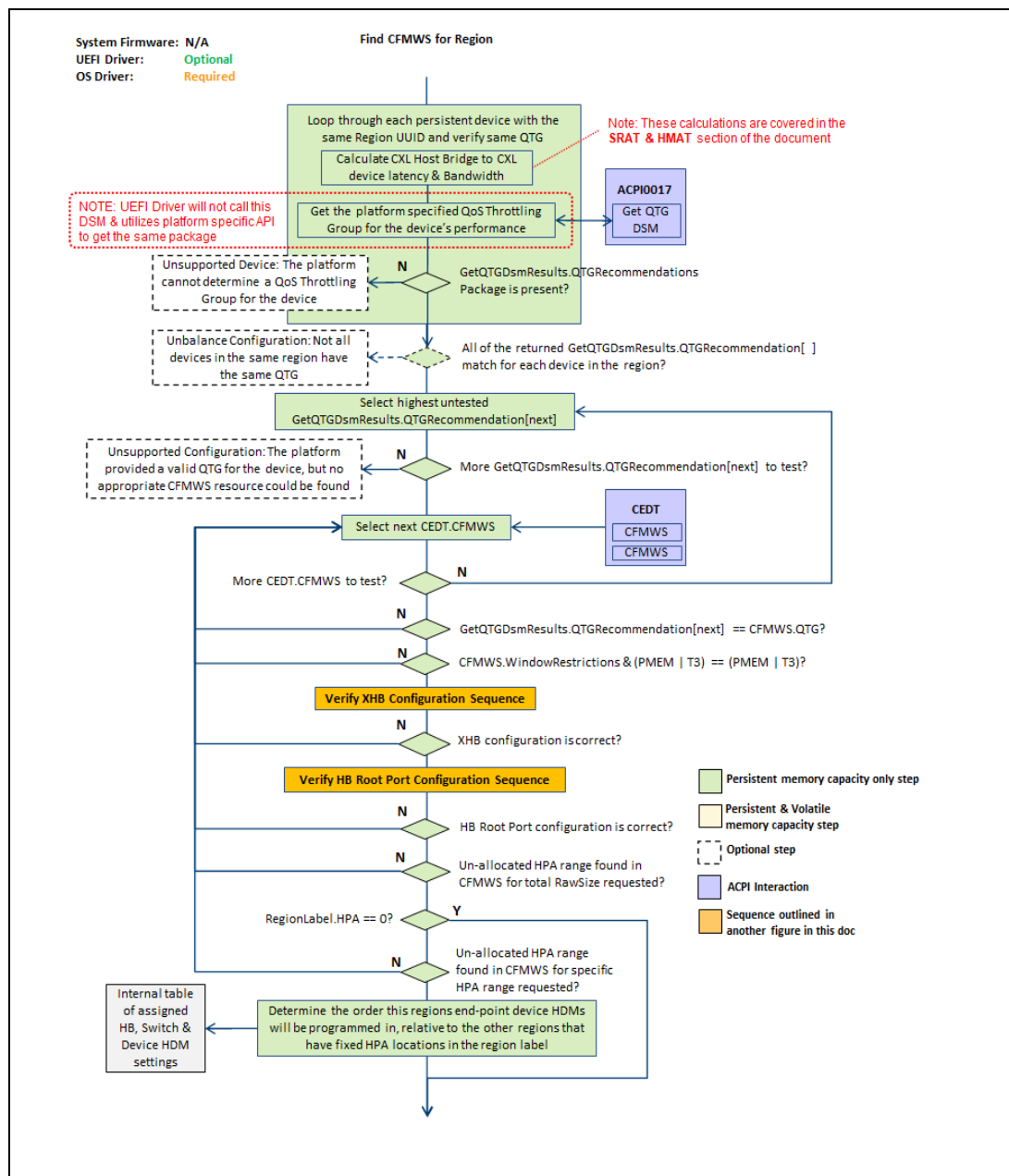
**Figure 44 - High-level sequence: Verifying region configuration**



## 2.13.12 Find CFMWS for region sequence

The following figure outlines the basic logic to find a suitable CFMWS for each region being configured. The sequence is performed anytime a persistent memory device is enumerated by the OS and UEFI drivers on every system boot, and by the OS drivers when handling a hot add of a device.

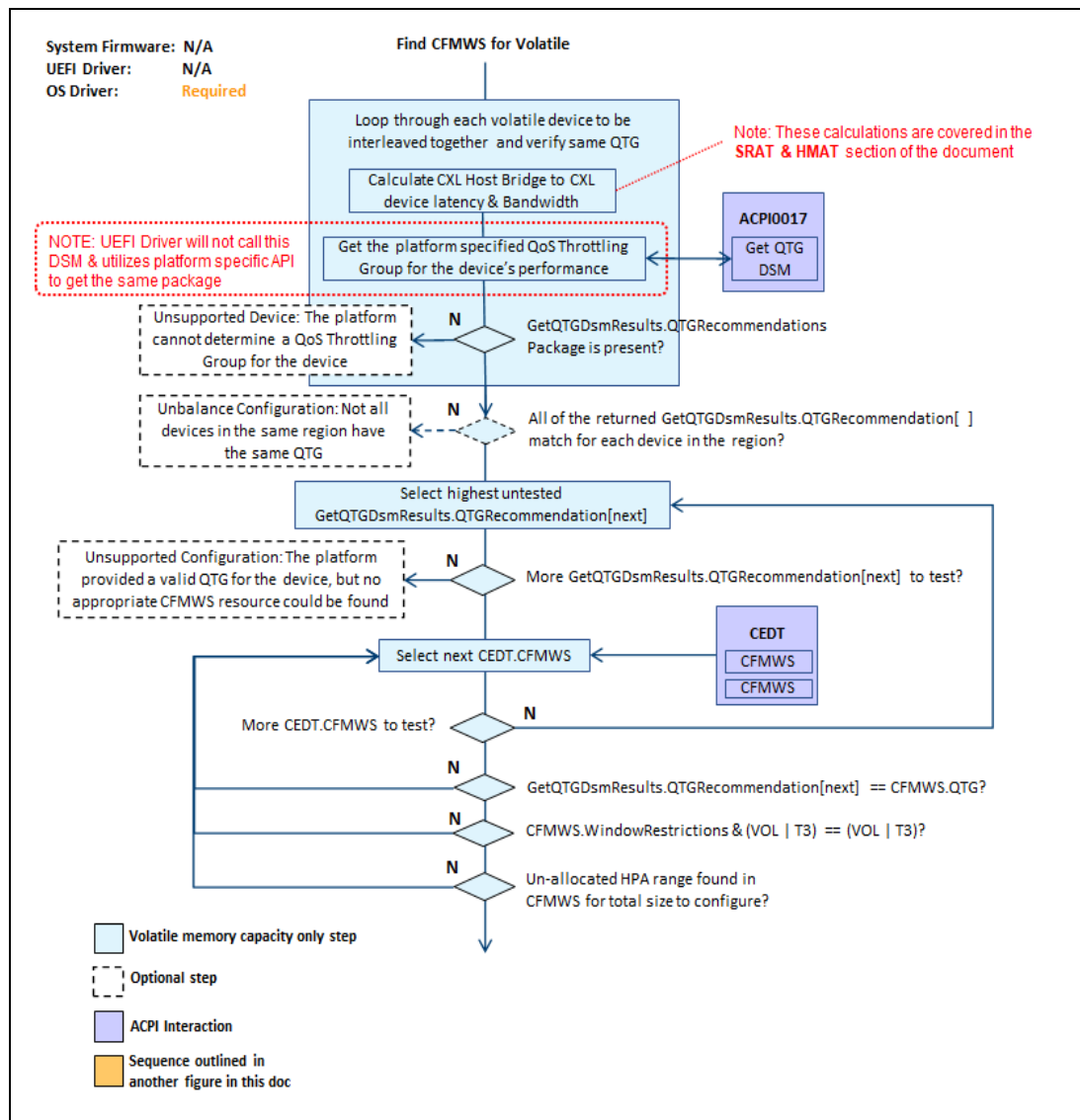
**Figure 45 - High-level sequence: Finding CFMWS for region**



### 2.13.13 Find CFMWS for volatile sequence

The following figure outlines the basic logic to find a suitable CFMWS for volatile memory being configured by the OS. The sequence is performed anytime a volatile memory device is enumerated by the OS drivers on every system boot, and by the OS drivers when handling a hot add of a device.

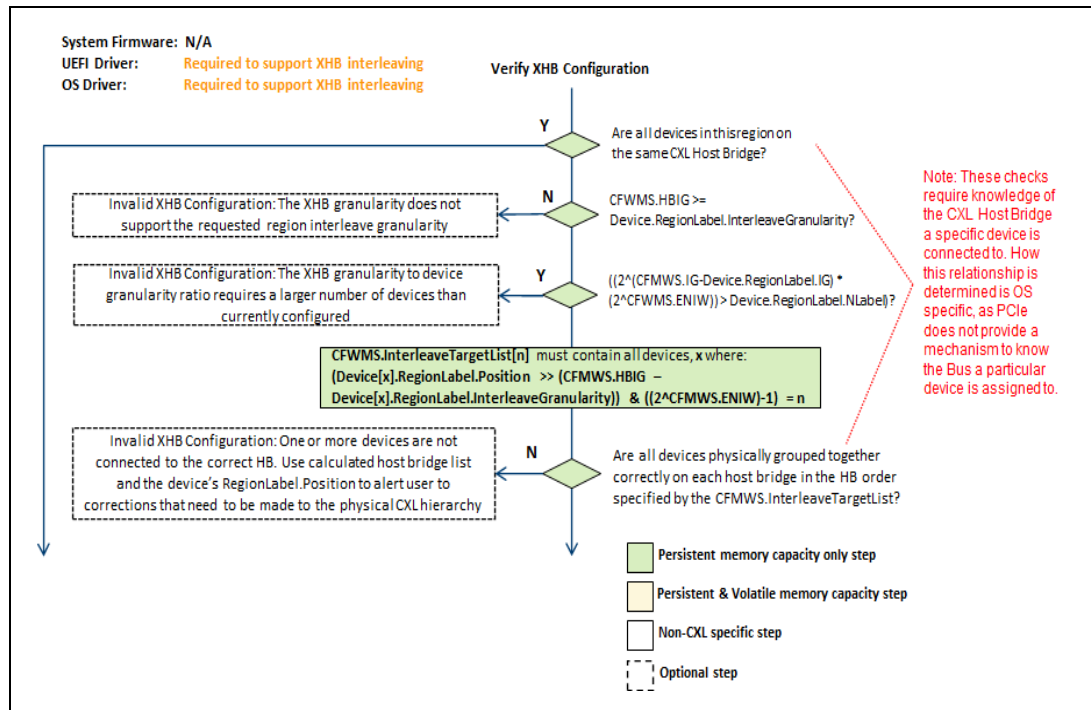
**Figure 46 - High-level sequence: Finding CFMWS for volatile**



## 2.13.14 Verify XHB configuration sequence

For platforms that support XHB regions, the following sequence outlines the basic steps the OS and UEFI drivers perform when verifying the XHB persistent memory region configuration. The sequence is performed anytime a persistent memory device is enumerated by the OS and UEFI drivers on every system boot, and by the OS drivers when handling a hot add of a device.

