Compute Express Link™ (CXL™)

Specification

June 2019

Revision: 1.1

LEGAL NOTICE FOR THIS PUBLICLY-AVAILABLE SPECIFICATION FROM COMPUTE EXPRESS LINK CONSORTIUM, INC.

This CXL **Specification Revision 1.1** (this "**CXL Specification**" or this "**document**") is owned by and is proprietary to Compute Express Link Consortium, Inc., a Delaware nonprofit corporation (sometimes referred to as "**CXL**" or the "**CXL Consortium**" or the "**Company**") and/or its successors and assigns.

NOTICE TO USERS WHO ARE MEMBERS OF THE CXL CONSORTIUM:

If you are a Member of the CXL Consortium (sometimes referred to as a "**CXL Member**"), and even if you have received this publicly-available version of this CXL Specification after agreeing to CXL Consortium's Evaluation Copy Agreement (a copy of which is available https://www.computeexpresslink.org/download-the-specification, each such CXL Member must also be in compliance with all of the following CXL Consortium documents, policies and/or procedures (collectively, the "**CXL Governing Documents**") in order for such CXL Member's use and/or implementation of this CXL Specification to receive and enjoy all of the rights, benefits, privileges and protections of CXL Consortium membership: (i) CXL Consortium's Intellectual Property Policy; (ii) CXL Consortium's Bylaws; (iii) any and all other CXL Consortium policies and procedures; and (iv) the CXL Member's Participation Agreement.

NOTICE TO NON-MEMBERS OF THE CXL CONSORTIUM:

If you are **not** a CXL Member and have received this publicly-available version of this CXL Specification, your use of this document is subject to your compliance with, and is limited by, all of the terms and conditions of the CXL Consortium's Evaluation Copy Agreement (a copy of which is available at https://www.computeexpresslink.org/download-the-specification**).**

In addition to the restrictions set forth in the CXL Consortium's Evaluation Copy Agreement, any references or citations to this document must acknowledge the Compute Express Link Consortium, Inc.'s sole and exclusive copyright ownership of this CXL Specification. The proper copyright citation or reference is as follows: **"© 2019-2020 COMPUTE EXPRESS LINK CONSORTIUM, INC. ALL RIGHTS RESERVED."** When making any such citation or reference to this document you are not permitted to revise, alter, modify, make any derivatives of, or otherwise amend the referenced portion of this document in any way without the prior express written permission of the Compute Express Link Consortium, Inc.

Except for the limited rights explicitly given to a non-CXL Member pursuant to the explicit provisions of the CXL Consortium's Evaluation Copy Agreement which governs the publicly-available version of this CXL Specification, nothing contained in this CXL Specification shall be deemed as granting (either expressly or impliedly) to any party that is not a CXL Member: (ii) any kind of license to implement or use this CXL Specification or any portion or content described or contained therein, or any kind of license in or to any other intellectual property owned or controlled by the CXL Consortium, including without limitation any trademarks of the CXL Consortium.; or (ii) any benefits and/or rights as a CXL Member under any CXL Governing Documents.

LEGAL DISCLAIMERS FOR ALL PARTIES:

THIS DOCUMENT AND ALL SPECIFICATIONS AND/OR OTHER CONTENT PROVIDED HEREIN IS PROVIDED ON AN "**AS IS**" BASIS. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, COMPUTE EXPRESS LINK CONSORTIUM, INC. (ALONG WITH THE CONTRIBUTORS TO THIS DOCUMENT) HEREBY DISCLAIM ALL REPRESENTATIONS, WARRANTIES AND/OR COVENANTS, EITHER EXPRESS OR IMPLIED, STATUTORY OR AT COMMON LAW, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, VALIDITY, AND/OR NON-INFRINGEMENT.
In the event this CXL Specification makes any references (including without limitation any incorporation by reference) to another standard's setting organization's or any other party's ("**Third Party**") content or work, including without limitation any specifications or standards of any such Third Party ("**Third Party Specification**"), you are hereby notified that your use or implementation of any Third Party Specification: (i) is not governed by any of the CXL Governing Documents; (ii) may require your use of a Third Party's patents, copyrights or other intellectual property rights, which in turn may require you to independently obtain a license or other consent from that Third Party in order to have full rights to implement or use that Third Party Specification; and/or (iii) may be governed by the intellectual property policy or other policies or procedures of the Third Party which owns the Third Party Specification. Any trademarks or service marks of any Third Party which may be referenced in this CXL Specification is owned by the respective owner of such marks.

NOTICE TO ALL PARTIES REGARDING THE PCI-SIG UNIQUE VALUE PROVIDED IN THIS CXL SPECIFICATION:

NOTICE TO USERS: THE UNIQUE VALUE THAT IS PROVIDED IN THIS CXL SPECIFICATION IS FOR USE IN VENDOR DEFINED MESSAGE FIELDS, DESIGNATED VENDOR SPECIFIC EXTENDED CAPABILITIES, AND ALTERNATE PROTOCOL NEGOTIATION ONLY AND MAY NOT BE USED IN ANY OTHER MANNER, AND A USER OF THE UNIQUE VALUE MAY NOT USE THE UNIQUE VALUE IN A MANNER THAT (A) ALTERS, MODIFIES, HARMS OR DAMAGES THE TECHNICAL FUNCTIONING, SAFETY OR SECURITY OF THE PCI-SIG ECOSYSTEM OR ANY PORTION THEREOF, OR (B) COULD OR WOULD REASONABLY BE DETERMINED TO ALTER, MODIFY, HARM OR DAMAGE THE TECHNICAL FUNCTIONING, SAFETY OR SECURITY OF THE PCI-SIG ECOSYSTEM OR ANY PORTION THEREOF (FOR PURPOSES OF THIS NOTICE, "**PCI-SIG ECOSYSTEM**" MEANS THE PCI-SIG SPECIFICATIONS, MEMBERS OF PCI-SIG AND THEIR ASSOCIATED PRODUCTS AND SERVICES THAT INCORPORATE ALL OR A PORTION OF A PCI-SIG SPECIFICATION AND EXTENDS TO THOSE PRODUCTS AND SERVICES INTERFACING WITH PCI-SIG MEMBER PRODUCTS AND SERVICES).

Contents

Figures

Tables

Revision History

Revision	Description	Date
1.0	Initial release.	March, 2019
1.1	Added Reserved and ALMP terminology definition to Terminology/Acronyms table and also alphabetized the entries. Completed update to CXL terminology (mostly figures); removed disclaimer re: old terminology. General typo fixes. Added missing figure caption in Transaction Layer chapter. Modified description of Deferrable Writes in Section 3.1.6 to be less restrictive. Added clarification in Section 3.2.5.13 that ordering between CXL.io traffic and CXL.cache traffic must be enforced by the device (e.g., between MSIs and D2H memory writes). Removed ExtCmp reference in ItoMWr & MemWr. Flit organization clarification: updated Figure 41 and added example with Figure 43. Fixed typo in Packing Rules MDH section with respect to H4. Clarified that Advanced Error Reporting (AER) is required for CXL. Clarification on data interleave rules for CXL.mem in Section 3.3.6. Updated Table 48, "ARB/MUX State Transition Table" on page 119 to add missing transitions and to correct transition conditions. Updated Section 5.1.1 to clarify rules for ALMP state change handshakes and to add rule around unexpected ALMPs. Updated Section 5.2.1 to clarify that ALMPs must be disabled when multiple protocols are not enabled. Updates to ARB/MUX flow diagrams. Fixed typos in the Physical Layer interleave example figures (LCRC at the end of the TLPs instead of IDLEs). Updated Table 52 to clarify protocol ID error detection and handling. Added Section 6.6 to clarify behavior out of recovery. Increased the HDM size granularity from 1MB to 256MB (defined in the Flex Bus Device DVSEC in Control and Status Registers chapter). Updated Viral Status in the Flex Bus Device DVSEC to RWS (from RW). Corrected the RCRB BAR definition so fields are RW instead of RWO. Corrected typo in Flex Bus Port DVSEC size value. Added entry to Table 63 to clarify that upper 7K of the 64K MEMBAR0 region is reserved. Corrected Table 65 so the PME-Turn_Off/Ack handshake is used consistently as a warning for both PCIe and CXL mode. Update Section 9.2.3 and Section 9.2.4 to remove references to EA and L2. Updated Section 11.2.2 to clarify device handling of non-function errors. Added additional latency recommendations to cover CXL.mem flows to Section 13.0; also changed wording to clarify that the latency guidelines are recommendations and not requirements. Added compliance test chapter.	June, 2019

§ §

1.0 Introduction

1.1 Audience

The information in this document is intended for anyone designing or architecting any hardware or software associated with Compute Express Link (CXL) or Flex Bus.

1.2 Terminology / Acronyms

Please refer to the PCI Express Specification for additional terminology and acronym definitions beyond those listed in Table 1.

Table 1. Terminology / Acronyms

Term / Acronym	Definition
Accelerator	Devices that may be used by software running on Host processors to offload or perform any type of compute or I/O task. Examples of accelerators include programmable agents (such as GPU/GPCPU), fixed-function agents, or reconfigurable agents such as FPGAs.
AiA	Accelerator Interfacing Architecture
AIC	Add In Card
ALMP	ARB/MUX Link Management Packet
CXL	Compute Express Link, a low-latency, high-bandwidth discrete or on-package link that supports dynamic protocol muxing of coherency, memory access, and IO protocols, thus enabling an accelerator to access system memory as a caching agent and/or Host system memory.
CXL.io	PCIe-based non coherent I/O protocol with enhancements for accelerator support.
CXL.mem	Memory access protocol that supports device-attached memory.
CXL.cache	Agent coherency protocol that supports device caching of Host memory.
DCOH	This is the agent on the device that is responsible for resolving coherency with respect to device caches and managing Bias states
DP	Downstream Port
Flex Bus	A flexible high-speed port that is statically configured to support either PCI Express or Compute Express Link.
Flex Bus.CXL	CXL protocol over a Flex Bus interconnect.
HBM	High Bandwidth Memory
Home Agent	This is the agent on the Host that is responsible for resolving system wide coherency for a given address
HDM	Host-managed Device Memory. Device-attached memory mapped to system coherent address space and accessible to Host using standard write-back semantics. Memory located on a CXL device can either be mapped as HDM or PDM.
MC	Memory Controller
MCP	Multi-chip Protocol, an on-package connection typically used between a CPU die and a companion die.
Smart I/O	Enhanced I/O with additional protocol support.
PCIe RCiEP	PCIe Root Complex Integrated Endpoint.

Table 1. Terminology / Acronyms

Term / Acronym	Definition
PDM	Private Device memory. Device-attached memory not mapped to system address space or directly accessible to Host as cacheable memory. Memory located on PCIe devices is of this type. Memory located on a CXL device can either be mapped as PDM or HDM.
RCEC	Root Complex Event Collector, collects errors from PCIe RCiEPs.
Reserved	The contents, states, or information are not defined at this time. Reserved register fields must be read only and must return 0 (all 0's for multi-bit fields) when read. Reserved encodings for register and packet fields must not be used. Any implementation dependent on a Reserved field value or encoding will result in an implementation that is not CXL-spec compliant. The functionality of such an implementation cannot be guaranteed in this or any future revision of this specification.
SVM	Shared Virtual Memory
SF	Snoop Filter
UP	Upstream Port
VMM	Virtual Machine Manager

1.3 Reference Documents

Table 2. Reference Documents

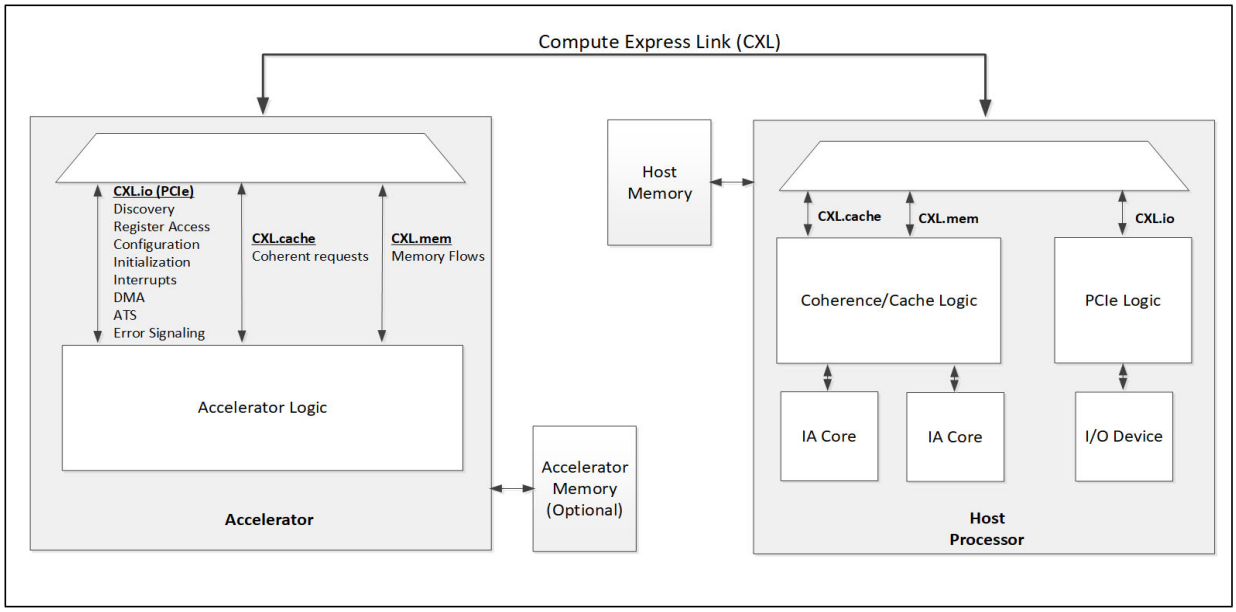
Document	Chapter Reference	Document No./Location
PCI Express Base Specification Revision 5.0	N/A	www.pcisig.com

1.4 Motivation and Overview

1.4.1 Compute Express Link

CXL is a dynamic multi-protocol technology designed to support a vast spectrum of accelerators. CXL provides a rich set of protocols that include I/O semantics similar to PCIe (i.e., CXL.io), caching protocol semantics (i.e., CXL.cache), and memory access semantics (i.e., CXL.mem) over a discrete or on-package link. Depending on the particular accelerator usage model, all of the protocols or only a subset of the protocols may be enabled; however, CXL.io is always required for discovery and enumeration, error reporting, and host physical address (HPA) lookup. A key benefit of CXL is that it provides a low-latency, high-bandwidth path for an accelerator to access the system. The figure below is a conceptual diagram showing a device attached to a Host processor via CXL. Note that the CXL link is shown as a direct-attached CPU link and cannot reside behind a PCIe switch (although this does not preclude the concept of a potential CXL switch in the future).

Figure 1. Conceptual Diagram of Accelerator Attached to Processor via CXL



1.4.2 Flex Bus

A Flex Bus port allows designs to choose between providing native PCIe protocol or CXL over a high-bandwidth, off-package link; the selection happens during boot time via auto negotiation and depends on the device that is plugged into the slot. Flex Bus uses PCIe electricals, making it compatible with PCIe retimers, and adheres to standard PCIe form factors for an add-in card.

Figure 2 provides a high-level diagram of a Flex Bus port implementation, illustrating both a slot implementation and a custom implementation where the device is soldered down on the motherboard. The slot implementation can accommodate either a Flex Bus.CXL card or a PCIe card. One or two optional retimers can be inserted between the CPU and the device to extend the distance. As illustrated in Figure 3, this flexible innovation port can be used to attach coherent accelerators or smart I/O to a Host processor.

Figure 2. CPU Flex Bus Port Example

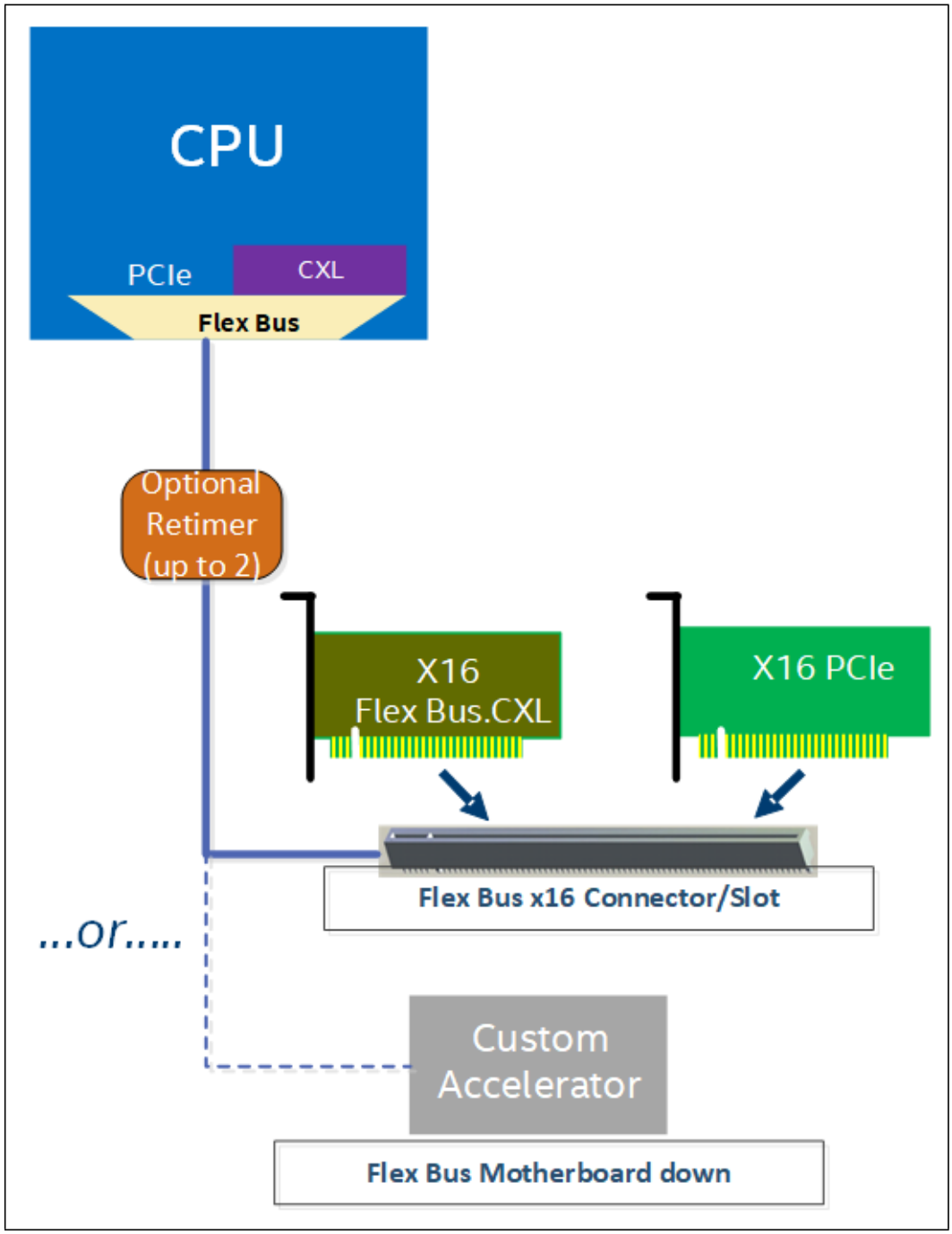


Figure 3. Flex Bus Usage Model Examples

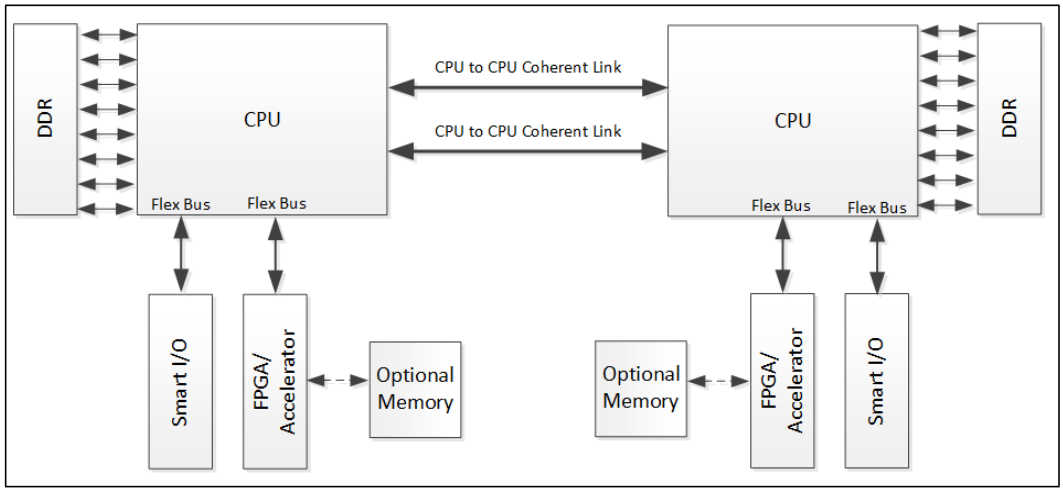
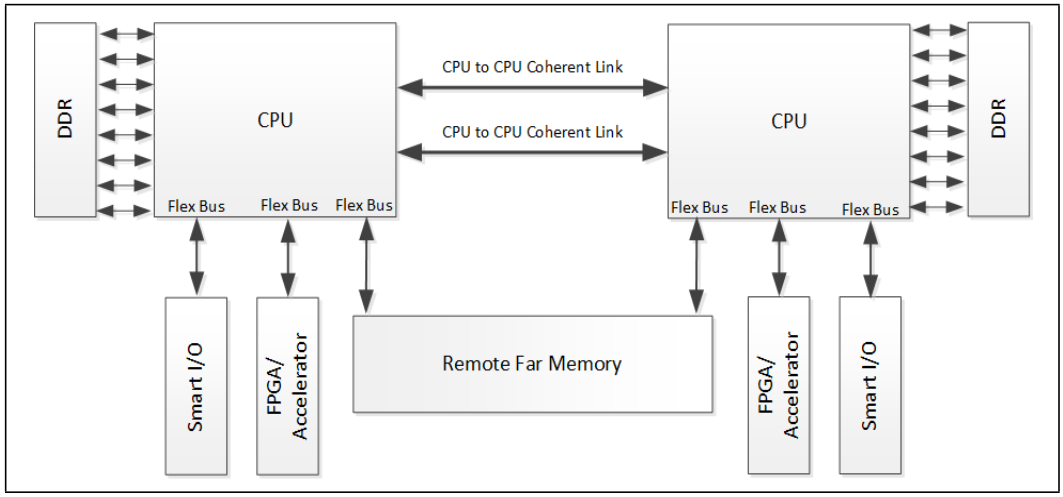


Figure 4 illustrates how a Flex Bus.CXL port can be used as a memory expansion port.

Figure 4. Remote Far Memory Usage Model Example



1.5 Flex Bus Link Features

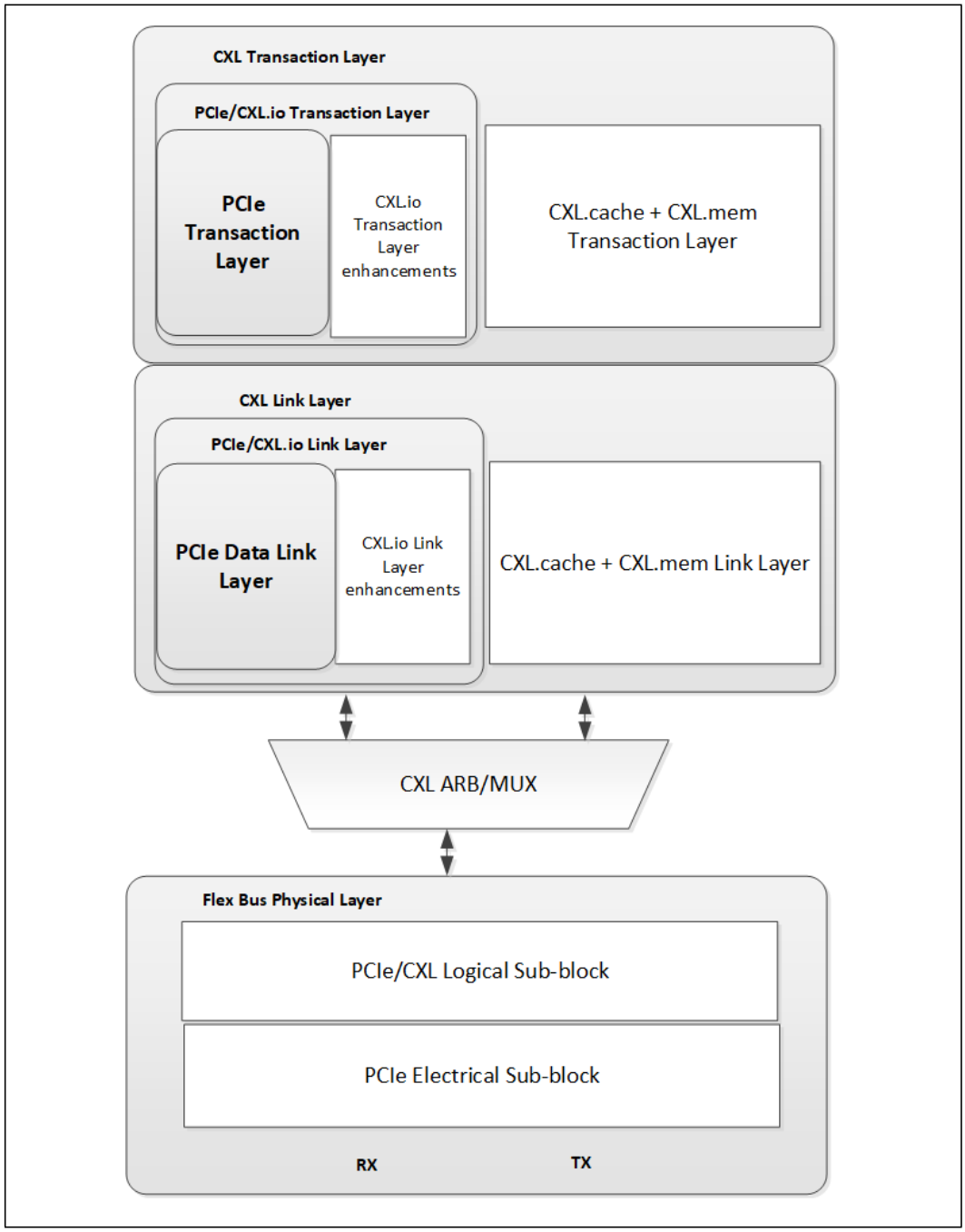
Flex Bus provides a point-to-point interconnect that can transmit native PCIe protocol or dynamic multi-protocol CXL to provide I/O, coherency, and memory protocol over PCIe electricals. The primary link attributes include support of the following features:

- Native PCIe mode, full feature support as defined in the PCIe specification
- CXL mode, as defined in this specification
- Static configuration of PCIe vs CXL protocol mode
- Signaling rate of 8 GT/s, 16 GT/s or 32 GT/s for CXL mode
- Link width support for x16, x8, x4, x2, and x1 (degraded mode) in CXL mode
- Bifurcation (aka Link Subdivision) support to x4 in CXL mode

1.6 Flex Bus Layering Overview

Flex Bus architecture is organized as multiple layers, as illustrated in Figure 5. The CXL transaction (protocol) layer is subdivided into logic that handles CXL.io and logic that handles CXL.mem and CXL.cache; the CXL link layer is subdivided in the same manner. Note that the CXL.mem and CXL.cache logic are combined within the transaction layer and within the link layer. The CXL link layer interfaces with the CXL ARB/MUX, which interleaves the traffic from the two logic streams. Additionally, the PCIe transaction and data link layers are optionally implemented and, if implemented, are converged with the CXL.io transaction and link layers, respectively. As a result of the link training process, the transaction and link layers are configured to operate in either PCIe mode or CXL mode. While a host CPU would most likely implement both modes, an accelerator AIC may choose to implement only the CXL mode. The logical sub-block of the Flex Bus physical layer is a converged logical physical layer that can operate in either PCIe mode or CXL mode, depending on the results of alternate mode negotiation during the link training process.

Figure 5. Conceptual Diagram of Flex Bus Layering



1.7 Document Scope

This document specifies the functional and operational details of the Flex Bus interconnect and the CXL protocol. It describes the CXL usage model and defines how the transaction, link, and physical layers operate. Reset, power management, and initialization/configuration flows are described. Additionally, RAS behavior is described. Please refer to the PCIe specification for PCIe protocol details.

The contents of this document are summarized in the following chapter highlights:

- Section 2.0, “Compute Express Link System Architecture” on page 23 – This chapter describes different profiles of devices that might attach to a CPU root complex over a CXL capable link. For each device profile, a description of the typical workload and system resource usage is provided along with an explanation of which CXL capabilities are relevant for that workload. Additionally, a Bias Based coherency model is introduced which optimizes the performance for accesses to device-attached memory depending on whether the memory is in host bias, during which the memory is expected to be accessed mainly by the Host, or device bias, during which the memory is expected to be accessed mainly by the device.
- Section 3.0, “Compute Express Link Transaction Layer” on page 29 – The transaction layer chapter is divided into subsections that describe details for CXL.io, CXL.cache, and CXL.mem. The CXL.io protocol is required for all implementations, while the other two protocols are optional depending on expected device usage and workload. The transaction layer specifies the transaction types, transaction layer packet formatting, transaction ordering rules, and crediting. The CXL.io protocol is based on the “Transaction Layer Specification” chapter of the PCIe base specification; any deltas from the PCIe base specification are described in this chapter. These deltas include PCIe Vendor_Defined Messages for reset and power management, modifications to the PCIe ATS request and completion formats to support accelerators, and Deferred Writes instruction definitions. For CXL.cache, this chapter describes the channels in each direction (i.e., request, response, and data), the transaction opcodes that flow through each channel, and the channel crediting and ordering rules. The transaction fields associated with each channel are also described. For CXL.mem, this chapter defines the message classes in each direction, the fields associated with each message class, and the message class ordering rules. Finally, this chapter provides flow diagrams that illustrate the sequence of transactions involved in completing host-initiated and device-initiated accesses to device-attached memory.
- Section 4.0, “Compute Express Link Link Layers” on page 78 – The link layer is responsible for reliable transmission of the transaction layer packets across the Flex Bus link. This chapter is divided into subsections that describe details for CXL.io and for CXL.cache and CXL.mem. The CXL.io protocol is based on the “Data Link Layer Specification” chapter of the PCIe base specification; any deltas from the PCIe base specification are described in this chapter. For CXL.cache and CXL.mem, the 528-bit flit layout is specified. The flit packing rules for selecting transactions from internal queues to fill the three slots in the flit are described. Other features described for CXL.cache and CXL.mem include the retry mechanism, link layer control flits, CRC calculation, and viral and poison.
- Section 5.0, “Compute Express Link ARB/MUX” on page 117 – The ARB/MUX arbitrates between requests from the CXL link layers and multiplexes the data to forward to the physical layer. On the receive side, the ARB/MUX decodes the flit to determine the target to forward transactions to the appropriate CXL link layer. Additionally, the ARB/MUX maintains virtual link state machines for every link layer it interfaces with, processing power state transition requests from the local link layers and generating ARB/MUX link management packets to communicate with the remote ARB/MUX.
- Section 6.0, “Flex Bus Physical Layer” on page 129 – The Flex Bus physical layer is responsible for training the link to bring it to operational state for transmission of

PCIe packets or CXL flits. During operational state, it prepares the data from the CXL link layers or the PCIe link layer for transmission across the Flex Bus link; likewise, it converts data received from the link to the appropriate format to pass on to the appropriate link layer. This chapter describes the deltas from the PCIe base specification to support the CXL mode of operation. The framing of the CXL flits and the physical layer packet layout are described. The mode selection process to decide between CXL mode or PCIe mode, including hardware autonomous negotiation and software controlled selection is also described. Finally, CXL low latency modes are described.

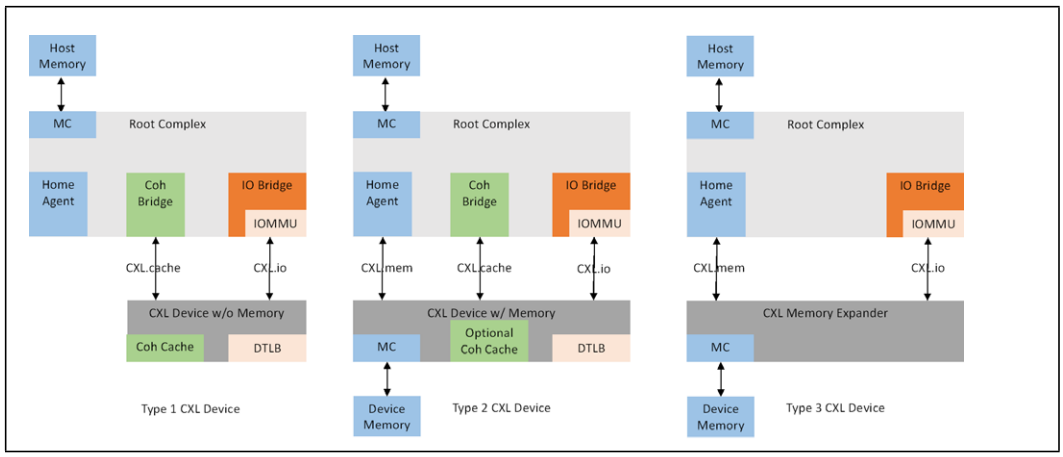
- Section 7.0, "Control and Status Registers" on page 145 – This chapter provides details of the Flex Bus and CXL control and status registers. It describes the various address spaces in which the registers are located. In the memory space, this chapter describes how the upstream and downstream port root complex register block (RCRB) regions are organized and how the upstream and downstream port MEMBAR0 regions are organized. It also differentiates between registers required to be implemented in a Flex Bus Host versus the registers required to be implemented in a CXL device.
- Section 8.0, "Reset, Initialization, Configuration and Manageability" on page 168 – This chapter describes the flows for boot, warm reset entry, cold reset entry, and sleep state entry; this includes the transactions sent across the link to initiate and acknowledge entry as well as steps taken by a CXL device to prepare for entry into each of these states. Additionally, this chapter describes the software enumeration model and how the BIOS view of the hierarchy differs from the OS view due to the fact that the CXL link is not exposed to the OS. This chapter discusses different accelerator topologies, i.e., single CPU, multiple CPUs, and multiple nodes; for each topology, software management of the multiple Flex Bus links involved is described.
- Section 9.0, "Power Management" on page 181 – This chapter provides details on protocol specific link power management and physical layer power management. It describes the overall power management flow in three phases: protocol specific PM entry negotiation, PM entry negotiation for ARB/MUX interfaces (managed independently per protocol), and PM entry process for the physical layer. The PM entry process for CXL.cache and CXL.mem is slightly different than the process for CXL.io; these processes are described in separate subsections in this chapter.
- Section 10.0, "Security" on page 193 – This chapter is a placeholder for non-product specific security requirements; currently there are no such requirements.
- Section 11.0, "Reliability, Availability and Serviceability" on page 194 – This chapter describes the RAS capabilities supported by a CXL host and a CXL device. It describes how various types of errors are logged and signaled to the appropriate hardware or software error handling agent. It describes the link down flow and the viral handling expectation. Finally, it describes the error injection requirements.
- Section 12.0, "Platform Architecture" on page 200 – This chapter provides details on the Flex Bus connector, platform topologies, AIC form factors, and AIC power envelope. It also discusses an out-of-band protocol detection mechanism.
- Section 13.0, "Performance Considerations" on page 203 – This chapter describes hardware and software considerations for optimizing performance across the Flex Bus link in CXL mode.
- Section 14.0, "CXL Compliance Testing" on page 204 – This chapter describes methodologies for ensuring that a device is compliant with the CXL specification.

§ §

2.0 Compute Express Link System Architecture

This section describes the performance advantages and key features of CXL. CXL is a high performance I/O bus architecture used to interconnect peripheral devices that can be either traditional non-coherent IO devices or accelerators with additional capabilities. The types of devices that can attach and the overall system architecture is described in the figure below.

Figure 6. CXL Device Types



Before we dive into the details of each type of CXL device, here's a foreword about where CXL is not applicable.

Traditional non-coherent IO devices rely primarily on standard Producer-Consumer ordering models and execute against Host-attached memory. For such devices, there's little interaction with the Host except for work submission and signaling on work completion boundaries. Such accelerators also tend to work on data streams or large contiguous data objects. These devices typically do not need the advanced capabilities provided by CXL and traditional PCIe is sufficient as an accelerator attach medium. The following sections describe various profiles of CXL devices.

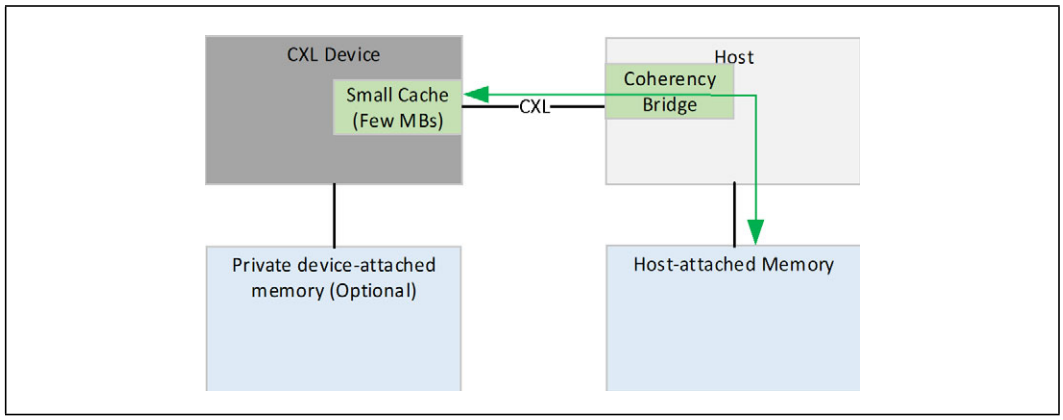
2.1 Type 1 CXL Device

Type 1 CXL devices have special needs for which having a fully coherent cache in the device becomes valuable. For such devices, standard Producer-Consumer ordering models do not work very well. One example of a device with a special need is to perform complex atomics that are not part of the standard suite of atomic operations present on PCIe.

Basic cache coherency allows an accelerator to implement any ordering model it chooses and allows it to implement an unlimited number of atomic operations. These tend to require only small amounts of cache which can easily be tracked by standard

processor snoop filter mechanisms. The size of cache that can be supported for such devices depends on the host's snoop filtering capacity. CXL supports such devices using its optional CXL.cache link over which an accelerator can use CXL.cache protocol for cache coherency transactions.

Figure 7. Type 1 - Device with Cache

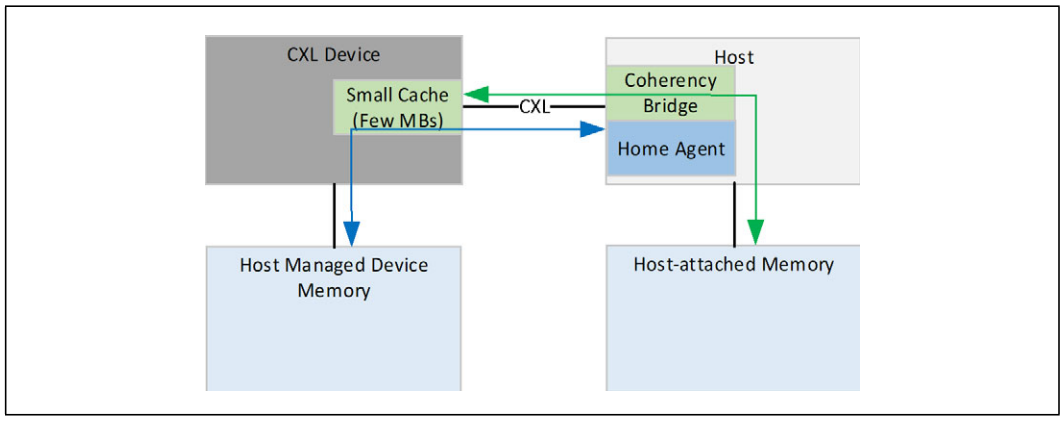


2.2 Type 2 Device

Type 2 devices are ones which have memory, for example DDR, High Bandwidth Memory (HBM) etc, attached to the device. These devices execute against memory but their performance comes from having massive bandwidth between the accelerator and device-attached memory. The key goal for CXL is to provide a means for the Host to push operands into device-attached memory and for the Host to pull results out of device-attached memory such that it doesn't add software and hardware cost that offsets the benefit of the accelerator. This spec refers to coherent system address mapped device-attached memory as Host-managed Device Memory (HDM).

There is an important distinction between HDM and traditional IO/PCIe Private Device Memory (PDM). An example of such a device is a GPGPU with attached GDDR. Such devices have treated device-attached memory as Private. This means that the memory is not accessible to the Host and is not coherent with the rest of the system. It is managed entirely by the device HW and driver and is used primarily as intermediate storage for the device with large datasets. The obvious disadvantage to a model such as this is that it involves large amounts of copies back and forth from the Host memory to device-attached memory as operands are brought in and results are written back. Please note that CXL does not preclude devices with PDM.

Figure 8. Type 2 Device - Device with Memory



At a high level, there are two models of operation that are envisaged for HDM. These are described below.

2.2.1 Bias Based Coherency Model

The Bias Based coherency model defines two states of bias for device-attached memory - Host Bias and Device Bias. When the device-attached memory is in Host Bias state, it appears to the device just as regular Host-attached memory does. That is, if the device needs to access it, it needs to send a request to the Host which will resolve coherency for the requested line. On the other hand, when the device-attached memory is in Device Bias state, the device is guaranteed that the Host does not have the line cached. As such, the device can access it without sending any transaction (request, snoops etc) to the Host whatsoever. It is important to note that the Host itself sees a uniform view of device-attached memory regardless of the bias state. In both modes, coherency is preserved for device-attached memory.

The key benefits of Bias Based coherency model are:

- Helps maintain coherency for device-attached memory which is mapped to system coherent address space.
- Helps the device access its local attached memory at high BW without incurring significant coherency overheads (e.g., snoops to the Host).
- Helps the Host access device-attached memory in a coherent, uniform manner, just as it would for Host-attached memory.