**Figure 47 - High-level sequence: Verify XHB configuration**

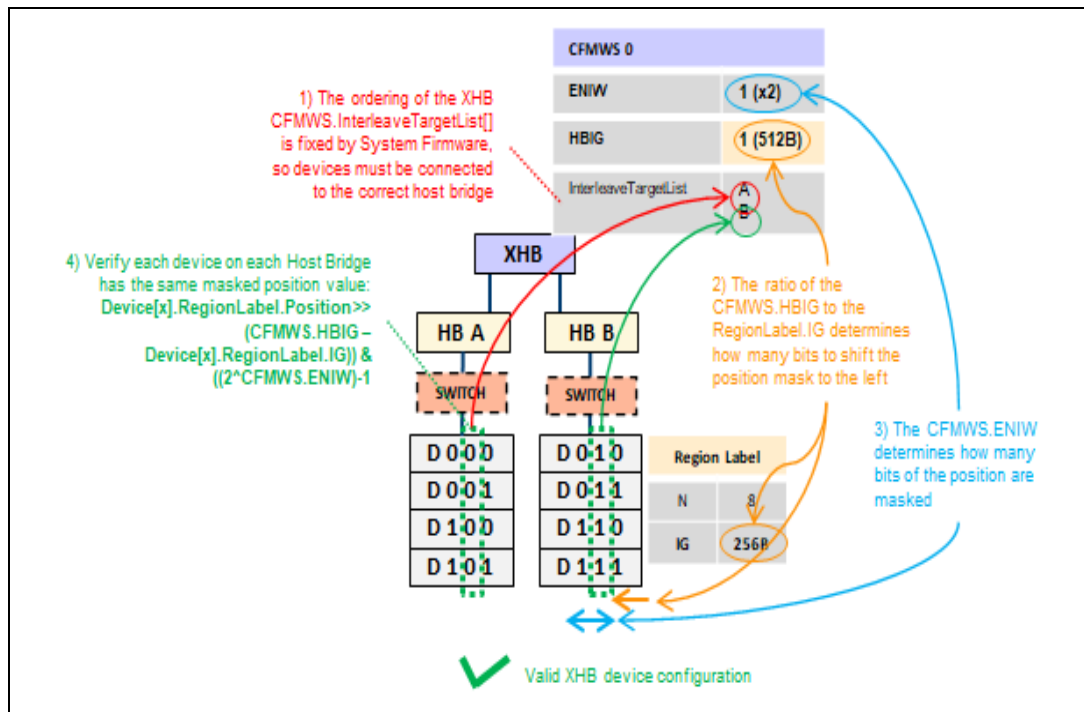


Here are some example region configurations and how this algorithm allows SW to determine correct device placement on each host bridge. This XHB algorithm does not need to check device ordering within a host bridge or intermediate switch that might be present. Those are covered in the following section for CXL Host Bridge root port ordering checks.

## 2.13.14.1 x2 XHB example configuration checks

Here is an example valid x2 XHB configuration and illustrated steps this algorithm executes:

**Figure 48 - Example valid x2 XHB configuration execution steps**

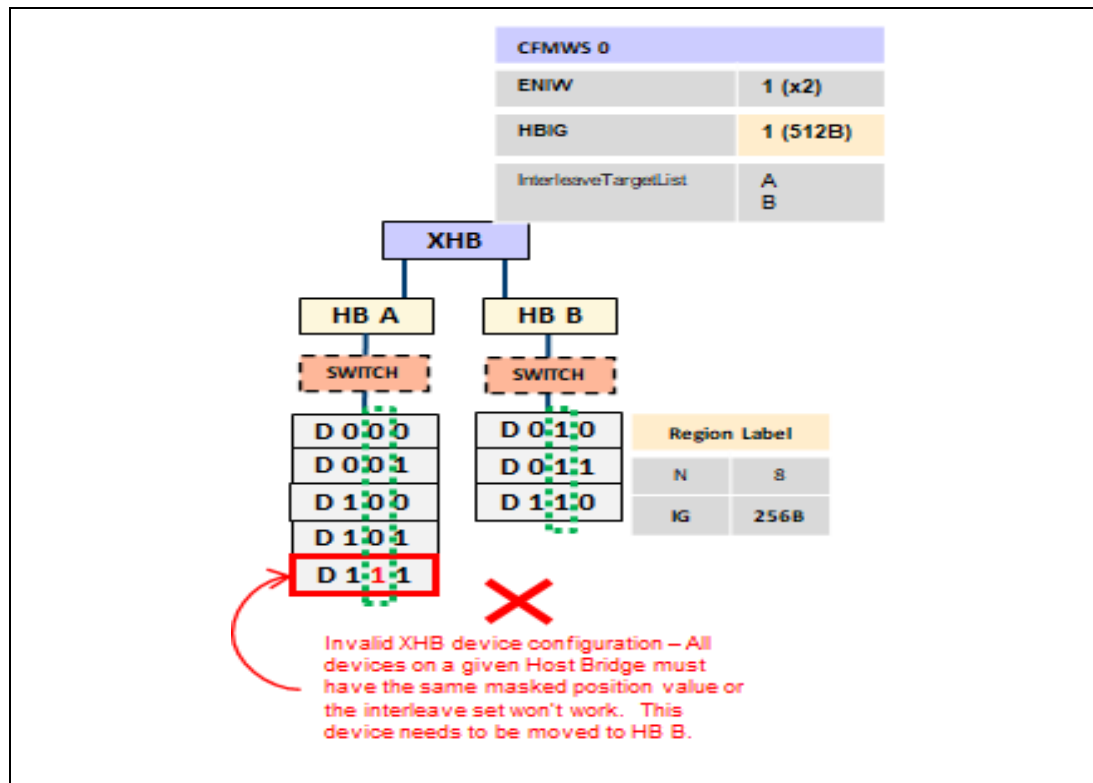


And the resulting calculations used to verify the configuration:

Configuration	XHB CFMWS	Device RegionLabel	RegionLabel. Position	CFMWS. HBIG - Dev. RegionLabel. IG	$((2^{\text{CFMWS.ENIW}}) - 1)$	n
-x2 XHB w x8 device region -4 devices on each HB -Different host bridge and device Interleave Granularity	CFMWS.ENIW = 1 (x2)  CFMWS.HBIG = 1 (512B)	NLabel = 8  Interleave Granularity = 0 (256B)	0000	1-0=1	$2^1-1=1$	0
			0001			0
			0010			1
			0011			1
			0100			0
			0101			0
			0110			1
			0111			1

Example of an invalid x2 XHB configuration that this check would catch:

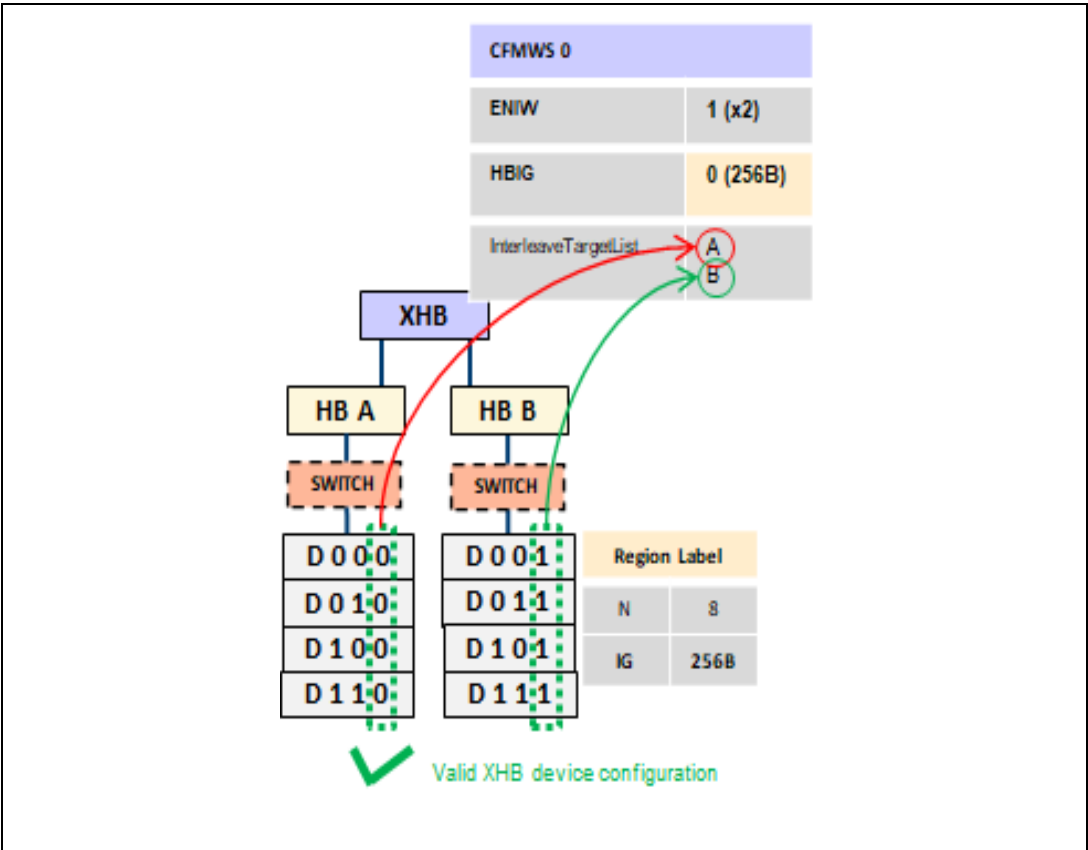
**Figure 49 - Example invalid x2 XHB configuration**





Here is another example valid x2 XHB configuration:

Figure 50 - Example valid x2 XHB configuration



And the resulting calculations used to verify the configuration:

Configuration	XHB CFMWS	Device RegionLabel	RegionLabel. Position	CFMWS. HBIG - Dev.Regio nLabel.I G	$((2^{CFMWS.ENIW}) - 1)$	n
-x2 XHB w x8 device region -4 devices on each HB -Same host bridge and device Interleave Granularity	CFMWS.E NIW = 1 (x2)  CFMWS. HBIG = 0 (256B)	NLabel = 8  Interleave Granularity = 0 (256B)	0000	0-0=0	$2^1-1=1$	0
			0001			1
			0010			0
			0011			1
			0100			0
			0101			1
			0110			0
			0111			1