To maintain Bias modes, a Type 2 CXL Device will:

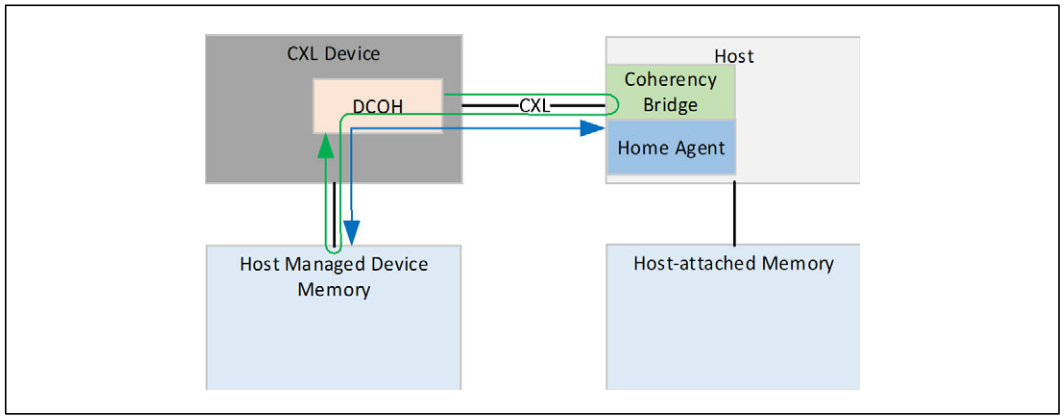
- Implement the Bias Table which tracks Bias on a page granularity (e.g., 1b per 4KB page) which can be cached in the device using a Bias Cache.
- Build support for Bias transitions using a Transition Agent (TA). This essentially looks like a DMA engine for “cleaning up” pages, which essentially means to flush the host’s caches for lines belonging to that page.
- Build support for basic load and store access to accelerator local memory for the benefit of the Host.

The bias modes are described in detail below.

2.2.1.1 Host Bias

The Host Bias mode typically refers to the part of the cycle when the operands are being written to memory by the Host during work submission or when results are being read out from the memory after work completion. During Host Bias mode, coherency flows allows for high throughput access from the Host to device-attached memory (as shown by the blue arrows in Figure 9) whereas device access to device-attached memory is not optimal since they need to go through the host (as shown in green arrows in Figure 9).

Figure 9. Type 2 Device - Host Bias

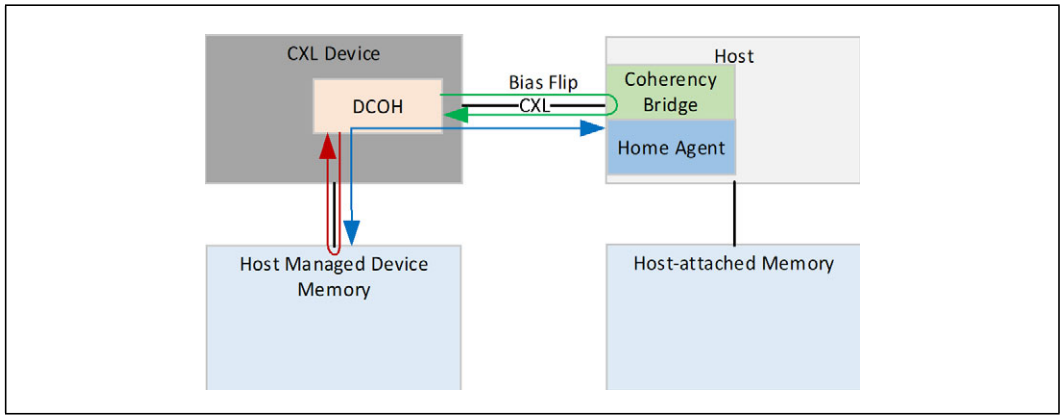


2.2.1.2 Device Bias

The Device Bias mode is used when the device is executing the work, between work submission and completion, and in this mode, the device needs high BW and low latency access to device-attached memory.

In this mode, device can access device-attached memory without consulting the Host's coherency engines (as shown in red arrows in Figure 10). The Host can still access device-attached memory but may be forced to give up ownership by the accelerator (as shown in green arrows in Figure 10). This results in the device seeing ideal latency & BW from device-attached memory, whereas the Host sees compromised performance.

Figure 10. Type 2 Device - Device Bias



2.2.1.3 Mode Management

There are two envisioned Bias Mode Management schemes – Software Assisted and HW Autonomous. CXL supports both modes. Examples of Bias Flows are present in Appendix A.

While two modes are described below, it is worth noting that strictly speaking, devices do not need to implement any bias. In this case, all of device-attached memory degenerates to Host Bias. This means that all accesses to device-attached memory must be routed through the Host. An accelerator is free to choose a custom mix of SW assisted and HW autonomous bias management scheme. The Host implementation is agnostic to any of the above choices.

2.2.1.4 Software Assisted Bias Mode Management

With Software Assistance, we rely on SW to know for a given page, which state of the work execution flow it resides in. This is useful for accelerators with phased computation with regular access patterns. Based on this, SW can best optimize the coherency performance on a page granularity by choosing Host or Device Bias modes appropriately.

Here are some characteristics of Software Assisted Bias Mode Management:

- Software Assistance can be used to have data ready at an accelerator before computation.
- If data is not moved to accelerator memory in advance, it is generally moved on demand based on some attempted reference to the data by the accelerator.
- In an "on demand" data fetch scenario, the accelerator must be able to find work to execute, for which data is already properly placed, or it must stall.
- Every cycle that an accelerator is stalled eats into its ability to add value over software running on a core.
- Large, complex, programmable accelerators, like GPUs are often able to find work to execute and hide data fetch latencies.
- Simple accelerators typically cannot hide data fetch latencies.

Efficient software assisted data/coherency management is critical to the aforementioned class of simple accelerators.

2.2.1.5 HW Autonomous Bias Mode Management

Software assisted coherency/data management is ideal for simple accelerators, but of lesser value to complex, programmable accelerators. At the same time, the complex problems frequently mapped to complex, programmable accelerators like GPUs present an enormously complex problem to programmers if software assisted coherency/data movement is a requirement. This is especially true for problems that split computation between Host and accelerator or problems with pointer based, tree based or sparse data sets.

With HW Autonomous Bias Mode Management, we do not rely on SW to appropriately manage page level coherency bias. Rather, it is the HW which makes predictions on the bias mode based on the requester for a given page and adapts accordingly. Key benefits for this model are:

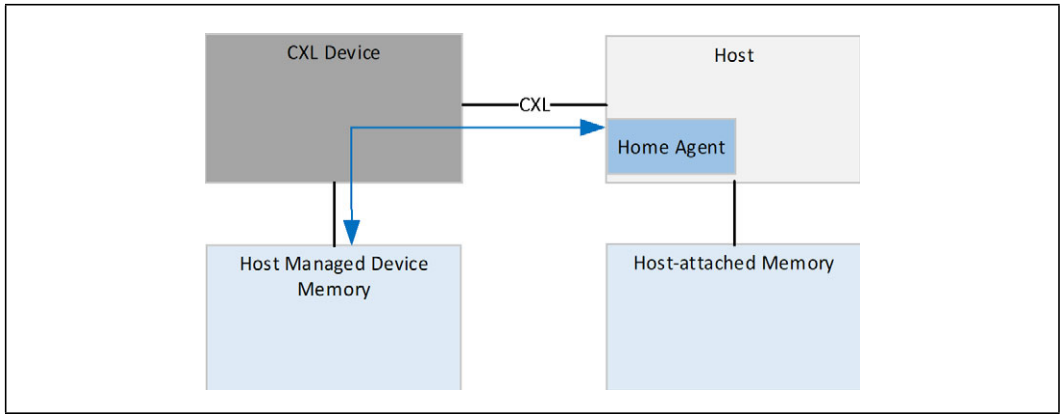
- Provide the same page granular coherency bias capability as in the software assisted model.
- Eliminate the need for SW to identify and schedule page bias transitions prior to offload execution.
- Provide hardware support for dynamic bias transition during offload execution.
- Hardware support for this model can be a simple extension to the software assisted model.
- Link flows and Host support is unaffected.
- Impact limited primarily to actions taken at the accelerator when a Host touches a Device Biased page and vice-versa.
- Note that even though this is an ostensible hardware driven solution, hardware need not perform all transitions autonomously – though it may do so if desired.

It is sufficient if hardware provide hints (e.g., "transition page X to bias Y now"), but leaves the actual transition operations under software control.

2.3 Type 3

A CXL Type 3 device is fundamentally different from other device Types in the sense that unlike other device types, it is not an active compute engine. Instead, a Type 3 device is primarily a memory expander for the Host as shown in the figure below.

Figure 11. Type 3 - Memory Expander



Since this is not an accelerator, the device does not make any requests over CXL.cache. The device operates primarily over CXL.mem to service requests sent from the Host. The CXL.io link is used device discovery, enumeration, error reporting and management. The CXL architecture is independent of memory technology and allows for a range of memory organization possibilities depending on support implemented in the Host.

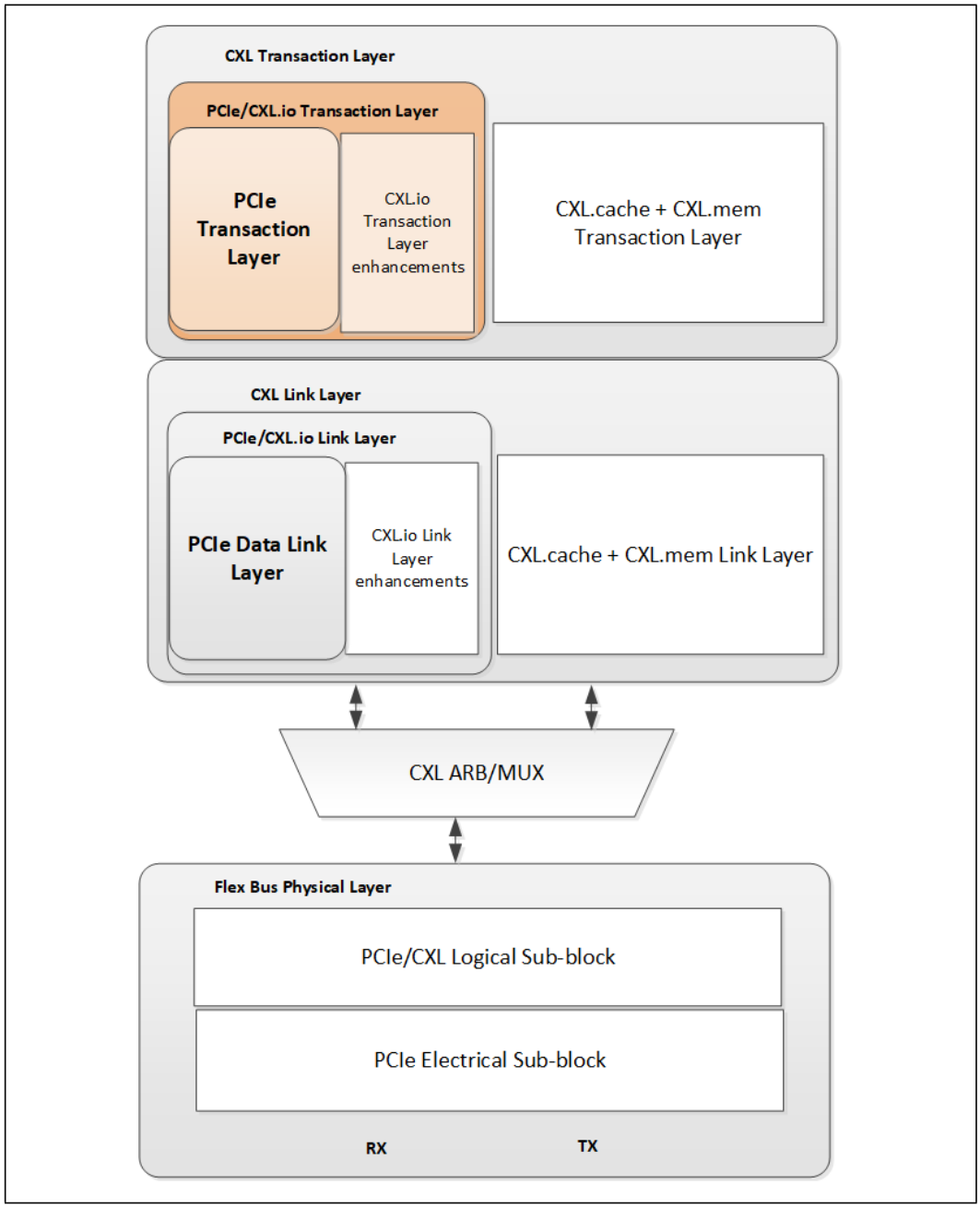
§ §

3.0 Compute Express Link Transaction Layer

3.1 CXL.io

CXL.io provides a non-coherent load/store interface for I/O devices. Figure 12 shows where the CXL.io transaction layer exists in the Flex Bus layered hierarchy. Transaction types, transaction packet formatting, credit-based flow control, virtual channel management, and transaction ordering rules follow the PCIe definition; please refer to the “Transaction Layer Specification” chapter of the PCI Express Base Specification for details. This chapter highlights notable PCIe operational modes or features that are used for CXL.io.

Figure 12. Flex Bus Layers -- CXL.io Transaction Layer Highlighted



3.1.1 PCIe Root Complex Integrated Endpoint

a CXL.io endpoint is exposed to software as a PCIe RCiEP. Please refer to the PCIe 5.0 Base Specification for more details.

3.1.2 CXL Power Management VDM Format

The CXL power management messages are sent as PCIe Vendor Defined Type0 messages with a 4DW data payload. These include the PMREQ, PMRESP, and PMGO messages. Figure 13 provides the format for the CXL PM VDM messages. The following are the characteristics of these messages:

- Fmt and Type fields are set to indicate message with data and routing of "Local-Terminate at Receiver"
- Message Code is set to Vendor Defined Type 0
- Vendor ID field is set to 8086h. (Note that this may change to include the CXL assigned vendor ID.)
- Byte 15 of the message header contains the VDM Code and is set to the value of "CXL PM Message." (68h)
- The 4DW Data Payload contains the CXL PM Logical Opcode (e.g., PMREQ, PMRESP, etc) and any other information related to the CXL PM message. Details of fields within the Data Payload are described in Table 3.

Figure 13. CXL Power Management Messages Packet Format

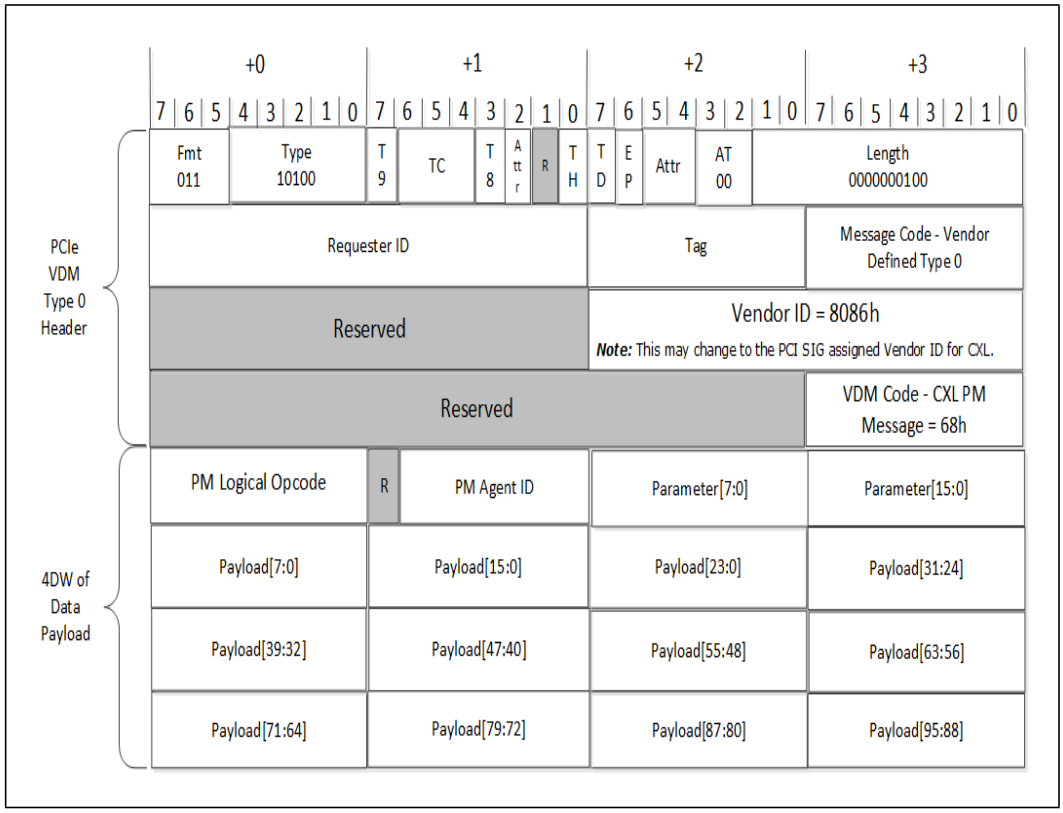


Table 3. CXL Power Management Messages -- Data Payload Fields Definitions

Field	Description	Notes
PM Logical Opcode[7:0]	Power Management Command: 00h - AGENT_INFO 02h - RESETPREP 04h - PMREQ (PMRESP and PMGO) FEh - CREDIT_RTN	
PM Agent ID[6:0]	Sender's ID: 1111111 - CXL device (Default)	A device does not consume this value when it receives a message from the Host. Host will send PM Agent ID for the CXL Device to use in the CREDIT_RTN msg.
Parameter[15:0]	CREDIT_RTN: Reserved AGENT_INFO: 0 - REQUEST (set) /RESPONSE_N (cleared) [7:1] - INDEX All others reserved PMREQ: 0 - REQUEST (set) /RESPONSE_N (cleared) 1 - EA 2 - GO All others reserved RESETPREP: 0 - REQUEST (set) /RESPONSE_N (cleared) All others reserved	

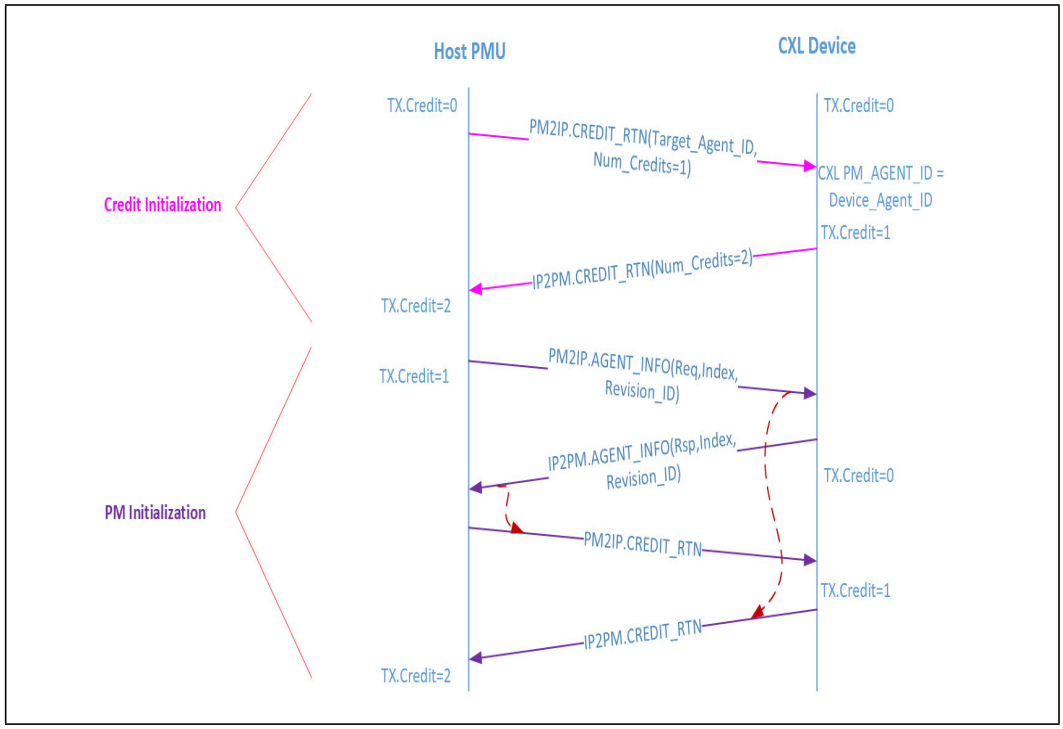
Table 3. CXL Power Management Messages -- Data Payload Fields Definitions

Field	Description	Notes
Payload[95:0]	CREDIT_RTN: 7:0 NUM_CREDITS 14:8 TARGET_AGENT_ID All others reserved AGENT_INFO: if Param.Index == 0, 7:0 - REVISION_ID all others reserved else all reserved RESETPREP: 7:0 - ResetType 0x01 => host space transition from S0 to S1; 0x03 => host space transition from S0 to S3; 0x04 => host space transition from S0 to S4; 0x05 => host space transition from S0 to S5; 0x10 => Host space Warm reset (host space partition reset without power down); 0x11 => Cold reset for host space (host space partition reset with powerdown); 0x21 => D3cold for Host space 15:8 - PrepType 0x00 => General Prep 0x01 => Early Prep; 0x02 => Reset Entry Start (first checkpoint for CXL device blocks during a Reset event/power state transition); 0x03 => Link Turnoff (typically the last checkpoint during a Reset event/power state transition); 17:16 - Phase 0x00 => Phase 0 0x01 => Phase 1 0x02 => Phase 2 0x03 => Phase 3 All others reserved PMREQ: 31:0 - PCIe LTR format All others reserved	CXL Agent must treat the TARGET_AGENT_ID field as Reserved when returning credits to Host. Only Index 0 is defined for AGENT_INFO, all other Index values are reserved.

3.1.2.1 Credit and PM Initialization

Figure 14 illustrates the use of PM2IP.CREDIT_RTN and PM2IP.AGENT_INFO messages to initialize Power Management messaging protocol intended to facilitate communication between the Host Power Management Unit and the CXL Device.

Figure 14. Power Management Credits and Initialization



The CXL device must be able to receive and process CREDIT_RTN messages without dependency on any other PM2IP messages. Also, CREDIT_RTN messages do not use a credit. The CREDIT_RTN messages are used to exchange and initialize the TX credits on each side, so that flow control can be managed appropriately. The credits being sent from either side represent the number of messages that side can receive from the other. CREDIT_RTN message is also used by the Host to assign a PM_AGENT_ID to the CXL Device. CXL Device must wait for the CREDIT_RTN message from the Host before initiating any IP2PM messages to the host.

A CXL device must support at least one credit - where a credit implies having sufficient buffering to sink a PM2IP message with 128 bits of payload.

After credit initialization, the CXL device must wait for an AGENT_INFO message from the Host. This message contains the Revision ID of the PM protocol of the Host. CXL Device must send its Revision ID to the Host in response to the AGENT_INFO Req from the host. Expectation is that the host and CXL device Revision IDs match - when there is a mismatch, Host PMU may implement a compatibility mode to work with CXL devices with older Revision ID. Alternatively, Host PMU may log the mismatch and report an error, if it does not know how to reliably function with a CXL device with a mis-matched Revision ID.

There is an expectation from the CXL device that it restores credits to the Host as soon as a message is received. Host PMU can have multiple messages in flight, if it was provided with multiple credits. Releasing credits in a timely manner will provide better performance for latency sensitive flows. Under no circumstances should the CXL device hold back a credit for longer than 10us.

The following list summarizes the rules that must be followed by a CXL Device.

- CXL Device must wait to receive PM2IP.CREDIT_RTN message before initiating any IP2PM messages
- CXL Device must use the PM_AGENT_ID that it receives in the first PM2IP message received from the Host Punit (Master)
- CXL Device must implement enough resources to sink and process any CREDIT_RTN messages without dependency on any other PM2IP or IP2PM messages or other message classes
- CXL Device must implement at least one credit to sink a PM2IP message
- CXL Device must return any credits to the Host Punit as soon as possible. Under no circumstances should the CXL device withhold a credit for longer than 10us

3.1.3 Optional PCIe Features Required for CXL

Table 4 lists optional features per the PCIe Specification that are required to enabled CXL.

Table 4. Optional PCIe Features Required For CXL

Optional PCIe Feature	Notes
Data Poisoning by transmitter	
ATS	Only required if .cache is present (e.g. only for Type 1 & Type 2 devices but not for Type 3 devices)
Additional VCs and TCs beyond VC0/TC0	VC0, optional VC1 for QoS
Advanced Error Reporting (AER)	

3.1.4 Error Propagation

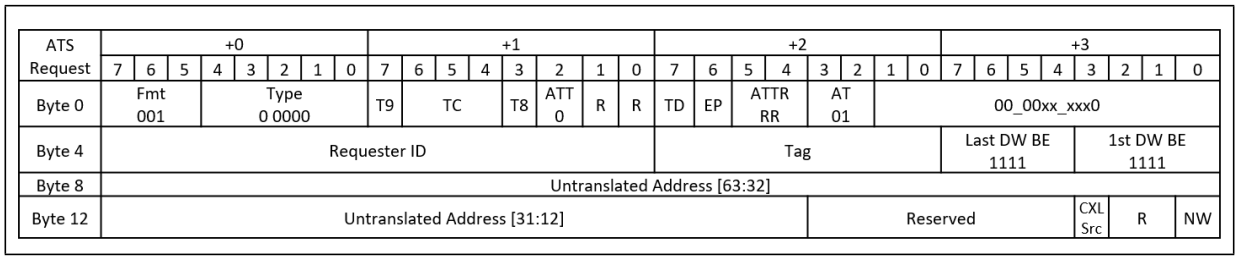
CXL.cache and CXL.mem errors detected by the device are propagated to the CPU over the CXL.io traffic stream. These errors are logged as correctable and uncorrectable internal errors in the PCIe AER registers.

3.1.5 Memory Type Indication on ATS

Requests to certain memory regions can only be issued on CXL.io and not on CXL.cache. It is up to the Host to decide what these memory regions are. For example, on x86 systems, the Host may choose to restrict access to Uncacheable (UC) type memory over CXL.io only. The Host indicates such regions by means of an indication on ATS completion to the device.

ATS requests sourced from a CXL device must set the “Source-CXL” bit.

Figure 15. ATS 64-bit Request with CXL Indication



64-bit: DWORD3, Byte 3, Bit 3; 32-bit: DWORD2, Byte 3, Bit 3Note: This bit is Reserved in the ATS request as defined by the PCIe spec.

ATS translation completion from the Host will carry the indication that requests to a given page can only be issued on CXL.io using the following bit, “Issue-on-CXL.io”, in the Translation Completion Data Entry:

Figure 16. ATS Translation Completion Data Entry with CXL indication

ATS Cpl Data Entry	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	Translated Address[63:32]																															
Byte 4	Translated Address[31:12]																S	N	CXL.io	Reserved					Global	Priv	Exe	U	W	R		

DWORD1, Byte 2, Bit 1

Note: This bit is Reserved in the ATS completion as defined by the PCIe spec.

3.1.6 Deferrable Writes

Deferrable Writes enable scalable work submission to a CXL device by multiple software entities without explicit locks or software synchronization. Deferrable Writes are downstream non-posted memory writes. The completion for a Deferrable Write allows the device to indicate whether the command was successfully accepted or if it needs to be deferred.

On CXL.io, a Deferrable Write is sent as a NPMemWr32/64 transaction which has the following encodings (please note that the encoding for NPMemWr32 is deprecated in PCIe):

Fmt[2:0] - 010b/011b

Type[4:0] - 11011b

Since Deferrable Write is non-posted, the device is expected to send a Cpl response. The Completion Status field in the Cpl (with a Byte Count of ‘4) indicates whether work was successfully accepted or not. Successful work submission is accompanied by a “Successful Completion” Completion Status. Unsuccessful work submission is accompanied by a “Memory Request Retry Status” Completion Status. The encoding for these are:

Successful Completion (SC) - 000b

Memory Request Retry Status (MRS) - 010b

3.2 CXL.cache

3.2.1 Overview

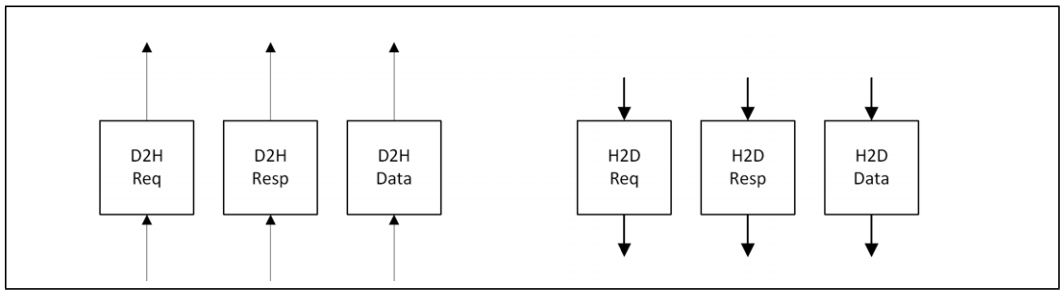
The CXL.cache protocol defines the interactions between the device and Host as a number of requests that each have at least one associated response message and sometimes a data transfer. The interface consists of three channels in each direction: Request, Response, and Data. The channels are named for their direction, D2H for Device to Host and H2D for Host to Device, and the transactions they carry, Request, Response, and Data.

D2H Request carries new requests from the Device to the Host. The requests typically target memory. Each request will receive zero, one or two responses and at most one 64-byte cacheline of data. The channel may be back pressured without issue. D2H Response carries all responses from the Device to the Host. Device responses to snoops indicate the state the line was left in the device caches, and may indicate that data is being returned to the Host to the provided data buffer. D2H responses need to be guaranteed to make progress or deadlocks may occur. They may still be blocked temporarily for link-layer credits, but should not require any other transaction to complete to free the credits. D2H Data carries all data and byte-enables from the Device to the Host. The data transfers can result either from implicit or explicit write-backs. In all cases a full 64-byte cacheline of data will be transferred. D2H Data transfers must make progress or deadlocks may occur. They may be blocked temporarily for link-layer credits, but must not require any other transaction to complete to free the credits.

H2D Request carries request from the Host to the Device. These are snoops to maintain coherency. Data may be returned for snoops. The request carries the location of the data buffer to which any return data should be written. H2D Requests may be back pressured for lack of device resources, however the resources must free up without needing D2H Requests to make progress. H2D Response carries ordering messages and pulls for write data. Each response carries the request identifier from the original device request to indicate where the response should be routed. For write data pull responses, the message carries the location where the data should be written. H2D Responses can only be blocked temporarily for link-layer credits. H2D Data delivers the data for device read requests. In all cases a full 64-byte cacheline of data will be transferred. H2D Data transfers can only be blocked temporarily for link-layer credits.

The CXL.cache interface has 3 main channels in each direction between the device and the Host. The three main channels are Request, Response, and Data as shown in the figure below. The independent channels allow different kinds of messages to use dedicated wires and achieve both decoupling and a higher effective throughput per wire.

Figure 17. CXL.cache Channels



3.2.2 CXL.cache Channel Description

3.2.2.1 Channel Ordering

In general, all of the CXL.cache channels must work independently of each other to ensure that forward progress is maintained. For example, since requests from the device to the Host to a given address X will be blocked by the Host before it collects all snoop responses for this address X, linking the channels would lead to deadlock.

However, there is a specific instance where ordering between channels must be maintained for the sake of correctness. The Host needs to wait until Global Ordering (GO) messages, sent on H2D Response, are observed by the device before sending

subsequent snoops for the same address. To limit the amount of buffering needed to track GO messages, the Host assumes that GO messages that have been sent over CXL.cache in a given cycle cannot be passed by snoops sent in a later cycle.

For transactions that have multiple concurrent messages within a single channel (e.g., FastGO and ExtCmp), the device/Host should assume that they can come in any order. For transactions that have disjoint messages on a single channel (e.g., WritePull and GO for WrInv) the device/Host must ensure they cross CXL.cache in order.

3.2.2.2 Channel Crediting

To maintain the modularity of the interface no assumptions can be made on the ability to send a message on a channel since at least link-layer credits may not be available at all times. Therefore, each channel must use a credit for sending any message and collect credit returns from the receiver. During operation, the receiver returns a credit whenever it has processed the message (i.e., freed up a buffer). It is not required that all credits are accounted for on either side, it is sufficient that credit counter saturates when full. If no credits are available, the sender must wait for the receiver to return one. The table below describes which channels must drain to maintain forward progress and which can be blocked indefinitely.

Table 5. CXL.cache Channel Crediting

Channel	Forward Progress Condition	Blocking condition	Description
D2H Request	Credited to Host	Indefinite	Needs Host buffer, could be held by earlier requests
D2H Response	Pre-allocated	Link-layer only, must make progress. Temporary back pressure is allowed.	Headed to specified Host buffer
D2H Data	Pre-allocated	Link-layer only, must make progress. Temporary back pressure is allowed.	Headed to specified Host buffer
H2D Request	Credited to device	Must make progress. Temporary back pressure is allowed.	May be temporarily back pressured due to lack of availability of D2H Response or D2H Data credits
H2D Response	Pre-allocated	Link-layer only, must make progress. Temporary back pressure is allowed.	Headed to specified device buffer
H2D Data	Pre-allocated	Link-layer only, must make progress. Temporary back pressure is allowed.	Headed to specified device buffer

3.2.3 CXL.cache Wire Description

The definition of each of the fields for each CXL.cache Channel is below.

3.2.3.1 D2H Request

Table 6. CXL.cache - D2H Request Fields

D2H Request	Width	Description
Valid	1	The request is valid.
Opcode	5	The opcode specifies the operation of the request. Details in Table 15
Address [51:6]	46	Carries the physical address of coherent requests.
CQID	12	Command Queue ID: The CQId field contains the ID of the tracker entry that is associated with the request. When the response and data is returned for this request, the CQId is sent in the response or data message indicating to the device which tracker entry originated this request. Implementation Note: CQID usage depends on the round-trip transaction latency and desired BW. To saturate link BW for a x16 link @32GT/s, 11 bits of CQID should be sufficient.
NT	1	For cacheable reads the NonTemporal field is used as a hint to indicate to the Host how it should be cached. Details in Table 7
RSVD	14	
Total	**79**	

Table 7. Non Temporal Encodings

NonTemporal	Definition
1b0	Default behavior. This is Host implementation specific.
1b1	Requested line should be moved to Least Recently Used (LRU) position

3.2.3.2 D2H Response

Table 8. CXL.cache - D2H Response Fields

D2H Response	Width	Description
Valid	1	The response is valid
Opcode	5	The opcode specifies the what kind of response is being signaled. Details in Table 18
UQID	12	Unique Queue ID: This is a reflection of the UQID sent with the H2D Request and indicates which Host entry is the target of the response
RSVD	2	
Total	**20**	

3.2.3.3 D2H Data

Table 9. CXL.cache - D2H Data Header Fields

D2H Data Header	Width	Description
Valid	1	The Valid signal indicates that this is a valid data message.
UQID	12	Unique Queue ID: This is a reflection of the UQID sent with the H2D Response and indicates which Host entry is the target of the data transfer.
ChunkValid	1	In case of a 32B transfer on CXL.cache, this indicates what 32 byte chunk of the cacheline is represented by this transfer. If not set, it indicates the lower 32B and if set, it indicates the upper 32B. This field is ignored for a 64B transfer.
Bogus	1	The Bogus field indicates that the data associated with this evict message was returned to a snoop after the D2H request was sent from the device but before a WritePull was received for the evict. This data is no longer the most current, so it should be dropped by the Host.
Poison	1	The Poison field is an indication that this data chunk is corrupted and should not be used by the Host.
RSVD	1	
Total	**17**	

3.2.3.3.1 Byte Enable

Although Byte Enable is technically part of the data header, it is not sent on the flit along with the rest of the data header fields. Instead, it is sent only if the value is not all 1's, as a data chunk as described in Section 4.2.2. The Byte Enable field is 64 bits wide and indicates which of the bytes are valid for the contained data.

3.2.3.4 H2D Request

Table 10. CXL.cache – H2D Request Fields

H2D Request	Width	Description
Valid	1	The Valid signal indicates that this is a valid request.
Opcode	3	The Opcode field indicates the kind of H2D request. Details in Table 19
Address [51:6]	46	The Address field indicates which cache line the request targets.
UQID	12	Unique Queue ID: This indicates which Host entry is the source of the request
RSVD	2	
Total	**64**	

3.2.3.5 H2D Response

Table 11. CXL.cache - H2D Response Fields

H2D Response	Width	Description
Valid	1	The Valid field indicates that this is a valid response to the device.
Opcode	4	The Opcode field indicates the type of the response being sent. Details in Table 20
RspData	12	The response Opcode determines how the RspData field is interpreted as shown in Table 20. Thus, depending on Opcode, it can either contain the UQID or the MESI information in bits [3:0] as shown in Table 13.
RSP_PRE	2	RSP_PRE carries performance monitoring information for requests that do not receive data. Details in Table 12
CQID	12	Command Queue ID: This is a reflection of the CQID sent with the D2H Request and indicates which device entry is the target of the response.
RSVD	1	
Total	**32**	

Table 12. RSP_PRE Encodings

RSP_PRE[1:0]	Response
00	Host Cache Miss to Local CPU socket memory
01	Host Cache Hit
10	Host Cache Miss to Remote CPU socket memory
11	Reserved

Table 13. Cache State Encoding for H2D Response

Cache State	Encoding
Invalid (I)	4'b0011
Shared (S)	4'b0001
Exclusive (E)	4'b0010
Modified (M)	4'b0110
Error (Err)	4'b0100

3.2.3.6 H2D Data

Table 14. CXL.cache - H2D Data Header Fields (Sheet 1 of 2)

H2D Data Header	Width	Description
Valid	1	The Valid field indicates that this is a valid data to the device.
CQID	12	Command Queue ID: This is a reflection of the CQID sent with the D2H Request and indicates which device entry is the target of the data transfer.
ChunkValid	1	In case of a 32B transfer on CXL.cache, this indicates what 32 byte chunk of the cacheline is represented by this transfer. If not set, it indicates the lower 32B and if set, it indicates the upper 32B. This field is ignored for a 64B transfer.

Table 14. CXL.cache - H2D Data Header Fields (Sheet 2 of 2)

H2D Data Header	Width	Description
Poison	1	The Poison field indicates to the device that this data is corrupted and as such should not be used.
GO-Err	1	The GO-ERR field indicates to the agent that this data is the result of an error condition and should not be cached or provided as response to snoops.
RSVD	8	
Total	**24**	

3.2.4 CXL.cache Transaction Description

3.2.4.1 Device to Host Requests

3.2.4.1.1 Device to Host (D2H) CXL.cache Request Semantics

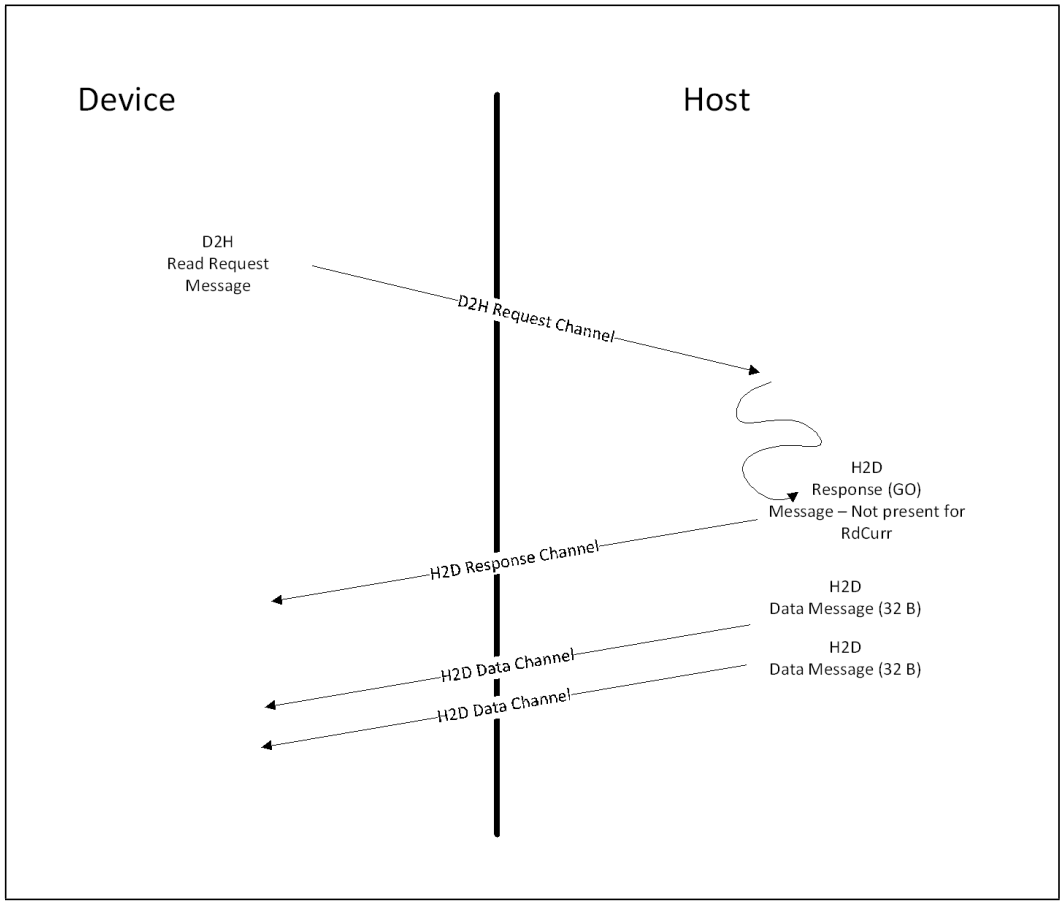
For device to Host requests there are four different semantics: CXL.cache Read, CXL.cache Read0, CXL.cache Read0/Write, and CXL.cache Write. All device to Host CXL.cache transactions fall into the one of these four semantics, though the allowable responses and restrictions for each request type within a given semantic are different.

3.2.4.1.2 CXL.cache Read

CXL.cache Reads must have a D2H request credit and send a request message on the D2H CXL.cache request channel. CXL.cache Read requests require zero or one response (GO) message and data messages totaling a single 64 byte cache line of data. Both the response, if present, and data messages are directed at the device tracker entry provided in the initial D2H request packet's CQid field. The device entry must remain active until all the messages from the Host have been received. To ensure forward progress the device must have a reserved data buffer to be able to accept all 64 bytes of data immediately after the request is sent. However, the device may temporarily be unable to accept data from the Host due to prior data returns not draining. Once both the response message and the data messages have been received from the Host, the transaction can be considered complete and the entry de-allocated from the device.

The figure below shows the elements required to complete a CXL.cache Read. Note that the response (GO) message can be received before, after, or between the data messages.

Figure 18. CXL.cache Read Behavior

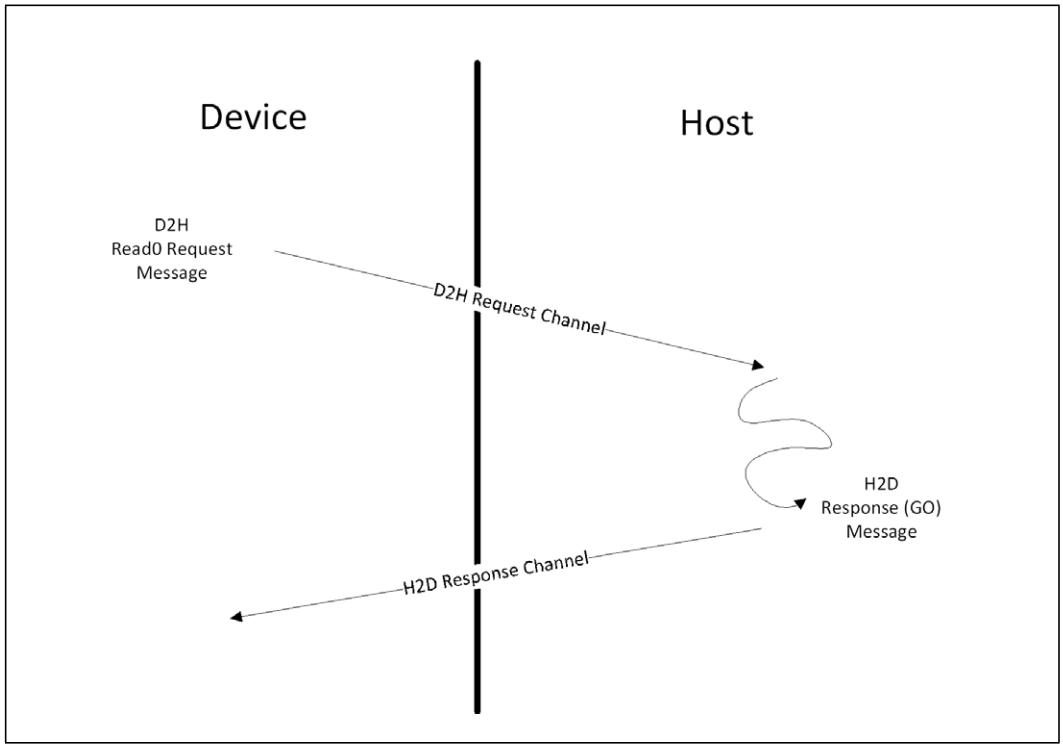


3.2.4.1.3 CXL.cache Read0

CXL.cache Read0 must have a D2H request credit and send a message on the D2H CXL.cache request channel. CXL.cache Read0 requests receive a response message but no data messages. The response message is directed at the device entry indicated in the initial D2H request message's CQId value. Once the GO message is received for these requests, they can be considered complete and the entry de-allocated from the device. A data message must not be sent by the Host for these transactions. Most special cycles (e.g., CLFlush) and other miscellaneous requests fall into this category. Details in Table 15.

The following figure shows the elements required to complete a CXL.cache Read0 transaction.

Figure 19. CXL.cache Read0 Behavior



3.2.4.1.4 CXL.cache Write

CXL.cache Write must have a D2H request credit before sending a request message on the D2H CXL.cache request channel. Once the Host has received the request message, it is required to send either two separate or one merged GO-I and WritePull message. The GO message must never arrive at the device before the WritePull but it can arrive at the same time in the combined message. If the transaction requires posted semantics then a combined GO-I/WritePull message can be used. If the transaction requires non-posted semantics, then WritePull will be issued first followed by the GO-I when the non-posted write is globally observed.

Upon receiving the GO-I message, the device will consider the store done from a memory ordering and cache coherency perspective, giving up snoop ownership of the cache line (if the CXL.cache message is an Evict).

The WritePull message triggers the device to send data messages to the Host totaling exactly 64 bytes of data, though any number of byte enables can be set.

A CXL.cache write transaction is considered complete by the device once the device has received the GO-I message, and has sent the required data messages. At this point the entry can be de-allocated from the device.

The Host considers a write to be done once it has received all 64 bytes of data, and has sent the GO-I response message. All device writes and Evicts fall into the CXL.cache Write semantic.

See Section Multiple Evicts to the same cache line for more information on restrictions around multiple active write transactions.

Figure 20 shows the elements required to complete a CXL.cache Write transaction (that matches posted behavior). The WritePull (or the combined GO_WritePull) message triggers the data messages. There are restrictions on Snoops and WritePulls. See Section Device/Host Snoop/WritePull Assumptions for more details.

Figure 21 shows a case where the WritePull is a separate message from the GO (for example: strongly ordered uncacheable write).

Figure 22 shows the Host FastGO plus ExtCmp responses for weakly ordered write requests.

Figure 20. CXL.cache Device to Host Write Behavior

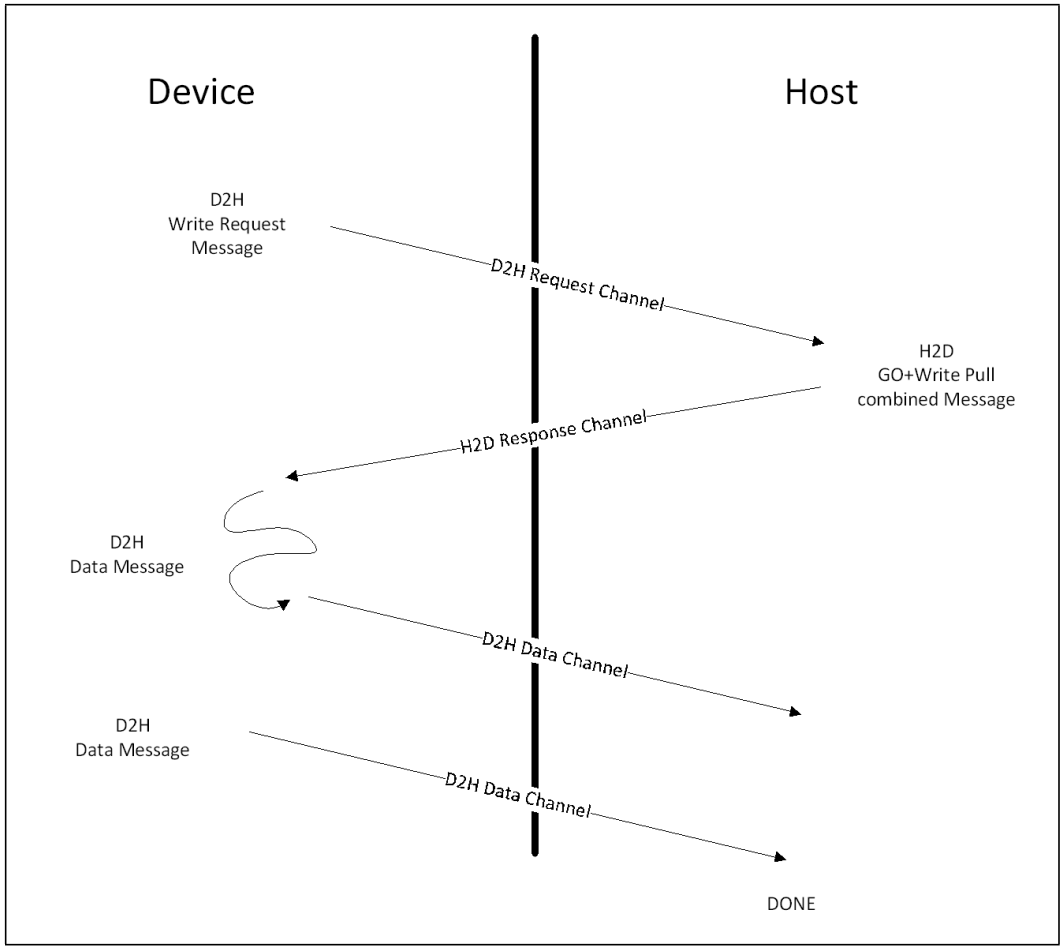


Figure 21. CXL.cache WrInv Transaction

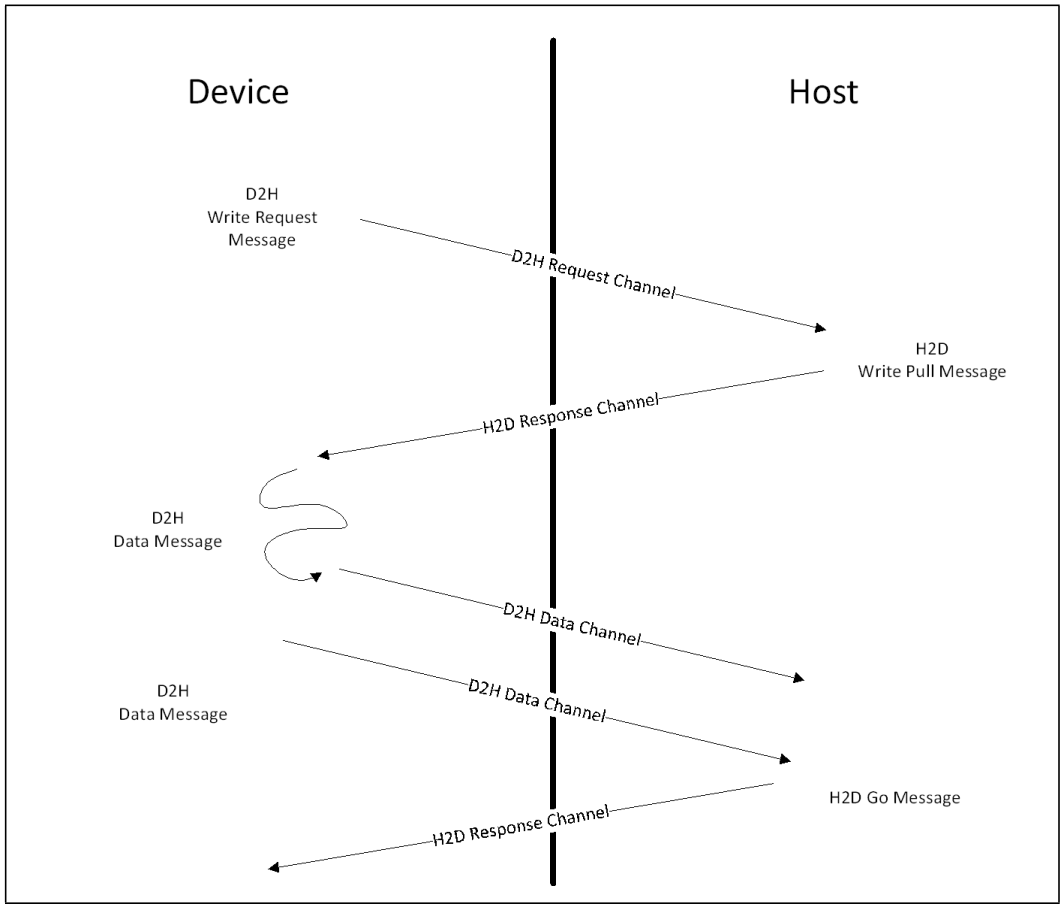
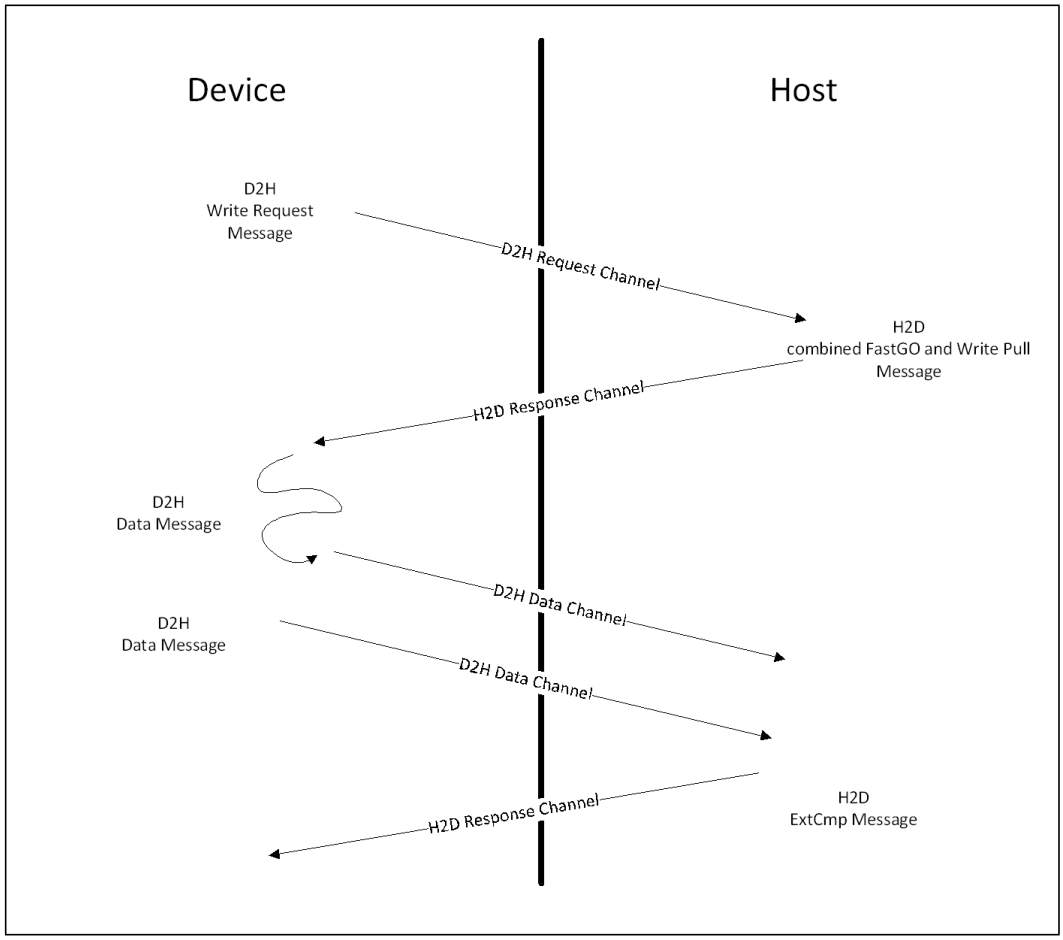


Figure 22. WOWrInv/F with FastGO/ExtCmp



3.2.4.1.5 CXL.cache Read0-Write Semantics

CXL.cache Read0-Write requests must have a D2H request credit before sending a request message on the D2H CXL.cache request channel. Once the Host has received the request message, it is required to send one merged GO-I and WritePull message.

The WritePull message triggers the device to send the data messages to the Host, which together transfer exactly 64 bytes of data though any number of byte enables can be set.

A CXL.cache Read0-write transaction is considered complete by the device once the device has received the GO-I message, and has sent the all required data messages. At this point the entry can be de-allocated from the device.

The Host considers a read0-write to be done once it has received all 64 bytes of data, and has sent the GO-I response message. ItoMWr falls into the Read0-Write category.

Figure 23. CXL.cache Read0-Write Semantics

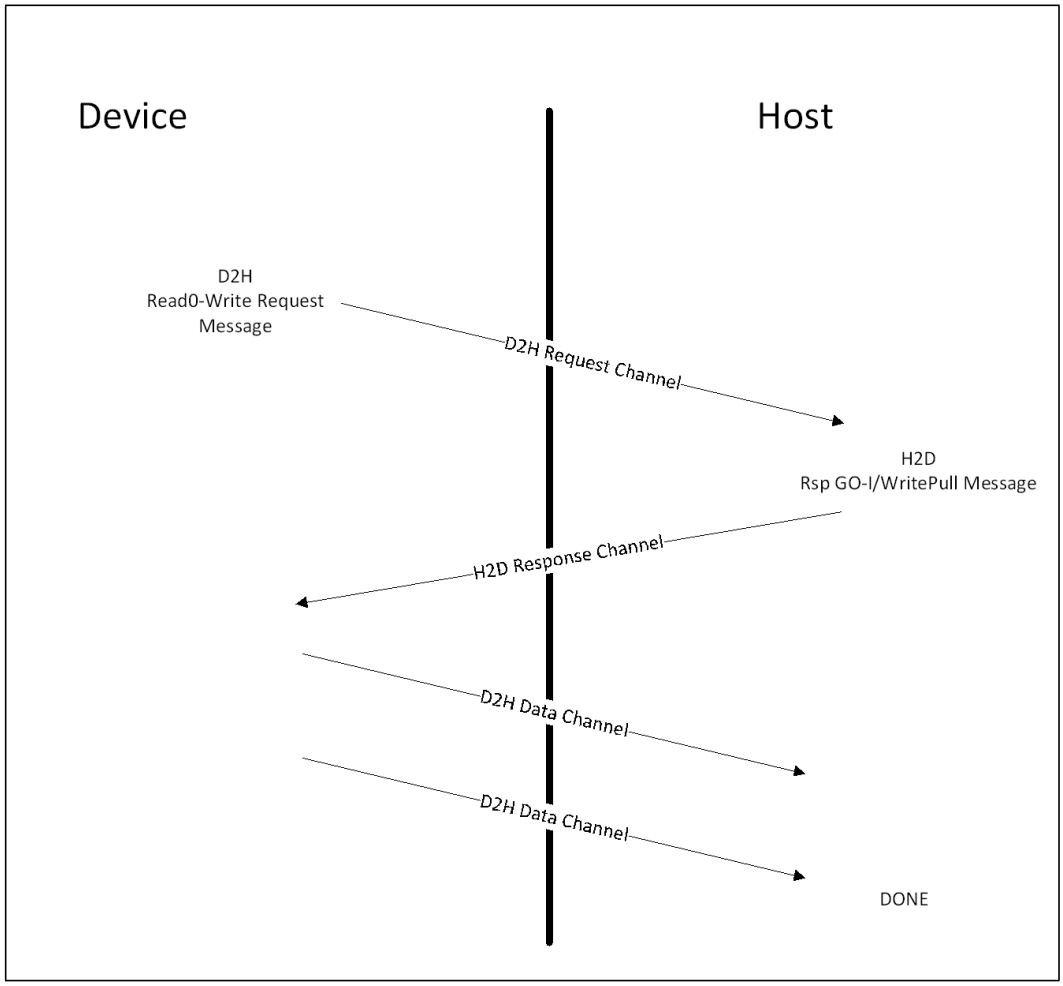


Table 15 summarizes all the opcodes available from Device to Host.

Table 15. CXL.cache. – Device to Host Requests (Sheet 1 of 2)

CXL.cache Opcode	Semantic	Opcode
RdCurr	Read	00001
RdOwn	Read	00010
RdShared	Read	00011
RdAny	Read	00100
RdOwnNoData	Read0	00101
ItoMWr	Read0-Write	00110
MemWr	Read0-Write	00111
CLFlush	Read0	01000
CleanEvict	Write	01001

Table 15. CXL.cache. – Device to Host Requests (Sheet 2 of 2)

CXL.cache Opcode	Semantic	Opcode
DirtyEvict	Write	01010
CleanEvictNoData	Write	01011
WOWrInv	Write	01100
WOWrInvF	Write	01101
WrInv	Write	01110
CacheFlushed	Read0	10000

3.2.4.1.6 RdCurr

These are full cache-line read requests from the device for lines to get the most current data, but not change the existing state in any cache, including in the Host. The Host does not need to track the cache-line in the device that issued the RdCurr. RdCurr gets a data but no GO. The device receives the line in the Invalid state which means it gets one use of the line and cannot cache it.

3.2.4.1.7 RdOwn

These are full cache-line reads requests from the device for lines to be cached in any writeable state. Typically RdOwn request receives the line in Exclusive (GO-E) or Modified (GO-M) state. Lines in Modified state must not be dropped, and have to be written back to the Host.

Under error conditions, a RdOwn request may receive the line in Invalid (GO-I) or Error (GO-Err) state. Both will return synthesized data of all1s. The device is responsible for handling the error appropriately.

3.2.4.1.8 RdShared

These are full cache-line read requests from the device for lines to cached in Shared state. Typically, RdShared request receives the line in Shared (GO-S) state.

Under error conditions, a RdShared request may receive the line in Invalid (GO-I) or Error (GO-Err) state. Both will return synthesized data of all1s. The device is responsible for handling the error appropriately.

3.2.4.1.9 RdAny

These are full cache-line read requests from the device for lines to cached in any state. Typically, RdAny request receives the line in Shared (GO-S), Exclusive (GO-E) or Modified (GO-M) state. Lines in Modified state must not be dropped, and have to be written back to the Host.

Under error conditions, a RdAny request may receive the line in Invalid (GO-I) or Error (GO-Err) state. Both will return synthesized data of all1s. The device is responsible for handling the error appropriately.

3.2.4.1.10 RdOwnNoData

These are requests to get exclusive ownership of the cache-line address indicated in the address field. The typical response is Exclusive (GO-E).

Under error conditions, a RdOwnNoData request may receive the line in Error (GO-Err) state. The device is responsible for handling the error appropriately.

3.2.4.1.11 ItoMWr

This command requests exclusive ownership of the cache-line address indicated in the address field and atomically writes the cache-line back to the Host. The device guarantees the entire line will be modified, so no data needs to be transferred to the device. The typical response is GO-I-WritePull, which is sent once the request is granted ownership. The device must not retain a copy of the line.

If an error occurs, then GO-Err-WritePull is sent instead. The device sends the data to the Host, which drops it. The device is responsible for handling the error as appropriate.

3.2.4.1.12 MemWr

The command behaves like the ItoMWr in that it atomically requests ownership of a cache-line and then writes a full cache-line back to the fabric. However, it differs from ItoMWr in where the data is written. Only if the command hits in a cache will the data be written there, on a miss the data will be written to directly to memory. The typical response is GO-I-WritePull once the request is granted ownership. The device must not retain a copy of the line.

If an error occurs, then GO-Err-WritePull is sent instead. The device sends the data to the Host, which drops it. The device is responsible for handling the error as appropriate.

3.2.4.1.13 ClFlush

This is a request to the Host to invalidate the cache-line specified in the address field. The typical response is GO-I that will be sent from the Host upon completion in memory.

Under error conditions, a ClFlush request may receive the line in the Error (GO-Err) state. The device is responsible for handling the error appropriately.

3.2.4.1.14 CleanEvict

This is a request to the Host to evict a full 64 byte Exclusive cache line from the device. Typically, CleanEvict receives GO-WritePull or GO-WritePullDrop. Receiving any means the device must relinquish snoop ownership of the line. For GO-WritePull the device will send the data as normal. For GO-WritePullDrop the device simply drops the data.

Once the device has issued this command and the address is subsequently snooped, but before the device has received the GO-WritePull or GO-WritePullDrop, the device must set the Bogus field in all D2H Data messages to indicate the data is now stale.

CleanEvict requests also guarantee to the Host that the device no longer contains any cached copies of this line. Only one CleanEvict from the device may be pending on CXL.cache for any given cache-line address.

CleanEvict is only expected for host-attached memory range of addresses. For device-attached memory range, the equivalent operation can be completed internally within the device without sending a transaction to the Host.

3.2.4.1.15 DirtyEvict

This is a request to the Host to evict a full 64 byte Modified cache line from the device. Typically, DirtyEvict receives GO-WritePull from the Host at which point the device must relinquish snoop ownership of the line and send the data as normal.

Once the device has issued this command and the address is subsequently snooped, but before the device has received the GO-WritePull, the device must set the Bogus field in all D2H Data messages to indicate the data is now stale.

DirtyEvict requests also guarantee to the Host that the device no longer contains any cached copies of this line. Only one DirtyEvict from the device may be pending on CXL.cache for any given cache-line address.

In error conditions, a GO-Err-WritePull will be received. The device will send the data as normal, and the Host will drop it. The device is responsible for handling the error as appropriate.

DirtyEvict is only expected for host-attached memory range of addresses. For device-attached memory range, the equivalent operation can be completed internally within the device without sending a transaction to the Host.

3.2.4.1.16 CleanEvictNoData

This is a request for the device to update the Host that a clean line is dropped in the device. The sole purpose of this request is to update any snoop filters in the Host and no data will be exchanged.

CleanEvictNoData is only expected for host-attached memory range of addresses. For device-attached memory range, the equivalent operation can be completed internally within the device without sending a transaction to the Host.

3.2.4.1.17 WOWrInv

This is a weakly ordered write invalidate line request of 0-63 bytes for write combining type stores. Any combination of byte enables may be set.

Typically, WOWrInv receives a FastGO-WritePull followed by an ExtCmp. Upon receiving the FastGO-WritePull the device sends the data to the Host. For host-attached memory, the Host sends the ExtCmp once the write is complete in memory.

In error conditions, a GO-Err-Writepull will be received. The device will send the data as normal, and the Host will drop it. The device is responsible for handling the error as appropriate. An ExtCmp will still be sent by the Host after the GO-Err in all cases.

3.2.4.1.18 WOWrInvF

Same as WOWrInv (rules and flows), except it is a write of 64 bytes.

3.2.4.1.19 WrInv

This is a write invalidate line request of 0-64 bytes. Typically WrInv receives a WritePull followed by a GO. Upon getting the WritePull the device sends the data to the Host. The Host sends GO once the write complete in memory (both, host-attached or device-attached).

In error conditions, a GO-Err is received. The device is responsible for handling the error as appropriate.

3.2.4.1.20 CacheFlushed

This is an indication sent by the device to inform the Host that its caches are flushed and it no longer contains any cache-lines in the Shared, Exclusive or Modified state. The Host can use this information to clear its snoop filters and block snoops to the device and return a GO. Once the device receives the GO, it is guaranteed to not receive any snoops from the Host until the device sends the next cacheable D2H Request.

Table 16. D2H Request (targeting non-device-attached memory) supported H2D Responses

D2H Request	WritePull	GO_WritePull	ExtCmp	GO_WritePull_Drop	FastGO_WritePull	GO_ERR_WritePull	GO-Err	GO-I	GO-S	GO-E	GO-M
CLFlush							X	X			
RdShared							X	X	X		
RdAny							X	X	X	X	X
ItoMWr		X				X					
MemWr		X				X					
CacheFlushed								X			
RdCurr											
RdOwn							X	X		X	X
RdOwnNoData							X			X	
CleanEvict		X		X							
DirtyEvict		X				X					
CleanEvictNoData								X			
WOWrInv			X		X	X					
WOWrInvF			X		X	X					
WrInv	X						X	X			

For requests targeting device-attached memory, if the region is in Device Bias, no transaction is expected on CXL.cache since the Device can complete those requests internally. If the region is in Host Bias, the table below shows how the device should expect the response.

Table 17. D2H Request (Targeting Device-attached Memory) Supported Responses (Sheet 1 of 2)

D2H Request	Response on CXL.mem	Response on CXL.cache
RdCurr	MemRdFwd	None
RdOwn	MemRdFwd	None
RdShared	MemRdFwd	None
RdAny	MemRdFwd	None
RdOwnNoData	MemRdFwd	None
ItoMWr	None	Same as host-attached memory
MemWr	None	Same as host-attached memory
CLFlush	MemRdFwd	None

Table 17. D2H Request (Targeting Device-attached Memory) Supported Responses (Sheet 2 of 2)

D2H Request	Response on CXL.mem	Response on CXL.cache
CleanEvict	NA	NA
DirtyEvict	NA	NA
CleanEvictNoData	NA	NA
WOWrInv	MemWrFwd	None
WOWrInvF	MemWrFwd	None
WrInv	None	Same as host-attached memory
CacheFlushed	None	Same as host-attached memory

CleanEvict, DirtyEvict and CleanEvictNoData targeting device-attached memory should always be completed internally by the device, regardless of bias state. For D2H Requests that receive a response on CXL.mem, the CQID associated with the CXL.cache request is reflected in the Tag of the CXL.mem MemRdFwd or MemWrFwd command. For MemRdFwd, the caching state of the line is reflected in the MetaValue field as described in Table 24.

3.2.4.2 Device to Host Response

Responses are directed at the Host entry indicated in the UQId field in the original H2D request message.

Table 18. D2H Response Encodings

Device CXL.cache Rsp	Opcode
RspIHitI	00100
RspVHitV	00110
RspIHitSE	00101
RspSHitSE	00001
RspSFwdM	00111
RspIFwdM	01111
RspVFwdV	10110

3.2.4.2.1 RspIHitI

In general, this is the response that a device provides to a snoop when the line was not found in any caches. If the device returns RspIHitI for a snoop, the Host can assume the line has been cleared from that device.

3.2.4.2.2 RspVHitV

In general, this is the response that a device provides to a snoop when the line was hit in the cache and no state change occurred. If the device returns an RspVHitV for a snoop, the Host can assume a copy of the line is present in one or more places in that device.

3.2.4.2.3 RspIHitSE

In general, this is the response that a device provides to a snoop when the line was hit in a clean state in at least one cache and is now invalid. If the device returns an RspIHitSE for a snoop, the Host can assume the line has been cleared from that device.

3.2.4.2.4 RspSHitSE

In general, this is the response that a device provides to a snoop when the line was hit in a clean state in at least one cache and is now downgraded to shared state. If the device returns an RspSHitSE for a snoop, the Host should assume the line is still in the device.

3.2.4.2.5 RspSFwdM

This response indicates to the Host that the line being snooped is now in S state in the device, after having hit the line in Modified state. The device may choose to downgrade the line to Invalid. This response also indicates to the Host snoop tracking logic that 64 bytes of data will be transferred on the D2H CXL.cache Data Channel to the Host data buffer indicated in the original snoop's destination (UQid).

3.2.4.2.6 RspIFwdM

(aka HITM) This response indicates to the Host that the line being snooped is now in I state in the device, after having hit the line in Modified state. The Host may now assume the device contains no more cached copies of this line. This response also indicates to the Host snoop tracking logic that 64 bytes of data will be transferred on the D2H CXL.cache Data Channel to the Host data buffer indicated in the original snoop's destination (UQid).

3.2.4.2.7 RspVFwdV

This response indicates that the device is returning the current data to the Host and leaving the state unchanged. The Host must only forward the data to the requestor since there is no state information.

3.2.4.3 Host to Device Requests

Snoops from the Host need not gain any credits besides local H2D request credits. The device will always send a Snoop Response message on the D2H CXL.cache Response channel. If the response is of the Rsp*Fwd* format, then the device must respond with 64 bytes of data via the D2H Data channel, directed at the UQid from the original snoop request message. If the response is not Rsp*Fwd*, the Host can consider the request complete upon receiving all of the snoop response messages. The device can stop tracking the snoop once the response has been sent for non-data forwarding cases, or after both the last chunk of data has been sent and the response has been sent.

The figure below shows the elements required to complete a CXL.cache snoop. Note that the response message can be received by the Host with any relative order with the data messages. The byte enable field is always all 1s for Snoop data transfers.

Figure 24. CXL.cache Snoop Behavior

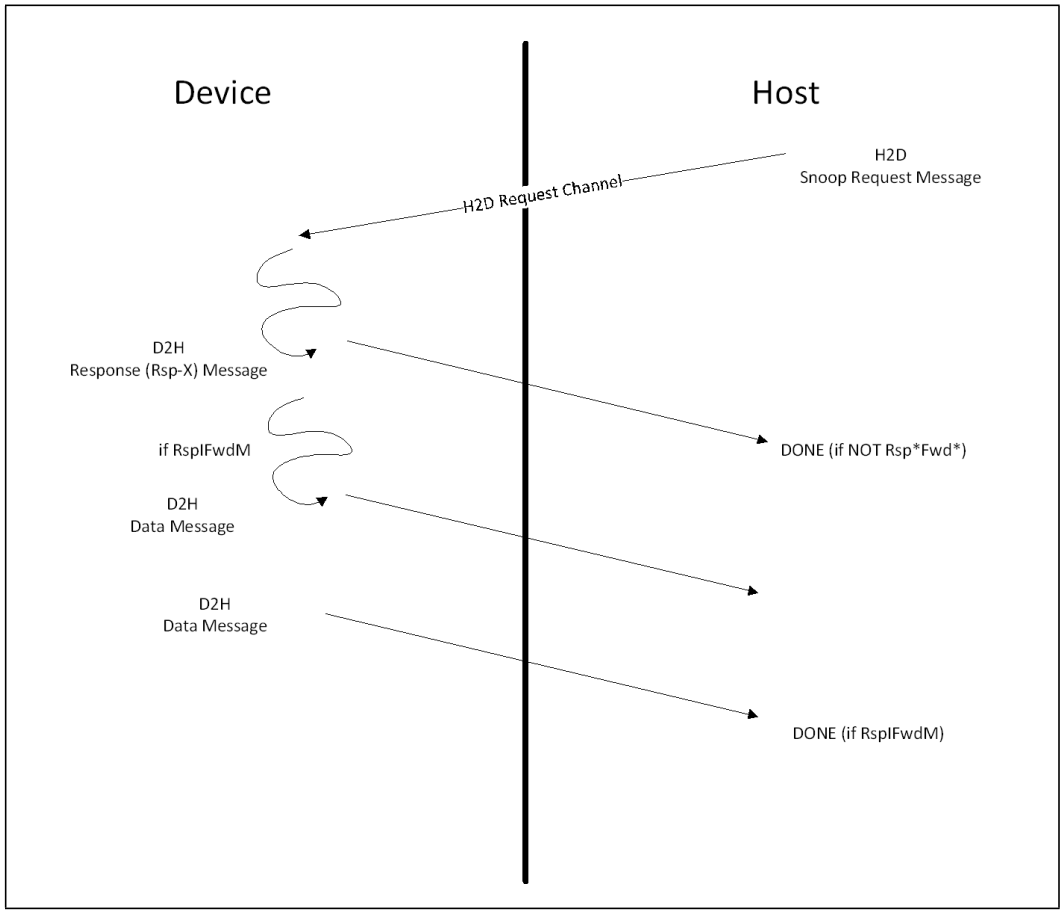


Table 19. CXL.cache – Mapping of Host to Device Requests & Responses

	Opcode	RspIHitI	RspVHitV	RspSHitSE	RspIHitSE	RspSFwdM	RspIFwdM	RspVFwdV
SnpData	'001	X		X		X	X	
SnpInv	'010	X			X		X	
SnpCurr	'011	X	X	X		X	X	X

3.2.4.3.1 SnpData

These are snoop requests from the Host for lines that are intended to be cached in either Shared or Exclusive state at the requester (the Exclusive state can be cached at the requester only if all devices respond with RspI). This type of snoop is typically

triggered by data read requests. A device that receives this snoop must either invalidate or downgrade all cache lines to Shared state. If the device holds dirty data it must return it to the Host.

3.2.4.3.2 SnpInv

These are snoop requests from the Host for lines that are intended to be granted ownership and Exclusive state at the requester. This type of snoop is typically triggered by write requests. A device that receives this snoop must invalidate all cache lines. If the device holds dirty data it must return it to the Host.

3.2.4.3.3 SnpCur

This snoop gets the current version of the line, but doesn't require change of any cache state in the hierarchy. It is only sent on behalf of the RdCurr request. If the device holds data in Modified state it must return it to the Host. The cache state can remain unchanged in both the device and Host, and the Host should not update its caches.

3.2.4.4 Host to Device Response

Table 20. H2D Response Opcode Encodings

H2D Response Class	Encoding	RspData
WritePull	0001	UQID
GO	0100	MESI
GO_WritePull	0101	UQID
ExtCmp	0110	Don't Care
GO_WritePull_Drop	1000	UQID
Fast_GO	1100	Don't Care
Fast_GO_WritePull	1101	UQID
GO_ERR_WritePull	1111	UQID

3.2.4.4.1 WritePull

This response tells the device to send the write data to the Host, but not to change the state of the line. This is used for WrInv where the data is needed before the GO-I can be sent. This is because GO-I is the notification that the write was completed by I/O.

3.2.4.4.2 GO

The Global Observation (GO) message conveys that read requests are coherent and that write requests are coherent and consistent. It is an indication that the transaction has been observed by the system device and the MESI state that is encoded in the RspType field indicates what state the data associated with the transaction should be put in for the requester's caches. Details in Table 11.

If the Host returns Modified state to the device, then the device is responsible for the dirty data and cannot drop the line without writing it back to the Host.

If the Host returns Invalid or Error state to the device, then the device must use the data at most once and not cache the data. Error responses to reads and cacheable write requests (for example, RdOwn or ItoMWr) will always be the result of an abort condition, so modified data can be safely dropped in the device.

3.2.4.4.3 GO_WritePull

This is a combined GO + WritePull message. No cache state is transferred to the device. The GO+WritePull message is used for posted write types.

3.2.4.4.4 ExtCmp

This response indicates that the data that was previously locally ordered (FastGO) has been observed throughout the system. Most importantly, accesses to memory will return the most up to date data.

3.2.4.4.5 GO_WritePull_Drop

This message has the same semantics as Go_WritePull, except that the device should not send data to the Host. This response can be sent in place of GO_WritePull when the Host determines that the data is not required. This response will never be sent for partial writes since the byte enables will always need to be transferred.

3.2.4.4.6 Fast_GO

Similar to GO, but only indicates that the request is locally observed. There will be a later ExtCmp message when the transaction is fully observable in memory. Devices that do not implement the Fast_GO feature may ignore this message and wait for the ExtCMP.

3.2.4.4.7 Fast_GO_WritePull

Similar to GO_WritePull, but only indicates that the request is locally observed. There will be a later ExtCmp message when the transaction is fully observable in memory. Devices that do not implement the Fast_GO feature may ignore the GO message and wait for the ExtCMP. Data must always be sent for the WritePull though. No cache state is transferred to the device.

3.2.4.4.8 GO_ERR_WritePull

Similar to GO_WritePull, but indicates that there was an error with the transaction that should be handled properly in the device. Data must be sent to the Host for the WritePull, and the Host will drop the data. No cache state is transferred to the device (assumed Error). An ExtCmp is still sent if it is expected by the originating request.

3.2.5 Cacheability Details and Request Restrictions

These details and restrictions apply to all devices.

3.2.5.1 GO-M Responses

GO-M responses from the Host indicate that the device is being granted the sole copy of modified data. The device must cache this data and write it back when it is done.

3.2.5.2 Device/Host Snoop-GO-Data Assumptions

When the Host returns a GO response to a device, the expectation is that a snoop arriving to the same address of the request receiving the GO would see the results of that GO. For example, if the Host sends GO-E for an RdOwn request followed by a snoop to the same address immediately afterwards, then one would expect the device to transition the line to M state and reply with an RspIFwdM response back to the Host. In order to implement this principle, CXL.cache link layer ensures that the device will receive the two messages in separate slots to make the order completely unambiguous.

When the Host is sending a snoop to the device, the requirement is that no GO response will be sent to any requests with that address in the device until after the Host has received a response for the snoop and all implicit writeback (IWB) data (dirty data forwarded in response to a snoop) has been received.

When the Host returns data to the device for a read type request, and GO for that request has not yet been sent to the device, the Host may not send a snoop to that address until after the GO message has been sent. Since the new cache state is encoded in the response message for reads, sending a snoop to an address without having received GO, but after having received data, is ambiguous to the device as to what the snoop response should be in that situation.

Fundamentally, the GO that is associated with a read request also applies to the data returned with that request. Sending data for a read request implies that that data is valid, meaning the device can consume it even if the GO has not yet arrived. The GO will arrive later and inform the device what state to cache the line in (if at all) and whether or not the data was the result of an error condition (such as hitting an address region the device was not allowed to access).

3.2.5.3 Device / Host Snoop / WritePull Assumptions

The device requires that the Host cannot have both a WritePull and H2D Snoop active on CXL.cache to a given 64 byte address. The Host may not launch a snoop to a 64 byte address until all WritePull data from that address has been received by the Host. Conversely, the Host may not launch a WritePull for a write until the Host has received the snoop response (including data in case of Rsp*Fwd*) for any snoops to the pending writes address. Any violation of these requirements will mean that the Bogus field on the D2H Data channel will be unreliable.

3.2.5.4 Snoop Responses and Data Transfer on CXL.cache Evicts

In order to snoop cache evictions (for example, DirtyEvict) and maintain an orderly transfer of snoop ownership from the device to the Host, cache evictions on CXL.cache must adhere to the following protocol.

If a device Evict transaction has been issued on the CXL.cache D2H request channel, but has not yet processed its WritePull from the Host, and a snoop hits the WB, the device must track this snoop hit. When the device begins to process the WritePull, the device must set the Bogus field in all of the D2H data messages sent to the Host. The intent is to communicate to the Host that the request data was already sent as IWB data, so the data from the Evict is potentially stale.

3.2.5.5 Multiple Snoops to the same address

The Host is only allowed to have one snoop outstanding to a given cache line address to a given device at one time. The Host must wait until it has received both the snoop response and all IWB data (if any) before dispatching the next snoop to that address.

3.2.5.6 Multiple Reads to the same cache line

Multiple read requests (cacheable or uncacheable) to the same cache line are allowed. The Host can freely reorder requests, so the device is responsible for ordering requests when required.

3.2.5.7 Multiple Evicts to the same cache line

Multiple Evicts to the same cache line are not allowed. The second Evict may only be issued after the first receives both the CXL.cache GO-I response and the WritePull.

Since Evict guarantees that the evicted cache line is otherwise in the initiating device, it is impossible to send another Evict without an intervening cacheable Read/Read0 request to that address.

3.2.5.8 Multiple WriteRequests to the same cache line

Multiple WrInv/WOWrInv/ItoMWr/MemWr to the same cache line are allowed to be outstanding on CXL.cache. The Host can freely reorder requests, so the device is responsible for ordering requests when required.

3.2.5.9 Normal Global Observation (GO)

Normal Global Observation (GO) responses are sent only after the Host has guaranteed that request will have next ownership of the requested cache line. GO messages for requests carry the cache line state permitted through the MESI state or indicate that the data should only be used once and whether or not an error occurred.

3.2.5.10 Relaxed Global Observation (FastGO)

FastGO is only allowed for requests that do not require strict ordering. The Host may return the FastGO once the request is guaranteed next ownership of the requested cache-line within the socket, but not necessarily in the system. Requests that receive a FastGO response and require completion messages are usually of the write combining memory type and the ordering requirement is that there will be a final completion (ExtCmp) message indicating that the request is at the stage where it is fully observed throughout the system.

3.2.5.11 Evict to Device-Attached Memory

Device Evicts to device-attached memory are not allowed on CXL.cache. The device is only allowed to issue WrInv, WOWrInv* to device-attached memory.

3.2.5.12 Memory Type on CXL.cache

To source requests on CXL.cache, devices need to get the Host Physical Address (HPA) from the Host by means of an ATS request on CXL.io. Due to memory type restrictions, on the ATS completion, the Host indicates to the device if a HPA can only be issued on CXL.io as described in Section 3.1.5. The device is not allowed to issue requests to such HPAs on CXL.cache.

3.2.5.13 General Assumptions

1. The Host will NOT preserve ordering of the CXL.cache requests as delivered by the device. The device must maintain the ordering of requests for the case(s) where ordering matters. For example, if D2H memory writes need to be ordered with respect to a MSI (on CXL.io), it is up to the device to implement the ordering. This is made possible by the non-posted nature of all requests on CXL.cache.
2. The order chosen by the Host will be conveyed differently for reads and writes. For reads, a Global Observation (GO) message conveys next ownership of the addressed cache line; the data message conveys ordering with respect to other transactions. For writes, the GO message conveys both next ownership of the line and ordering with respect to other transactions.
3. The device may cache ownership and internally order writes to an address if a prior read to that address received either GO-E or GO-M.
4. For reads from the device, the Host transfers ownership of the cache line with the GO message, even if the data response has not yet been received by the device. The device must respond to a snoop to a cache line which has received GO, but if

data from the current transaction is required (e.g., a RdOwn to write the line) the data portion of the snoop is delayed until the data response is received.