## 2.13.14.2 x4 XHB example configuration checks

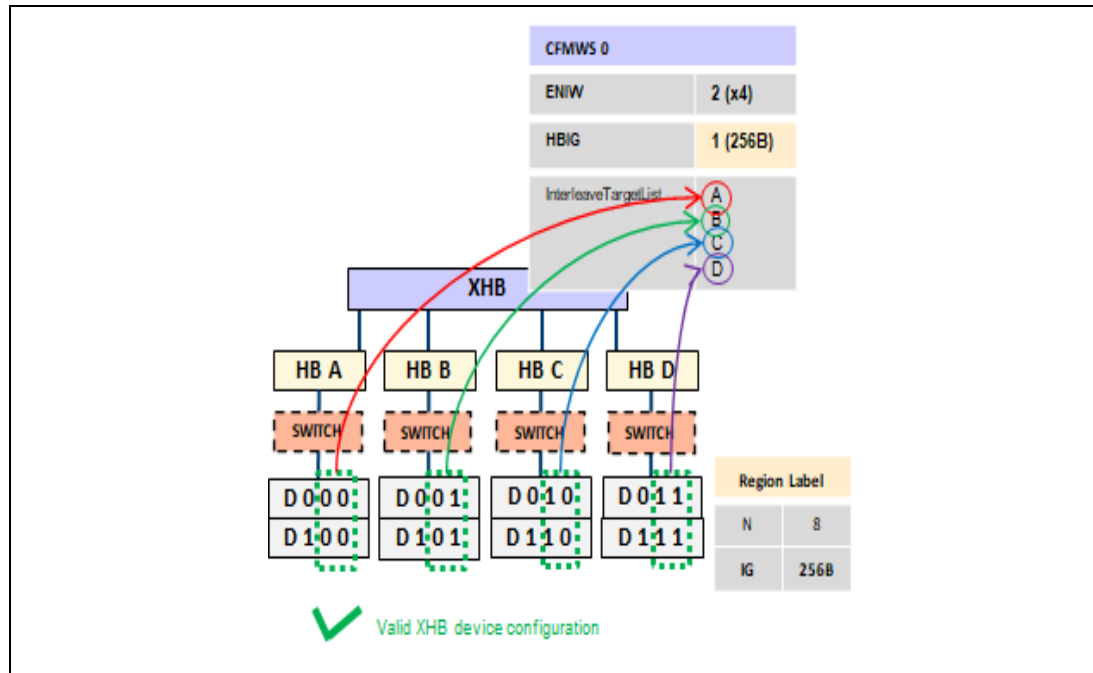
Here is an invalid x4 XHB configuration example that would require more devices than are connected:

Configuration	XHB CFMWS	Device RegionLabel	RegionLabel. Position	CFMWS. HBIG - Dev.RegionLabel.IG	$((2^{CFMWS.ENIW}) - 1)$	n
-x4 XHB w x8 device region -2 devices on each HB -Different host bridge and device Interleave Granularity	CFMWS.ENIW = 2 (x4)  CFMWS.HBIG = 2 (1K)	NLabel = 8  Interleave Granularity = 0 (256B)		<p>Invalid configuration: The CFMWS.IG will send 1K down each HB so with 256 for device IG there would need to be <math>1K/256B = 4</math> devices on each HB which is 16 devices and is <math>&gt; Dev.RegionLabel.NLabel</math>.</p> <p>This is caught by the following test in the previous flow:            If <math>((2^{(CFMWS.IG - Dev.RegionLabel.IG)} * (2^{CFMWS.ENIW})) &gt; Dev.RegionLabel.NLabel)</math> //invalid configuration</p>		



Here is another example valid x4 XHB configuration:

**Figure 52 - Example valid x4 XHB configuration**

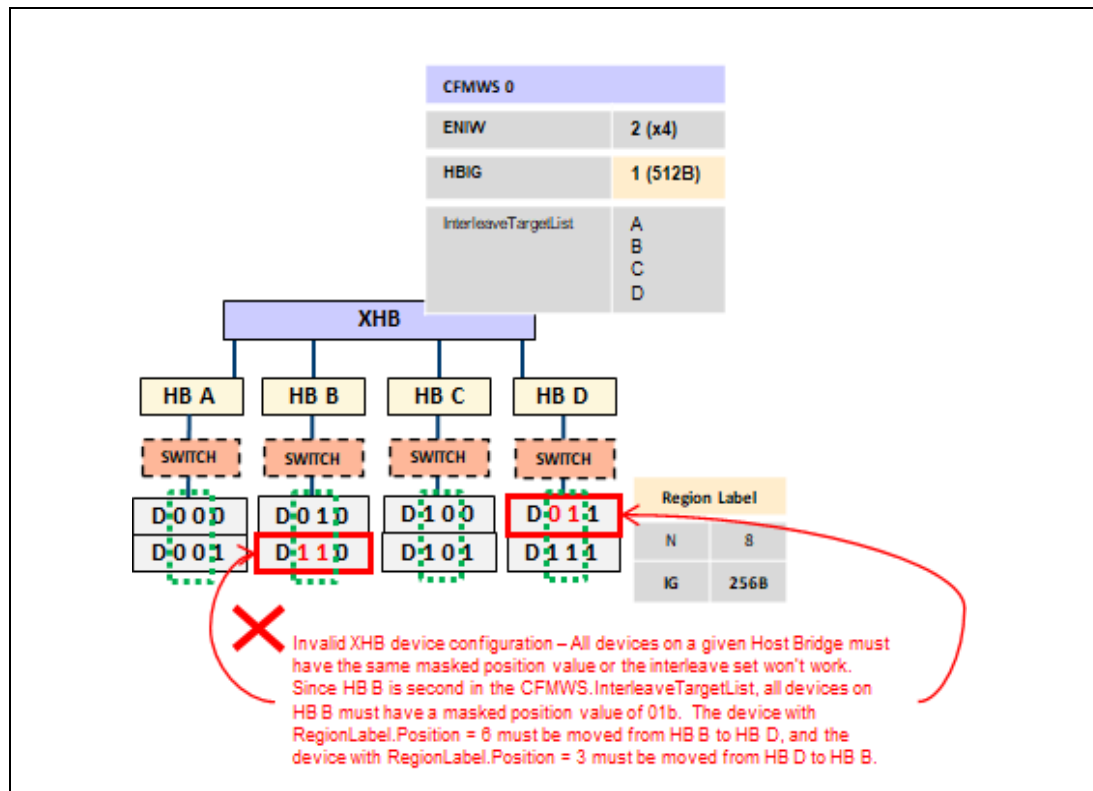


And the resulting calculations used to verify the configuration:

Configuration	XHB CFMWS	Device RegionLabel	RegionLabel. Position	CFMWS. HBIG - Dev.RegionLabel.I G	$((2^{CFMWS.ENIW}) - 1)$	n
-x4 XHB w x8 device region -2 devices on each HB -Same host bridge and device Interleave Granularity	CFMWS.E NIW = 2 (x4)  CFMWS. HBIG = 0 (256B)	NLabel = 8  Interleave Granularity = 0 (256B)	0000	0-0=0	$2^2-1=11b$	00
			0001			01
			0010			10
			0011			11
			0100			00
			0101			01
			0110			10
			0111			11

Example of an invalid x4 XHB configuration that this check would catch:

**Figure 53 - Example invalid x4 XHB configuration**



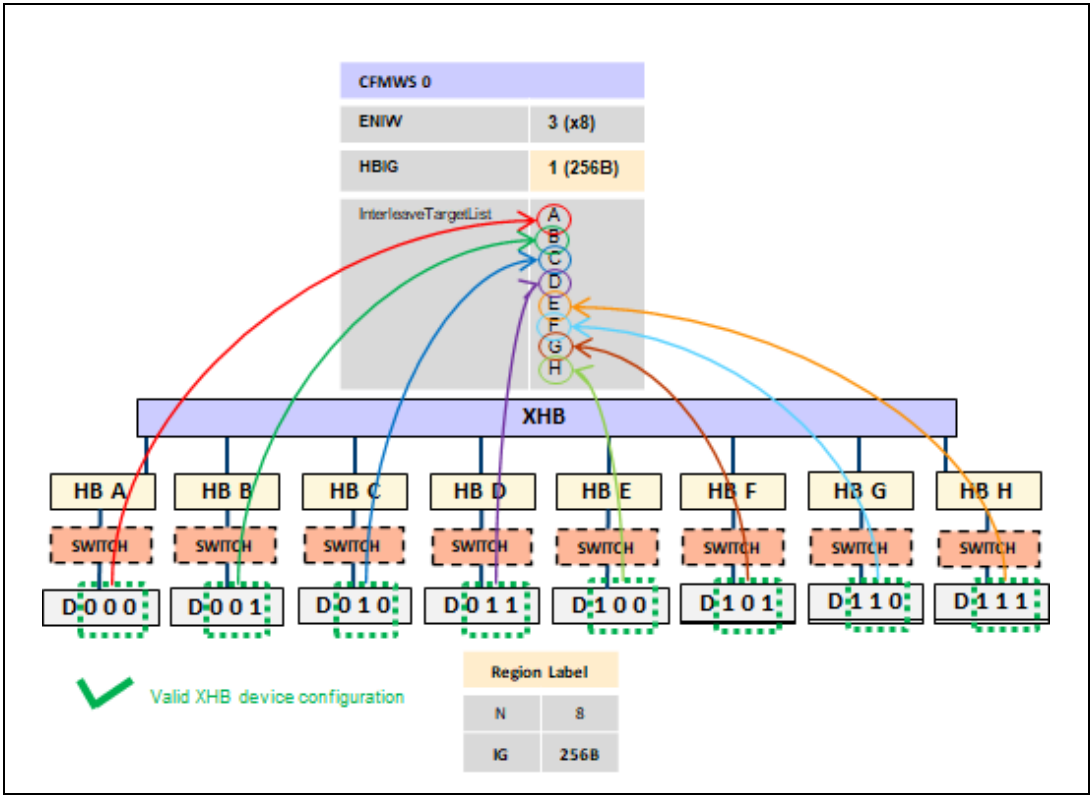
### 2.13.14.3 x8 XHB example configuration checks

Here is an invalid x8 XHB configuration example that would require more devices than are connected:

Configuration	XHB CFMWS	Device RegionLabel	RegionLabel. Position	CFMWS. HBIG – Dev.RegionLabel.IG	$((2^{CFMWS.ENIW}) - 1)$	n
-x8 XHB w x8 device region -1 devices on each HB -Different host bridge and device Interleave Granularity	CFMWS.ENIW = 3 (x8)  CFMWS. HBIG = 2 (1K)	NLabel = 8  Interleave Granularity = 0 (256B)		<p>Invalid configuration: The CFMWS.IG will send 512B down each HB so with 256 for device IG there would need to be 1K/256B = 4 devices on each HB which is 16 devices and is &gt; Dev.RegionLabel.NLabel.</p> <p>This is caught by the following test in the previous flow:            If <math>((2^{(CFMWS.IG - Dev.RegionLabel.IG)} * (2^{CFMWS.ENIW})) &gt; Dev.RegionLabel.NLabel)</math> //invalid configuration</p>		

Here is an example valid x8 XHB configuration:

Figure 54 - Example valid x8 XHB configuration



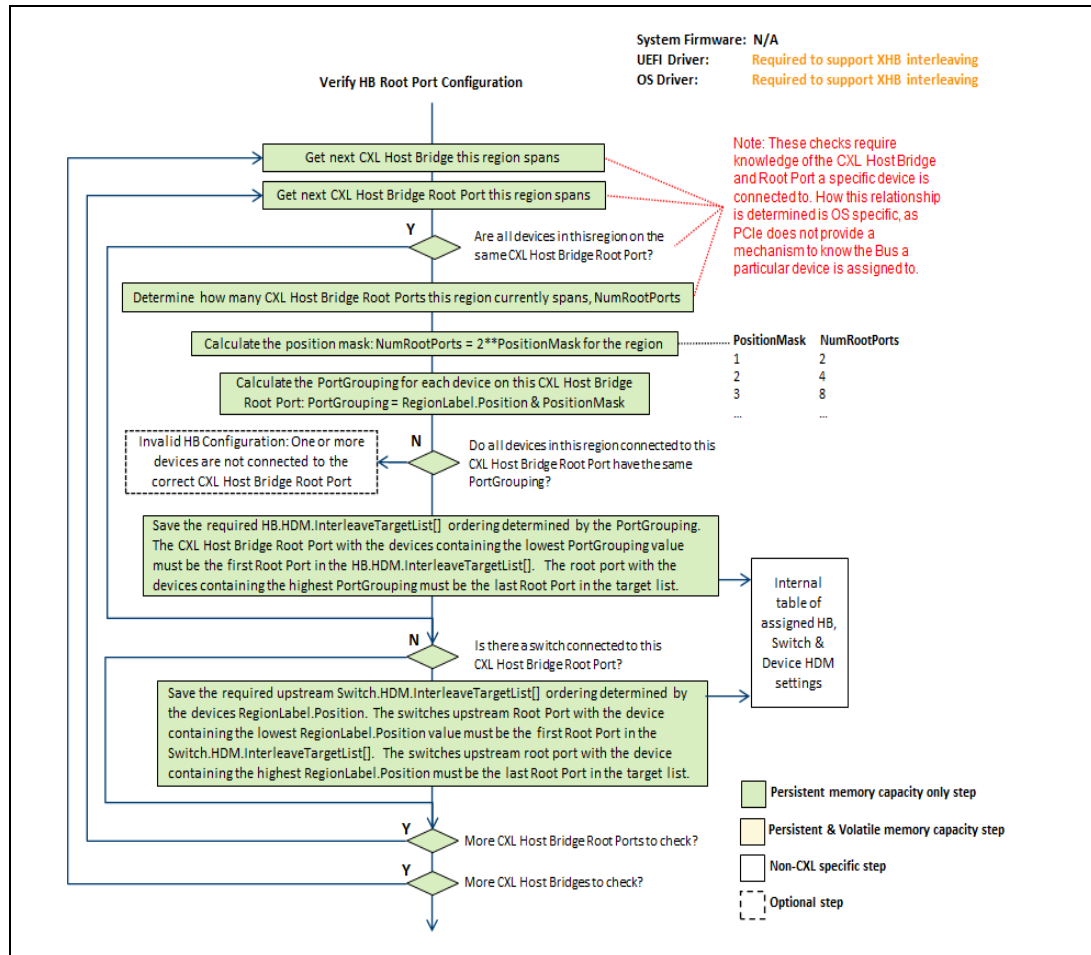
And the resulting calculations used to verify the configuration:

Configuration	XHB CFMWS	Device RegionLabel	RegionLabel. Position	CFMWS. HBIG - Dev.Regio nLabel.I G	$((2^{CFMWS.ENIW}) - 1)$	n
-x8 XHB w x8 device region -1 device on each HB -Same host bridge and device Interleave Granularity	CFMWS.E NIW = 3 (x8)  CFMWS. HBIG = 0 (256B)	NLabel = 8  Interleave Granularity = 0 (256B)	0000	0-0=0	$2^3-1=111b$	000
			0001			001
			0010			010
			0011			011
			0100			100
			0101			101
			0110			110
			0111			111

## 2.13.15 Verify HB root port configuration sequence

The following sequence outlines the basic steps the OS and UEFI drivers perform when verifying that each device in each region is grouped correctly on each CXL Host Bridge root port of the requested region configuration. The sequence is performed anytime a persistent memory device is enumerated by the OS and UEFI drivers on every system boot, and by the OS drivers when handling a hot add of a device.

**Figure 55 - High-level sequence: Verify HB root port configuration**



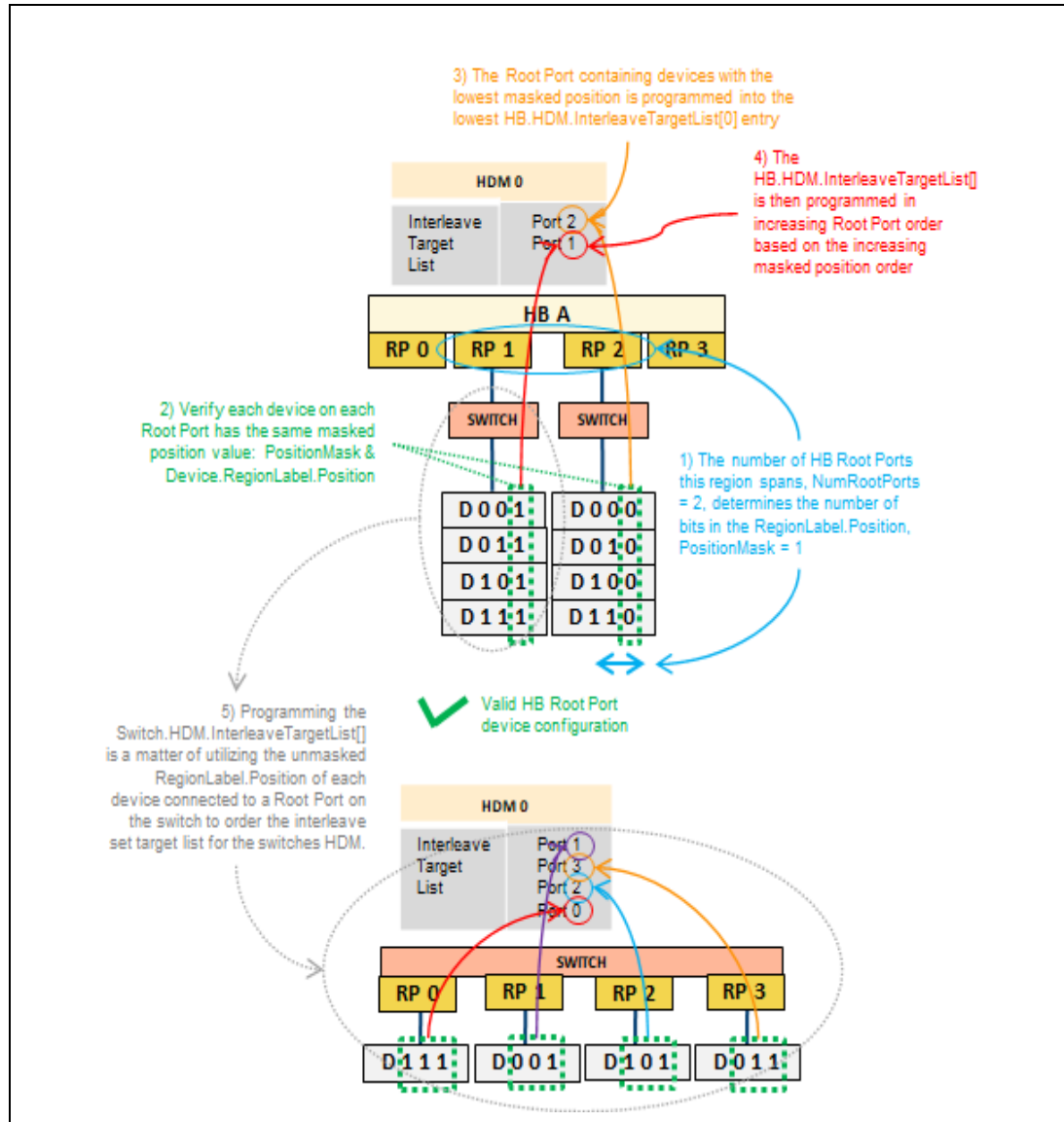
The following sections outline examples of the previous CXL Host Bridge root port configuration checks required for persistent memory regions.



### 2.13.15.1 Region spanning 2 HB root ports example configuration checks

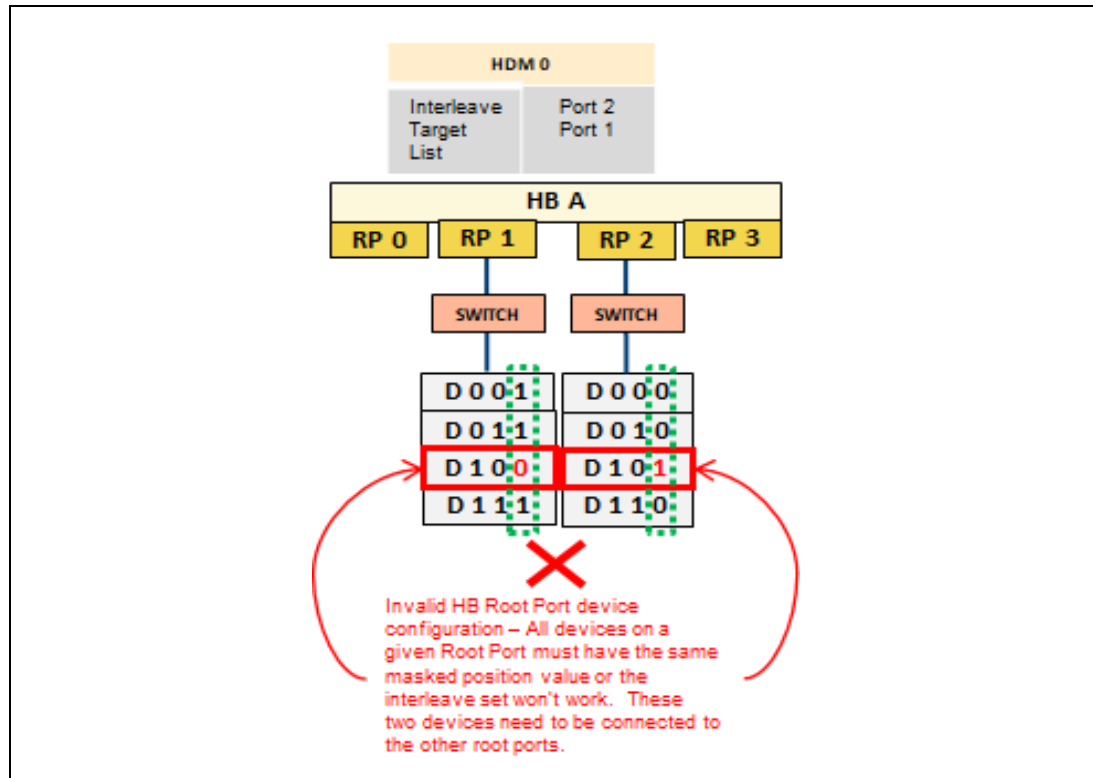
Here is an example valid region spanning 2 HB root ports and illustrated steps this algorithm executes:

**Figure 56 - Example valid region spanning 2 HB root ports**



Example of an invalid region spanning 2 HB root ports that this check would catch:

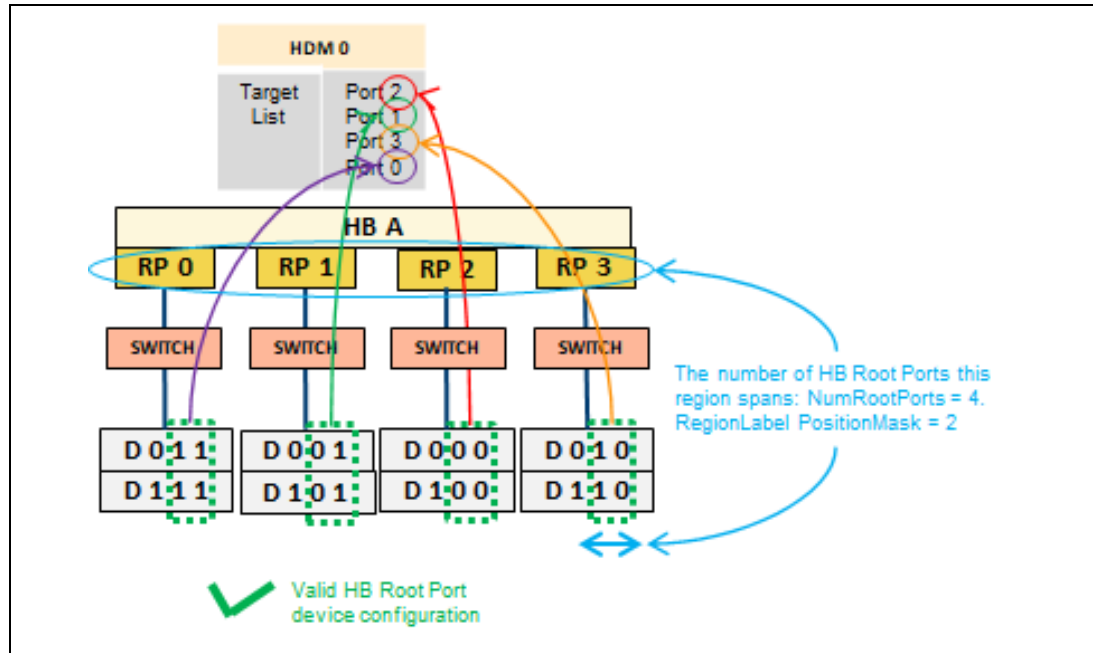
**Figure 57 - Example invalid region spanning 2 HB root ports**



## 2.13.15.2 Region spanning 4 root ports example configuration checks

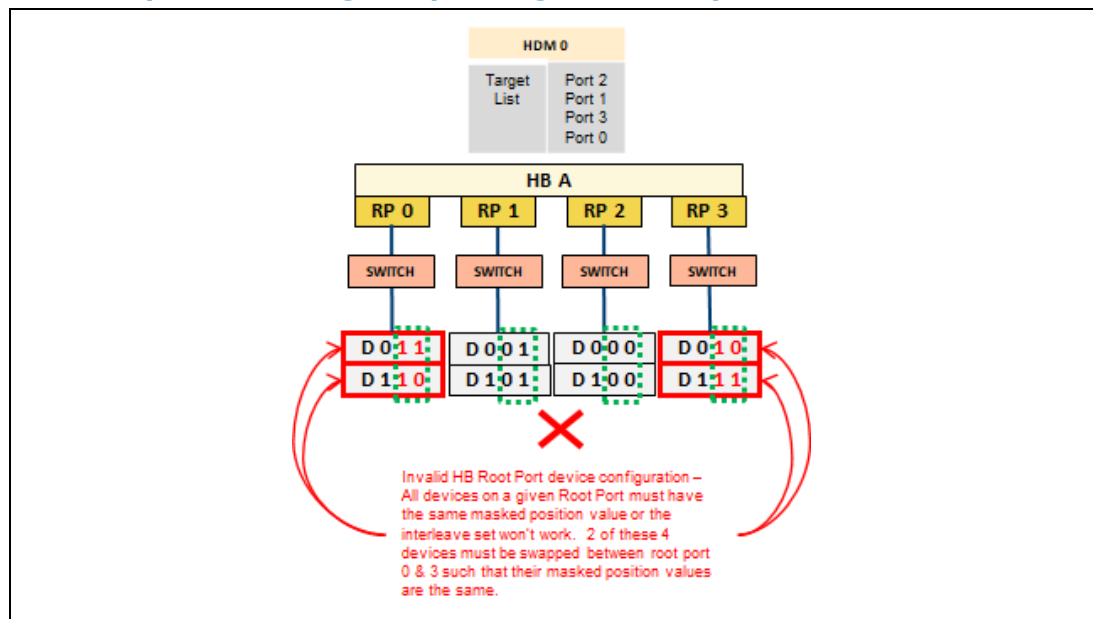
Here is an example valid region spanning 4 HB root ports:

**Figure 58 - Example valid region spanning 4 HB root ports**



Example of an invalid region spanning 2 HB root ports that this check would catch:

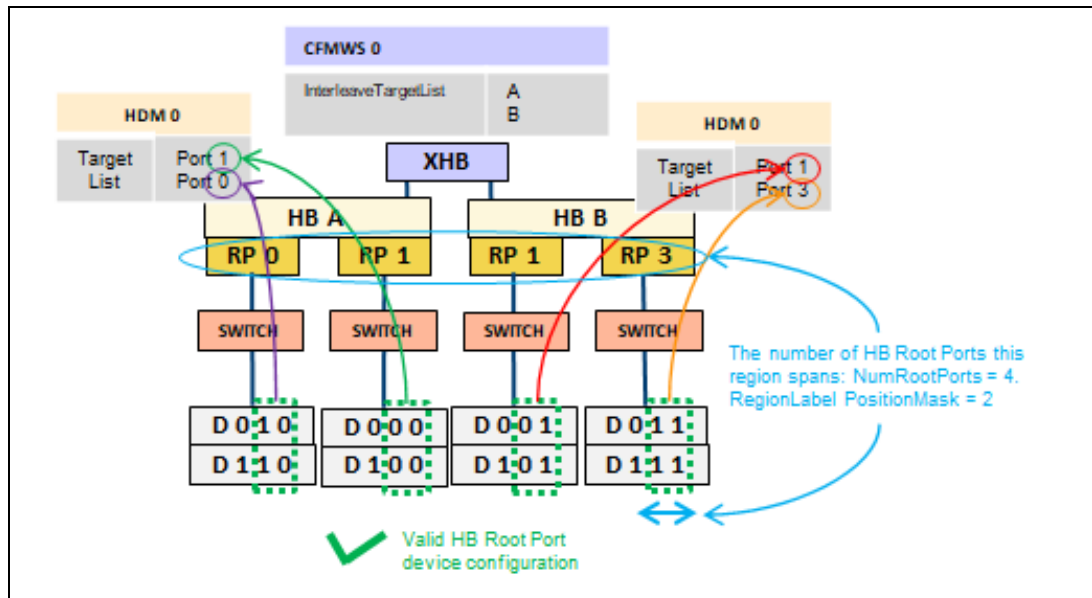
**Figure 59 - Example invalid region spanning 4 HB root ports**



### 2.13.15.3 Region spanning 4 HB root ports and x2 XHB example configuration checks

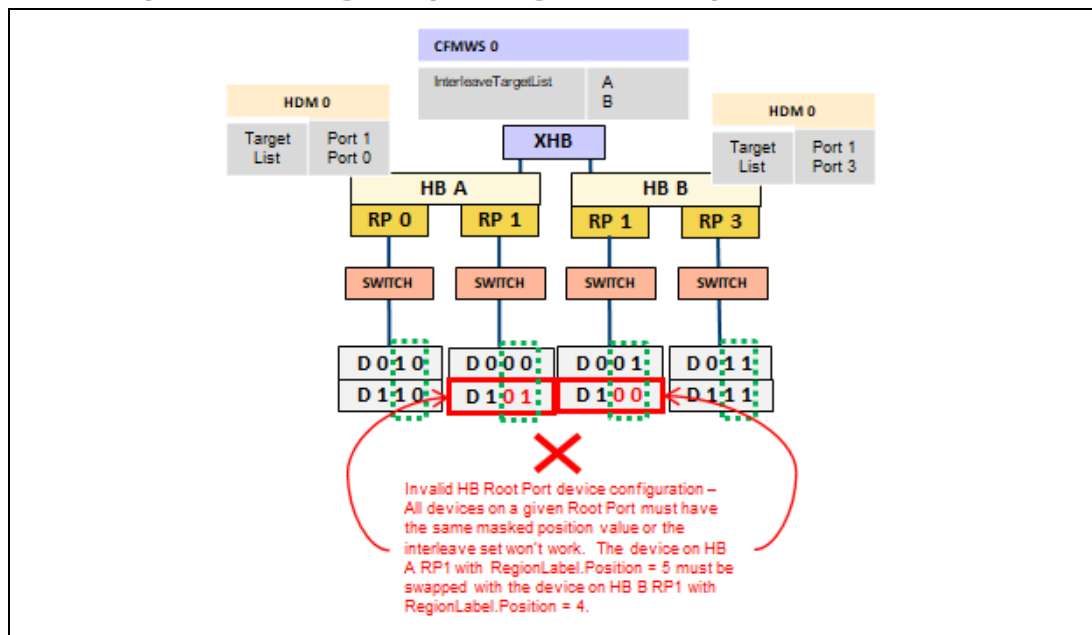
Here is an example valid region spanning 4 HB root ports on a x2 XHB:

**Figure 60 - Example valid region spanning 4 HB root ports on a x2 XHB**



Example of an invalid region spanning 2 HB root ports on a x2 XHB that this check would catch:

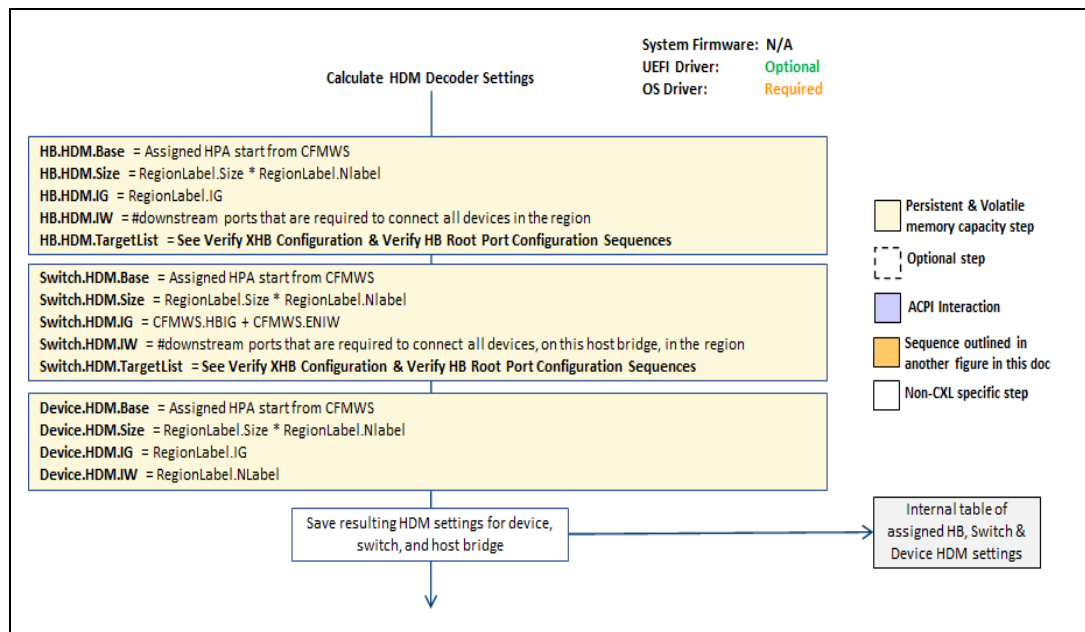
**Figure 61 - Example invalid region spanning 4 HB root ports on a x2 XHB**



## 2.13.16 Calculate HDM decoder settings sequence

The following figure outlines the basic high-level sequence the UEFI and OS drivers will execute in preparation for programming the HDM decoders.