5. The Host must not send a snoop for an address where the device has received a data response for a previous read transaction but has not yet received the GO. Refer to Section 3.2.5.2
6. Write requests (other than Evicts) such as WrInv, WOWrInv*, ItoMWr and MemWr will never respond to WritePulls with data marked as Bogus.
7. The Host must not send two cache-line data responses to the same device request. The device may assume one-time use ownership (based on the request) and begin processing for any part of a cache line received by the device before the GO message. Final state information will arrive with the GO message, at which time the device can either cache the line or drop it depending on the response.
8. For a given transaction, H2D Data transfers may be done in any order, and may be interleaved with data transfers for other transactions.
9. D2H Data transfer of a cache line must come in consecutive packets with no interleaved transfers from other lines. The data must come in natural chunk order, that is, 64B transfers must complete the lower 32B half first, since snoops are always cache line aligned.
10. Device snoop responses in D2H Response must not be dependent on any other channel or on any other requests in the device besides the availability of credits in the D2H Response channel. The Host must guarantee that the responses will eventually be serviced and return credits to the device.
11. The Host must not send a second snoop request to an address until all responses, plus data if required, for the prior snoop are collected.
12. H2D Response and H2D Data messages to the device must drain without the need for any other transaction to make progress.
13. The Host must not return GO-M for data that is not actually modified with respect to memory.
14. The Host must not write unmodified data back to memory.
15. Except for WOWrInv and WOWrInF, all other writes are strongly ordered

3.3 CXL.mem

3.3.1 Introduction

The CXL Memory Protocol is called CXL.mem, and it is a transactional interface between the CPU and Memory. It uses the phy and link layer of Compute Express Link (CXL) when communicating across dies. The protocol can be used for multiple different Memory attach options including when the Memory Controller is located in the Host CPU, when the Memory Controller is within an Accelerator device, or when the Memory Controller is moved to a memory buffer chip. It applies to different Memory types (volatile, persistent etc) and configurations (flat, hierarchical etc) as well.

The coherency engine in the CPU interfaces with the Memory (Mem) using CXL.mem requests and responses. In this configuration, the CPU coherency engine is regarded as the CXL.mem Master and the Mem device is regarded as the CXL.mem Subordinate. The CXL.mem Master is the agent which is responsible for sourcing CXL.mem requests (reads, writes etc) and a CXL.mem Subordinate is the agent which is responsible for responding to CXL.mem requests (data, completions etc).

When the Subordinate is an Accelerator, CXL.mem protocol assumes the presence of a device coherency engine (DCOH). This agent is assumed to be responsible for implementing coherency related functions such as snooping of device caches based on CXL.mem commands and update of Meta Data fields. Support for memory with Meta

Data is optional but this needs to be negotiated with the Host in advance. The negotiation mechanisms is outside the scope of this specification. If Meta Data is not supported by device-attached memory, the DCOH will still need to use the Host supplied Meta Data updates to interpret the commands. If Meta Data is supported by device-attached memory, it can be used by Host to implement a coarse snoop filter for CPU sockets.

CXL.mem transactions from Master to Subordinate are called "M2S" and transactions from Subordinate to Master are called “S2M”.

Within M2S transactions, there are two message classes:

- Request without data - generically called Requests (Req)
- Request with Data - (RwD)

Similarly, within S2M transactions, there are two message classes:

- Response without data - generically called No Data Response (NDR)
- Response with data - generically called Data Response (DRS)

The next sections describe the above message classes and opcodes in detail.

3.3.2 M2S Request (Req)

The Req message class generically contains reads, invalidates and signals going from the Master to the Subordinate.

Table 21. M2S Request Fields

Field	Bits	Description
Valid	1	The valid signal indicates that this is a valid request
MemOpcode	4	Memory Operation – This specifies which, if any, operation needs to be performed on the data and associated information. Details in Table 22
MetaField	2	Meta Data Field – Up to 3 Meta Data Fields can be addressed. This specifies which, if any, Meta Data Field needs to be updated. Details of Meta Data Field in Table 23. If the Subordinate does not support memory with Meta Data, this field will still be used by the DCOH for interpreting Host commands as described in Table 24
MetaValue	2	Meta Data Value - When MetaField is not No-Op, this specifies the value the field needs to be updated to. Details in Table 24. If the Subordinate does not support memory with Meta Data, this field will still be used by the device coherence engine for interpreting Host commands as described in Table 24
SnpType	3	Snoop Type - This specifies what snoop type, if any, needs to be issued by the DCOH and the minimum coherency state required by the Host. Details in Table 25
Address[51:5]	47	This field specifies the Host Physical Address associated with the MemOpcode. Addr[5] is provisioned for future usages such as critical chunk first.
Tag	16	The Tag field is used to specify the source entry in the Master which is pre-allocated for the duration of the CXL.mem transaction. This value needs to be reflected with the response from the Subordinate so the response can be routed appropriately. The exceptions are the MemRdFwd and MemWrFwd opcodes as described in Table 22
TC	2	Traffic Class - This can be used by the Master to specify the Quality of Service associated with the request. This is reserved for future usage.
RSVD	10	Reserved
Total	**87**	

Table 22. M2S Req Memory Opcodes

Opcode	Description	Encoding
MemInv	Invalidation request from the Master. Primarily for Meta Data updates. No data read or write required. If SnpType field contains valid commands, perform required snoops.	‘0000
MemRd	Normal memory data read operation. If MetaField contains valid commands, perform Meta Data updates. If SnpType field contains valid commands, perform required snoops.	‘0001
MemRdData	Normal Memory data read operation. MetaField & MetaValue to be ignored. Instead, update Meta0-State as follows: If initial Meta0-State value = ‘I’, update Meta0-State value to ‘A’ Else, no update required If SnpType field contains valid commands, perform required snoops.	‘0010
MemRdFwd	This is an indication from the Host that data can be directly forwarded from device-attached memory to the device without any completion to the Host. This is typically sent as a result of a CXL.cache D2H read request to device-attached memory. The Tag field contains the reflected CQID sent along with the D2H read request. The SnpType is always NoOp for this Opcode. The caching state of the line is reflected in Meta0-State value.	‘0011
MemWrFwd	This is an indication from the Host to the device that it owns the line and can update it without any completion to the Host. This is typically sent as a result of a CXL.cache D2H write request to device-attached memory. The Tag field contains the reflected CQID sent along with the D2H write request. The SnpType is always NoOp for this Opcode. The caching state of the line is reflected in Meta0-State value.	‘0100
MemInvNT	This is similar to the MemInv command except that the NT is a hint that indicates the invalidation is non-temporal and the writeback is expected soon. However, this is a hint and not a guarantee.	‘1001
Reserved	Reserved	‘0110 ‘0111 ‘Others

Table 23. Meta Data Field Definition

Meta Field	Description	Encoding
Meta0 - State	Update the Metadata bits with the value in the Meta Data Value field. Details of MetaValue associated with Meta0-State in Table 24	00
Reserved	Reserved	01 10
No-Op	No meta data operation. Ignore value in MetaValue field	11

Table 24. Meta0-State Value Definition

Encoding	Description
2'b00	Invalid (I) - Indicates the Host does not have a cacheable copy of the line. The DCOH can use this information to grant exclusive ownership of the line to the device. When paired with a MemOpcode = MemInv and SnpType = SnpInv, this is used to communicate that the device should flush this line from its caches, if cached, to device-attached memory.
2'b10	Any (A) - Indicates the Host may have an shared, exclusive or modified copy of the line. The DCOH can use this information to interpret that the Host likely wants to update the line and the device should not be given a copy of the line without first sending a request to the Host.
2'b11	Shared (S) - Indicates the Host may have at most a shared copy of the line. The DCOH can use this information to interpret that the Host does not have an exclusive or modified copy of the line. If the device wants a shared or current copy of the line, the DCOH can provide this without sending a request to the Host. If the device wants an exclusive copy of the line, the DCOH will have to send a request to the Host first.
2'b01	Reserved

Table 25. Snoop Type Definition

SnpType Description	Description	Encoding
No-Op	No snoop needs to be performed	000
SnpData	Snoop may be required - the requester needs at least a Shared copy of the line. Device may choose to give an exclusive copy of line as well.	001
SnpCur	Snoop may be required - the requester needs the current value of the line. Requester guarantees the line will not be cached. Device need not change the state of the line in its caches, if present.	010
SnpInv	Snoop may be required - the requester needs an exclusive copy of the line.	011
Reserved		1xx

Valid M2S request semantics are described below.

Table 26. M2S Req Usage (Sheet 1 of 2)

M2S Req	Meta Field	Meta Value	SnpType	S2M NDR	S2M DRS	Description
MemRd	Meta0-State	A	SnpInv	Cmp-E	MemData	The Host wants an exclusive copy of the line
MemRd	Meta0-State	S	SnpData	Cmp-S or Cmp-E	MemData	The Host wants a shared copy of the line
MemRd	No-Op	NA	SnpCur	Cmp	MemData	The Host wants a non-cacheable but current value of the line
MemRd	No-Op	NA	SnpInv	Cmp	MemData	The Host wants a non-cacheable value of the line and the device should invalidate the line from its caches
MemInv	Meta0-State	A	SnpInv	Cmp-E	NA	The Host wants ownership of the line without data

Table 26. M2S Req Usage (Sheet 2 of 2)

M2S Req	Meta Field	Meta Value	SnpType	S2M NDR	S2M DRS	Description
MemInvNT	Meta0-State	A	SnpInv	Cmp-E	NA	The Host wants ownership of the line without data. However, the Host expects this to be non-temporal and may do a writeback soon.
MemInv	Meta0-State	I	SnpInv	Cmp	NA	The Host wants the device to invalidate the line from its caches
MemRdData	NA	NA	SnpData	Cmp-S or Cmp-E	MemData	The Host wants a cacheable copy in either exclusive or shared state

3.3.3 M2S Request with Data (RwD)

The Request with Data (RwD) message class generally contains writes from the Master to the Subordinate.

Table 27. M2S RwD Fields

Field	Bits	Description
Valid	1	The valid signal indicates that this is a valid request
MemOpcode	4	Memory Operation – This specifies which, if any, operation needs to be performed on the data and associated information. Details in Table 28
MetaField	2	Meta Data Field – Up to 3 Meta Data Fields can be addressed. This specifies which, if any, Meta Data Field needs to be updated. Details of Meta Data Field in Table 23. If the Subordinate does not support memory with Meta Data, this field will still be used by the DCOH for interpreting Host commands as described in Table 24
MetaValue	2	Meta Data Value - When MetaField is not No-Op, this specifies the value the field needs to be updated to. Details in Table 24. If the Subordinate does not support memory with Meta Data, this field will still be used by the device coherence engine for interpreting Host commands as described in Table 24
SnpType	3	Snoop Type - This specifies what snoop type, if any, needs to be issued by the DCOH and the minimum coherency state required by the Host. Details in Table 25
Address[51:6]	46	This field specifies the Host Physical Address associated with the MemOpcode.
Tag	16	The Tag field is used to specify the source entry in the Master which is pre-allocated for the duration of the CXL.mem transaction. This value needs to be reflected with the response from the Subordinate so the response can be routed appropriately.
TC	2	Traffic Class - This can be used by the Master to specify the Quality of Service associated with the request. This is reserved for future usage.
Poison	1	This indicates that the data contains an error. The handling of poisoned data is device specific. Please refer to the Chapter 12 for more details.
RSVD	10	
Total	**87**	

Table 28. M2S RwD Memory Opcodes

Opcode	Description	Encoding
MemWr	Memory write command. Used for full line writes. If MetaField contains valid commands, perform Meta Data updates. If SnpType field contains valid commands, perform required snoops. If the snoop hits a Modified cacheline in the device, the DCOH will invalidate the cache and write the data from the Host to device-attached memory.	‘0001
MemWrPtl	Memory Write Partial. Contains 64 byte enables, one for each byte of data. If MetaField contains valid commands, perform Meta Data updates. If SnpType field contains valid commands, perform required snoops. If the snoop hits a Modified cacheline in the device, the DCOH will need to perform a merge, invalidate the cache and write the contents back to device-attached memory.	‘0010
Reserved	Reserved	Others

The definition of other fields are consistent with M2S Req (refer to M2S Request (Req)). Valid M2S RwD semantics are described below.

Table 29. M2S RwD Usage

M2S Req	Meta Field	Meta Value	SnpType	S2M NDR	Description
MemWr	Meta0-State	I	No-Op	Cmp	The Host wants to write the line back to memory and does not retain a cacheable copy.
MemWr	Meta0-State	A	No-Op	Cmp	The Host wants to write the line back to memory and retains a cacheable copy in shared, exclusive or modified state.
MemWr	Meta0-State	I	SnpInv	Cmp	The Host wants to write the line back to memory and does not retain a cacheable copy. In addition, the Host did not get ownership of the line before doing this write and needs the device to snoop-invalidate its caches before doing the write back to memory.
MemWrPtl	Meta0-State	I	SnpInv	Cmp	Same as the above row except the data being written is partial and the device needs to merge the data if it finds a copy of the line in its caches.

3.3.4 S2M No Data Response (NDR)

The NDR message class contains completions and indications from the Subordinate to the Master.

Table 30. S2M NDR Fields

Field	Bits	Description
Valid	1	The valid signal indicates that this is a valid request
Opcode	3	Memory Operation – This specifies which, if any, operation needs to be performed on the data and associated information. Details in Table 31
MetaField	2	Meta Data Field – For devices that support memory with meta data, this is a reflection of the value sent in the associated M2S Req or M2S RwD. For devices that do not, this field is a don’t care.

Table 30. S2M NDR Fields

Field	Bits	Description
MetaValue	2	Meta Data Value – For M2S Req, for devices that support memory with meta data, this is the initial value of the Meta Data Field as read from memory for a M2S Req that does not return a S2M DRS. For M2S RwD and for devices that do not support memory with meta data, this field is a don't care.
Tag	16	Tag - This is a reflection of the Tag field sent with the associated M2S Req or M2S RwD.
RSVD	4	
Total	**28**	

Opcodes for the NDR message class are defined in the table below.

Table 31. S2M NDR Opcodes

Opcode	Description	Encoding
Cmp	Completions for Writebacks, Reads and Invalidates	'000
Cmp-S	Indication from the DCOH to the Host for Shared state	'001
Cmp-E	Indication from the DCOH to the Host for Exclusive ownership	'010

Definition of other fields are the same as for M2S message classes.

3.3.5 S2M Data Response (DRS)

The DRS message class contains memory read data from the Subordinate to the Master.

The fields of the DRS message class are defined in the table below.

Table 32. S2M DRS Fields

Field	Bits	Description
Valid	1	The valid signal indicates that this is a valid request.
Opcode	3	Memory Operation – This specifies which, if any, operation needs to be performed on the data and associated information. Details in Table 33.
MetaField	2	Meta Data Field – For devices that support memory with meta data, this is a reflection of the value sent in the associated M2S Req or M2S RwD. For devices that do not, this field is a don't care.
MetaValue	2	Meta Data Value – For M2S Req, for devices that support memory with meta data, this is the initial value of the Meta Data Field as read from memory. For M2S RwD and for devices that do not support memory with meta data, this field is a don't care.
Tag	16	Tag - This is a reflection of the Tag field sent with the associated M2S Req or M2S RwD.
Poison	1	This indicates that the data contains an error. The handling of poisoned data is Host specific. Please refer to the Chapter 12 for more details.
RSVD	15	
Total	**40**	

Table 33. S2M DRS Opcodes

Opcode	Description	Encoding
MemData	Memory read data. Sent in response to Reads.	'000

3.3.6 Forward Progress & Ordering Rules

- Req & RwD message classes, each, need to be credited independently between each hop in a multi-hop fabric. Back pressure, due to lack of resources at the destination, is allowed. However, these must eventually drain without dependency on any other traffic type.
- No transaction should pass a MemRdFwd or a MemWrFwd, if the transaction and MemRdFwd or MemWrFwd are to the same cacheline address.
  Reason: As described in Table 22, MemRdFwd and MemWrFwd opcodes, sent on the Req message class are, in fact, responses to CXL.cache D2H requests. The reason the response for certain CXL.cache D2H requests are on CXL.mem Req channel is to ensure subsequent requests from the Host to the same address remain ordered behind it. This allows the host and device to avoid race conditions. An example of a transaction flow is shown Figure 37.
- Apart from the above, there is no ordering requirement for the Req, RwD, NDR & DRS message classes or for different addresses within the Req message class.
- NDR & DRS message classes, each, need to be pre-allocated at the source. This guarantees that the responses can sink and ensures forward progress.
- On CXL.mem, a strongly ordered write request needs to be completed before another transaction is issued to the same address.
- CXL.mem requests need to make forward progress at the device without any dependency on any device initiated request. This includes any request from the device on CXL.io or CXL.cache.
- M2S & S2M Data transfer of a cache line must occur with no interleaved transfers from other lines. The data must come in natural chunk order, that is, 64B transfers must complete the lower 32B half first.

3.4 Transaction Flows to Device-Attached Memory

3.4.1 Flows for Type 1 and Type 2 Devices

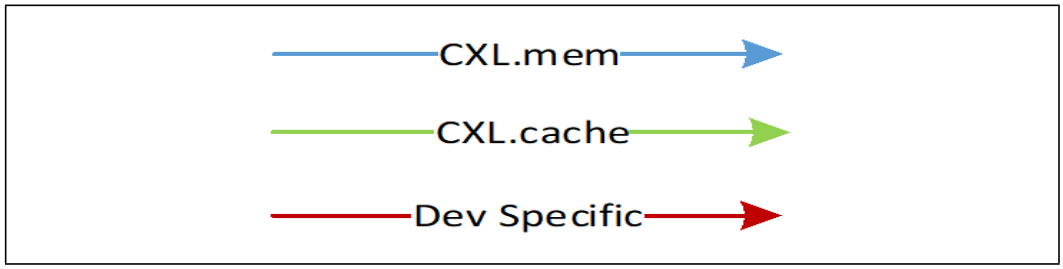
3.4.1.1 Notes and Assumptions

The transaction flow diagrams below are intended to be illustrative of the flows between the Host and device for access to device-attached Memory using the Bias Based Coherency mechanism described in Section 2.0. However, these flows are not comprehensive of every Host and device interaction. The diagrams below make the following assumptions:

- The device contains a coherency engine which is called DCOH in the diagrams below.
- The DCOH contains a Snoop Filter which tracks any caches (called Dev cache) implemented on the device. This is not strictly required and the device is free to choose an implementation specific mechanism as long as the coherency rules are obeyed.
- The DCOH contains a Bias Table lookup mechanism. The implementation of this is device specific.

- The device specific aspects of the flow, illustrated using Red flow arrows, need not conform exactly to the pictures below. These can be implemented in a device specific manner.

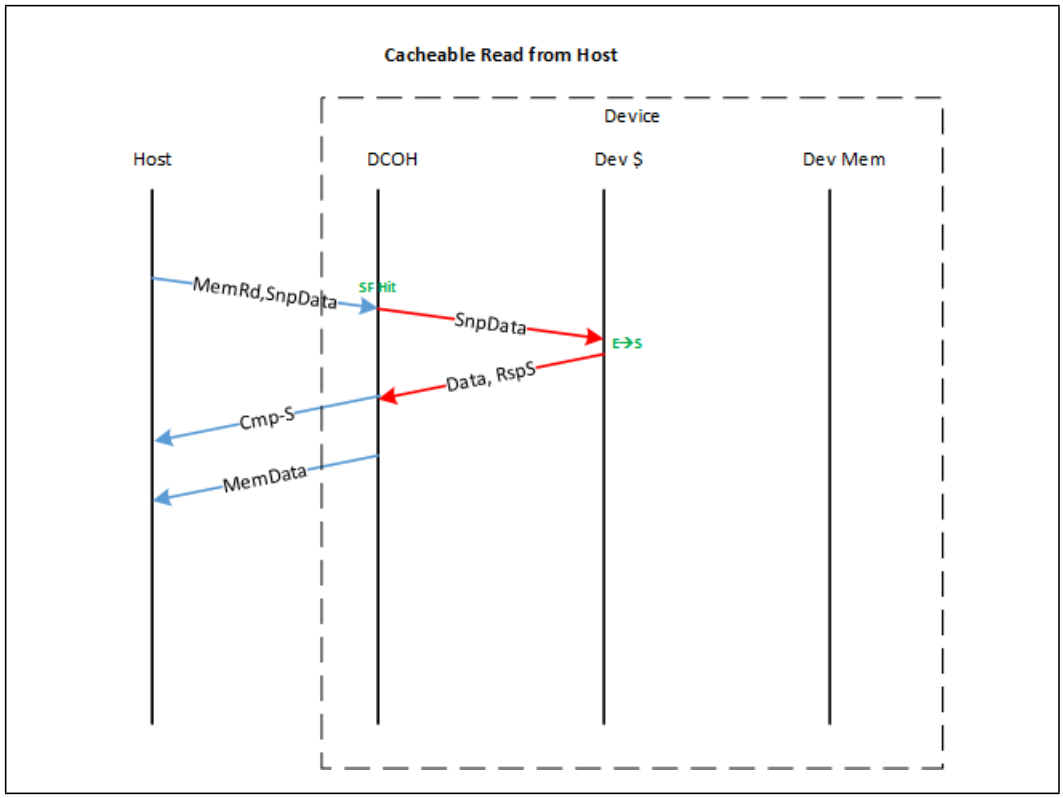
Figure 25. Legend



3.4.1.2 Requests from Host

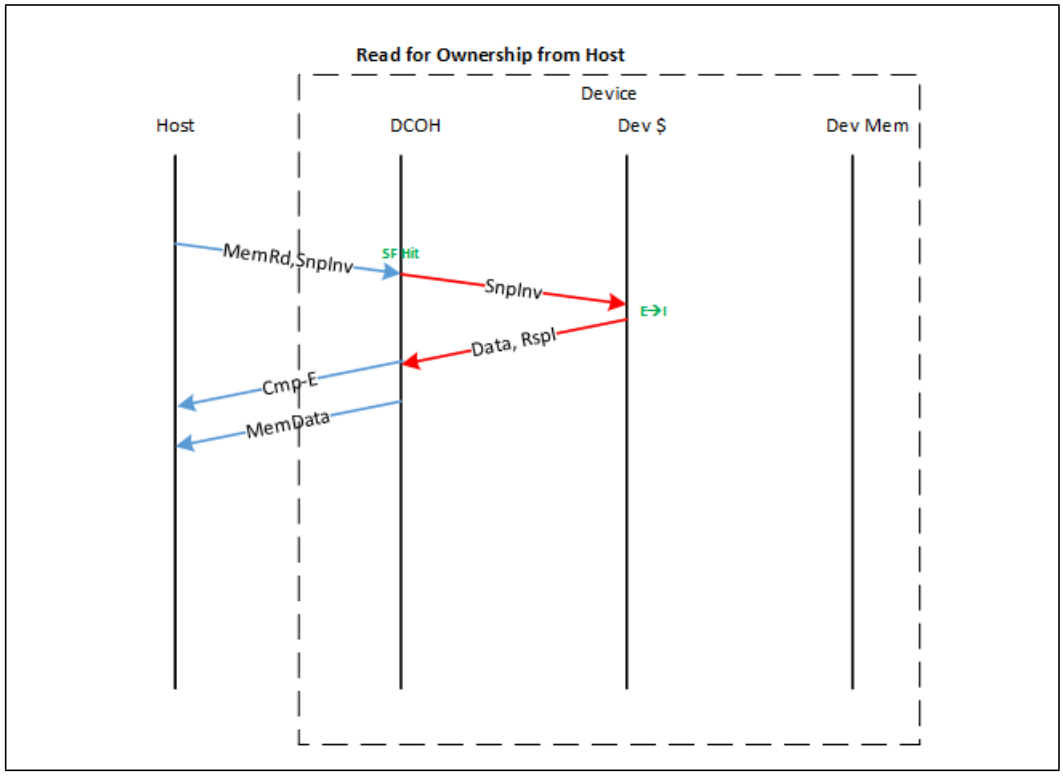
Please note that the flows shown in this section (Requests from Host) do not change on the CXL interface regardless of the bias state of the target region. This effectively means that the device needs to give the Host a consistent response, as expected by the Host and shown below.

Figure 26. Example Cacheable Read from Host



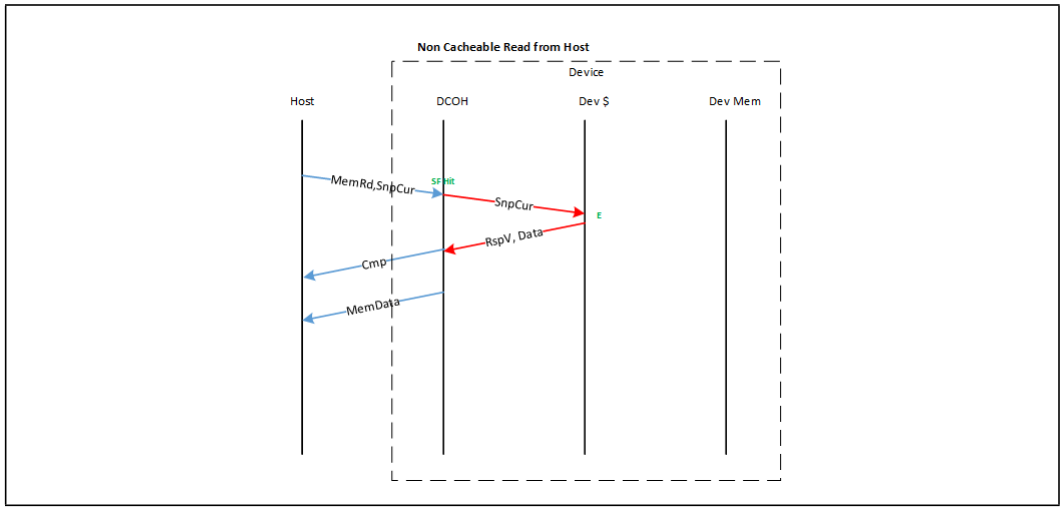
In the above example, the Host requested a cacheable non-exclusive copy of the line. The non-exclusive aspect of the request is communicated using the “SnpData” semantic. In this example, the request got a snoop filter hit in the DCOH, which caused the device cache to be snooped. The device cache downgraded the state from Exclusive to Shared and returned the Shared data copy to the Host. The Host is told of the state of the line using the Cmp-S semantic.

Figure 27. Example Read for Ownership from Host



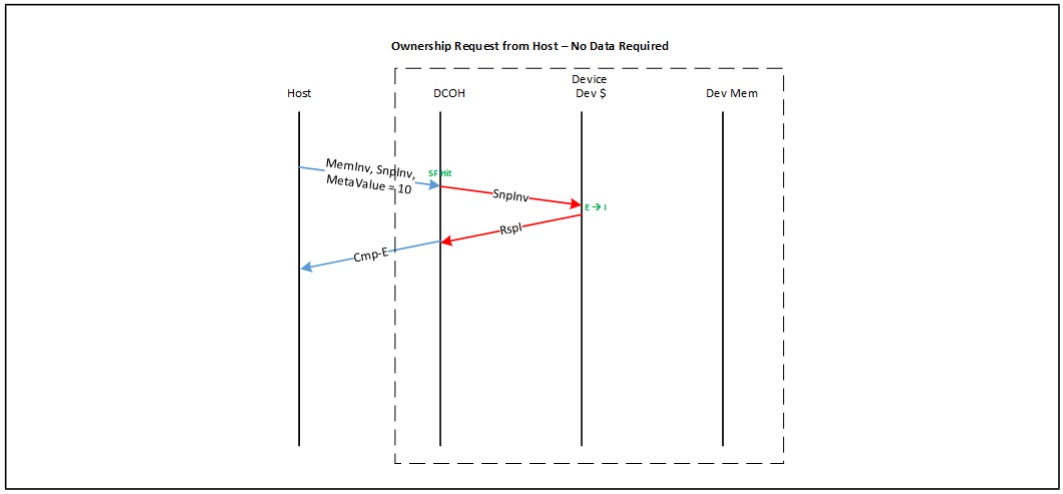
In the above example, the Host requested a cacheable exclusive copy of the line. The exclusive aspect of the request is communicated using the “SnpInv” semantic, which asks the device to invalidate its caches. In this example, the request got a snoop filter hit in the DCOH, which caused the device cache to be snooped. The device cache downgraded the state from Exclusive to Invalid and returned the Exclusive data copy to the Host. The Host is told of the state of the line using the Cmp-E semantic.

Figure 28. Example Non Cacheable Read from Host



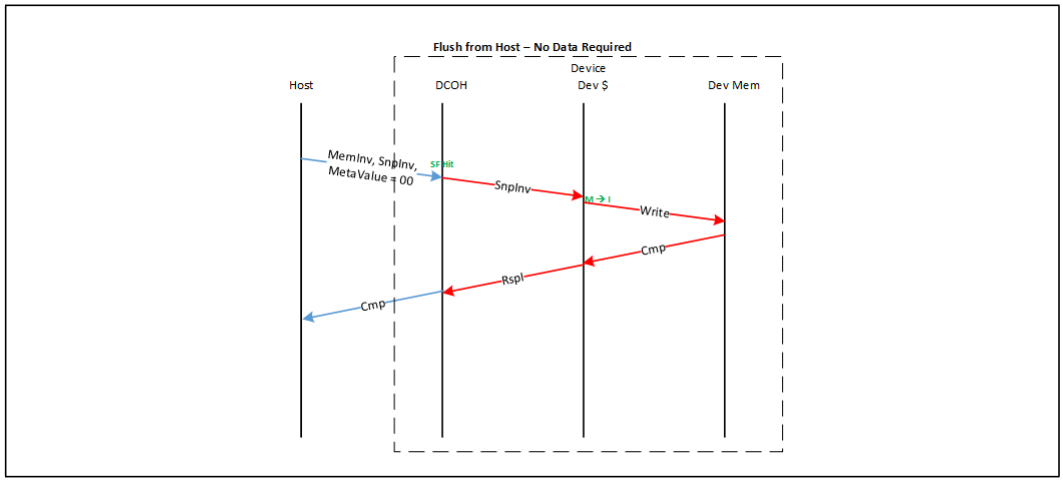
In the above example, the Host requested a non-cacheable copy of the line. The non-cacheable aspect of the request is communicated using the "SnpCurr" semantic. In this example, the request got a snoop filter hit in the DCOH, which caused the device cache to be snooped. The device cache did not need to change its caching state; however, it gave the current snapshot of the data. The Host is told that it is not allowed to cache the line using the Cmp semantic.

Figure 29. Example Ownership Request from Host - No Data Required



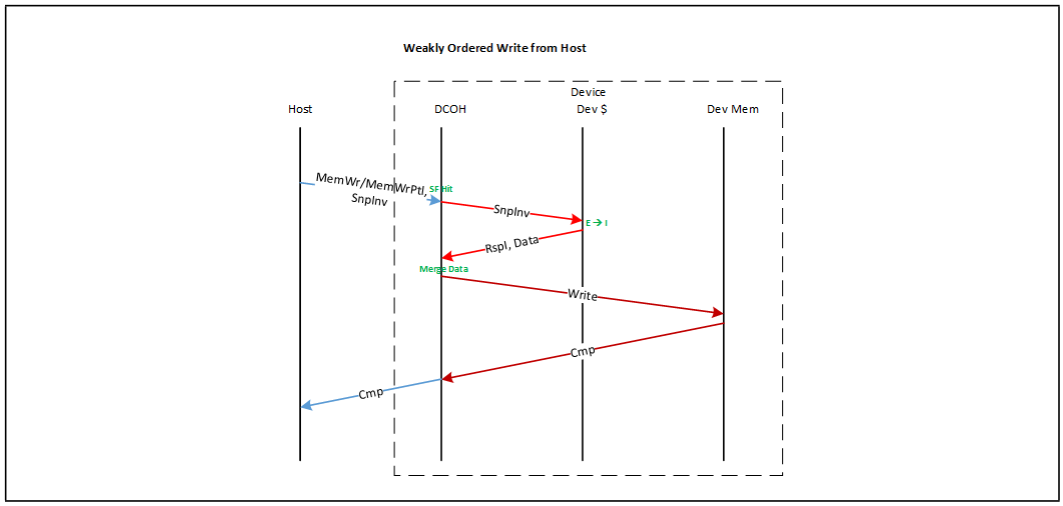
In the above example, the Host requested exclusive access to a line without requiring the device to send data. It communicates that to the device using an opcode of MemInv with a MetaValue of '10 (Any), which is significant in this case. It also asks the device to invalidate its caches with the SnpInv command. The device invalidates its caches and gives exclusive ownership to the Host as communicated using the Cmp-E semantic.

Figure 30. Example Flush from Host



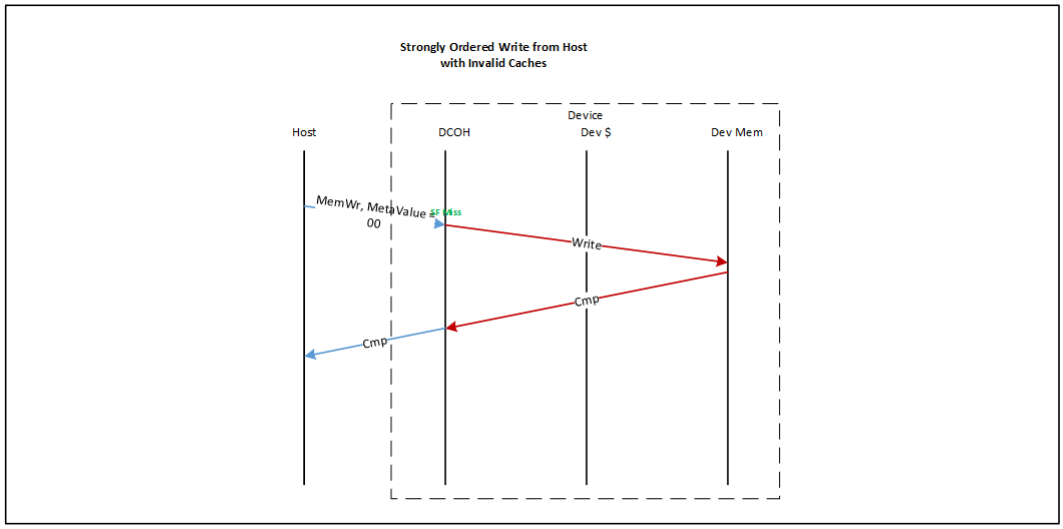
In the above example, the Host wants to flush a line from all caches, including the device's caches, to memory. To do so, it uses an opcode of MemInv with a MetaValue of '00 (Invalid) and a SnpInv. The device flushes its caches and returns a Cmp indication to the Host.

Figure 31. Example Weakly Ordered Write from Host



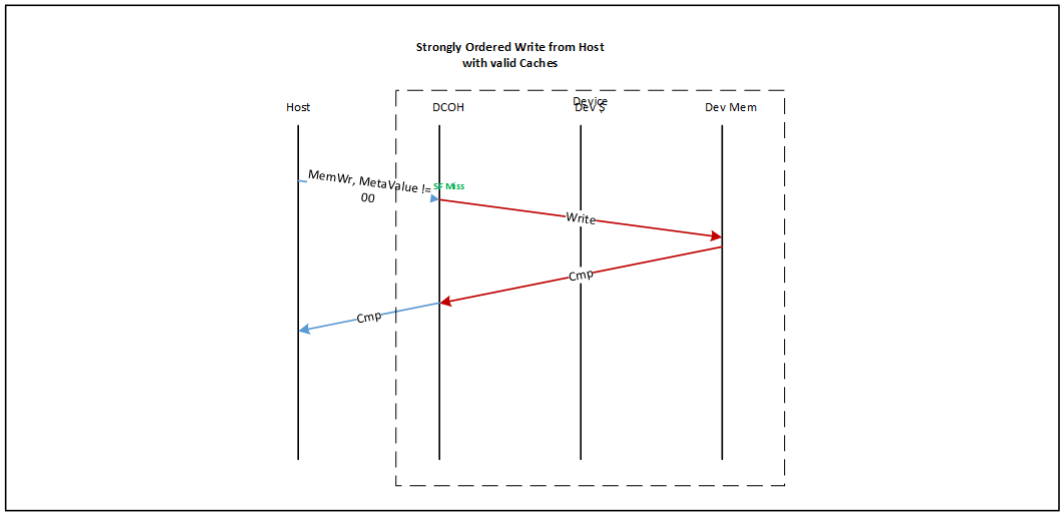
In the above example, the Host issues a weakly ordered write (partial or full line). The weakly ordered semantic is communicated by the embedded SnpInv. In this example, the device had a copy of the line cached. This resulted in a merge within the device before writing it back to memory and sending a Cmp indication to the Host.

Figure 32. Example Strongly Ordered Write from Host with Invalid Host Caches



In the above example, the Host performed a strongly ordered write while guaranteeing to the device that it no longer has a valid cached copy of the line. The strong ordering is demonstrated by the fact that the Host didn't need to snoop the device's caches which means it previously acquired an exclusive copy of the line. The guarantee on no valid cached copy is indicated by a MetaValue of '00 (Invalid).

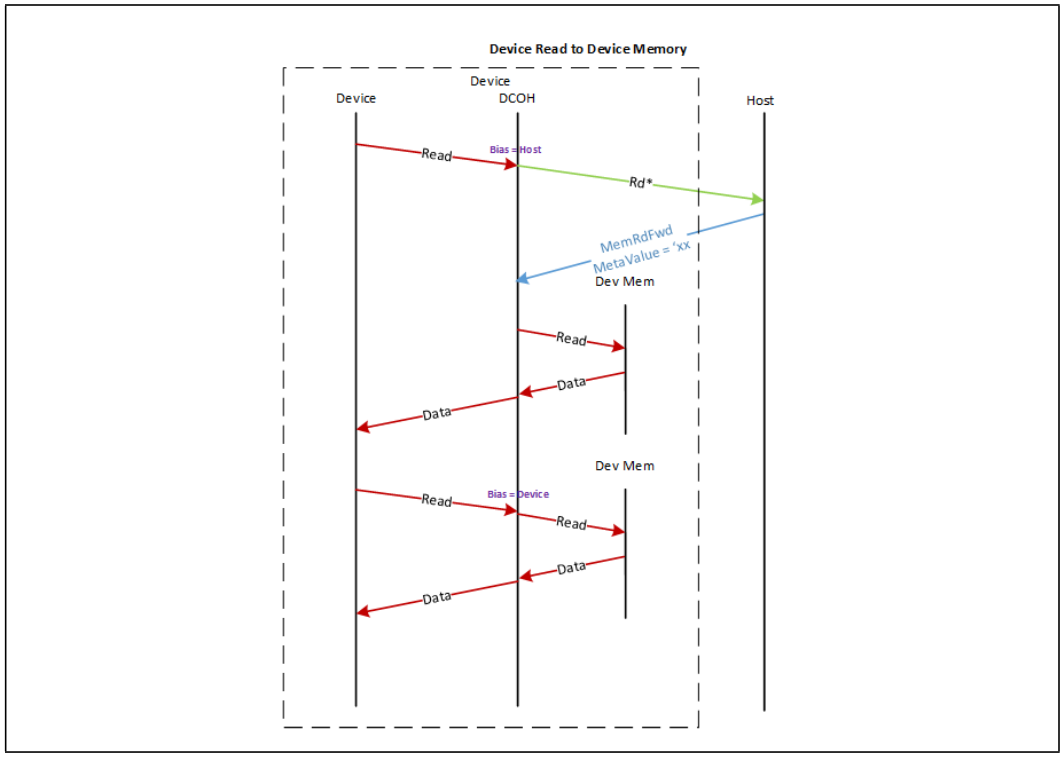
Figure 33. Example Strongly Ordered Write from Host with Valid Caches



The above example is the same as the previous one except that the Host chose to retain a valid cacheable copy of the line after the write. This is communicated to the device using a MetaValue of not '00 (Invalid).

3.4.1.3 Requests from Device in Host & Device Bias

Figure 34. Example Device Read to Device-Attached Memory

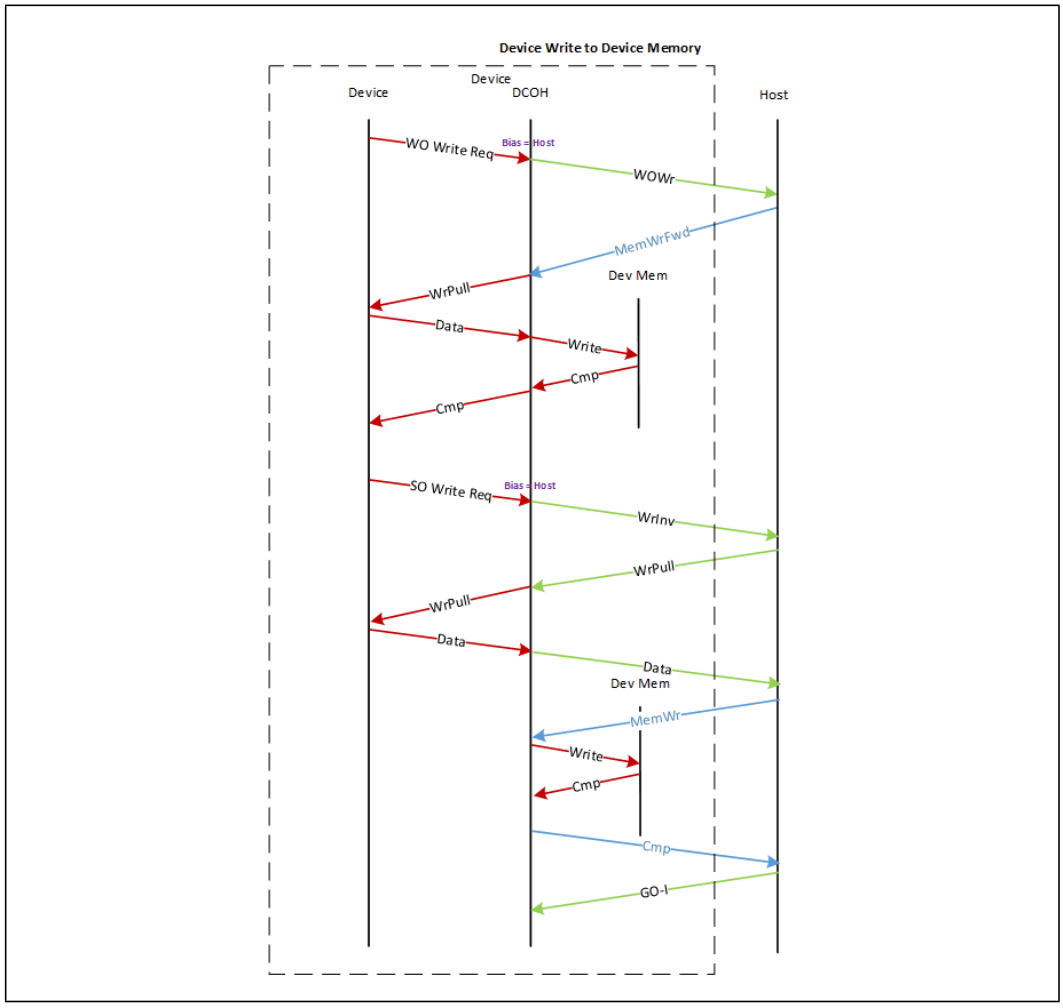


There are two flows shown above.