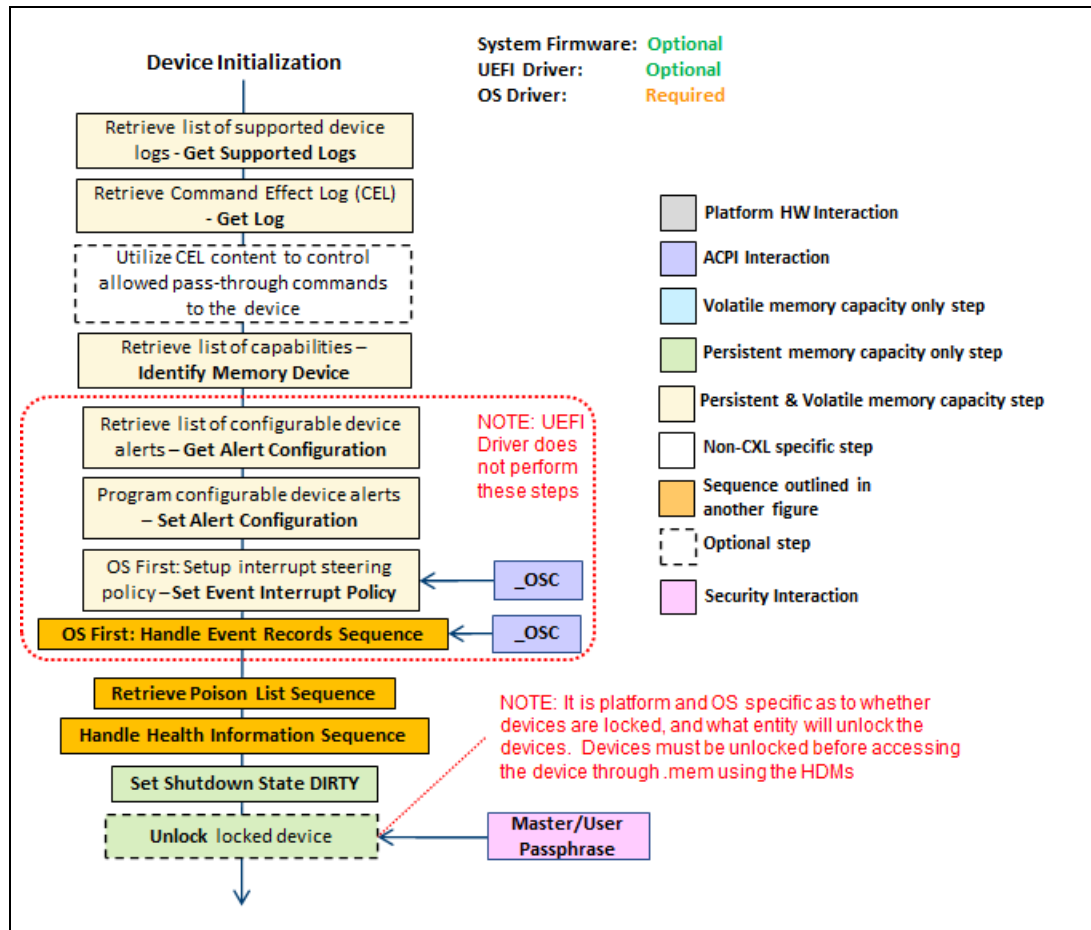
**Figure 62 - High-level sequence: Calculate HDM decoder settings**



## 2.13.17 Device initialization sequence

The following figure outlines the basic high-level sequence the platform UEFI and OS drivers will execute to handle the memory device initialization.

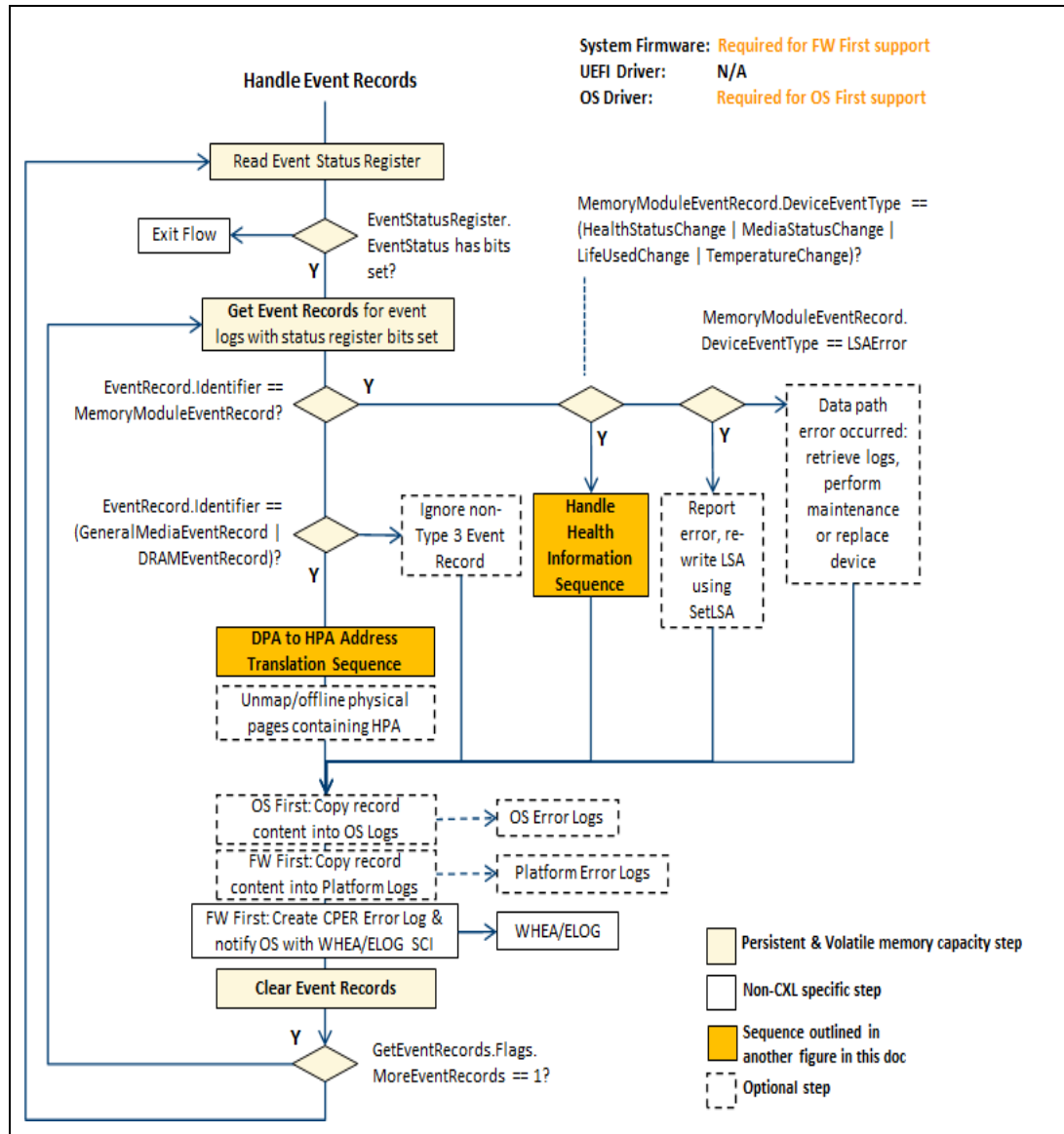
**Figure 63 - High-level sequence: Device initialization**



## 2.13.18 Handle event records sequence

The following figure outlines the basic high-level flow that the System Firmware and OS drivers will need to execute to retrieve outstanding event records from each of the device's event logs.

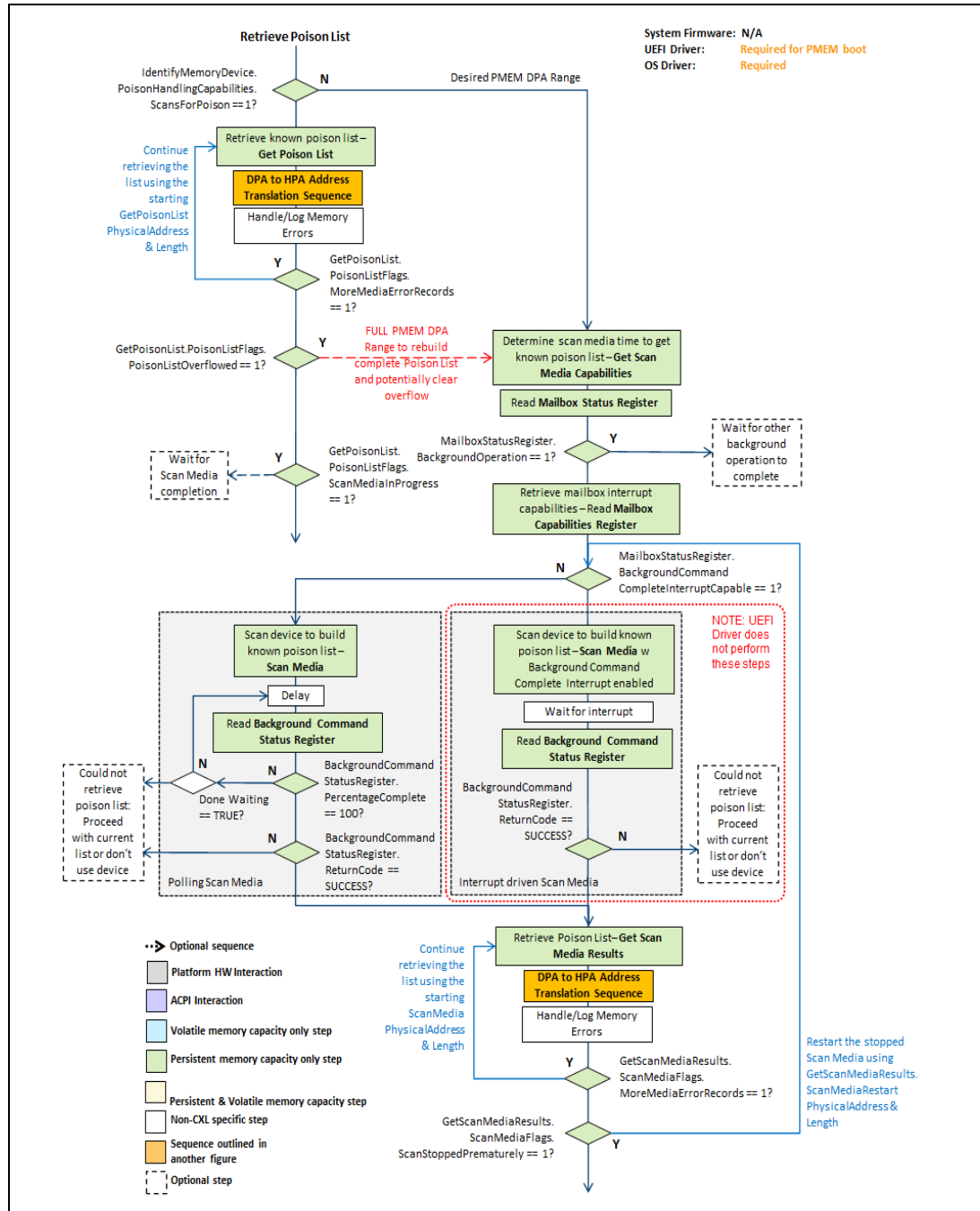
**Figure 64 - High-level sequence: Handle event records**



## 2.13.19 Retrieve poison list sequence

The following figure outlines the basic high-level sequence the UEFI and OS drivers will execute to retrieve the list of poisoned address locations the CXL end point device is tracking.

**Figure 65 - High-level sequence: Retrieve poison list**

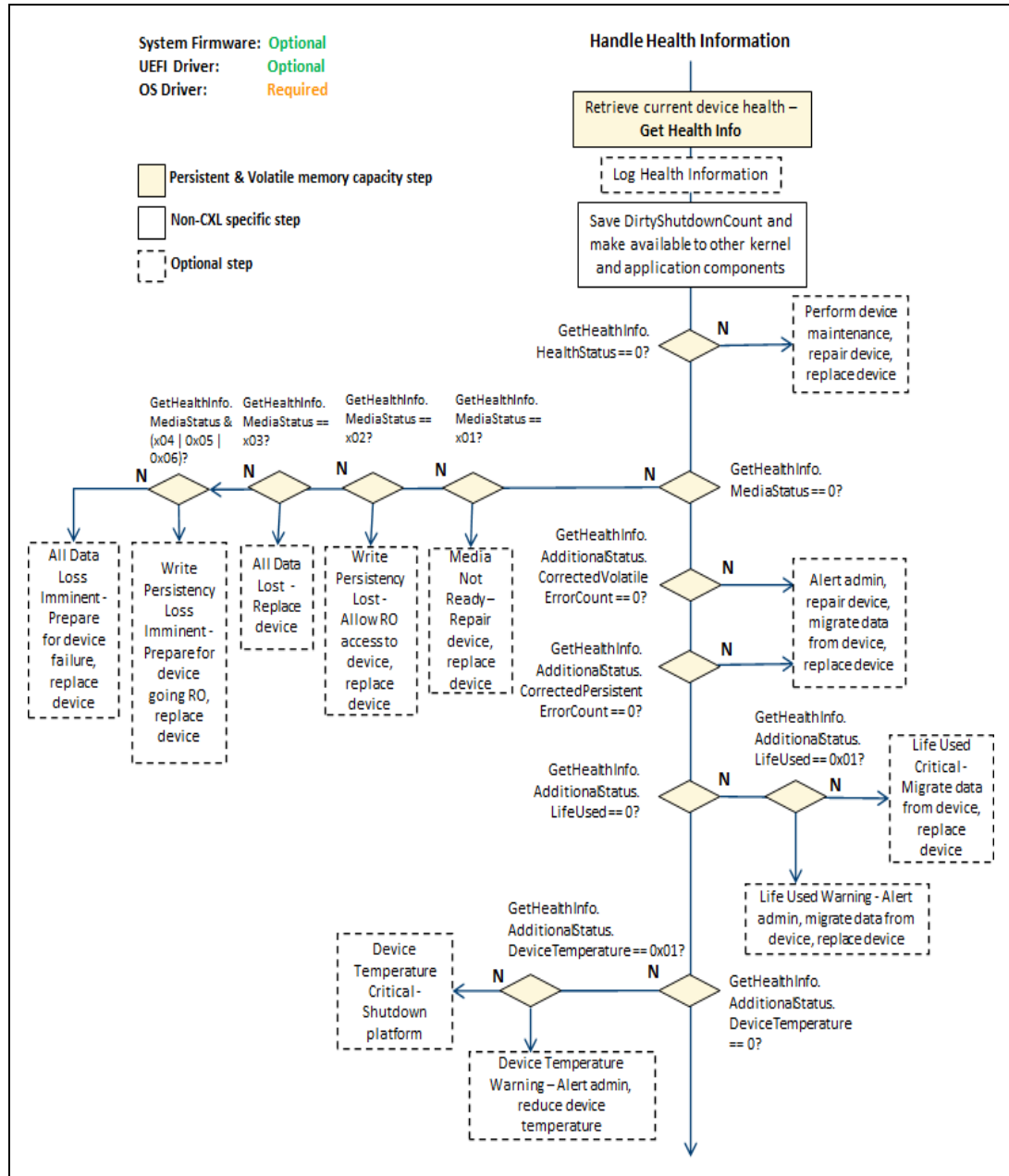




## 2.13.20 Handle health information sequence

The following figure outlines the basic high-level sequence the System Firmware, UEFI and OS drivers will execute to handle the memory device's initial health information.

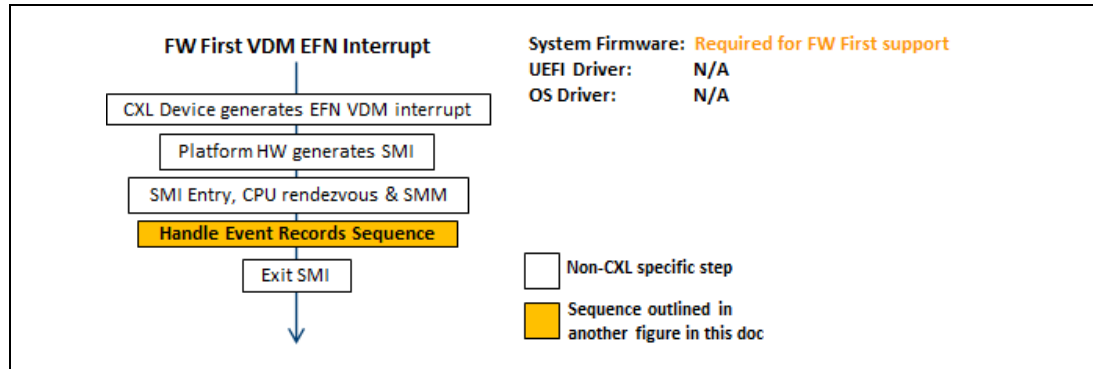
**Figure 66 - High-level sequence: Handle health information**



## 2.13.21 FW first event interrupt sequence

The following figure outlines the basic high-level FW first asynchronous interrupt sequence.

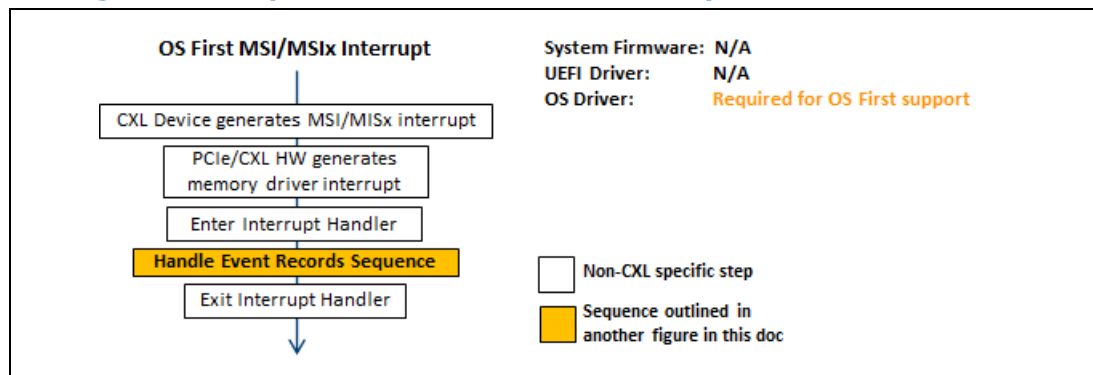
**Figure 67 - High-level sequence: FW first event interrupt**



## 2.13.22 OS first event interrupt sequence

The following figure outlines the basic high-level OS first asynchronous interrupt sequence.

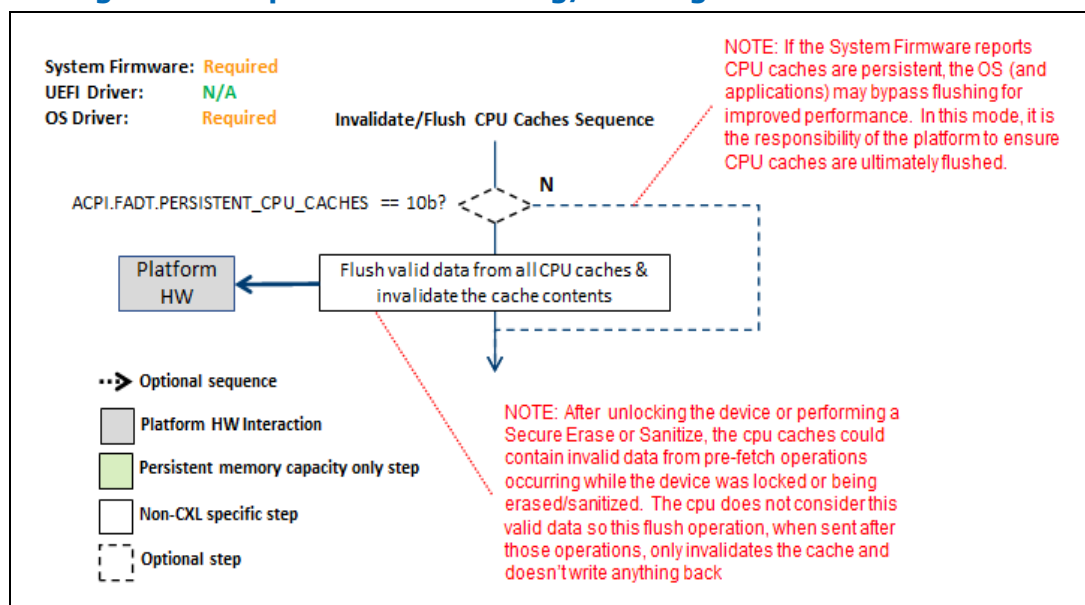
**Figure 68 - High-level sequence: OS first event interrupt**



## 2.13.23 Invalidating/Flushing CPU caches sequence

The following figure outlines the basic high-level flow that the System Firmware and OS drivers will need to execute to invalidate or flush CPU caches to persistent memory.