In the first one, a device read to device attached memory happened to find the line in Host bias. Since it is in Host bias, the device needs to send the request to the Host to resolve coherency. The Host, after resolving coherency, sends a MemRdFwd on CXL.mem to complete the transaction, at which point the device can complete the read internally.

In the second flow, the device read happened to find the line in Device Bias. Since it is in Device Bias, the read can be completed entirely within the device itself and no request needs to be sent to the Host.

Figure 35. Example Device Write to Device-Attached Memory in Host Bias

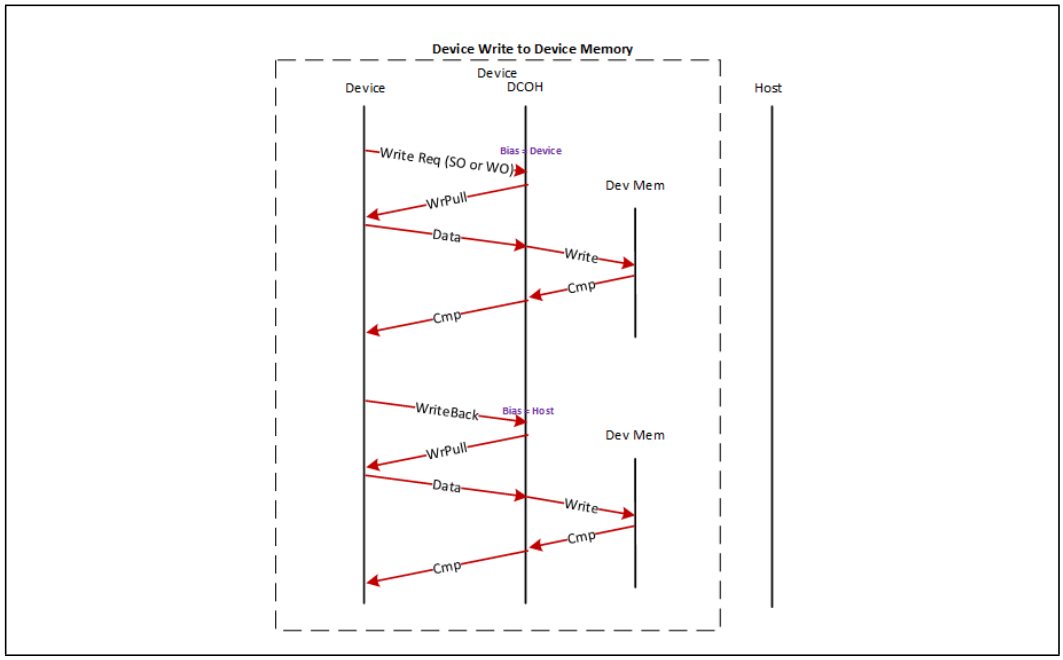


There are two flows shown above, both with the line in Host Bias: a weakly ordered write request and a strongly ordered write request.

In the case of the weakly ordered write request, the request is issued by the device to the Host to resolve coherency. The Host resolves coherency and sends a CXL.mem MemWrFwd opcode which carries the completion for the WOWrInv* command on CXL.cache. The CQID associated with the CXL.cache WOWrInv* command is reflected in the Tag of the CXL.mem MemWrFwd command. At this point, the device is allowed to complete the write internally. After sending the MemWrFwd, since the Host no longer fences against other accesses to the same line, this is considered a weakly ordered write.

In the second flow, the write is strongly ordered. To preserve the strongly ordered semantic, the Host fences against other accesses while this write completes. However, as can be seen, this involves two transfers of the data across the link, which is not efficient. Unless strongly ordered writes are absolutely required, better performance can be achieved with weakly ordered writes.

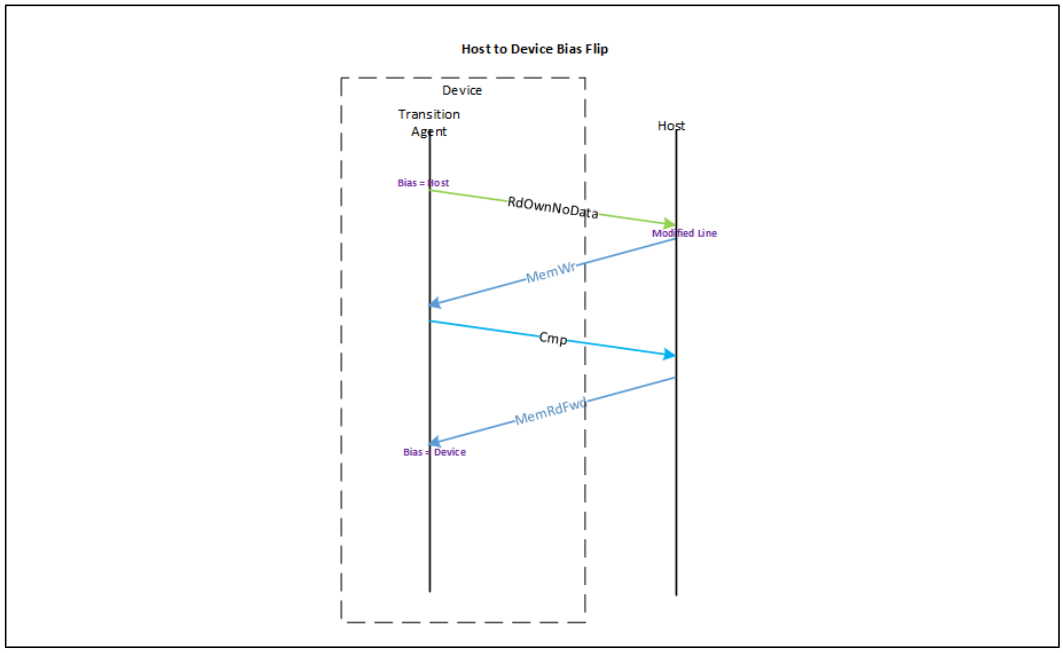
Figure 36. Example Device Write to Device-Attached Memory



Again, two flows are shown above. In the first case, if a weakly or strongly ordered write finds the line in Device Bias, the write can be completed entirely within the device without having to send any indication to the Host.

The second flow shows a device writeback to device-attached memory. Please note that if the device is doing a writeback to device-attached memory, regardless of bias state, the request can be completed within the device without having to send a request to the Host.

Figure 37. Example Host to Device Bias Flip

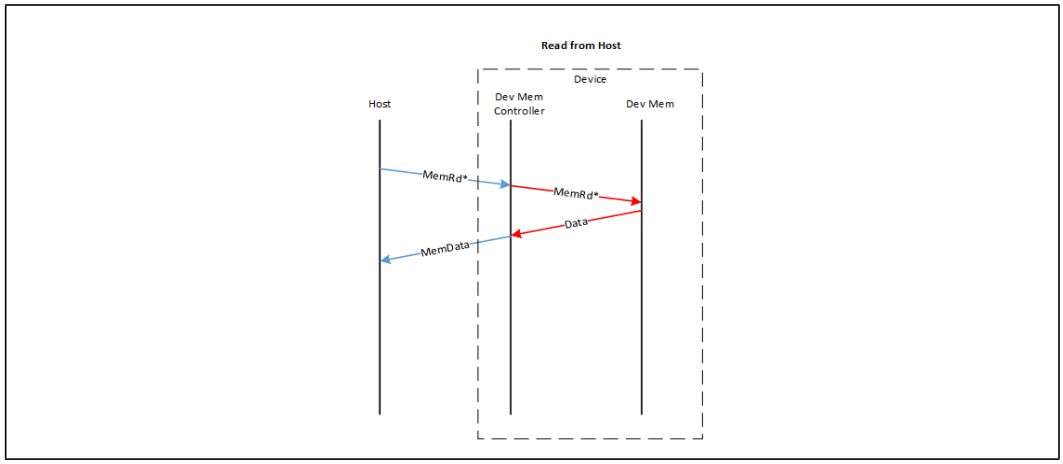


Please note that the MemRdFwd will carry the CQID of the RdOwnNoData transaction in the Tag. The reason for putting the RdOwnNoData completion (MemRdFwd) on CXL.mem is to ensure that subsequent requests from the Host to the same address are ordered behind the MemRdFwd. This allows the device to assume ownership of a line as soon as it receives a MemRdFwd without having to monitor requests from the Host.

3.5 Flows for Type 3 Devices

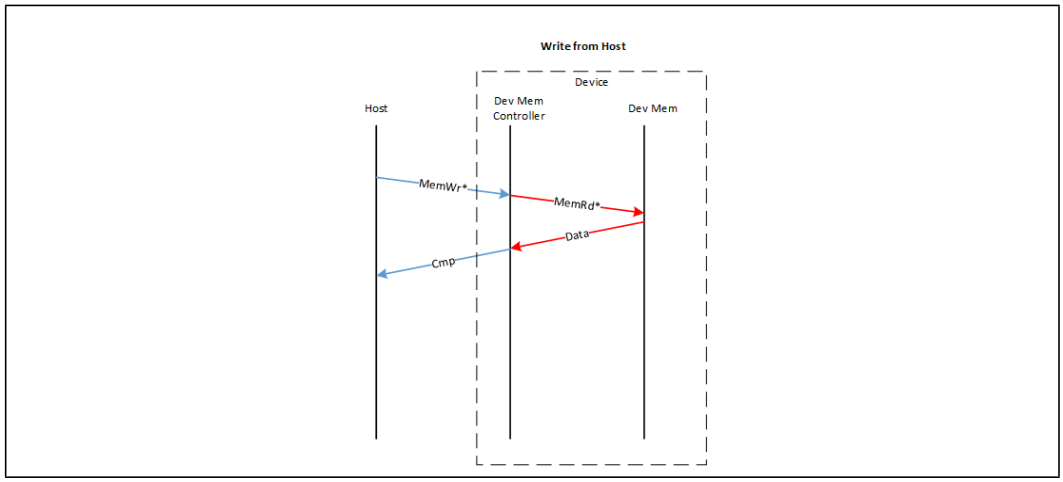
Type 3 devices are memory expanders which neither cache host memory, nor require active management of a device cache by the Host. Thus, Type 3 devices do not have a DCOH agent. As such, the Host treats these devices as disaggregated memory controllers. This allows the transaction flows to Type 3 devices to be simplified to just two classes, reads and writes, as shown below. The legend shown in Figure 25 also applies to the transaction flows shown below.

Figure 38. Read from Host



The key difference between reads to Type 1 and Type 2 devices versus Type 3 devices is that there is no S2M NDR associated with it. Writes to Type 3 device always complete with a S2M NDR Cmp message.

Figure 39. Write from Host



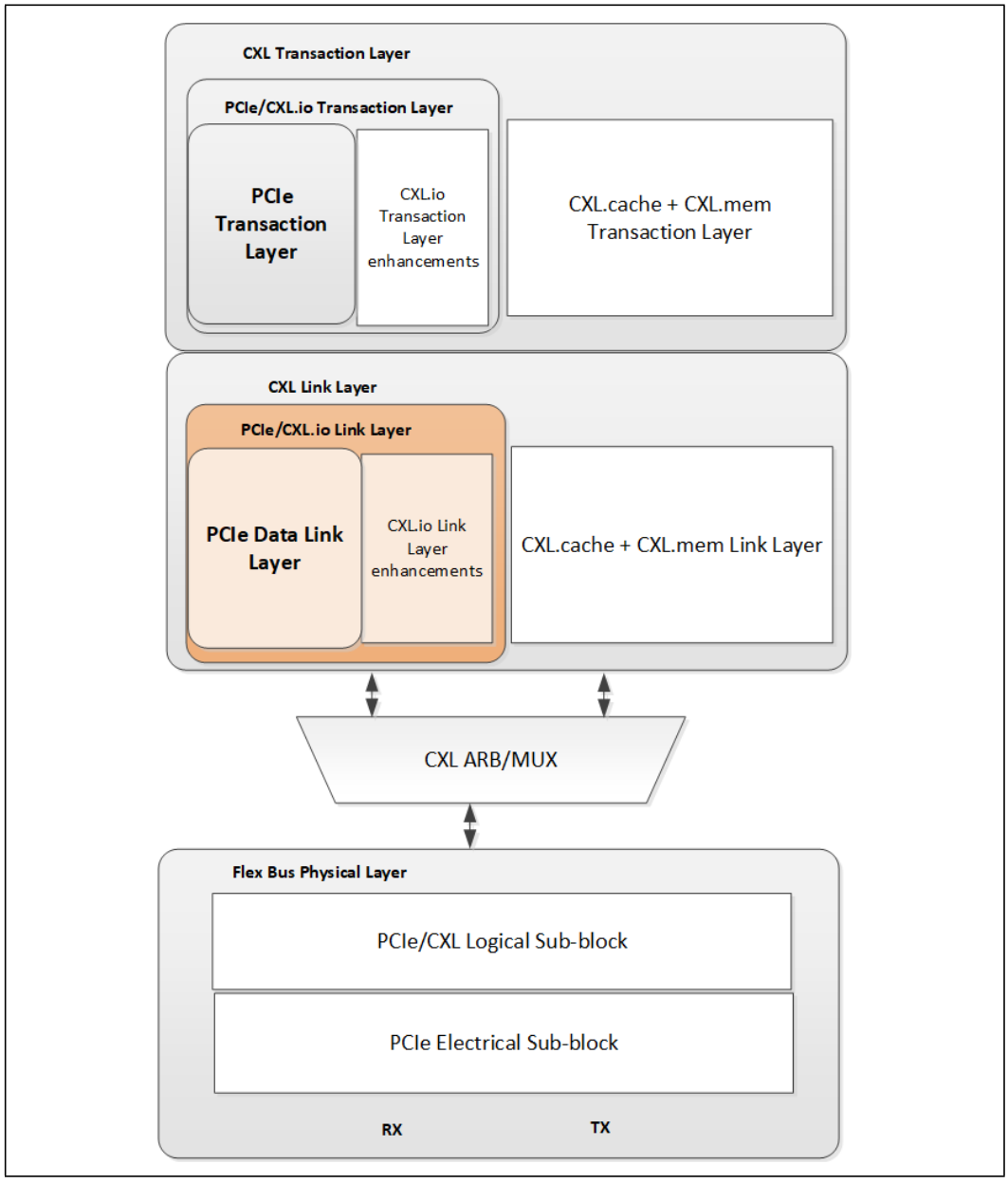
§ §

4.0 Compute Express Link Link Layers

4.1 CXL.io Link Layer

The CXL.io link layer acts as an intermediate stage between the CXL.io transaction layer and the Flex Bus Physical layer. Its primary responsibility is to provide a reliable mechanism for exchanging transaction layer packets (TLPs) between two components on the link. The PCIe Data Link Layer is utilized as the link layer for CXL.io Link layer. Please refer to chapter titled “Data Link Layer Specification” in PCI Express Base Specification for details.

Figure 40. Flex Bus Layers -- CXL.io Link Layer Highlighted



In addition, the CXL.io link layer implements the framing/deframing of CXL.io packets. CXL.io utilizes the Encoding for 8.0 GT/s and Higher data rates only, refer to section entitled “Encoding for 8.0GT/s and Higher Data Rates” in the PCI Express Base Specification for details.

This chapter highlights the notable framing and application of symbols to lanes that are specific for CXL.io. Note that when viewed on the link, the framing symbol to lane mapping will be shifted due to additional CXL framing (i.e., two bytes of Protocol ID and two reserved bytes) and also due to interleaving with other CXL protocols.

For CXL.io, only the x16 Link transmitter and receiver framing requirements described in the PCI Express Base Specification apply irrespective of the negotiated link width. The framing related rules for N = 1, 2, 4 and 8 do not apply. For downgraded Link widths, where number of active lanes is less than x16, a single x16 data stream is formed using x16 framing rules and transferred over x16/(degraded link width) degraded link width streams.

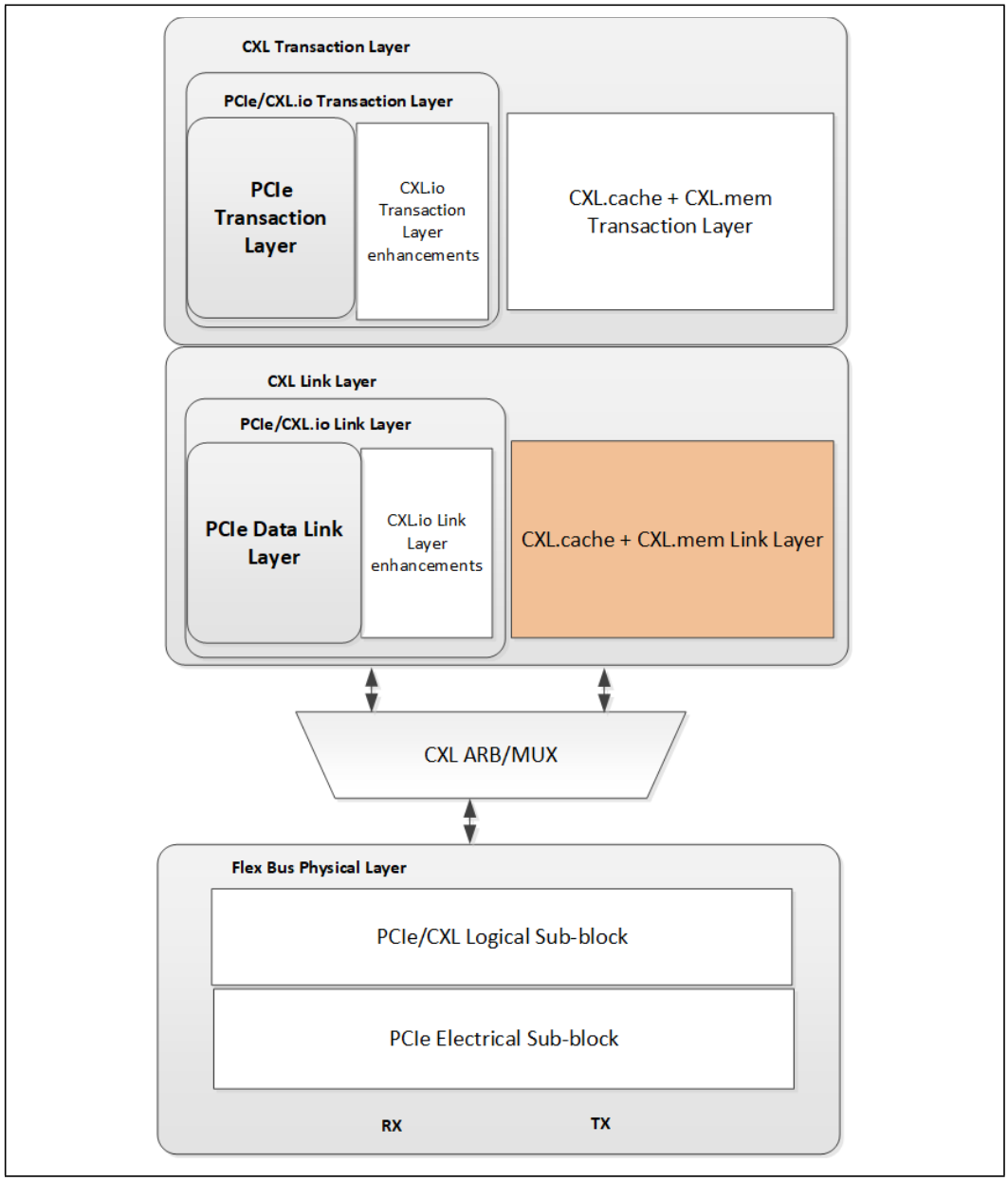
CXL.io link layer forwards a framed IO packet to the Flex Bus Physical layer. The Flex Bus Physical layer framing rules are defined in Section 6.0.

4.2 CXL.mem and CXL.cache Common Link Layer

4.2.1 Introduction

The figure below shows where the CXL.cache and CXL.mem link layer exists in the Flex Bus layered hierarchy.

Figure 41. Flex Bus Layers -- CXL.cache + CXL.mem Link Layer Highlighted



As previously mentioned, CXL.cache & CXL.mem protocols use a common Link Layer. This chapter defines the properties of this common Link Layer. Protocol information, including definition of fields, opcodes, transaction flows etc can be found in Section 3.2 and Section 3.3.

4.2.2 High-Level CXL.cache/CXL.mem Flit Overview

The CXL.cache/mem flit size is a fixed 528b. There are 2B of CRC code and 4 slots of 16B each as shown below.

Figure 42. CXL.cache / .mem Protocol Flit Overview

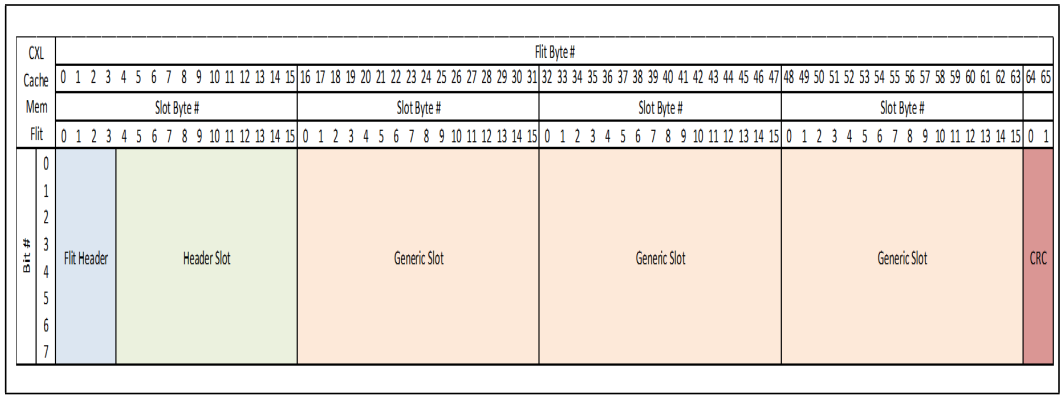
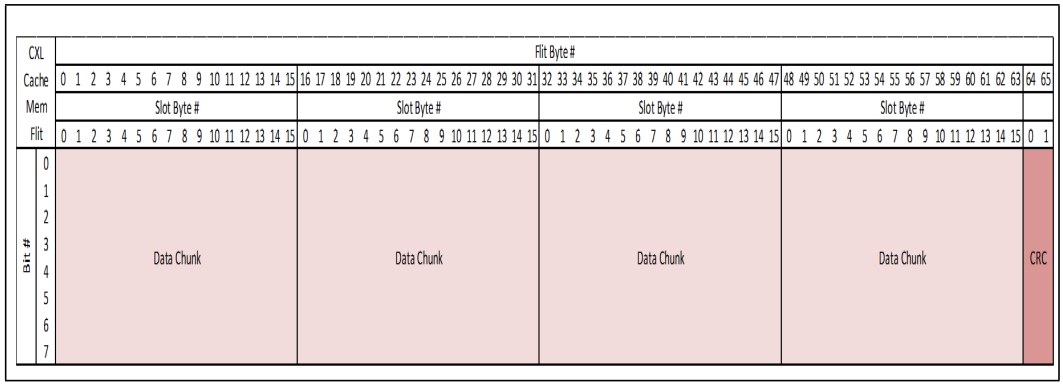
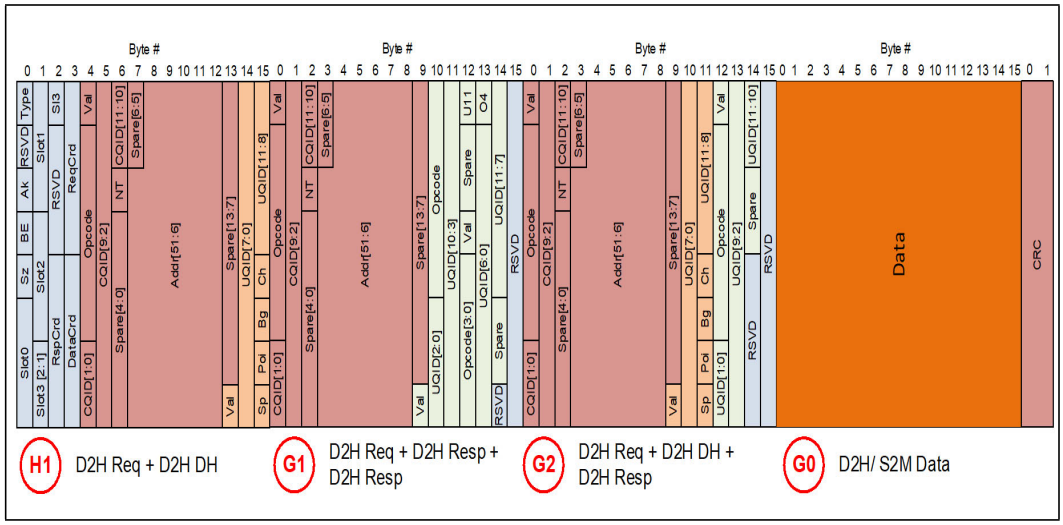


Figure 43. CXL.cache / .mem All Data Flit Overview



An example of a Protocol Flit in the device to Host direction is shown below. For detailed descriptions of slot formats please refer to Section 4.2.3

Figure 44. Example of a Protocol Flit from device to Host



A “Header” Slot is defined as one that carries a “Header” of link-layer specific information, including the definition of the protocol-level messages contained in the rest of the header as well as in the other slots in the flit.

A “Generic Request/Response Slot” is defined as one that holds one or more small CXL.cache messages.

A “Generic Data Slot” carries just 16B of data. A 64B Cache-line is transferred with 4 such generic data slots.

The flit can be composed of a Header Slot and 3 Generic Slots or possibly with only Generic data Slots.

The flit header utilizes the same definition for both the Upstream as well as the Downstream ports summarized in the table below.

Table 34. CXL.cache/CXL.mem Flit Header Definition

Field Name	Brief Description	Size
Flit Type	This field distinguishes between a Protocol or a Control Flit	1
Acknowledgment	This is an acknowledgment of 8 successful flit transfers	1
BE	Byte Enable	1
Sz	Size	1
ReqCrd	Request Credit Return	4
DataCrd	Data Credit Return	4
RspCrd	Response Credit Return	4
Slot 0	Slot 0 Format Type	3
Slot 1	Slot 1 Format Type	3
Slot 2	Slot 2 Format Type	3

Table 34. CXL.cache/CXL.mem Flit Header Definition

Field Name	Brief Description	Size
Slot 3	Slot 3 Format Type	3
RSVD	Reserved	4
Total		32

In general, bits or encodings that are not defined will be marked “Reserved” or “RSVD” in this specification. These bits should be set to 0 by the sender of the packet and the receiver should ignore them. Please also note that certain fields with static 0/1 values will be checked by the receiving Link Layer when decoding a packet. Checking of these bits reduces the probability of silent error under conditions where the CRC check fails to detect a long burst error. For example, LLCTRL (control flits) have several static bits defined. A LLCTRL flit that passes the CRC check but fails the static bit check should be treated as a fatal error. Logging and reporting of such errors is device specific.

The following describes how the flit header information is encoded.

Table 35. Flit Type Encoding

	Flit Type	Description
0	Protocol	This is a flit that carries CXL.cache or CXL.mem protocol related information
1	Control	This is a flit inserted by the link layer purely for link layer specific functionality. These flits are not exposed to the upper layers.

The **Acknowledgment** field is used as part of the link layer retry protocol to signal CRC-passing receipt of flits from the remote transmitter. The transmitter sets the Ak bit to acknowledge successful receipt of 8 flits; a clear Ak bit is ignored by the receiver.

The **Byte Enable** and **Size** fields have to do with the variable size of data messages. To reach its efficiency targets, the CXL.cache/mem link layer assumes that generally all bytes are enabled for most data, and that data is transmitted at the full cache line granularity. When all bytes are enabled, the link layer does not transmit the byte enable bits, but instead clears the Byte Enable field of the corresponding flit header. When the receiver decodes that the Byte Enable field is clear, it must regenerate the byte enable bits as all ones before passing the data message on to the transaction layer. If the Byte Enable bit is set, the link layer Rx expects an additional data chunk slot containing byte enable information. Note that this will always be the last slot of data sent.

Similarly, the **Size** field reflects the fact that the CXL.cache protocol allows transmission of data at the half cache line granularity. When the Size bit is set, the link layer Rx expects four slots of data chunks, corresponding to a full cache line. When the Size bit is clear, it expects only two slots of data chunks. In the latter case, each half cache line transmission will be accompanied by its own data header. A critical assumption of packing the Size and Byte Enable information in the flit header is that the Tx flit packer may begin at most one data message per flit.

The following table describes legal values of Sz and BE for various data transfers.

Table 36. Legal values of Sz & BE Fields

Type of Data Transfer	32B Transfer Possible?	BE Possible?
CXL.cache H2D Data	Yes	No

Table 36. Legal values of Sz & BE Fields

Type of Data Transfer	32B Transfer Possible?	BE Possible?
CXL.mem M2S Data	No	Yes
CXL.cache D2H Data	Yes	Yes
CXL.mem S2M Data	Yes	No

The transmitter sets the Credit Return fields to indicate resources available in the co-located receiver for use by the remote transmitter. Credits are given for transmission per message class, which is why the flit header contains independent Request, Response, and Data Credit Return fields. The granularity of credits is per transfer. For data transfers, this means 1 credit allows for one data transfer, regardless of whether the transfer is 64B, 32B or contains Byte Enables. These fields are encoded exponentially, as delineated in the table below.

Table 37. CXL.cache/CXL.mem Credit Return Encodings

Credit Return Encoding[3]	Protocol
0	CXL.cache
1	CXL.mem
Credit Return Encoding[2:0]	**Number of Credits**
000	0
001	1
010	2
011	4
100	8
101	16
110	32
111	64

Finally, the Slot Format Type fields encode the Slot Format of both the header itself and of the other generic slots in the flit (if the Flit Type bit specifies that the flit is a Protocol Flit). The subsequent sections detail the protocol message contents of each slot format, but the table below provides a quick reference for the Slot Format field encoding.

Table 38. Slot Format Field Encoding (Sheet 1 of 2)

Slot Format Encoding	H2D/M2S		D2H/S2M	
	Slot 0	Slots 1,2 and 3	Slot 0	Slots 1, 2 and 3
000	H0	G0	H0	G0
001	H1	G1	H1	G1
010	H2	G2	H2	G2
011	H3	G3	H3	G3
100	H4	G4	H4	G4

Table 38. Slot Format Field Encoding (Sheet 2 of 2)

Slot Format Encoding	H2D/M2S		D2H/S2M	
101	H5	G5	H5	G5
110	RSVD	RSVD	RSVD	G6
111	RSVD	RSVD	RSVD	RSVD

The following tables describe the actual slot format and the type of message contained by each format for both directions.

Table 39. H2D/M2S Slot Formats

Format to Req Type Mapping	H2D/M2S	
	Type	Size
H0	CXL.cache Req + CXL.cache Resp	96
H1	CXL.cache Data Header + 2 CXL.cache Resp	88
H2	CXL.cache Req + CXL.cache Data Header	88
H3	4 CXL.cache Data Header	96
H4	CXL.mem RwD Header	87
H5	CXL.mem Req Only	87
G0	CXL.cache/ CXL.mem Data Chunk	128
G1	4 CXL.cache Resp	128
G2	CXL.cache Req + CXL.cache Data Header + CXL.cache Resp	120
G3	4 CXL.cache Data Header + CXL.cache Resp	128
G4	CXL.mem Req + CXL.cache Data Header	111
G5	CXL.mem RwD Header + CXL.cache Resp	119

Table 40. D2H/S2M Slot Formats (Sheet 1 of 2)

Format to Req Type Mapping	D2H/S2M	
	Type	Size
H0	CXL.cache Data Header + 2 CXL.cache Resp + CXL.mem NDR	85
H1	CXL.cache Req + CXL.cache Data Header	96
H2	4 CXL.cache Data Header + CXL.cache Resp	88
H3	CXL.mem DRS Header + CXL.mem NDR	68
H4	2 CXL.mem NDR	56
H5	2 CXL.mem DRS Header	80
G0	CXL.cache/ CXL.mem Data Chunk	128
G1	CXL.cache Req + 2 CXL.cache Resp	119

Table 40. D2H/S2M Slot Formats (Sheet 2 of 2)

Format to Req Type Mapping	D2H/S2M	
G2	CXL.cache Req + CXL.cache Data Header + CXL.cache Resp	116
G3	4 CXL.cache Data Header	68
G4	CXL.mem DRS Header + 2 CXL.mem NDR	96
G5	3 CXL.mem NDR	84
G6	3 CXL.mem DRS Header	120

4.2.3 Slot Format Definition

4.2.3.1 RSVD Fields

Flit, slot and message bits that are not defined will be marked “RSVD” in this specification. RSVD bits should be set to 0 by the sender and the receiver should ignore them.