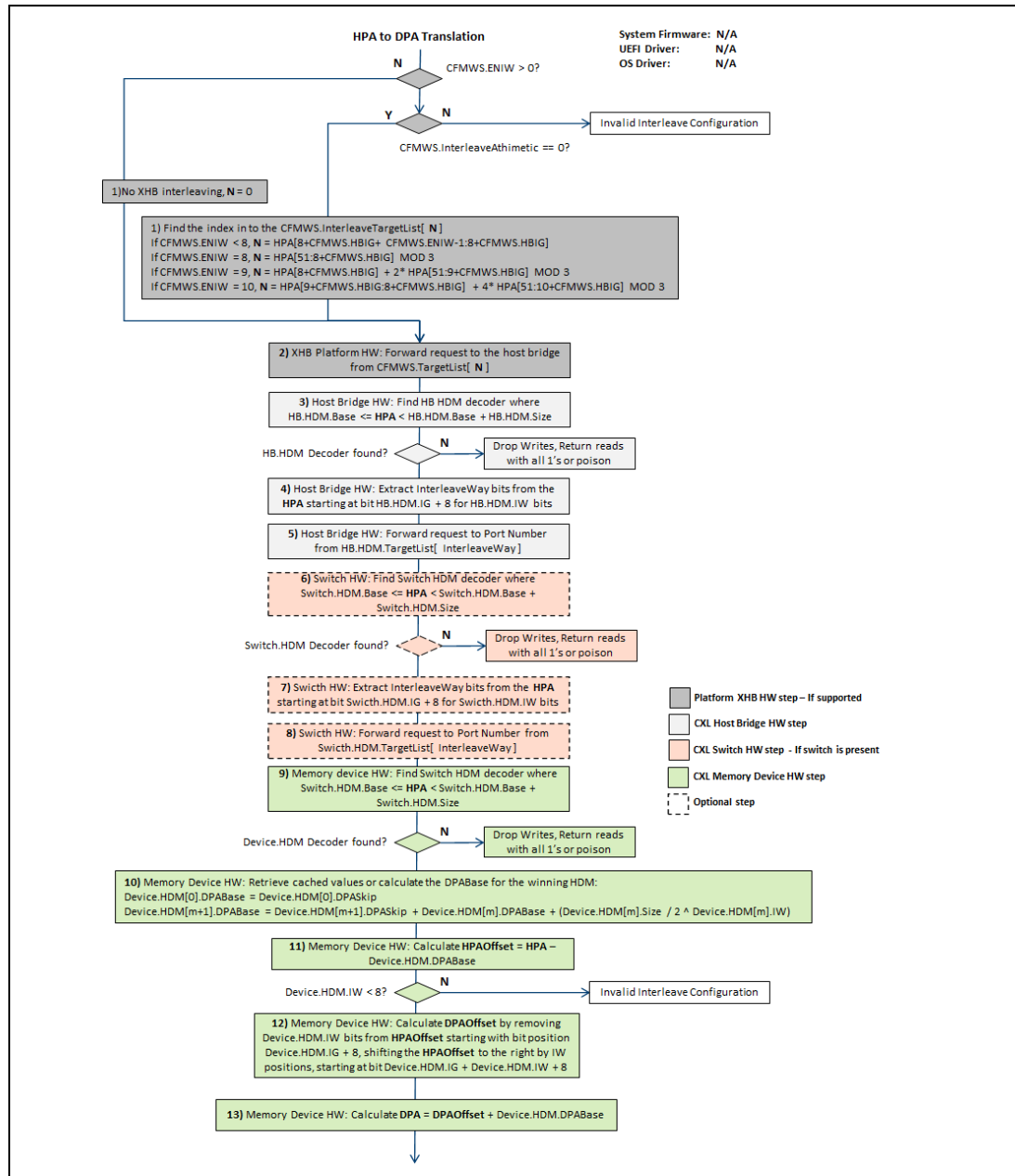
**Figure 69 - High-level sequence: Invalidating/flushing CPU caches**



## 2.13.24 HPA to DPA translation sequence

The following figure outlines the Host Physical Address to Device Physical Address translation that occurs in the Platform XHB HW and HDM decoder HW for the CXL Host Bridge, CXL Switch, and CXL Memory Device. This is the logic outlined in the CXL specification.

**Figure 70 - High-level sequence: HPA to DPA translation**



Here are some example HPA to DPA translations referencing the steps in the previous flow.

HPA to DPA Translation	Example 1	Example 2
Configuration description	See <b>Memory Region Example – Region interleaved across 2 CXL switches</b> for configuration -1 HB -2 Switch -4 devices on each HB -x8 pmem interleave set	See <b>Memory Region Example – Region interleaved across 2 HBs</b> for configuration -x2 XHB interleave -4 devices on each HB -x8 pmem interleave set
Platform HW XHB configuration	N/A	CFMWS.Base 0 CFMWS.WindowsSize 1GB CFMWS.ENIW 1 (x2) CFMWS.HBIG 2 (1K) Target List: HB A, HB B
HB HDM Configuration	Base 0 Size 1GB DPA Skip 0 IG 2 (1K) IW 1 (x2) TargetList: Port0, Port1	Base 0 Size 1GB DPA Skip 0 IG 3 (2K) IW 2 (x4) TargetList: Port0, Port1, Port2, Port3
Switch HDM configuration	Base 0 Size 1GB  DPA Skip 0 IG 3 (2K) IW 2 (x4) TargetList: Port0, Port1, Port 2, Port3	N/A
Device HDM configuration	Base 0 Size 1GB DPA Skip 0 IG 2 (1K) IW 3 (x8)	Base 0 Size 1GB DPA Skip 0 IG 2 (1K) IW 3 (x8)
<b>Example HPA - Host Physical Address</b>	<b>HPA 0x3400</b>	<b>HPA 0x3400</b>
Step 1: XHB Platform HW extract InterleaveWays	N/A	Platform HW: Extract 1 bit (CFMWS.ENIW) from HPA starting at bit 10 (CFMWS.HBIG + 8) = 1b
Step 2: XHB Platform HW forward request to host bridge	N/A	Platform HW: Forward request to CFMWS.TargetList[ 1 ] = HB B
Step 3: HB find decoder	HB A: Decoder 0	HB B: Decoder 0
Step 4: HB extract InterleaveWays	HB A: Extract 1 bit (IW) from HPA starting at bit 10 (IG+8) = 1b	HB B: Extract 2 bit (IW) from HPA starting at bit 11 (IG+8) = 10b
Step 5: HB forward request to port	HB A: Forward request to TargetList[ 1 ] = Port1 (Switch C)	HB B: Forward request to TargetList[ 2 ] = Port2 (Device 6)
Step 6: Switch find decoder	Switch C: Decoder 0	N/A

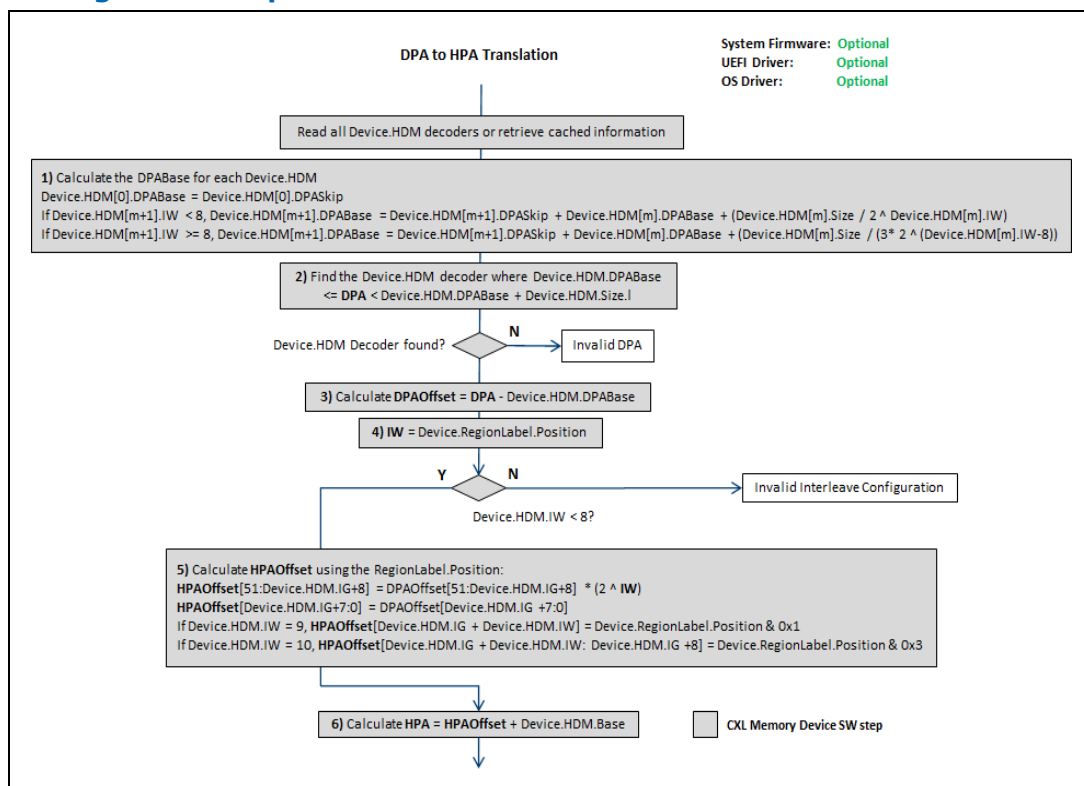
HPA to DPA Translation	Example 1	Example 2
Step 7: Switch extract InterleaveWays	Switch C: Extract 2 bits (IW) from HPA starting at bit 11 (IG+8) = 10b	N/A
Step 8: Switch forward request to port	Switch C: Forward request to TargetList[ 2 ] = Port2 (Device 6)	N/A
Step 9: Device find decoder	Device 6: Decoder 0	Device 6: Decoder 0
Step 10: Device calculate DBA Base	Device 6: Decoder0.DPABase = 0	Device 6: Decoder0.DPABase = 0
Step 11: Device calculate HPAOffset	Device 6: 0x1400	Device 6: 0x1400
Step 12: Device calculate DPAOffset	Device 6: Remove 3 bits (Device.IW) from HPAOffset starting at bit 10 (Device.IG + 8), shift upper address bits right 3 bits = 0x400	Device 6: Remove 3 bits (Device.IW) from HPAOffset starting at bit 10 (Device.IG + 8), shift upper address bits right 3 bits = 0x400
Step 13: Device calculate DPA	Device 6: 0x400	Device 6: 0x400
<b>Example final translated DPA</b>	<b>DPA 0x400 on Device 6</b>	<b>DPA 0x400 on Device 6</b>

## 2.13.25 DPA to HPA translation sequence

The following figure outlines the Device Physical Address to Host Physical Address translation that System Firmware (FW First), UEFI and OS drivers may need to implement if consuming General Media Event Records that report a DPA from the device.

Note that when the IW bits are added back into the address, software needs to insert the correct number for that device in the IW which is equivalent to that device's RegionLabel.Position.

**Figure 71 - High-level sequence: DPA to HPA translation**



Here are some example DPA to HPA translations referencing the steps in the previous flow.

<b>DPA to HPA Translation</b>	<b>Example 1</b>	<b>Example 2</b>
Configuration description	See <b>Memory Region Example – Region interleaved across 2 CXL switches</b> for configuration -1 HB -2 Switch -4 devices on each HB -x8 pmem interleave set	See <b>Memory Region Example – Region interleaved across 2 HBs</b> for configuration -x2 XHB interleave -4 devices on each HB -x8 pmem interleave set
Device HDM configuration	Base 0 Size 1GB DPA Skip 0 IG 2 (1K) IW 3 (x8)	Base 0 Size 1GB DPA Skip 0 IG 2 (1K) IW 3 (x8)
<b>Example DPA - Host Physical Address</b>	<b>DPA 0x400 on Device 6</b>	<b>DPA 0x400 on Device 6</b>
Step 1: Calculate DBA Base for each device HDM	Device 6: Decoder0.DPABase = 0	Device 6: Decoder0.DPABase = 0
Step 2: Find device decoder	Device 6: Decoder 0	Device 6: Decoder 0
Step 3: Calculate DPAOffset for decoder	Device 6: 0x400	Device 6: 0x400
Step 4: Use Device.RegionLabel.Position as the IW	IW = RegionLabel.Position = 5	IW = RegionLabel.Position = 5
Step 5: Calculate HPAOffset	Device 6: Insert 3 bits (Device.IW) into DPAOffset starting at bit 10 (Device.IG + 8), shift upper address bits left 3 bits = 0x3400	Device 6: Insert 3 bits (Device.IW) from HPAOffset starting at bit 10 (Device.IG + 8), shift upper address bits left 3 bits = 0x3400
<b>Step 6: Calculate HPA from HPAOffset</b>	<b>HPA 0x3400</b>	<b>HPA 0x3400</b>



## 2.13.26 GPF sequence

The following figure outlines the high-level sequence for GPF. The System Firmware, UEFI and OS Drivers, and the platform HW/FW all play apart in the GPF sequence.

**Figure 72 - High-level sequence: GPF**