4.2.3.2 H2D & M2S Formats

Figure 45. H0 - H2D Req + H2D Resp

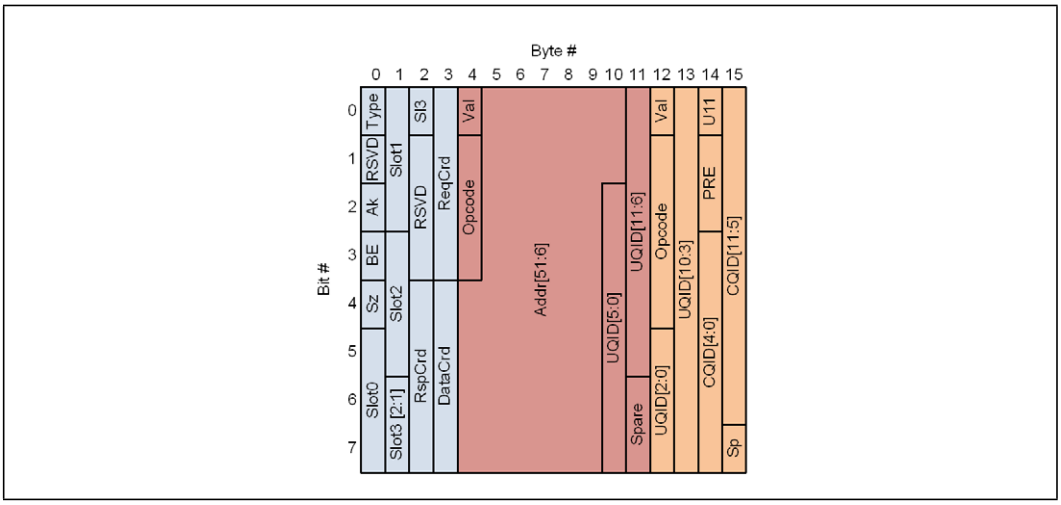


Figure 46. H1 - H2D Data Header + H2D Resp + H2D Resp

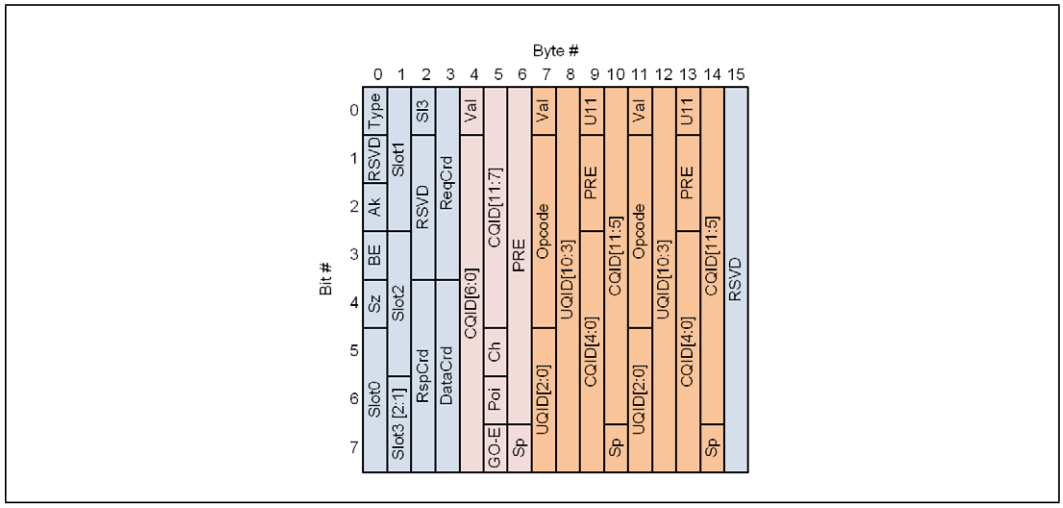


Figure 47. H2 - H2D Req + H2D Data Header

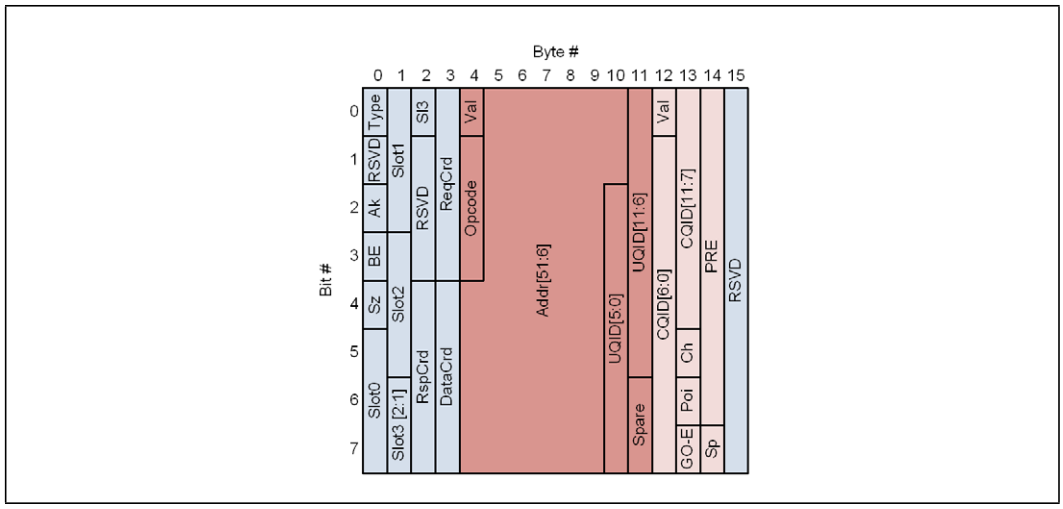


Figure 48. H3 - 4 H2D Data Header

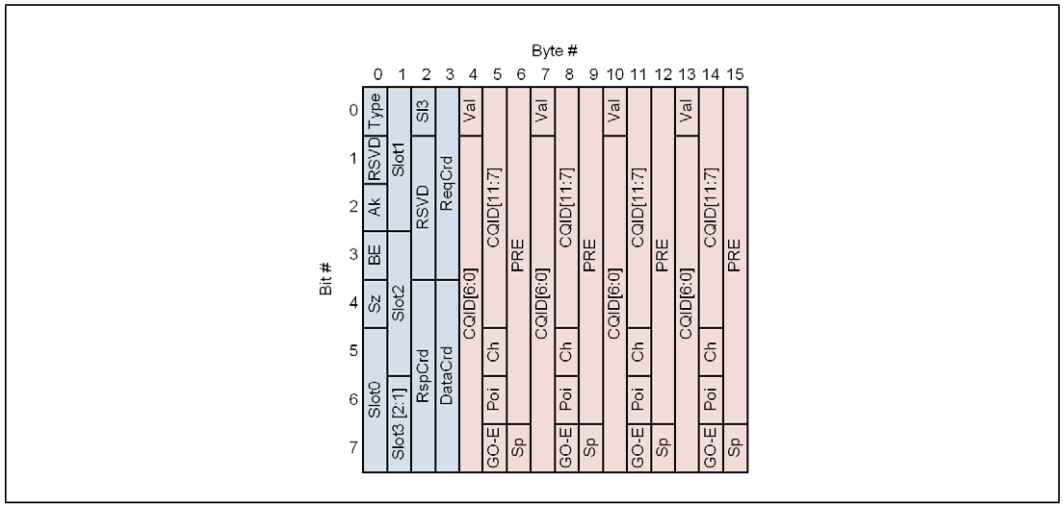


Figure 49. H4 - M2S RwD Header

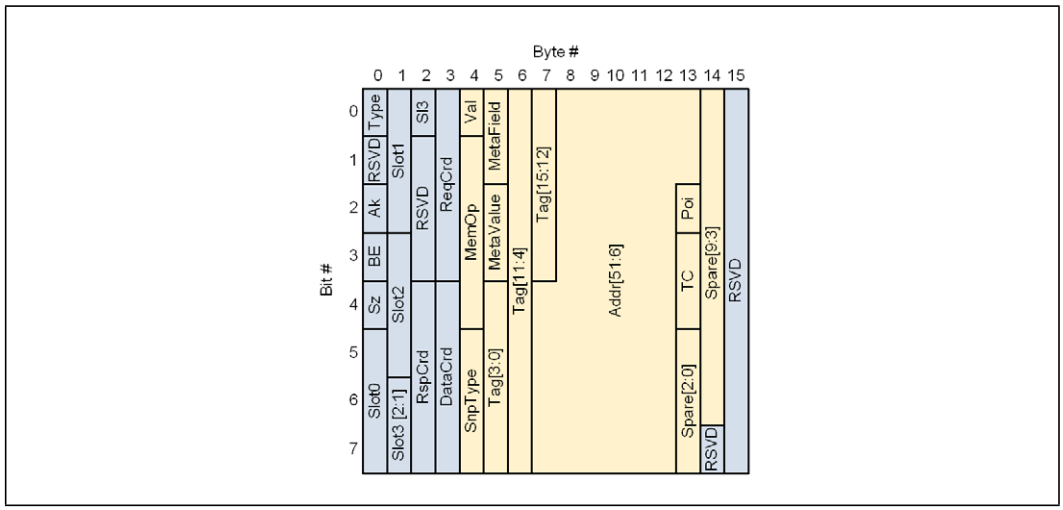


Figure 50. H5 - M2S Req

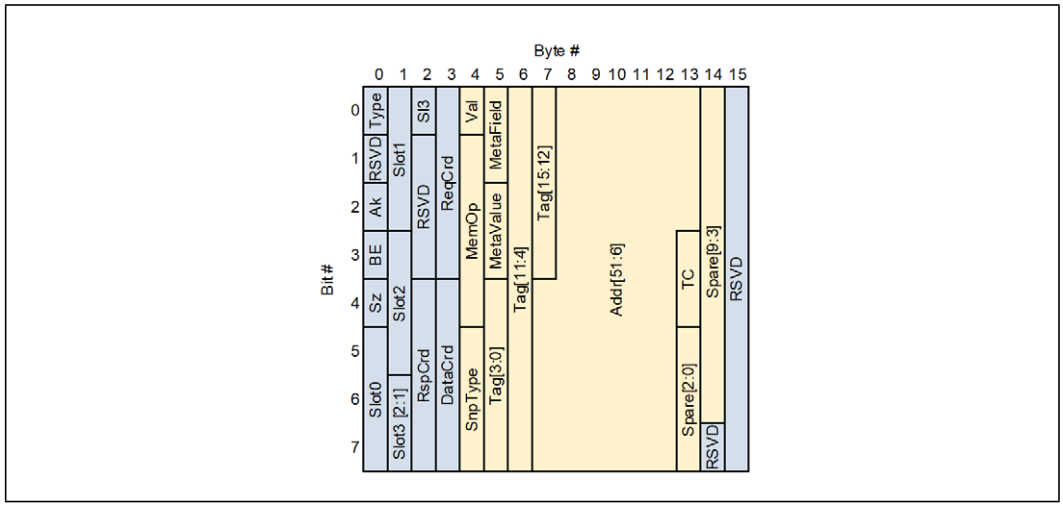


Figure 51. G0 - H2D / M2S Data

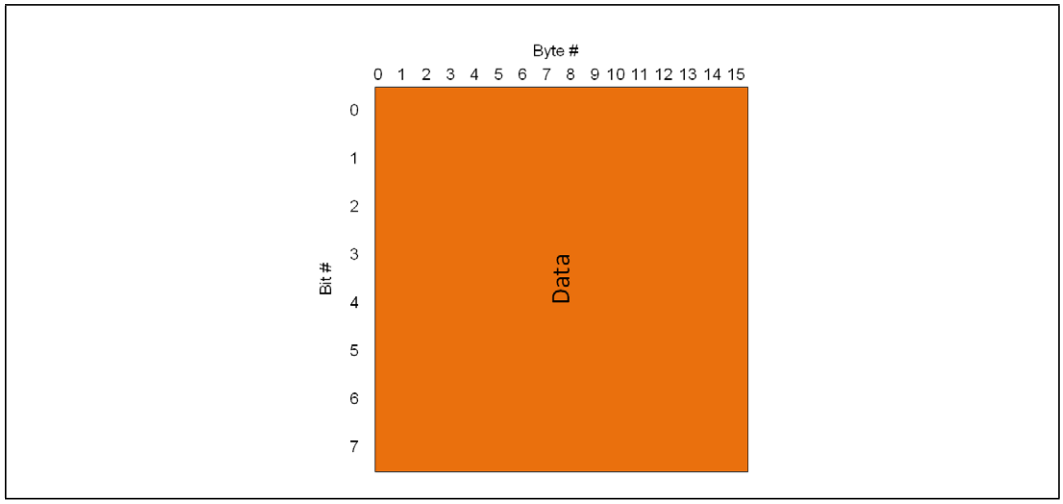


Figure 52. G0 - M2S Byte Enable

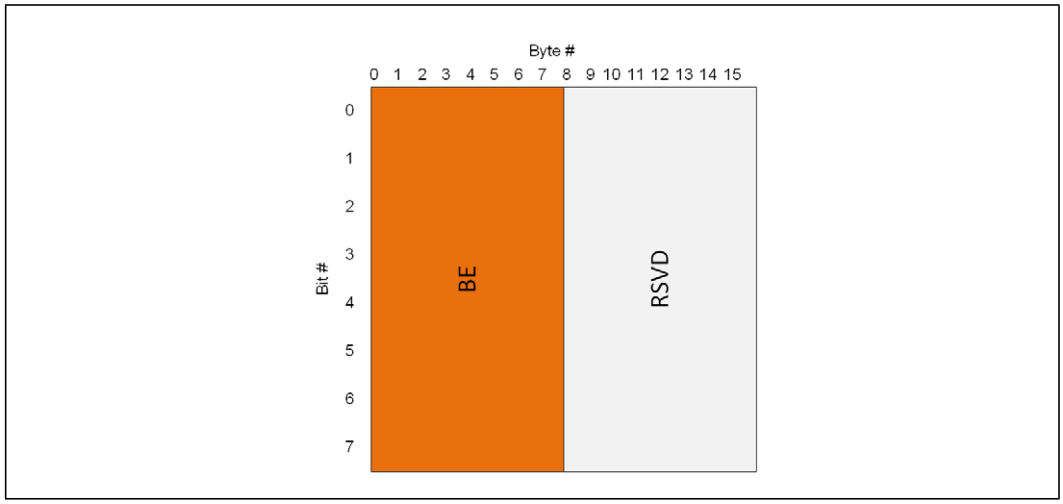


Figure 53. G1 - 4 H2D Resp

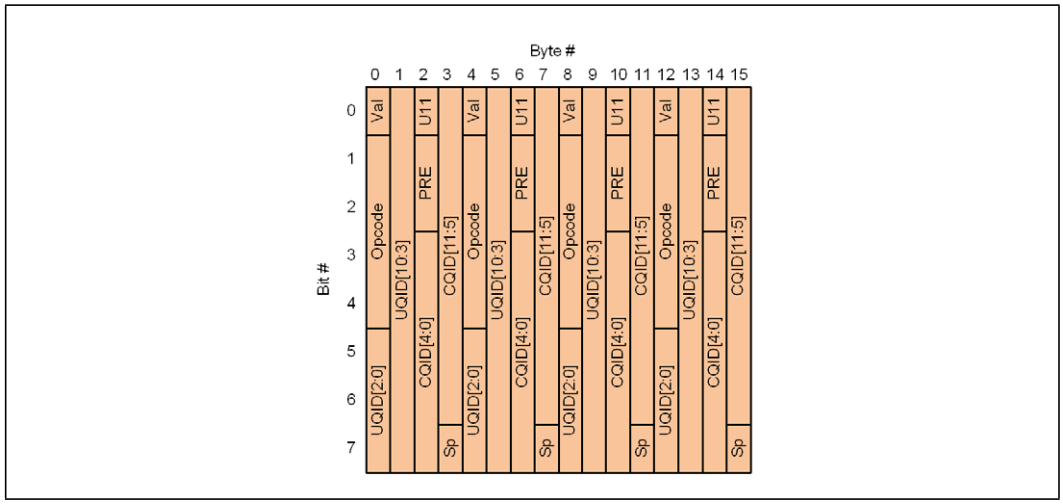


Figure 54. G2 - H2D Req + H2D Data Header + H2D Resp

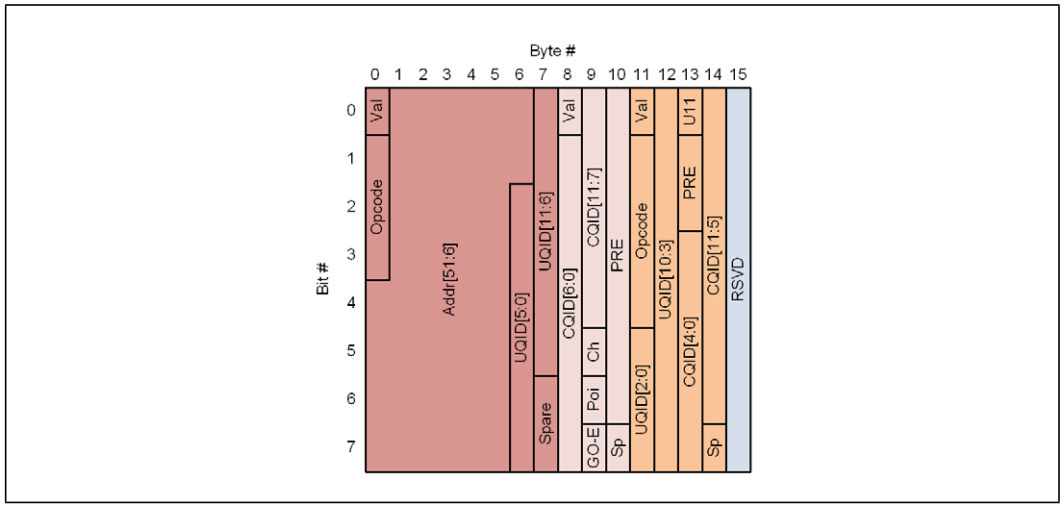


Figure 55. G3 - 4 H2D Data Header + H2D Resp

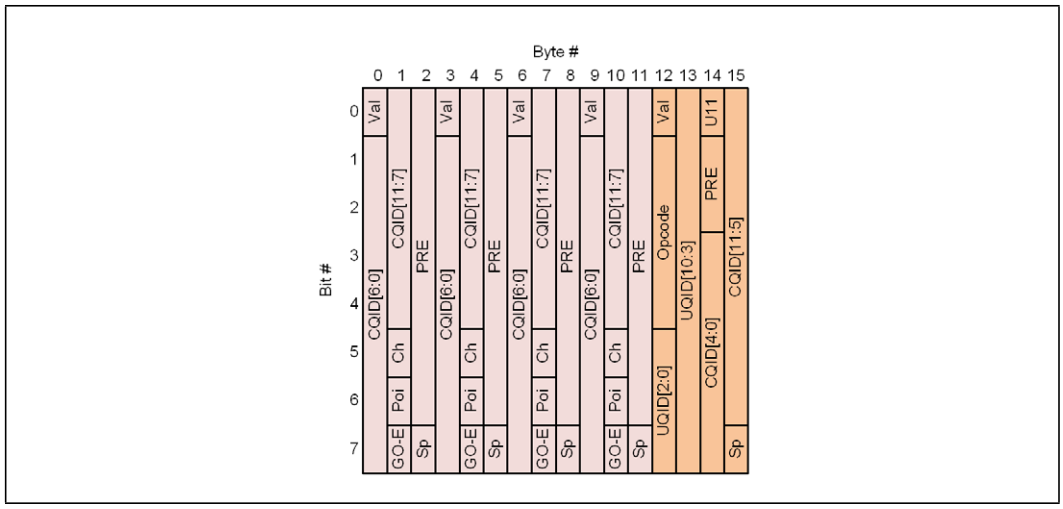


Figure 56. G4 - M2S Req + H2D Data Header

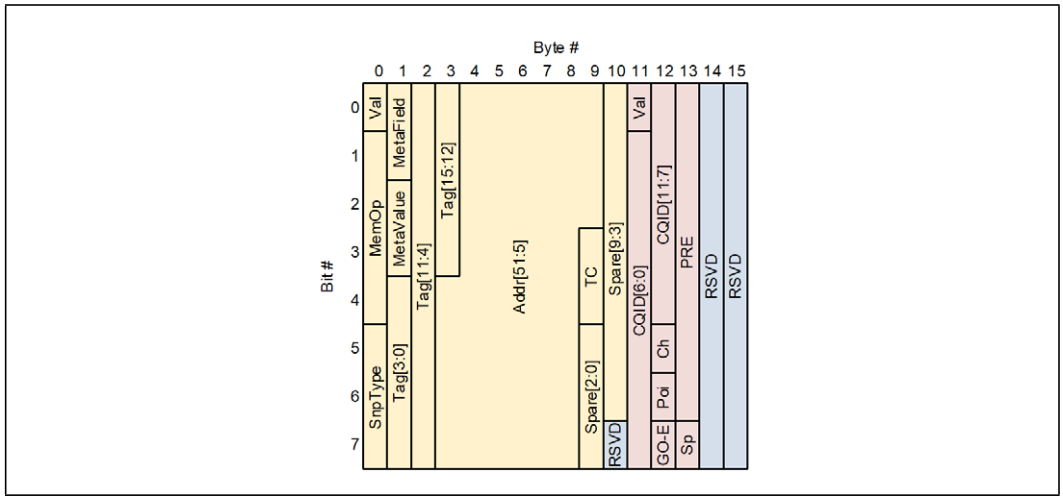
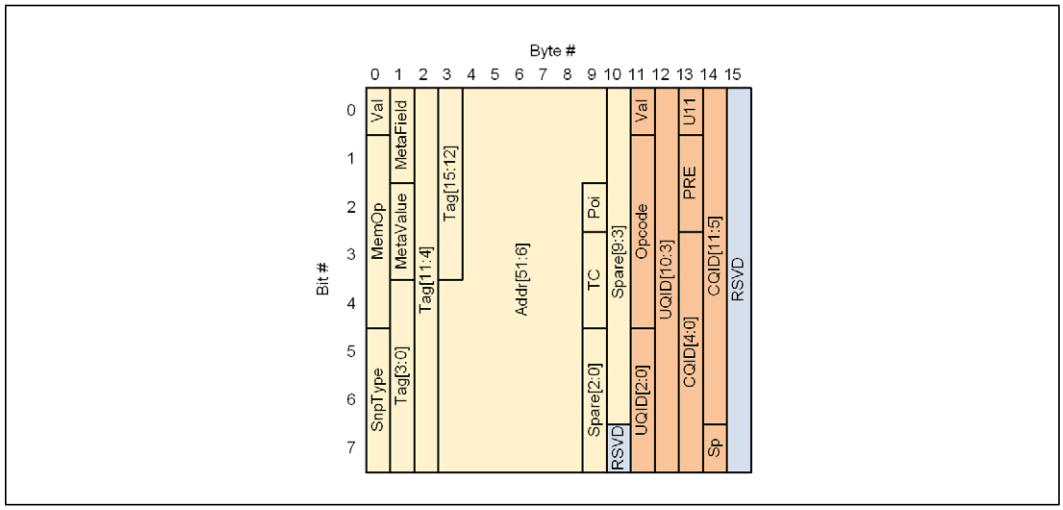


Figure 57. G5 - M2S RwD Header + H2D Resp



4.2.3.3 D2H & S2M Formats

Figure 58. H0 - D2H Data Header + 2 D2H Resp + S2M NDR

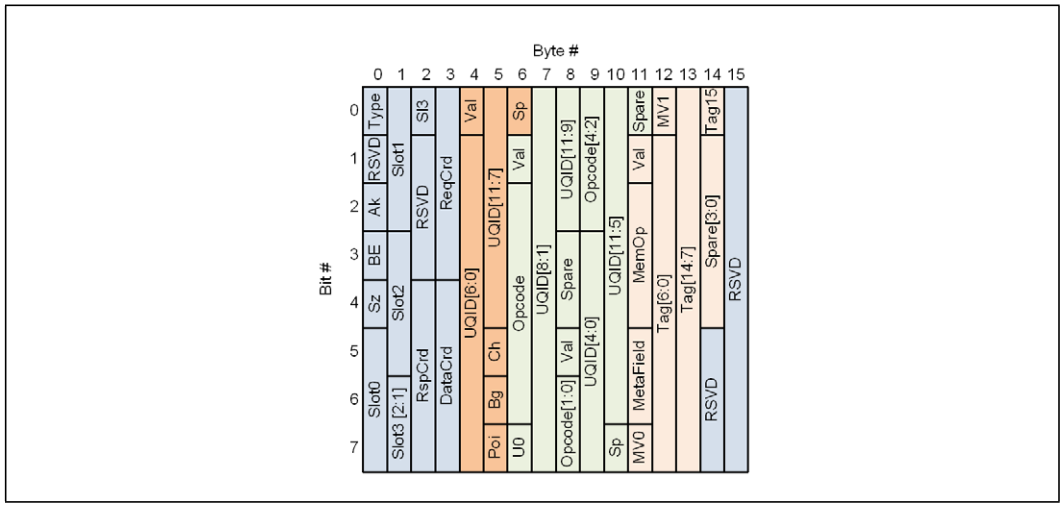


Figure 59. H1 - D2H Req + D2H Data Header

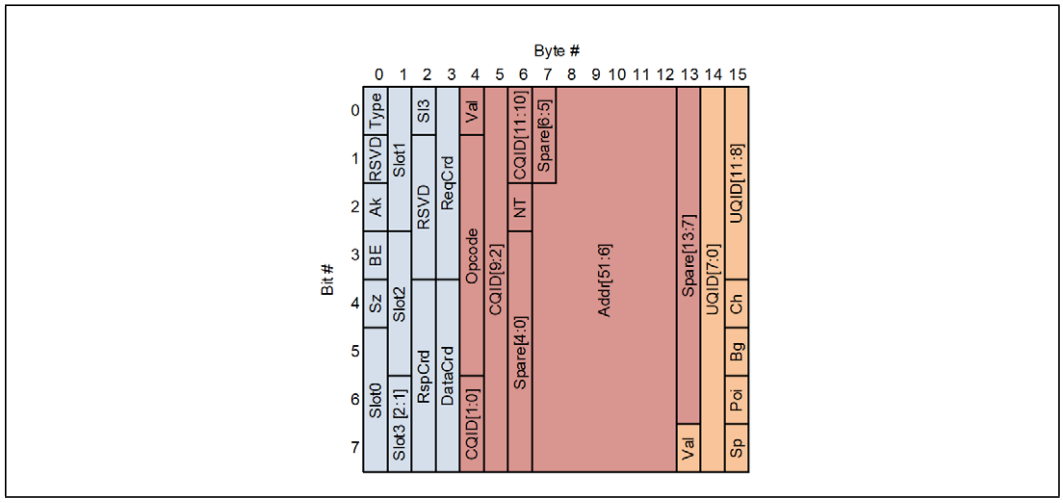


Figure 60. H2 - 4 D2H Data Header + D2H Resp

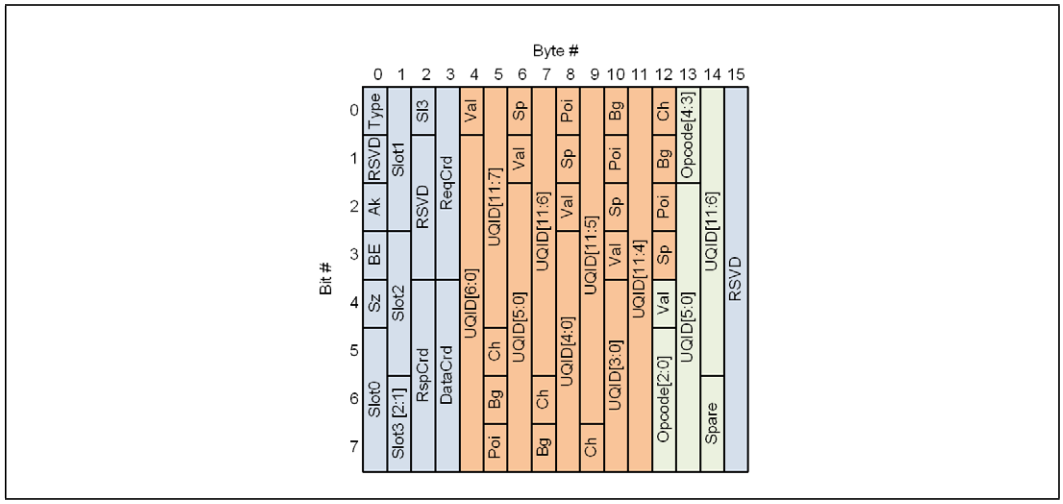


Figure 61. H3 - S2M DRS Header + S2M NDR

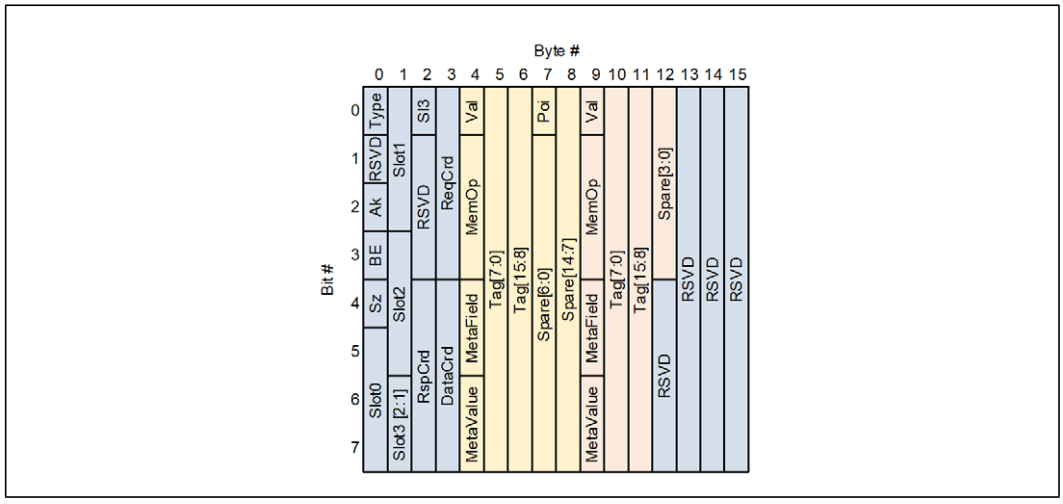


Figure 62. H4 - 2 S2M NDR

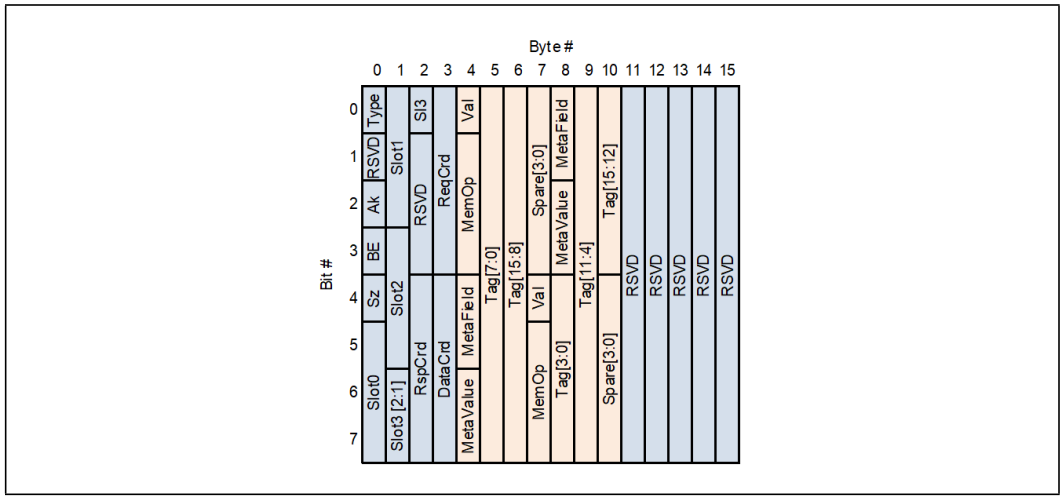


Figure 63. H5 - 2 S2M DRS

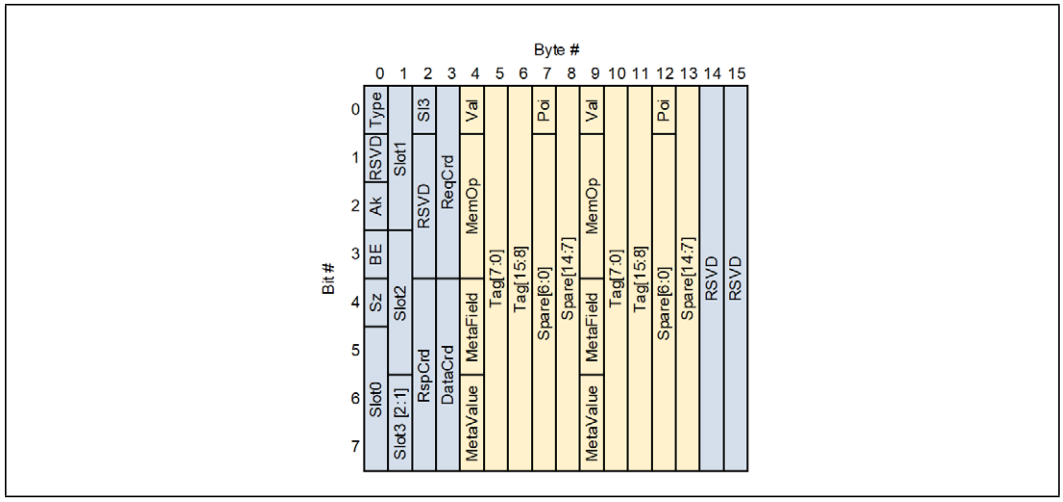


Figure 64. G0 - D2H Data

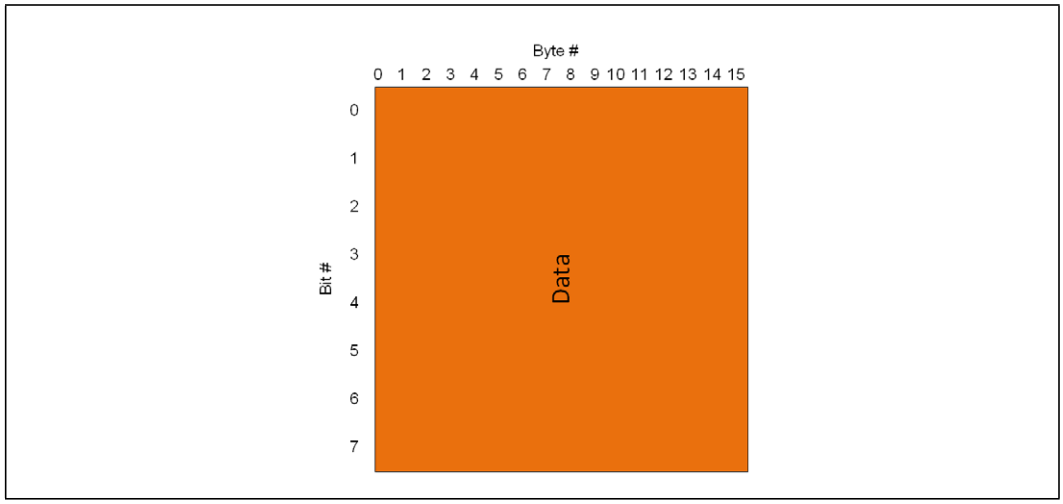


Figure 65. G0 - D2H/S2M Byte Enable

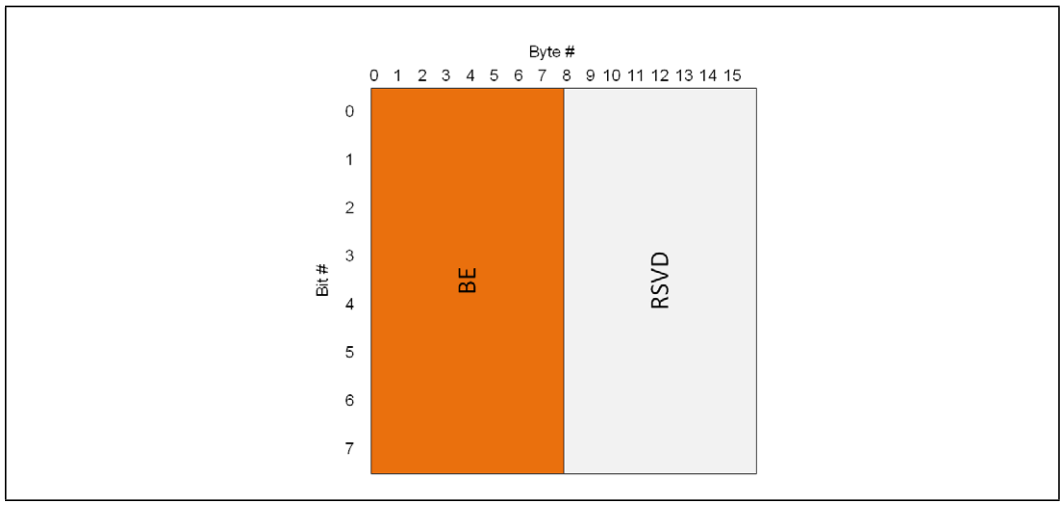


Figure 66. G1 - D2H Req + 2 D2H Resp

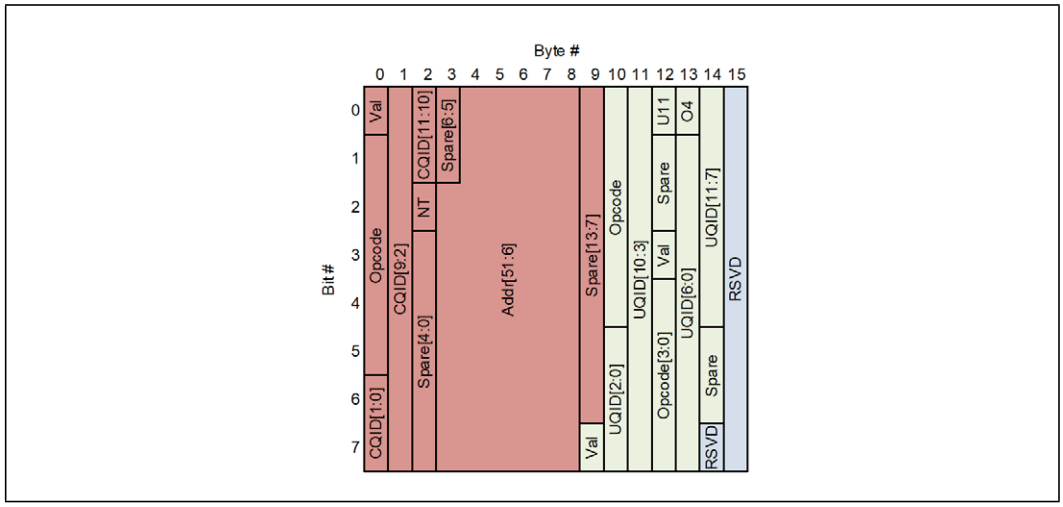


Figure 67. G2 - D2H Req + D2H Data Header + D2H Resp

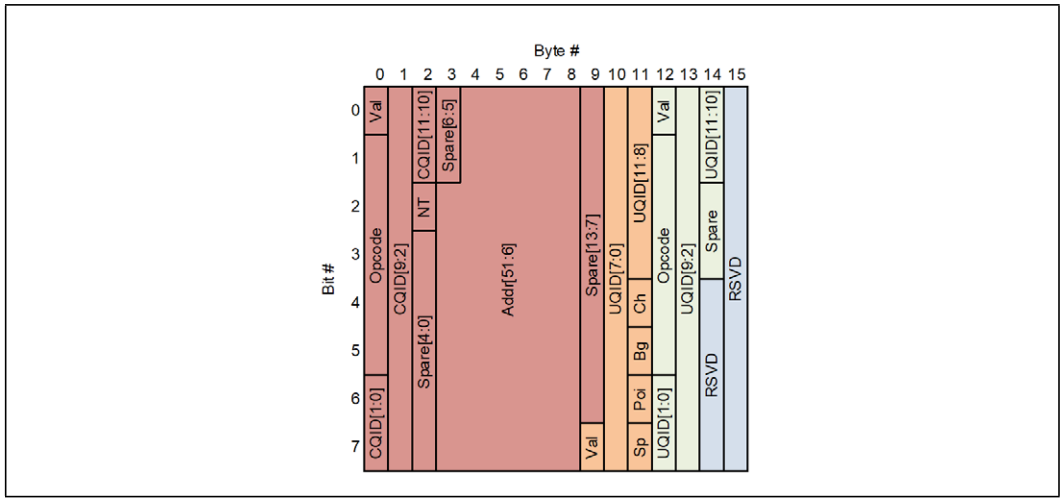


Figure 68. G3 - 4 D2H Data Header

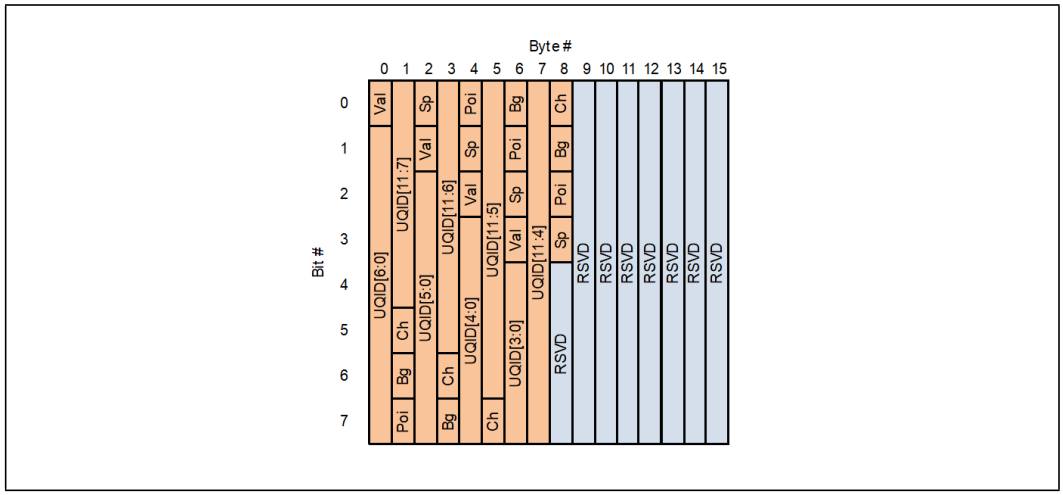


Figure 69. G4 - S2M DRS Header + 2 S2M NDR

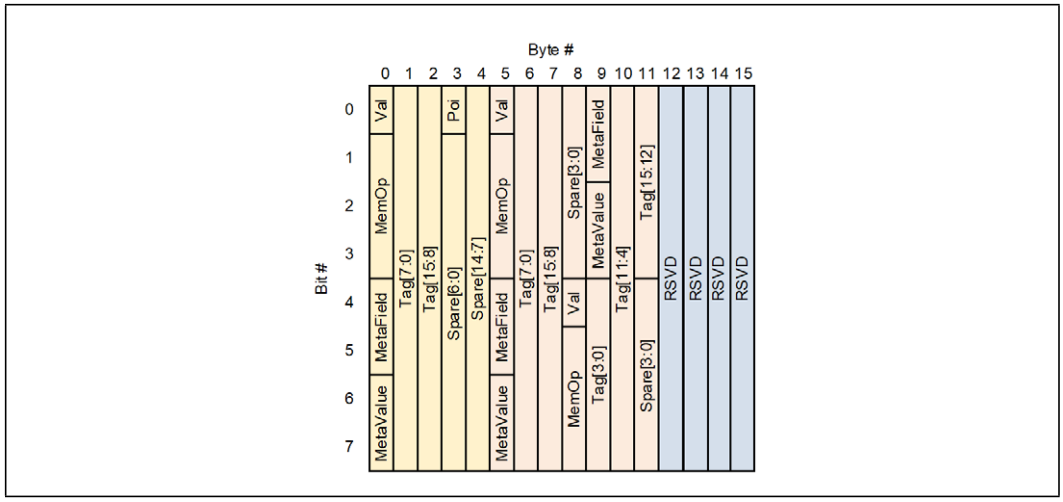


Figure 70. G5 - 3 S2M NDR

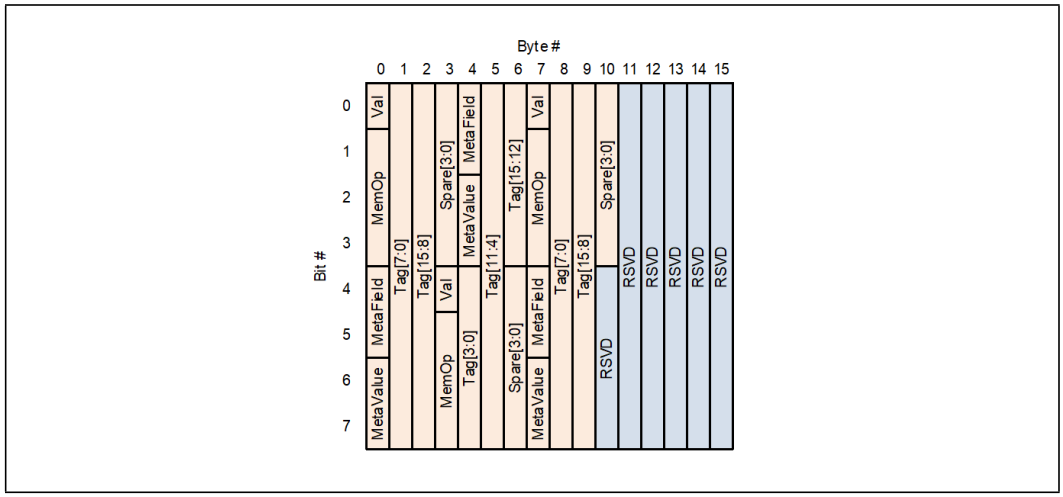
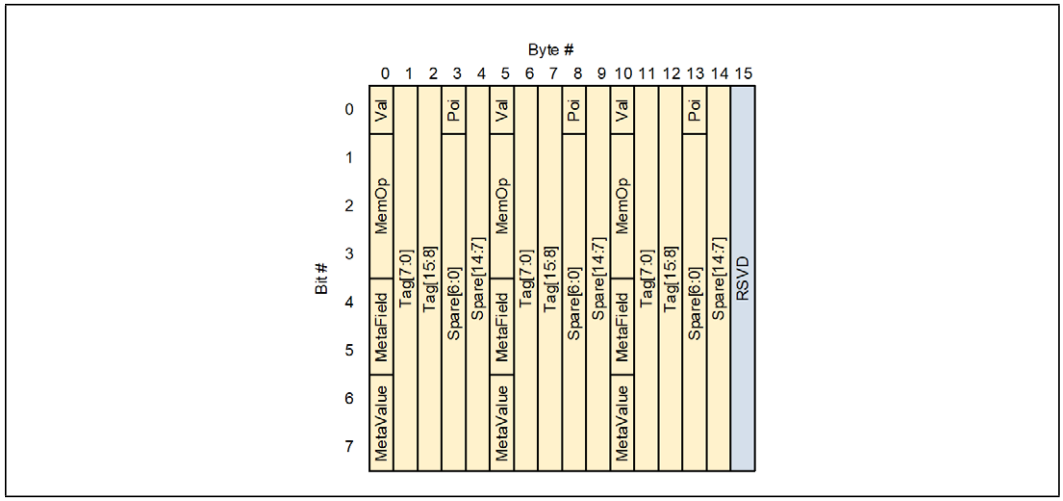


Figure 71. G6 - 3 S2M DRS



4.2.4 Link Layer Registers

Architectural registers associated with CXL.cache and CXL.mem have been defined in Section 7.2.2.1.15

4.2.5 Flit Packing Rules

The packing rules are defined below. Here, for the purpose of bidding, it is assumed that a given queue has credits towards the RX and any protocol dependencies (SNP-GO ordering, for example) have already been considered:

- Rollover is defined as any time a data transfer needs more than one flit. Note that a data chunk which contains 128b (format G0), can only be scheduled in Slots 1, 2 and 3 of a protocol flit since Slot 0 has only 96b available, as 32b are taken up by the flit header. The following rules apply to Rollover data chunks.

  - If there's a rollover of more than 3 16B data chunks, the next flit must necessarily be an all data flit.
  - If there's a rollover of 3 16B data chunks, Slots 1, Slots 2 and Slots 3 must necessarily contain the 3 rollover data chunks. Slot 0 will be packed independently (it is allowed for Slot 0 to have the Data Header for the next data transfer).
  - If there's a rollover of 2 16B data chunks, Slots 1 and Slots 2 must necessarily contain the 2 rollover data chunks. Slot 0 and Slot 3 will be packed independently.
  - If there's a rollover of 1 16B data chunk, Slot 1 must necessarily contain the rollover data chunk. Slot 0, Slot 2 and Slot 3 will be packed independently.
  - If there's no rollover, each of the 4 slots will be packed independently.
- Care must be taken to ensure fairness between packing of CXL.mem & CXL.cache transactions. Similarly, care must be taken to ensure fairness between channels within a given protocol. The exact mechanism to ensure fairness is implementation specific.
- Valid messages within a given slot need to be tightly packed. Which means, if a slot contains multiple possible locations for a given message, the Tx must pack the message in the first available location before advancing to the next available location.
- Valid messages within a given flit need to be tightly packed. Which means, if a flit contains multiple possible slots for a given message, the Tx must pack the message in the first available slot before advancing to the next available slot.
- If a valid Data Header is packed in a given slot, the next available "data-slots" (Slot 1, Slot 2, Slot 3 or an all-data flit) will be guaranteed to have data associated with the header. The Rx will use this property to maintain a shadow copy of the Tx Rollover counts. This enables the Rx to expect all-data flits where a flit header is not present.
- For data transfers, the Tx must send 16B data chunks in cacheline order. That is, chunk order 01 for 32B transfers and chunk order 0123 for 64B transfers.
- A slot with more than one data header (e.g. H5 in the S2M direction, or G3 in the H2D direction) is called a multi-data header slot or a MDH slot. MDH slots can only be sent for full cache line transfers when both 32B chunks are available to pack immediately. That is, BE = 0, Sz = 1. A MDH slot can only be used if both end points support MDH (defeature defined in Section 7.2.2.1.22)
- A MDH slot format must be chosen by the Tx only if there is more than 1 valid Data Header to pack in that slot.
- Control flits cannot be interleaved with all data flits. This also implies that when an all-data flit is expected following a protocol flit (due to Rollover), the Tx cannot send a Control flit before the all data flit.
- For non-MDH containing flits, there can be at most 1 valid Data Header in that flit. Also, a MDH containing flit cannot be packed with another valid Data Header in the same flit.
- The maximum number of messages that can be sent in a given flit (sum, across all slots) is:

  D2H Request --> 4

  D2H Response --> 2

  D2H Data Header --> 4

  D2H Data --> 4*16B

  S2M NDR --> 2

  S2M DRS Header --> 3

  S2M DRS Data --> 4*16B

H2D Request --> 2
H2D Response --> 4
H2D Data Header --> 4
H2D Data --> 4*16B
M2S Req --> 2
M2S DRS Header --> 1
M2D DRS Data --> 4*16B

4.2.6 Link Layer Control Flit

Link Layer Control flits do not follow flow control rules applicable to protocol flits. That is, they can be sent from an entity without any credits. These flits must be processed and consumed by the receiver within the period to transmit a flit on the channel since there are no storage or flow control mechanisms for these flits. The following table lists all the Controls Flits supported by the CXL.cache/CXL.mem link layer.

Table 41. CXL.cache/CXL.mem Link Layer Control Types

LLCTRL Type Encoding	LLCTRL Type Name	Description	Retriable? (Enters the LLRB)
0b0001	RETRY	Link layer retry flit	No
0b0000	LLCRD	Flit containing only link layer credit return and/or Ack information, but no protocol information.	Yes
0b1100	INIT	Link layer initialization flit	Yes

A detailed description of the control flits is present below.

Table 42. CXL.cache/CXL.mem Link Layer Control Details (Sheet 1 of 2)

Flit Type	LLCTRL	SubType	SubType Description	Payload	Payload Description
LLCRD	0000	0000	NA	NA	NA
		0001	Acknowledge	2:0	Acknowledge[2:0]
				3	RSVD
				7:4	Acknowledge[7:4]
				63:8	RSVD
		Others	RSVD	NA	NA
Retry	0001	0000	RETRY.Idle	63:0	RSVD
		0001	RETRY.Req	7:0	Requester's Retry Sequence Number (Eseq)
				15:8	RSVD
				20:16	Contains NUM_RETRY
				25:21	Contains NUM_PHY_REINIT (for debug)
				63:26	RSVD
		0010	RETRY.Ack	0	Empty: The Empty indicates that the LLR contains no valid data and therefore the NUM_RETRY value should be reset
				1	Viral: The Viral bit indicates that the transmitting agent is in a Viral state

Table 42. CXL.cache/CXL.mem Link Layer Control Details (Sheet 2 of 2)

				2	RSVD
				7:3	Contain an echo of the NUM_RETRY value from the LLR.Req
				15:8	Contains the WrPtr value of the retry queue for debug purposes
				23:16	Contains an echo of the Eseq from the LLR.Req
				31:24	Contains the NumFreeBuf value of the retry queue for debug purposes
				63:32	RSVD
		0011	RETRY.Frame	NA	Flit required to be sent before a RETRY.Req or RETRY.Ack flit to allow said flit to be decoded without risk of aliasing.
		Others	RSVD	NA	NA
Init	1100	1000	INIT.Param	3:0	Interconnect Version: Version of AL the port is compliant with. CXL 1.0 = '0001 Others Reserved
				7:4	RSVD
				12:8	RSVD
				23:13	RSVD
				31:24	LLR Wrap Value: Value after which LLR sequence counter should wrap to zero. The default value of this field is 9, until an error-free INIT.Param flit is received.
		Others	RSVD	NA	NA

In the LLCRD flit, the total number of flit acknowledgments being returned is determined by creating the Full_Ack return value, where

Full_Ack = {Acknowledgment[7:4],Ak,Acknowledgment[2:0]}, where the Ak bit is from the flit header.

The flit formats for the control flit are illustrated below.

Figure 72. LLCRD Flit Format (Only Slot 0 is Valid. Others are Reserved)

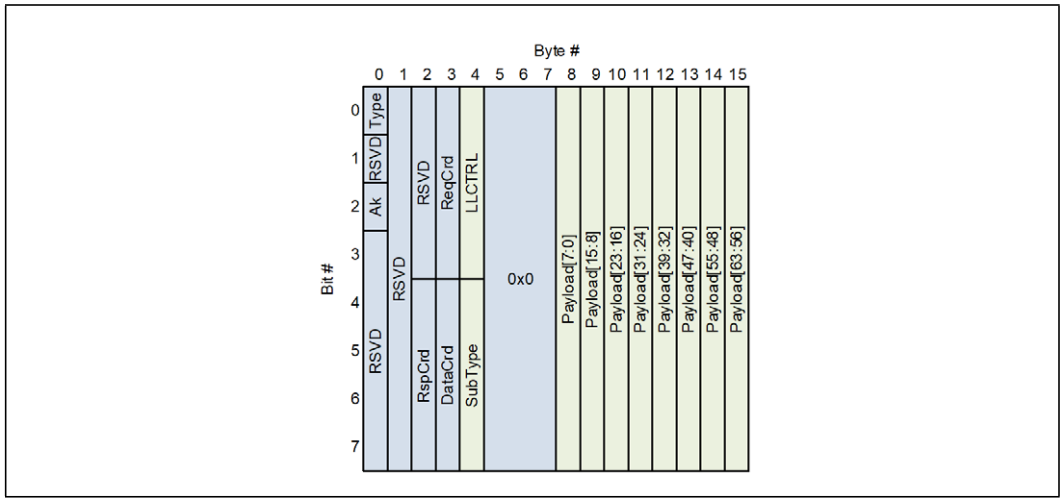


Figure 73. Retry Flit Format (Only Slot 0 is Valid. Others are Reserved)

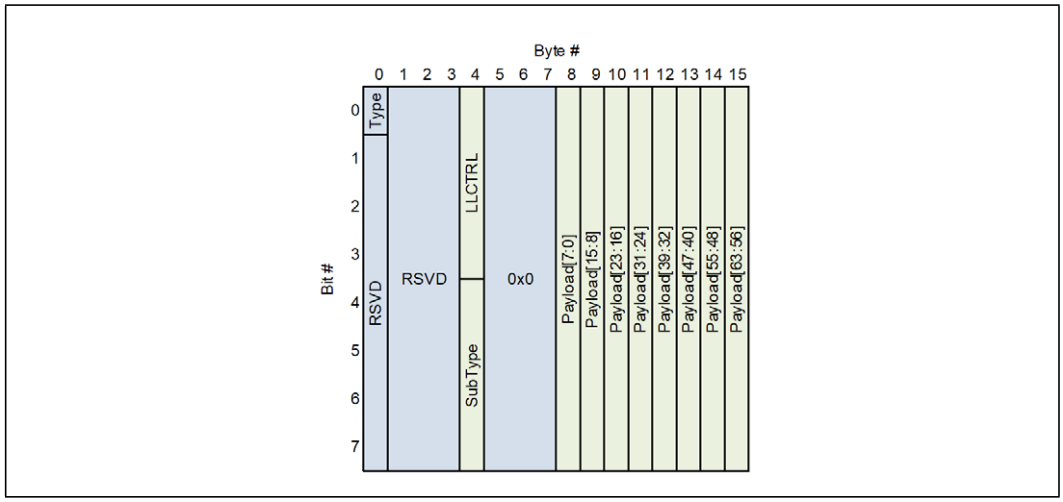
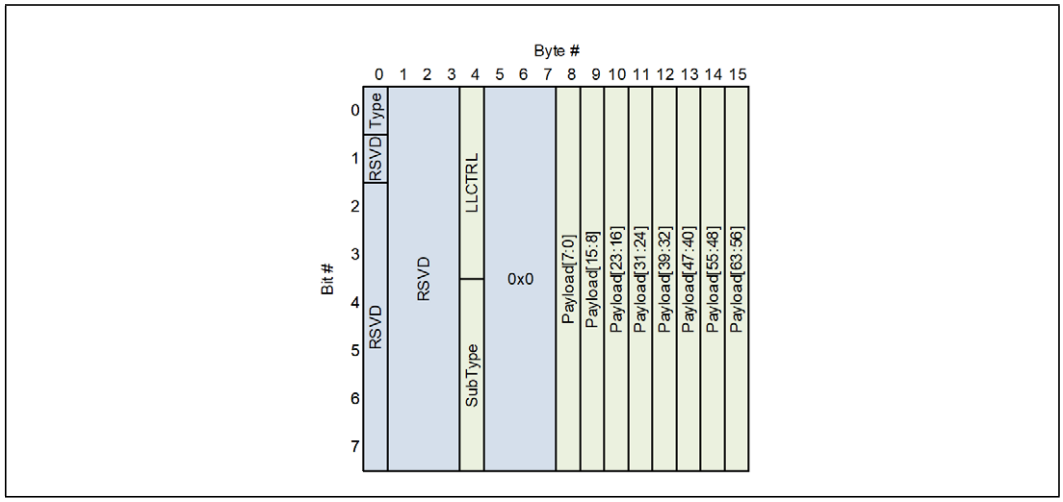


Figure 74. Init Flit Format (Only Slot 0 is Valid. Others are Reserved)



Note: The RETRY.Req and RETRY.Ack flits belong to the type of flit that receiving devices must respond to even in the shadow of a previous CRC error. In addition to checking the CRC of a RETRY flit, the receiving device should also check as many defined bits (those listed as having hardcoded 1/0 values) as possible in order to increase confidence in qualifying an incoming flit as a RETRY message.

4.2.7 Link Layer Initialization

Link Layer Initialization is expected to be started after any physical layer reset is complete and the link has trained successfully to L0. After reset, the Cache/Mem Link Layer can only send LLCTRL-Retry flits until Link Initialization is complete. The following describes how the link layer is initialized and credits are exchanged.

- The Tx portion of the Link Layer must wait until the Rx portion of the Link Layer has received at least one valid flit that is CRC clean before sending the LLCTRL-INIT.Param flit. Before this condition is met, the Link Layer must transmit only LLCTRL-Retry flits.
  - If for any reason the Rx portion of the Link Layer is not ready to begin processing flits beyond LLCTRL-INIT and LLCTRL-Retry, the Tx will stall transmission of LLCTR-INIT.Param flit
- The LLCTRL-INIT.Param flit must be the first non-LLCTRL-Retry flit transmitted by the Link Layer
- The Rx portion of the Link Layer must be able to receive an LLCTRL-INIT.Param flit immediately upon completion of Physical Layer initialization because the very first valid flit may be a LLCTRL-INIT.Param
- Received LLCTRL-INIT.Param values (i.e., LLR Wrap Value) must be made “active”, that is, applied to their respective hardware states within 8 flit clocks of error-free reception of LLCTRL-INIT.Param flit.
- Any non-Retry flits received before LLCTRL-INIT.Param flit will trigger an Uncorrectable Error.
- Only a single LLCTRL-INIT.Param flit should be created by the Tx portion of the Link Layer after reset. Any CRC error conditions with an LLCTRL-INIT.Param flit will be dealt with by the Retry state machine and replaced from the Link Layer Retry Buffer.

- Receipt of an LLCTRL-INIT.Param flit after an LLCTRL-INIT.Param flit has already been received should be considered an Uncorrectable Error.
- It is the responsibility of the Rx to transmit credits to the sender using standard credit return mechanisms after link initialization. Each entity should know how many buffers it has and set its credit return counters to these values. Then, during normal operation, the standard credit return logic will return these credits to the sender.
- Immediately after link initialization, the credit exchange mechanism will use the LLCRD flit format.
- It is possible that the receiver will make available more credits than the sender can track for a given message class. For correct operation, it is therefore required that the credit counters at the sender be saturating.
- Credits should be sized to achieve desired levels of bandwidth considering round-trip time of credit return latency. This is implementation and usage dependent.

4.2.8 CXL.cache/CXL.mem Link Layer Retry

The link layer provides recovery from transmission errors using retransmission, or Link Layer Retry (LLR). The sender buffers every flit sent in a local link layer retry buffer (LLRB). To uniquely identify flits in this buffer, the retry scheme relies on sequence numbers which are maintained within each device. Unlike in PCIe, CXL.cache/.mem sequence numbers are not communicated between devices with each flit to optimize link efficiency. The exchange of sequence numbers occurs only through link layer control (LLCTRL) flits during a LLR sequence. The sequence numbers are set to a predetermined value (zero) during reset and they are implemented using a wrap-around counter. The counter wraps back to zero after reaching the depth of the retry buffer. This scheme makes the following assumptions:

- The round-trip delay between devices is more than the maximum of the link layer clock or flit period.
- All protocol flits are stored in the retry buffer. See Section 4.2.8.5.1 for further details on the handling of non-retryable control flits.

Note that for efficient operation, the size of the retry buffer must be more than the round-trip delay. This includes:

- Time to send a flit from the sender
- Flight time of the flit from sender to receiver
- Processing time at the receiver to detect an error in the flit
- Time to accumulate and, if needed, force Ack return and send embedded Ack return back to the sender
- Flight time of the Ack return from the receiver to the sender
- Processing time of Ack return at the original sender

Otherwise, the LLR scheme will introduce latency, as the transmitter will have to wait for the receiver to confirm correct receipt of a previous flit before the transmitter can free space in its LLRB and send a new flit. Note that the error case is not significant because transmission of new flits is effectively stalled until successful retransmission of the erroneous flit anyway.

4.2.8.1 LLR Variables

The retry scheme maintains two state machines and several state variables. Although the following text describes them in terms of one transmitter and one receiver, both the transmitter and receiver side of the retry state machines and the corresponding state variables are present at each device because of the bidirectional nature of the link.

Since both sides of the link implement both transmitter and receiver state machines, for clarity this discussion will use the term “local” to refer to the entity that detects a CRC error, and “remote” to refer to the entity that sent the flit that was received erroneously.

The receiving device uses the following state variables to keep track of the sequence number of the next flit to arrive.

- **ESeq**: This indicates the expected sequence number of the next valid flit at the receiving link layer entity. ESeq is incremented by one (modulo the size of the LLRB) on error-free reception of a retryable flit. ESeq stops incrementing after an error is detected on a received flit until retransmission begins (RETRY.Ack message is received). Link layer reset initializes ESeq to 0. Note that there is no way for the receiver to tell whether it has detected an error on a non-retryable control flit. In this case it will initiate the link layer retry flow as usual, and effectively the transmitter will replay from the first retryable flit sent after that non-retryable control flit.

The sending entity maintains two indices into its LLRB, as indicated below.

- **WrPtr**: This indexes the entry of the LLRB that will record the next new flit. When an entity sends a flit, it copies that flit into the LLRB entry indicated by the WrPtr and then increments the WrPtr by one (modulo the size of the LLRB). This is implemented using a wrap-around counter that wraps around to 0 after reaching the depth of the LLRB. Certain LLCTRL flits do not affect the WrPtr. WrPtr stops incrementing after receiving an error indication at the remote entity (RETRY.Req message), until normal operation resumes again (all flits from the LLRB have been retransmitted). WrPtr is initialized to 0 and is incremented only when a flit is put into the LLRB.
- **RdPtr**: This is used to read the contents out of the LLRB during a retry scenario. The value of this pointer is set by the sequence number sent with the retransmission request (RETRY.Req message). The RdPtr is incremented by one (modulo the size of the LLRB) whenever a flit is sent, either from the LLRB in response to a retry request or when a new flit arrives from the transaction layer and irrespective of the states of the local or remote retry state machines. If a flit is being sent when the RdPtr and WrPtr are the same, then it indicates that a new flit is being sent, otherwise it must be a flit from the retry queue.

The LLR scheme uses an explicit acknowledgment that is sent from the receiver to the sender to remove flits from the LLRB at the sender. The acknowledgment is indicated via an ACK bit in the headers of flits flowing in the reverse direction. In CXL.cache, a single ACK bit represents 8 acknowledgments. Each entity keeps track of the number of available LLRB entries and the number of received flits pending acknowledgment through the following variables.

- **NumFreeBuf**: This indicates the number of free LLRB entries at the entity. NumFreeBuf is decremented by 1 whenever an LLRB entry is used to store a transmitted flit. NumFreeBuf is incremented by the value encoded in the Ack/ Full_Ack field of a received flit. NumFreeBuf is initialized at reset time to the size of the LLRB. The maximum number of retry queues at any entity is limited to 255 (8 bit counter). Also, note that the retry buffer at any entity is never filled to its capacity, therefore NumFreeBuf is never ‘0. If there is only 1 retry buffer entry available, then the sender cannot send an ACK bearing flit. This restriction is required to avoid ambiguity between a full or an empty retry buffer during a retry sequence that may result into incorrect operation. This implies if there are only 2 retry buffer entries left (NumFreeBuf = 2), then the sender can send an Ack bearing flit only if the outgoing flit encodes a value of at least 1, else a LLCRD control flit is sent. This is required to avoid deadlock at the link layer due to retry buffer becoming full at both entities on a link and their inability to send ACK through header flits.

- **NumAck**: This indicates the number of acknowledgments accumulated at the receiver. NumAck increments by 1 when a retryable flit is received. NumAck is decremented by 8 when the ACK bit is set in the header of an outgoing flit. If the outgoing flit is coming from the LLRB and its ACK bit is set, NumAck does not decrement. At initialization, NumAck is set to 0. The minimum size of the NumAck field is the size of the LLRB. NumAck at each entity must be able to keep track of at least 255 acknowledgments.

The LLR protocol requires that the number of retry queue entries at each entity must be at least 23 entries (Size of Forced Ack (16) + Max All-Data-Flit (5) + 2) to prevent deadlock.

4.2.8.2 ACK Forcing

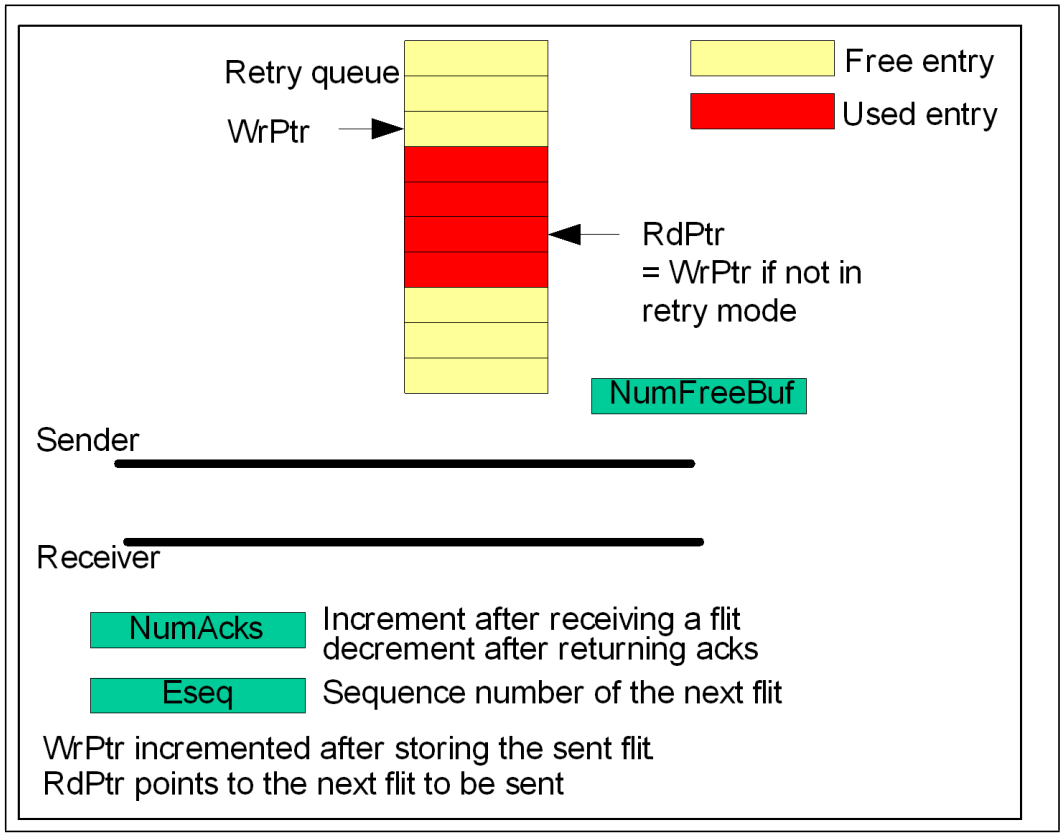
Recall that the LLR protocol requires space available in the LLRB to transmit a new flit, and that the sender must receive explicit acknowledgment from the receiver before freeing space in the LLRB. In scenarios where the traffic flow is very asymmetric, this requirement could result in traffic throttling and possibly even starvation.

Suppose that the A⟶B direction has very heavy traffic, but there is no traffic at all in the B⟶A direction. In this case A could exhaust its LLRB size, while B never has any return traffic in which to embed Acks. In CXL we want to minimize injected traffic to reserve bandwidth for the other traffic stream(s) sharing the link.

To avoid starvation, CXL must still permit Ack forcing (injection of a non-traffic flit to carry an Ack return), but this function is more heavily constrained so as not to waste bandwidth. In CXL, when B has accumulated at least 16 Acks to return, B's CXL.cache/mem link layer will inject a LLCRD flit for Ack return.

The CXL.cache link layer must accumulate a minimum of 8 Acks to set the ACK bit. If Ack forcing occurred after the accumulation of 8 Acks, it could result in a negative beat pattern where real traffic always arrives soon after a forced Ack, but not long enough after for enough Acks to re-accumulate to set the ACK bit. In the worst case this could double the bandwidth consumption of the CXL.cache side. By waiting for at least16 Acks to accumulate, the CXL.cache/mem link layer ensures that it can still opportunistically return Acks in any real traffic that arrives after a forced Ack return.

Figure 75. Retry Buffer and Related Pointers.



4.2.8.3 LLR Control Flits

The LLR Scheme uses several LLCTRL (link layer control) flits of the RETRY format to communicate the state information and the implicit sequence numbers between the entities.

- RETRY.Req: This flit is sent from the entity that received a flit in error to the sending entity. The flit contains the expected sequence number (ESeq) at the receiving entity, indicating the index of the flit in the retry queue at the remote entity that must be retransmitted. It also contains the NUM_RETRY value of the sending entity.
- RETRY.Ack: This flit is sent from the entity that is responding to an error detected at the remote entity. It contains a reflection of the NUM_RETRY value from the corresponding Retry.Req message. The flit contains the WrPtr value at the sending entity for debug purposes only. The WrPtr value should not be used by the retry state machines in any way. This flit will be followed by the flit identified for retry by the ESeq number.
- RETRY.Idle: This flit is sent during the retry sequence when there are no other protocol flits to be sent (see Section 4.2.8.5.2 for details) or a retry queue is not ready to be sent. For example, it can be used for debug purposes for designs that need additional time between sending the RETRY.Ack and the actual contents of the LLR queue.
- RETRY.Frame: This flit is sent in conjunction with a RETRY.Req or RETRY.Ack flit to prevent aliased decoding of these flits. See Section Section 4.2.8.5 for further details.

The table below describes the impact of RETRY messages on the local and remote retry state machines. In this context, the “sender” refers to the Device sending the message and the “receiver” refers to the Device receiving the message. Note that how this maps to which device detected the CRC error and which sent the erroneous message depends on the message type; e.g., for a RETRY.Req sequence, the sender detected the CRC error, but for a RETRY.Ack sequence, it’s the receiver that detected the CRC error.

4.2.8.4 RETRY Framing Sequences

Recall that the CXL.cache flit formatting specifies an all-data flit for link efficiency. This flit is encoded as part of the header of the preceding flit and contains no header information of its own. This introduces the possibility that the data contained in this flit could happen to match the encoding of a RETRY flit.

This introduces a problem at the receiver. It must be certain to decode the actual RETRY flit, but it must not falsely decode an aliasing data flit as a RETRY flit. In theory it might use the header information of the stream it receives in the shadow of a CRC error to determine whether it should attempt to decode the subsequent flit. Therefore the receiver cannot know with certainty which flits to treat as header-containing (decode) and which to ignore (all-data).

CXL introduces the RETRY.Frame flit for this purpose to disambiguate a control sequence from an all-data flit (ADF). Due to MDH, 5 ADF can be sent back-to-back. Hence, a RETRY.Req sequence comprises 5 RETRY.Frame flits immediately followed by a RETRY.Req flit, and a RETRY.Ack sequence comprises 5 RETRY.Frame flits immediately followed by a RETRY.Ack flit. This is shown in Figure 76.

Table 43. Control Flits and Their Effect on Sender and Receiver States

RETRY Message	Sender State	Receiver State
RETRY.Idle	Unchanged.	Unchanged.
RETRY.Frame + RETRY.Req Sequence	Local Retry State Machine (LRSM) is updated. NUM_RETRY is incremented. See Section 4.2.8.5.1	Remote Retry State Machine (RRSM) is updated. RdPtr is set to ESeq sent with the flit. See Section 4.2.8.5.3
RETRY.Frame + RETRY.Ack Sequence	RRSM is updated.	LRSM is updated.
RETRY.Frame, RETRY.Req, or RETRY.Ack message that is not as part of a valid framed sequence	Unchanged.	Unchanged (drop the flit).

Note: A RETRY.Ack sequence that arrives when a RETRY.Ack is not expected will be treated as an error by the receiver. Error resolution in this case is device specific though it is recommended that this results in the machine halting operation. It is recommended that this error condition not change the state of the LRSM.

4.2.8.5 LLR State Machines

The LLR scheme is implemented with two state machines: Remote Retry State Machine (RRSM) and Local Retry State Machine (LRSM). These state machines are implemented by each entity and together determine the overall state of the transmitter and receiver at the entity. The states of the retry state machines are used by the send and receive controllers to determine what flit to send and the actions needed to process a received flit.

4.2.8.5.1 Local Retry State Machine (LRSM)

This state machine is activated at the entity that detects an error on a received flit. The possible states for this state machine are:

- RETRY_LOCAL_NORMAL: This is the initial or default state indicating normal operation (no CRC error has been detected).
- RETRY_LLRREQ: This state indicates that the receiver has detected an error on a received flit and a RETRY.Req sequence must be sent to the remote entity.
- RETRY_LOCAL_IDLE: This state indicates that the receiver is waiting for a RETRY.Ack sequence from the remote entity in response to its RETRY.Req sequence. The implementation may require sub-states of RETRY_LOCAL_IDLE to capture, for example, the case where the last flit received is a Frame flit and the next flit expected is a RETRY.Ack.
- RETRY_PHY_REINIT: The state machine remains in this state for the duration of a physical layer reset.
- RETRY_ABORT: This state indicates that the retry attempt has failed and the link cannot recover. Error logging and reporting in this case is device specific. This is a terminal state.

The local retry state machine also has the three counters described below. The counters and thresholds described below are implementation specific.

- **TIMEOUT**: This counter is enabled whenever a RETRY.Req request is sent from an entity and the LRSM state becomes RETRY_LOCAL_IDLE. The TIMEOUT counter is disabled and the counting stops when the LRSM state changes to some state other than RETRY_LOCAL_IDLE. The TIMEOUT counter is reset to 0 at link layer initialization and whenever the LRSM state changes from RETRY_LOCAL_IDLE to RETRY_LOCAL_NORMAL or RETRY_LLRREQ. The TIMEOUT counter is also reset when the Physical layer returns from re-initialization (the LRSM transition through RETRY_PHY_REINIT to RETRY_LLRREQ). If the counter has reached its threshold without receiving a Retry.Ack sequence, then the RETRY.Req request is sent again to retry the same flit. See Section 4.2.8.5.2 for a description of when TIMEOUT increments. Note: It is suggested that the value of TIMEOUT should be no less than 4096 transfers.
- **NUM_RETRY**: This counter is used to count the number of RETRY.Req requests sent to retry the same flit. The counter remains enabled during the whole retry sequence (state is not RETRY_LOCAL_NORMAL). It is reset to 0 at initialization. It is also reset to 0 when a RETRY.Ack sequence is received or whenever the LRSM state is RETRY_LOCAL_NORMAL and an error-free retryable flit is received. The counter is incremented whenever the LRSM state changes from RETRY_LOCAL_LLRREQ to RETRY_LOCAL_IDLE. If the counter reaches a threshold (called MAX_NUM_RETRY), then the local retry state machine transitions to the RETRY_PHY_REINIT. The NUM_RETRY counter is also reset when the Physical layer returns from re-initialization (the LRSM transition through RETRY_PHY_REINIT to RETRY_LLRREQ). Note: It is suggested that the value of MAX_NUM_RETRY should be no less than 0xA.
- **NUM_PHY_REINIT**: This counter is used to count the number of physical layer re-initializations generated during a LLR sequence. The counter remains enabled during the whole retry sequence (state is not RETRY_LOCAL_NORMAL). It is reset to 0 at initialization and when receipt of a retryable flit triggers a transition from RETRY_LOCAL_IDLE or RETRY_LOCAL_NORMAL to RETRY_LOCAL_NORMAL. The counter is incremented whenever the LRSM changes from RETRY_LLRREQ to RETRY_PHY_REINIT. If the counter reaches a threshold (called MAX_NUM_PHY_REINIT) instead of transitioning from RETRY_LLRREQ to RETRY_PHY_REINIT, the LRSM will transition to RETRY_ABORT. The NUM_PHY_REINIT counter is also reset whenever a Retry.Ack sequence is received

with the Empty bit set. Note: It is suggested that the value of MAX_NUM_PHY_REINIT should be no less than 0xA.

Note that the condition of TIMEOUT reaching its threshold is not mutually exclusive with other conditions that cause the LRSM state transitions. Retry.Ack sequences can be assumed to never arrive at the time that the retry requesting device times out and sends a new RETRY.Req sequence (by appropriately setting the value of TIMEOUT – see Section Section 4.2.8.5.2). If this case occurs, no guarantees are made regarding the behavior of the device (behavior is "undefined" from a Spec perspective and is not validated from an implementation perspective). Consequently, the LLR Timeout value should not be reduced unless it can be certain this case will not occur. If an error is detected at the same time as TIMEOUT reaches its threshold, then the error on the received flit is ignored, TIMEOUT is taken and a repeat Retry.Req sequence is sent to the remote entity.

Table 44. Local Retry State Transitions (Sheet 1 of 2)

Current Local Retry State	Condition	Next Local Retry State	Actions
RETRY_LOCAL_NORMAL	An error free retryable flit is received.	RETRY_LOCAL_NORMAL	Increment NumFreeBuf using the amount specified in the ACK or Full_Ack fields. Increment NumAck by 1. Increment Eseq by 1. NUM_RETRY is reset to 0. NUM_PHY_REINIT is reset to 0. Received flit is processed normally by the link layer.
RETRY_LOCAL_NORMAL	Error free non-retryable flit (other than Retry.Req sequence) is received.	RETRY_LOCAL_NORMAL	Received flit is processed.
RETRY_LOCAL_NORMAL	Error free Retry.Req sequence is received.	RETRY_LOCAL_NORMAL	RRSM is updated.
RETRY_LOCAL_NORMAL	Error is detected on a received flit.	RETRY_LLRREQ	Received flit is discarded.
RETRY_LOCAL_NORMAL	PHY_RESET / PHY_REINIT detected.	RETRY_PHY_REINIT	None.
RETRY_LLRREQ	NUM_RETRY == MAX_NUM_RETRY and NUM_PHY_REINIT == MAX_NUM_PHY_REINIT	RETRY_ABORT	Indicate link failure.
RETRY_LLRREQ	NUM_RETRY == MAX_NUM_RETRY and NUM_PHY_REINIT < MAX_NUM_PHY_REINIT	RETRY_PHY_REINIT	If an error-free Retry.Req or Retry.Ack sequence is received, process the flit. Any other flit is discarded. Reset sent to physical layer. Increment NUM_PHY_REINIT.
RETRY_LLREQ	NUM_RETRY < MAX_NUM_RETRY and a Retry.Req sequence has not been sent.	RETRY_LLRREQ	If an error-free Retry.Req or Retry.Ack sequence is received, process the flit. Any other flit is discarded.
RETRY_LLRREQ	NUM_RETRY < MAX_NUM_RETRY and a Retry.Req sequence has been sent.	RETRY_LOCAL_IDLE	If an error free Retry.Req or Retry.Ack sequence is received, process the flit. Any other flit is discarded. Increment NUM_RETRY.
RETRY_LLRREQ	PHY_RESET/PHY_REINIT detected.	RETRY_PHY_REINIT	None.
RETRY_LLREQ	Error is detected on a received flit	RETRY_LLREQ	Received flit is discarded.

Table 44. Local Retry State Transitions (Sheet 2 of 2)

Current Local Retry State	Condition	Next Local Retry State	Actions
RETRY_PHY_REINIT	Physical layer still in reinit.	RETRY_PHY_REINIT	None.
RETRY_PHY_REINIT	Physical layer returns from Reinit.	RETRY_LLRREQ	Received flit is discarded. NUM_RETRY is reset to 0.
RETRY_LOCAL_IDLE	Retry.Ack sequence is received and NUM_RETRY from Retry.Ack matches the value in the local entity	RETRY_LOCAL_NORMAL	TIMEOUT is reset to 0. If Retry.Ack sequence is received with Empty bit set, NUM_RETRY is reset to 0 and NUM_PHY_REINIT is reset to 0.
RETRY_LOCAL_IDLE	Retry.Ack sequence is received and NUM_RETRY from Retry.Ack does NOT match the value in the local entity	RETRY_LOCAL_IDLE	Any received retryable flit is discarded
RETRY_LOCAL_IDLE	TIMEOUT has reached its threshold.	RETRY_LLRREQ	TIMEOUT is reset to 0.
RETRY_LOCAL_IDLE	Error is detected on a received flit.	RETRY_LOCAL_IDLE	Any received retryable flit is discarded.
RETRY_LOCAL_IDLE	A flit other than RETRY.Ack/ Retry.Req sequence is received.	RETRY_LOCAL_IDLE	Any received retryable flit is discarded.
RETRY_LOCAL_IDLE	A Retry.Req sequence is received.	RETRY_LOCAL_IDLE	RRSM is updated.
RETRY_LOCAL_IDLE	PHY_RESET / PHY_REINIT detected.	RETRY_PHY_REINIT	None.
RETRY_ABORT	A flit is received.	RETRY_ABORT	Any received retryable flit is discarded.

4.2.8.5.2 TIMEOUT Definition

After the local receiver has detected a CRC error, triggering the LRSM, the local Tx sends a RETRY.Req sequence to initiate LLR. At this time, the local Tx also starts its TIMEOUT counter.

The purpose of this counter is to decide that either the Retry.Req sequence or corresponding Retry.Ack sequence has been lost, and that another RETRY.Req attempt should be made. Recall that it is a fatal error to receive multiple Retry.Ack sequences (i.e., a subsequent Ack without a corresponding Req is unexpected). Therefore, the link layer must guarantee that it not send another Retry.Req sequence until it is certain it will not receive a Retry.Ack sequence for a previously sent Req. Thus the purpose of the TIMEOUT counter is to estimate the worst-case latency for a Retry.Req sequence to reach the remote side and for the corresponding Retry.Ack sequence to return.

Certain unpredictable events (such as physical layer re-initialization, low power transitions, etc.) that interrupt link availability could add a very large amount of latency to the RETRY round-trip. To make the TIMEOUT robust to such events, instead of incrementing per link layer clock, TIMEOUT increments whenever the local Tx transmits a flit, protocol or control. Due to the TIMEOUT protocol, it must force injection of RETRY.Idle flits if it has no real traffic to send, so that the TIMEOUT counter continues to increment.

4.2.8.5.3 Remote Retry State Machine (RRSM)

The remote retry state machine is activated at an entity if a flit sent from that entity is received in error by the local receiver, resulting in a link layer retry request (Retry.Req sequence) from the remote entity. The possible states for this state machine are:

- RETRY_REMOTE_NORMAL: This is the initial or default state indicating normal operation.
- RETRY_LLRACK: This state indicates that a link layer retry request (Retry.Req sequence) has been received from the remote entity and a Retry.Ack sequence followed by flits from the retry queue must be (re)sent.

The remote retry state machine transitions are described in the table below.

Table 45. Remote Retry State Transition

Current Remote Retry State	Condition	Next Remote Retry State
RETRY_REMOTE_NORMAL	Any flit, other than error free Retry.Req sequence, is received.	RETRY_REMOTE_NORMAL
RETRY_REMOTE_NORMAL	Error free Retry.Req sequence received.	RETRY_LLRACK
RETRY_LLRACK	Retry.Ack sequence not sent.	RETRY_LLRACK
RETRY_LLRACK	Retry.Ack sequence sent.	RETRY_REMOTE_NORMAL
RETRY_LLRACK	Physical Layer Reinitialization	RETRY_REMOTE_NORMAL

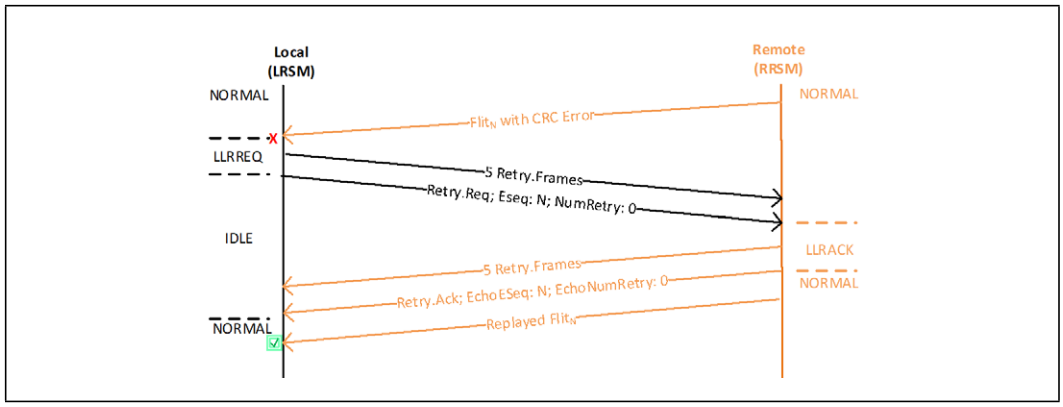
Note: In order to select the priority of sending flits, the following rules apply:

1. Whenever the RRSM state becomes RETRY_LLRACK, the entity must give priority to sending the LLCTRL flit with Retry.Ack

2. Except RRSM state of RETRY_LLRACK, the priority goes to LRSM state of RETRY_LLRREQ and in that case the entity must send a LLCTRL flit with Retry.Req over all other flits.

The overall sequence of replay is shown in Figure 76.

Figure 76. CXL.cache / mem Replay Diagram



4.2.8.6 Interaction with Physical Layer Reset or Reinitialization

On detection of a physical layer reset or reinitialization, the receiver side of the link layer must force a link layer retry on the next flit. Forcing an error will either initiate LLR or cause a current LLR to follow the correct error path. The LLR will ensure that no flits are dropped during the physical layer reset. Without initiating a LLR it is possible that packets/flits in flight on the physical wires could be lost or the sequence numbers could get mismatched.

Upon detection of a physical layer reset, the LLR RRSM needs to be reset to its initial state and any instance of Retry.Ack sequence needs to be cleared in the link layer and physical layer. The device needs to make sure it receives a Retry.Req sequence before it ever transmits a RETRY.Ack sequence.

4.2.8.7 CXL.cache / CXL.mem Flit CRC

The CXL.cache Link Layer uses a 16b CRC for transmission error detection. The 16b CRC is over the 528 bit flit. The assumptions about the type errors is as follows:

- Bit ordering runs down each lane
- Bit Errors occur randomly or in bursts down a lane, with majority of errors single bit random errors.
- Random errors can statistically cause multiple bit errors in a single flit, so it is more likely to get 2 errors in a flit then 3 errors, and more likely to get 3 errors in a flit then 4 errors, and so on...
- There is no requirement for primitive polynomial (a polynomial that generates all elements of an extension field from a base field) since we do have a fixed payload. Primitive may be the result, but it's not required.

4.2.8.7.1 CRC-16 Polynomial and Detection Properties

The CRC polynomial to be used is 0x1f053.

The 16b CRC Polynomial has the following properties:

- 16 Bit Burst Detection – Provides 2 Adjacent wire protection for 8UI flit
- All Single, double, and triple bit errors detected
- Polynomial selection based on best 4-bit error detection characteristics and perfect 1, 2, 3-bit error detection

4.2.8.7.2 CRC-16 Computation

Below are the 384 bit data masks for use with an XOR tree to produce the 16 CRC bits. The mask bit order is CRC[N]=DM[527:016]. Data Mask bits [527:016] are applied to the Flit bits [527:016], as flit bits [015:000] are defined to be CRC[15:00].

The Flit Data Mask for the 16 CRC bits is located in the table below.

Figure 77. CRC Data Mask for 527 bit Flit

	Flit Bit Location															
	527 - 496	495 - 464	463 - 432	431 - 400	399 - 368	367 - 336	335 - 304	303 - 272	271 - 240	239 - 208	207 - 176	175 - 144	143 - 112	111 - 80	79 - 48	47 - 16
CRC15	EF9C_D9F9	C4BB_B83A	3E84_A97C	D7AE_DA13	FAEB_01B8	5B20_4A4C	AE1E_79D9	7753_5D21	DC7F_DD6A	38F0_3E77	F5F5_2A2C	636D_B05C	3978_EA30	CD50_E0D9	9B06_93D4	746B_2431
CRC14	9852_B505	26E6_6427	21C6_FDC2	BC79_B71A	079E_8164	76B0_6F6A	F911_4535	CCFA_F3B1	3240_33DF	2488_214C	0F0F_BF3A	52DB_6872	25C4_9F28	ABF8_90B5	5685_DA3E	4E5E_B629
CRC13	23B5_837B	57C8_8A29	AE67_D79D	8992_019E	F924_410A	6078_7DF9	D296_DB43	912E_24F9	455F_C485	AAB4_2ED1	F272_F5B1	4A00_0465	2B9A_A5A4	98AC_A883	3044_7ECB	5344_7F25
CRC12	7E46_1844	6F5F_FD2E	E9B7_42B2	1367_DADC	8679_213D	6B1C_74B0	4755_1478	BFC4_4F5D	7ED0_3F28	EDAA_291F	0CCC_50F4	C66D_B26E	ACB5_B8E2	8106_B498	0324_ACB1	DDC9_1BA3
CRC11	50BF_D5DB	F314_46AD	4A5F_0825	DE1D_377D	B9D7_9126	EEAE_7014	8DB4_F3E5	28B1_7A8F	6317_C2FE	4E25_2AF8	7393_0256	005B_696B	6F22_3641	8DD3_BA95	9A94_C58C	9A8F_A9E0
CRC10	A85F_EAED	F98A_2356	A52F_8412	EF0E_9BBE	DCEB_C893	7757_380A	46DA_79F2	9458_BD47	B18B_E17F	2712_957C	39C9_812B	002D_B4B5	B791_1B20	C6E9_DD4A	CD4A_62C6	4D47_D4F0
CRC09	542F_F576	FCC5_11AB	5297_C209	7787_4DDF	6E75_E449	BBAB_9C05	236D_3CF9	4A2C_5EA3	D8C5_F0BF	9389_4ABE	1CE4_C095	8016_DA5A	DBC8_8D90	6374_EEA5	66A5_3163	26A3_EA78
CRC08	2A17_FABB	7E62_88D5	A94B_E104	BBC3_A6EF	B73A_F224	DDD5_CE02	91B6_9E7C	A516_2F51	EC62_F85F	C9C4_A55F	0E72_604A	C00B_6D2D	6DE4_46C8	31BA_7752	B352_98B1	9351_F53C
CRC07	150B_FD5D	BF31_446A	D4A5_F082	5DE1_D377	DB9D_7912	6EEA_E701	48DB_4F3E	528B_17A8	F631_7C2F	E4E2_52AF	8739_3025	6005_B696	B6F2_2364	18DD_3BA9	59A9_4C58	C9A8_FA9E
CRC06	8A85_FEAE	DF98_A235	6A52_F841	2EF0_E9BB	EDCE_BC89	3775_7380	A46D_A79F	2945_88D4	7B18_BE17	F271_2957	C39C_9812	B002_DB4B	5B79_11B2	0C6E_9DD4	ACD4_A62C	64D4_7D4F
CRC05	AADE_26AE	AB77_E920	8BAD_D55C	40D6_AECE	0C0C_5FFC	C09A_F38C	FC28_AA16	E3F1_98CB	E1F3_8261	C1C8_AADC	143B_6625	3B6C_DDF9	94C4_62E9	CB67_AE33	CD6C_C0C2	4601_1A96
CRC04	D56F_1357	55BB_F490	45D6_EAAE	206B_5767	0606_2FFE	604D_79C6	7E14_550B	71F8_CC65	F0F9_C130	E0E4_556E	0A1D_B312	9DB6_6EFC	CA62_3174	E5B3_D719	E6B6_6061	2300_8D4B
CRC03	852B_5052	6E66_4272	1C6F_DC2B	C79B_71A0	79E8_1647	6B06_F6AF	9114_535C	CFAF_3B13	2403_3DF2	4882_14C0	F0FB_F3A5	2DB6_8722	5C49_F28A	BF89_0B55	685D_A3E4	E5EB_6294
CRC02	C295_A829	3733_2139	0E37_EE15	E3CD_B8D0	3CF4_0B23	B583_7B57	C88A_29AE	67D7_9D89	9201_9EF9	2441_0A60	787D_F9D2	96DB_4391	2E24_F945	5FC4_85AA	B42E_D1F2	72F5_B14A
CRC01	614A_D414	9B99_909C	871B_F70A	F1E6_DC68	1E7A_0591	DAC1_BDAB	E445_14D7	33EB_CEC4	C900_CF7C	9220_8530	3C3E_FCE9	4B6D_A1C8	9712_7CA2	AFE2_42D5	5A17_68F9	397A_D8A5
CRC00	DF39_B3F3	8977_7074	7D09_52F9	AF5D_B427	F5D6_0370	B640_9499	5C3C_F3B2	EEA6_BA43	B8FF_BAD4	71E0_7CEF	EBEA_5458	C6DB_60B8	72F1_D461	9AA1_C1B3	360D_27A8	E8D6_4863

4.2.9 CXL.cache-Side Poison and Viral

4.2.9.1 Viral

Viral is a containment feature as described in Section 11.4, “CXL Viral Handling” on page 198. As such, when the local socket is in a viral state, it is the responsibility of all off-die interfaces to convey this state to the remote side for appropriate handling. CXL.cache/mem side conveys viral status information. As soon as the viral status is detected locally, the link layer forces a CRC error on the next outgoing flit. If there is no traffic to send, the transmitter will send a LLCRD flit with a CRC error. It then embeds viral status information in the LLR.Ack message it generates as part of the defined CRC error recovery flow.

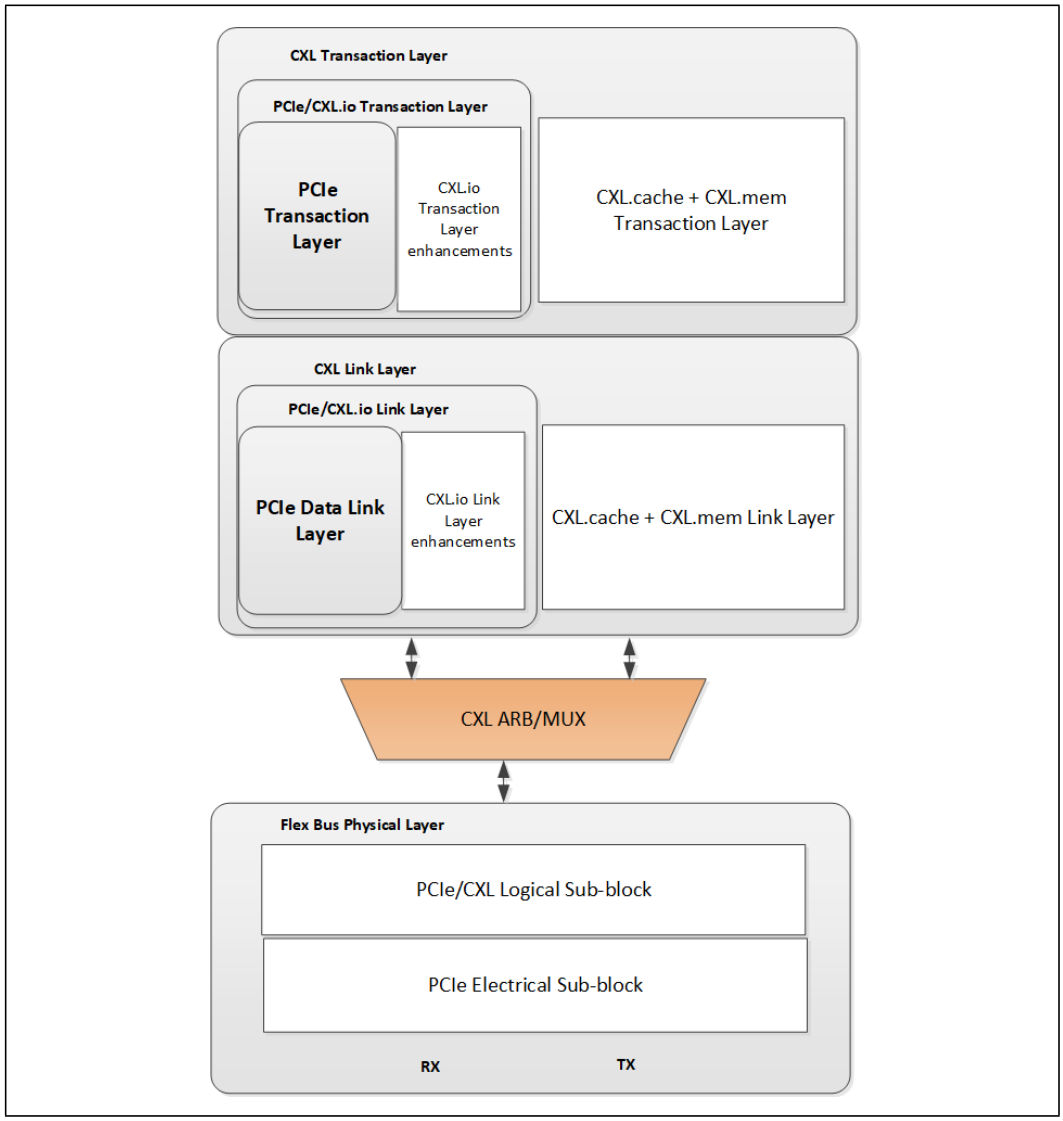
There are two primary benefits to this methodology. First, by using the LLR.Ack to convey viral status, we do not have to allocate a bit for this in protocol flits. Second, it allows immediate indication of viral and reduces the risk of race conditions between the viral distribution path and the datapath. These risks could be particularly exacerbated by the large CXL.cache flit size and the potential limitations in which components (header, slots) allocate dedicated fields for viral indication.

§ §

5.0 Compute Express Link ARB/MUX

The figure below shows where the CXL ARB/MUX exists in the Flex Bus layered hierarchy. The ARB/MUX provides dynamic muxing of the CXL.io and CXL.cache/CXL.mem link layer control and data signals to interface with the Flex Bus physical layer.

Figure 78. Flex Bus Layers -- CXL ARB/MUX Highlighted



In the transmit direction, the ARB/MUX arbitrates between requests from the CXL link layers and multiplexes the data. It also processes power state transition requests from the link layers: resolving them to a single request to forward to the physical layer, maintaining virtual link state machines (vLSMs) for each link layer interface, and generating ARB/MUX link management packets (ALMPs) to communicate the power state transition requests across the link on behalf of each link layer. Please refer to Section 9.4, Section 9.5, and Section 9.6 for more details on how the ALMPs are utilized in the overall flow for power state transitions. In PCIe mode or single protocol mode, the ARB/MUX is bypassed, and thus ALMP generation by the ARB/MUX is disabled.

In the receive direction, the ARB/MUX determines the protocol associated with the CXL flit and forwards the flit to the appropriate link layer. It also processes the ALMP packets, participating in any required handshakes and updating its vLSMs as appropriate.

5.1 Virtual LSM States

The ARB/MUX maintains vLSMs for each CXL link layer it interfaces with, transitioning the state based on power state transition requests it receives from the local link layer or from the remote ARB/MUX on behalf of a remote link layer. Table 46 below lists the different possible states for the vLSMs. PM States and Retrain are virtual states that can differ across interfaces (CXL.io and CXL.cache and CXL.mem), however all other states such as LinkReset, LinkDisable and LinkError are forwarded to the Link Layer and are therefore synchronized across interfaces.

Table 46. Virtual LSM States Maintained Per Link Layer Interface

Virtual LSM State	Description
Reset	Power-on default state during which initialization occurs
Active	Normal operational state
L1.1	Power savings state, from which the link can enter Active via Retrain
L1.2	Power savings state, from which the link can enter Active via Retrain
L1.3	Power savings state, from which the link can enter Active via Retrain
L1.4	Power savings state, from which the link can enter Active via Retrain
DAPM	Deepest Allowable PM State (not a resolved state; a request that resolves to an L1 substate)
SLEEP_L2	Power savings state, from which the link must go through Reset to reach Active
LinkReset	Reset propagation state resulting from software or hardware initiated reset
LinkError	Link Error state due to hardware detected errors
LinkDisable	Software controlled link disable state
Retrain	Transitory state that transitions to Active

Note: When the Physical Layer enters Hot-Reset or LinkDisable state, that state is communicated to all link layers as LinkReset or LinkDisable respectively. No ALMPs are exchanged, irrespective of who requested, for these transitions.

The ARB/MUX looks at the status of each vLSM to resolve to a single state request to forward to the physical layer as specified in Table 47. For Example if current vLSM[0] state is L1.1 (row = L1.1) and current vLSM[1] state is Active (column = Active), then the resolved request from the ARB/MUX to the Physical layer will be Active.

Table 47. ARB/MUX Multiple Virtual LSM Resolution Table

Resolved Request from ARB/MUX to Flex Bus Physical Layer (Row = current vLSM[0] state; Column = current vLSM[1] state)	Reset	Active	L1.1	L1.2	L1.3	L1.4	SLEEP_L2
Reset	RESET	Active	L1.1	L1.2	L1.3	L1.4	SLEEP_L2
Active	Active	Active	Active	Active	Active	Active	Active
L1.1	L1.1	Active	L1.1	L1.1	L1.1	L1.1	L1.1
L1.2	L1.2	Active	L1.1	L1.2	L1.2	L1.2	L1.2
L1.3	L1.3	Active	L1.1	L1.2	L1.3	L1.3	L1.3
L1.4	L1.4	Active	L1.1	L1.2	L1.3	L1.4	L1.4
SLEEP_L2	SLEEP_L2	Active	L1.1	L1.2	L1.3	L1.4	SLEEP_L2

Note: Table 47 is presented as a suggestion, not a requirement.

When any of the above link layers request for LinkReset or LinkError, the ARB/MUX will unconditionally propagate the request to the Physical layer ignoring the direction of the state consolidator.

Table 48 describes the conditions under which a vLSM transitions from one state to the next. A transition to the next state happens after all the steps in the trigger conditions column are complete. Some of the trigger conditions are sequential and indicate a series of actions from multiple sources. For example, on the transition from Active to L1.x state on an upstream port, the state transition will not occur until the vLSM has received a request to enter L1.x from the Link Layer followed by the vLSM sending a Request ALMP{L1.x} to the remote vLSM. Next the vLSM must wait to receive a Status ALMP{L1.x} from the remote vLSM. Once all these conditions are met in sequence, the vLSM will transition to the L1.x state as requested.

Table 48. ARB/MUX State Transition Table

Current vLSM State	Next State	Upstream Port Trigger Condition	Downstream Port Trigger Condition
Active	L1.x	Upon receiving a Request to enter L1.x from Link Layer, the ARB/MUX must initiate a Request ALMP{L1.x} and receive a Status ALMP{L1.x} from the remote vLSM	Upon receiving a Request to enter L1.x from Link Layer and receiving a Request ALMP{L1.x} from the Remote vLSM, the ARB/MUX must send Status ALMP{L1.x} to the remote vLSM
	L2	Upon receiving a Request to enter L2 from Link Layer the ARB/MUX must initiate a Request ALMP{L2} and receive a Status ALMP{L2} from the remote vLSM	Upon receiving a Request to enter L2 from Link Layer and receiving a Request ALMP{L2} from the Remote vLSM the ARB/MUX must send Status ALMP{L2} to the remote vLSM
L1 (Physical Layer LTSSM also in L1)	Retrain (Physical LTSSM in Recovery)	Upon receiving an ALMP Active request from remote ARB/MUX	Upon receiving an ALMP Active request from remote ARB/MUX
L1 (Physical Layer LTSSM in L0)	Retrain (Physical Layer LTSSM in L0)	Upon receiving an ALMP Active request from remote ARB/MUX	Upon receiving an ALMP Active request from remote ARB/MUX

Table 48. ARB/MUX State Transition Table

Current vLSM State	Next State	Upstream Port Trigger Condition	Downstream Port Trigger Condition
Active (Physical Layer in Recovery)	Retrain	LogPHY enters Retrain from Active state	LogPHY enters Retrain from Active state
Retrain (LTSSM in Recovery)	Active	Link Layer stops requesting Retrain and after State Status ALMP is sent and received which resolves to Active state	Link Layer stops requesting Retrain and after State Status ALMP is sent and received which resolves to Active state
Retrain (Exit from L1 state) (LTSSM in Recovery)	Active	Link Layer is requesting Active and the following conditions are met: Sent and received State Status ALMP; Entry to Active ALMP exchange protocol is complete (See Section 5.1.1.2)	Link Layer is requesting Active and the following conditions are met: Sent and received State Status ALMP; Entry to Active ALMP exchange protocol is complete (See Section 5.1.1.2)
ANY (Except Disable/ LinkError)	LinkReset	Indication of LinkReset from Physical Layer	Indication of LinkReset from Physical Layer
ANY (Except LinkError)	Disabled	Indication of Disabled from Physical Layer	Indication of Disabled from Physical Layer
ANY	LinkError	Directed to enter LinkError from Link Layer or indication of LinkError from Physical Layer	Directed to enter LinkError from Link Layer or indication of LinkError from Physical Layer
Retrain	LinkError	Implementation Specific	Implementation Specific
LinkError	Reset	Implementation Specific	Implementation Specific
LinkReset	Reset	Implementation Specific	Implementation Specific
Reset	Active	Link Layer is asking for Active and Entry to Active ALMP exchange protocol is complete (See Section 5.1.1.2)	Link Layer is asking for Active and Entry to Active ALMP exchange protocol is complete (See Section 5.1.1.2)

5.1.1 Rules for Virtual LSM State Transitions Across Link

This section refers to vLSM state transitions.

5.1.1.1 General Rules

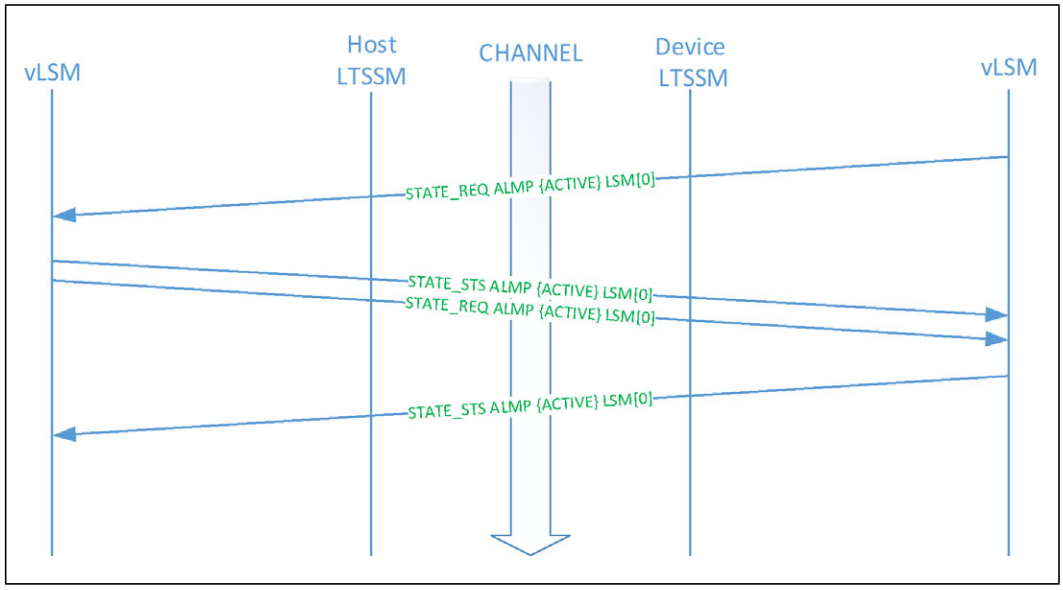
- The link cannot operate for any other protocols if CXL.io protocol is down. (CXL.io operation is a minimum requirement)

5.1.1.2 Entry to Active Exchange Protocol

The ALMP protocol required for the entry to active consists of 4 ALMP exchanges between the local and remote vLSMs as seen in Figure 79. Entry to active begins with a Active State Request ALMP sent to the remote vLSM which responds with a Active State Status ALMP. The only valid response to an Active State Request is an Active State Status once the corresponding vLSM is ready. The remote vLSM must also send an Active State Request ALMP to the local vLSM which responds with an Active State Status ALMP.

Once all four ALMPs are received, the vLSM states transition to Active State.

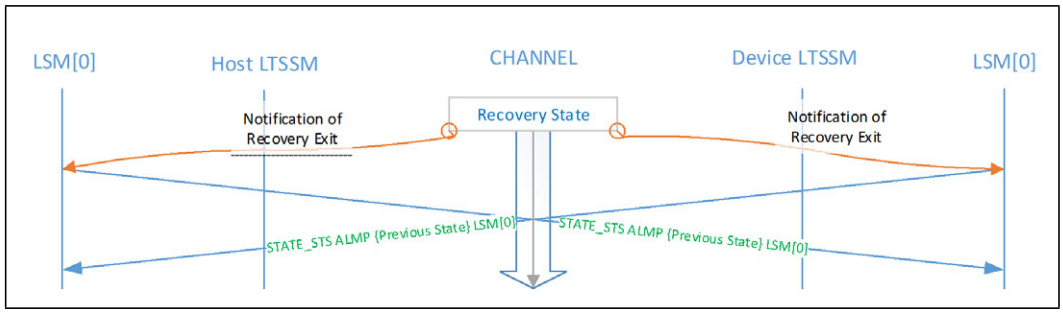
Figure 79. Entry to Active Protocol Exchange



5.1.1.3 Status Synchronization Protocol

As a part of Recovery, all active vLSMs transition into the Retrain state. A State Status ALMP is sent by each vLSM after the indication of LTSSM Recovery exit is received, as shown in Figure 80. The exchange of State Status ALMPs is all that is needed to synchronize the vLSM. The state indicated in the State Status ALMPs for synchronization is the state of the vLSM before entry to LTSSM Recovery. Therefore the ARB/MUX must take a snapshot of its vLSM states when notified that the Physical Layer enters Recovery and before it transitions its vLSMs to Retrain.

Figure 80. Status Synchronization



5.1.1.4 State Request ALMP

The following rules apply for sending a State Request ALMP. A State Request ALMP is sent to request a state change to Active or PM. For PM, the request can only be initiated by the Upstream ARB/MUX.

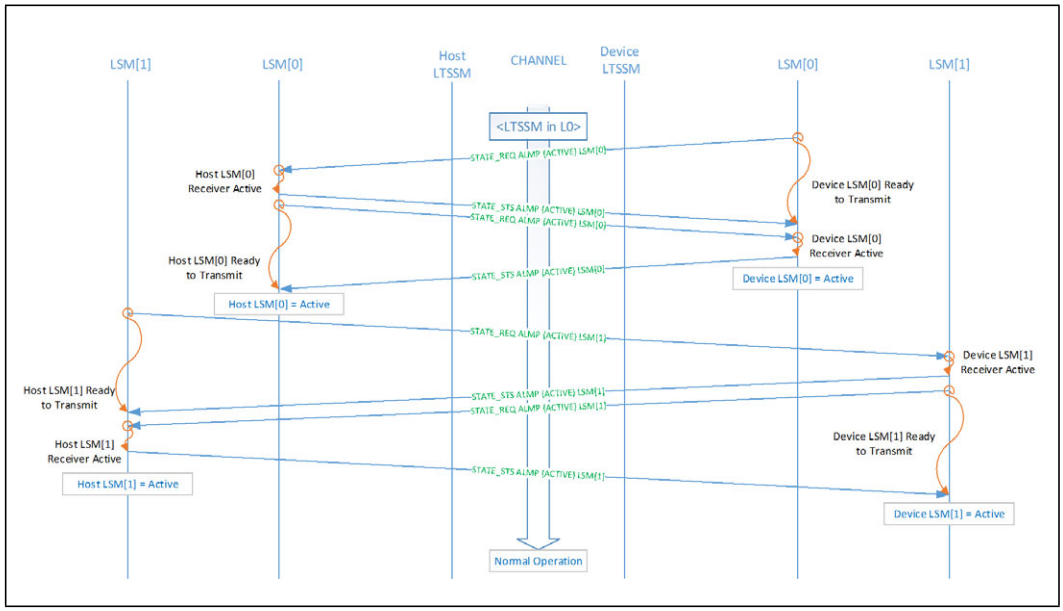
5.1.1.4.1 For Entry Into Active

- All Recovery state operations must complete before the entry to Active sequence starts
- An ALMP State Request is sent to initiate the entry into Active State.

- A vLSM must send a Request and receive a Status before the transmitter is considered active.

Figure 81 shows an example of entry into the Active state. The flows in Figure 81 show four independent actions (ALMP handshakes) that may not necessarily happen in the order or small time-frame shown. The vLSM transmitter and receiver may become active independently. Both transmitter and receiver must be active before the vLSM state is Active. The transmitter becomes active after a vLSM has transmitted and received Status ALMP{Active}. The receiver becomes active after a vLSM receives a Request ALMP{Active} and sends a Status ALMP{Active} in return.

Figure 81. CXL Entry to Active Flow

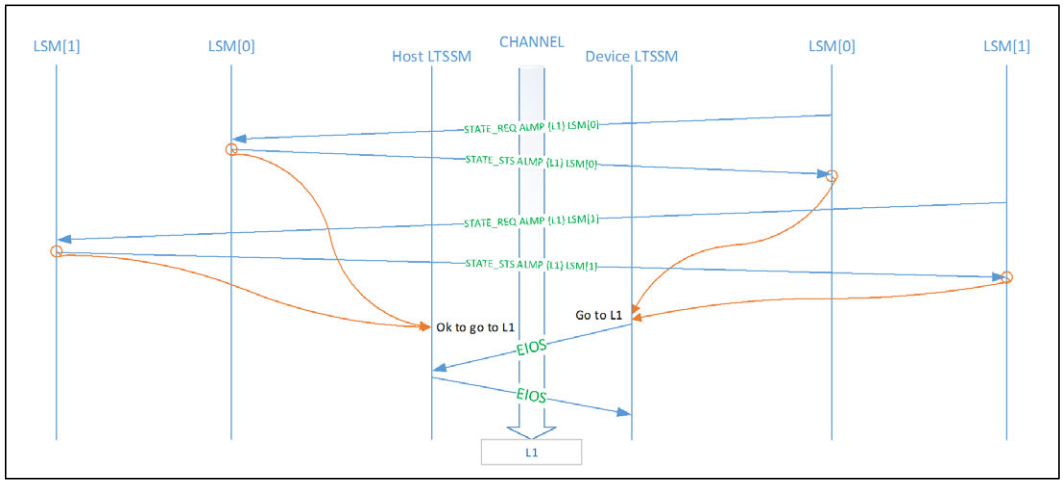


5.1.1.4.2 For Entry into PM State (L1/L2)

- An ALMP State Request is sent to initiate the entry into PM States
- A vLSM must send a Request and receive a Status before the transmitter is placed into a low power state.

Figure 82 shows an example of Entry to PM State (L1) initiated by the device side ARB/MUX. Each vLSM will be ready to enter L1 State once the vLSM has sent a Request ALMP{L1} and received a Status ALMP{L1} in return or the vLSM has received a Request ALMP{L1} and sent a Status ALMP{L1} in return. The vLSMs operate independently and actions may not complete in the order or the timeframe shown. Once all vLSMs are ready to enter PM State (L1), the Channel will complete EIOS exchange and enter L1.

Figure 82. CXL Entry to PM State



5.1.1.5 State Status ALMP

5.1.1.5.1 When State Request ALMP is received

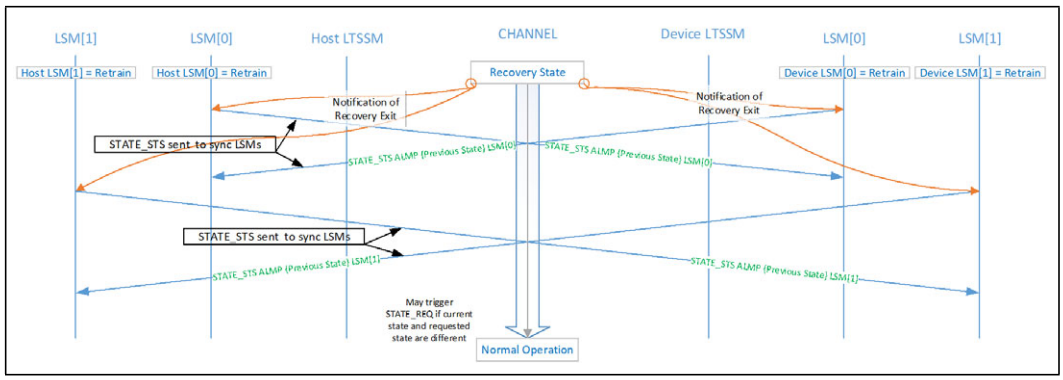
- A State Status ALMP is sent after a State Request ALMP is received for entry into Active State or PM States when entry to the PM state is accepted. No State Status ALMP is sent if the PM state is not accepted. See Section 9.4, “Compute Express Link Power Management” on page 186 for more details.

5.1.1.5.2 Recovery State

- A vLSM cannot conduct any other communication on the link coming out of recovery until it has sent and received State Status ALMP.
- The vLSM will enter Recovery if a State Status ALMP is received without a State Request first being sent by the vLSM except when the vLSM is coming out of Retrain, as shown in Figure 85.

Figure 83 shows a general example of Recovery exit. The state sent in the State Status ALMP exchange is the state of the vLSM prior to it going into the Retrain state.

Figure 83. CXL Recovery Exit Flow



On Exit from Recovery, the vLSMs on either side of the channel will send a Status ALMP in order to synchronize the LSMs. The Status ALMP will provide the state of the vLSM prior to it entering Retrain. The Status ALMPs for synchronization may trigger a State Request ALMP if the provided state and the requested state are not the same, as seen in Figure 84. The ALMP for synchronization may trigger a re-entry to recovery if the vLSMs on either side of the channel are not in the same state out of Retrain, as seen in Figure 85. If the provided states from both vLSMs are the same as the requested state prior to the Recovery, the vLSMs are considered synchronized and will continue normal operation, see figure Figure 83.

Figure 84 shows an example of the exit from a PM State (L1) through Recovery. The Host LSM[0] in L1 state receives the Active Request, and the link enters Recovery. After the exit from recovery, each vLSM sends Status ALMP{L1} (State of the vLSM before Recovery entry) to synchronize the vLSMs. Because the state in the Status ALMP for synchronization is not equal to the requested state that triggered the entry to recovery, Request ALMP{Active} and Status ALMP{Active} handshakes are completed to enter Active State.

Figure 84. CXL Exit from PM State

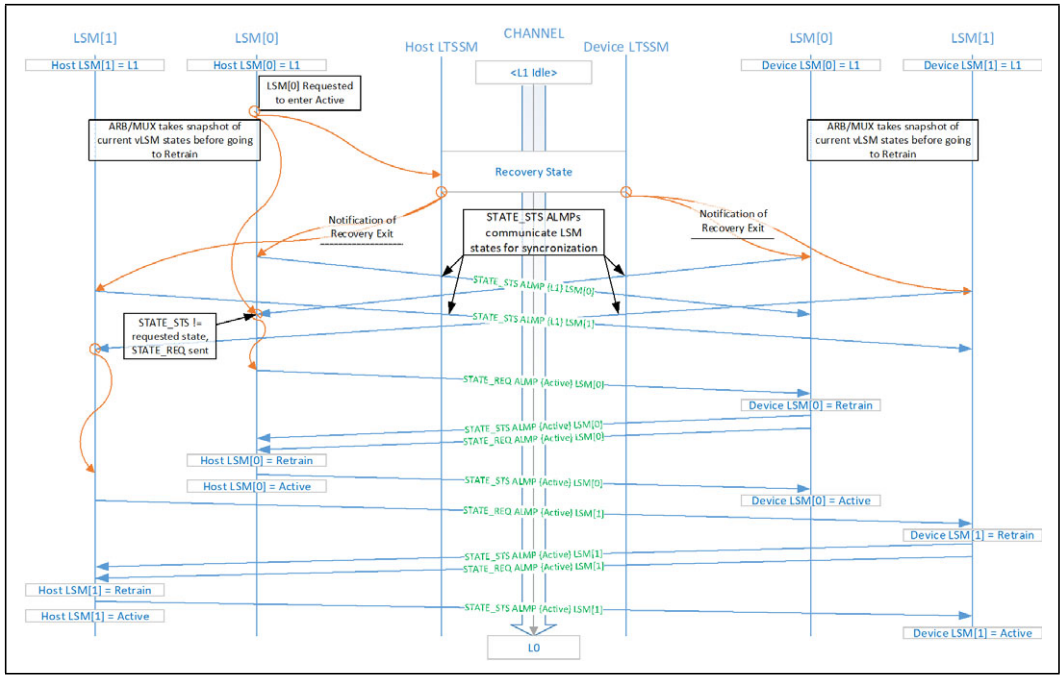
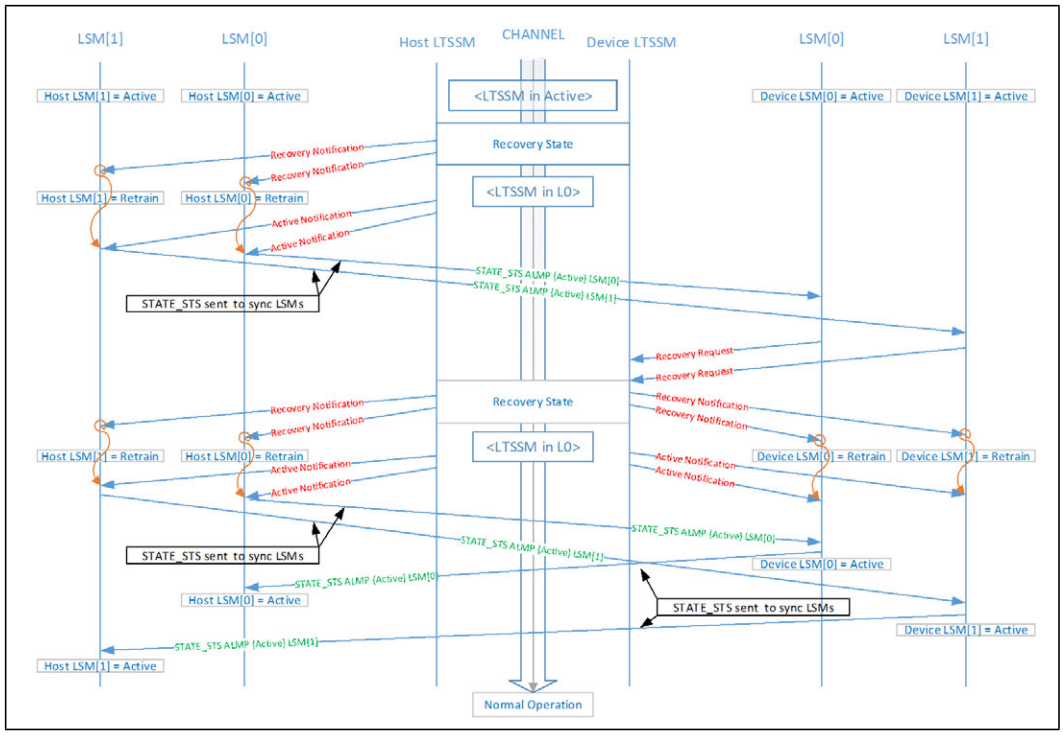


Figure 85 shows an example of error in the Recovery flow. The error shown is one example of how an error may occur where the device does not properly receive the request to go to Retrain and therefore remains in Active. On indication of exit from Recovery from the LTSSM the Host LSMs send Status ALMP{Active} to synchronize vLSM across the Channel. Since the Device side received Status ALMP{Active} without first sending a Request ALMP or being in Retrain State, the Device LSM requests the Physical Layer to enter recovery. Recovery flow is then entered by both Host and Device and exited and synchronized correctly.

Figure 85. CXL Recovery Error Flow



5.1.1.6 Unexpected ALMPs

The following situations describe circumstances where an unexpected ALMP will cause entry to Retrain State:

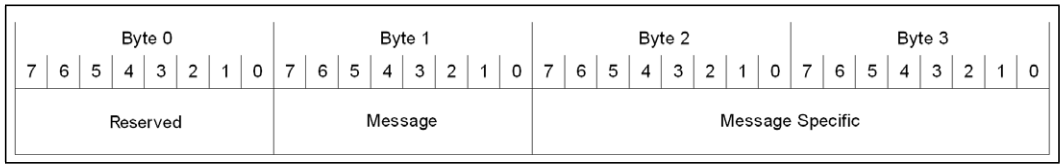
- When in the Synchronization portion on exit from Retrain, any ALMP other than a Status ALMP is considered an unexpected ALMP and will trigger recovery.
- When an Active Request ALMP has been sent, any ALMP other than an Active State Status ALMP that is sent in response is considered an unexpected ALMP and will trigger recovery.

5.2 ARB/MUX Link Management Packets

The ARB/MUX uses ALMPs to communicate virtual link state transition requests and responses associated with each link layer to the remote ARB/MUX.

An ALMP is a 1DW packet with format shown in Figure 86 below. The message code used in Byte 1 of the ALMP is 0000_1000b. This 1DW packet is replicated four times on the lower 16-bytes of a 528-bit flit to provide data integrity protection; the flit is zero padded on the upper bits. If the ARB/MUX detects an error in the ALMP, it initiates a retrain of the link.

Figure 86. ARB/MUX Link Management Packet Format



Bytes 2 and 3 of the ALMP packet is as shown in Table 49 below. ALMPs can be request or status type. The local ARB/MUX initiates transition of a remote vLSM using a request ALMP. After receiving a request ALMP, the local ARB/MUX processes the transition request and returns a status ALMP to indicate that the transition has occurred. If the transition request is not accepted, no status ALMP is sent and both local and remote vLSMs remain in their current state.

Table 49. ALMP Byte 2 and Byte 3 Encoding

Byte2 Bit	Description
3:0	Virtual LSM State Encoding: 0000: NOP/Reset (for Status ALMP only) 0001: ACTIVE 0010: Reserved 0011: DEEPEST ALLOWABLE PM STATE/Reserved (for Status ALMP only) 0100: IDLE_L1.1 0101: IDLE_L1.2 0110: IDLE_L1.3 0111: IDLE_L1.4 1000: L2 1001: LINKRESET (for Status ALMP only) 1010: LINKERROR (for Status ALMP only) 1011: Retrain (for Status ALMP only) 1100: DISABLE (for Status ALMP only) 1101: Reserved 1110: Reserved 1111: Reserved
6:4	Reserved
7	Request/Status Type 1: Virtual LSM Request ALMP 0: Virtual LSM Status ALMP
Byte3 Bit	**Description**
3:0	Virtual LSM Instance Number: Indicates the targeted Virtual LSM interface when there are multiple Virtual LSMs present. 0000: Reserved 0001: ALMP for CXL.io 0010: ALMP for CXL.cache and CXL.mem Note: When a single Virtual LSM is present, the ARB/MUX should be bypassed.
7:4	Reserved

5.2.1 ARB/MUX Bypass Feature

The ARB/MUX must disable generation of ALMPs when there is no dynamic multiplexing of CXL.io with other CXL protocols, e.g., when the Flex Bus link is operating in PCIe mode or when only CXL.io protocol is enabled. Determination of the bypass condition can be via hwinit or during link training.

5.3 Arbitration and Data Multiplexing/Demultiplexing

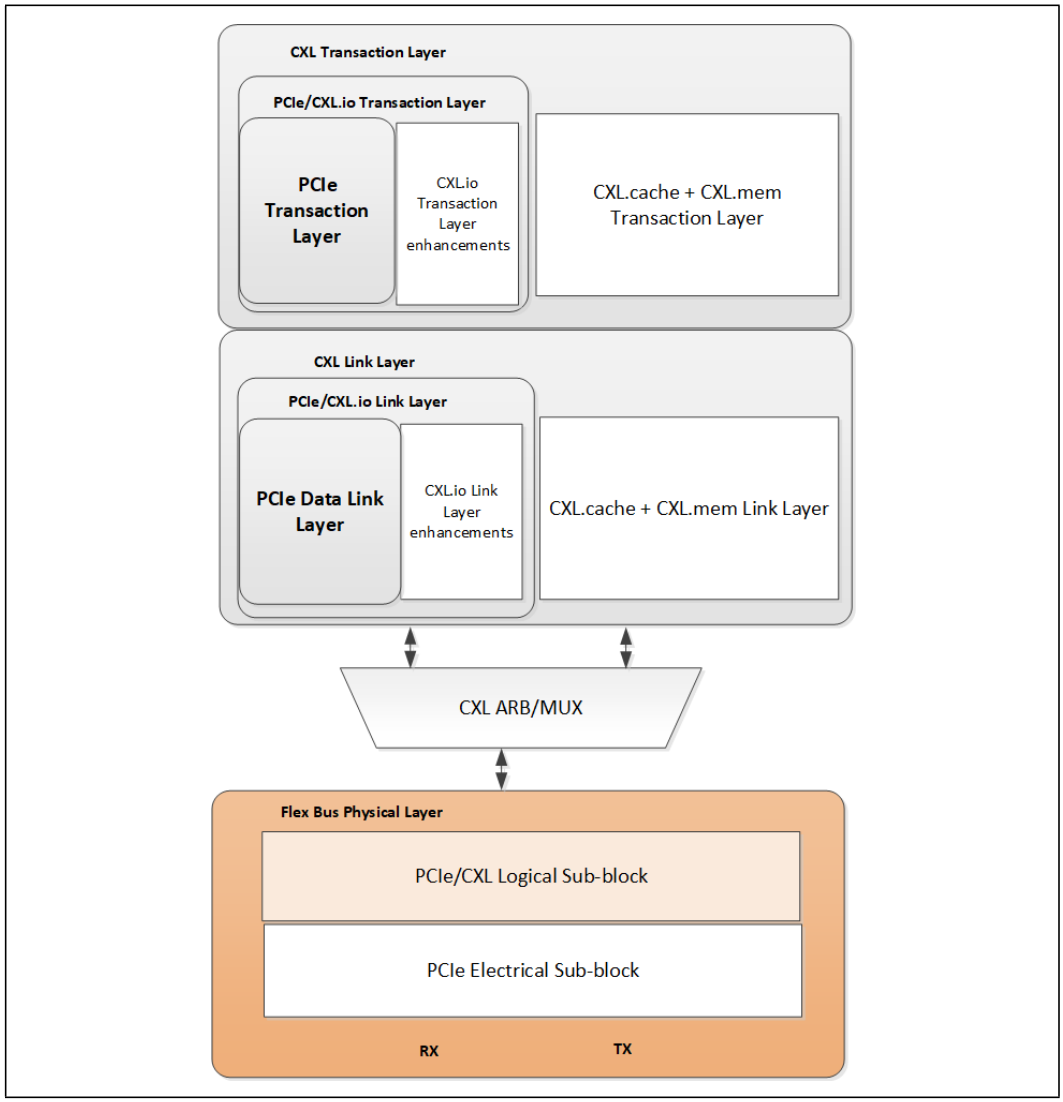
The ARB/MUX is responsible for arbitrating between requests from the CXL link layers and multiplexing the data based on the arbitration results. The arbitration policy is implementation specific as long as it satisfies the timing requirements of the higher level protocols being transferred over the Flex Bus link. Additionally, there must be a way to program the relative arbitration weightages associated with the CXL.io and CXL.cache+CXL.mem link layers as they arbitrate to transmit traffic over the Flex Bus link. See Section 7.2.2.2.1 for more details. Interleaving of traffic between different CXL protocols is done at the 528-bit flit boundary.

§ §

6.0 Flex Bus Physical Layer

6.1 Overview

Figure 87. Flex Bus Layers -- Physical Layer Highlighted



The figure above shows where the Flex Bus physical layer exists in the Flex Bus layered hierarchy. On the transmit side, the Flex Bus physical layer prepares data received from either the PCIe link layer or the CXL link layer for transmission across the Flex Bus link.

On the receive side, the Flex Bus physical layer deserializes the data received on the Flex Bus link and converts it to the appropriate format to forward to the PCIe/CXL link layer. The Flex Bus physical layer consists of a logical sub-block, aka the logical PHY, and an electrical sub-block. The logical PHY operates in PCIe mode during initial link training and switches over to CXL mode, if appropriate, depending on the results of alternate mode negotiation, during recovery after training to 2.5 GT/s. The electrical sub-block follows the PCIe specification.

In CXL mode, normal operation occurs at x16 link width and 32 GT/s link speed. Bifurcation (aka link subdivision) into x8 and x4 widths is supported in CXL mode. Degraded modes of operation include 8 GT/s or 16 GT/s link speed and smaller link widths down to x1. Table 50 summarizes the supported CXL combinations. In PCIe mode, the link supports all widths and speeds defined in the PCIe specification, as well as the ability to bifurcate.

Table 50. Flex Bus.CXL Link Speeds and Widths for Normal and Degraded Mode

Link Speed	Native Width	Degraded Modes Supported
32 GT/s	x16	x16 @16 GT/s or 8 GT/s; x8, x4, x2, or x1 @32 GT/s or 16 GT/s or 8 GT/s
32 GT/s	x8	x8 @16 GT/s or 8 GT/s; x4, x2, or x1 @32 GT/s or 16 GT/s or 8 GT/s
32 GT/s	x4	x4 @16 GT/s or 8 GT/s; x2 or x1 @32 GT/s or 16 GT/s or 8 GT/s
32 GT/s	x2	x2 @16 GT/s or 8 GT/s; x1 @32 GT/s or 16 GT/s or 8 GT/s

This chapter focuses on the details of the logical PHY. The Flex Bus logical PHY is based on the PCIe logical PHY; PCIe mode of operation follows the PCIe specification exactly while Flex Bus.CXL mode has deltas from PCIe that affect link training and framing. Please refer to the “Physical Layer Logical Block” chapter of the PCI Express Base Specification for details on PCIe mode of operation. The Flex Bus.CXL deltas are described in this chapter.

6.2 Flex Bus.CXL Framing and Packet Layout

The Flex Bus.CXL framing and packet layout is described in this section for x16,x8,x4, x2, and x1 widths.

6.2.1 Ordered Set Blocks and Data Blocks

Flex Bus.CXL uses the PCIe concept of ordered set blocks and data blocks. Each block spans 128 bits per lane and potentially two bits of sync header per lane.

Ordered set blocks are used for training, entering and exiting electrical idle, transitions to data blocks, and clock tolerance compensation; they are the same as defined in the PCIe base specification. A 2-bit sync header with value 01b is inserted before each 128 bits transmitted per lane in an ordered set block when 128/130b encoding is used; in the latency optimized mode, there is no sync header.

Data blocks are used for transmission of the flits received from the CXL link layer. A 16-bit Protocol ID field is associated with each 528-bit flit payload (512 bits of payload + 16 bits of CRC) received from the link layer, which is striped across the lanes on an 8-

bit granularity; the placement of the protocol ID depends on the width. A 2-bit sync header with value 10b is inserted before every 128 bits transmitted per lane in a data block when 128/130b encoding is used; in the latency optimized mode, there is no sync header. A 528-bit flit may traverse the boundary between data blocks.

Transitions between ordered set blocks and data blocks are indicated in a couple of ways. One way is via the 2-bit sync header of 01b for ordered set blocks and 10b for data blocks. The second way is via the use of Start of Data Stream (SDS) ordered sets and End of Data Stream (EDS) tokens. Unlike PCIe where the EDS token is explicit, Flex Bus.CXL encodes the EDS token in the protocol ID value.

6.2.2 Protocol ID[15:0]

The 16-bit protocol ID field specifies whether the transmitted flit is CXL.io, CXL.cache/CXL.mem, or some other payload. The table below provides a list of valid 16-bit protocol ID encodings. Encodings that include an implied EDS token signify that the next block is an ordered set block. Implied EDS tokens can only occur with the last flit transmitted in a data block; flits that cross the data block boundary cannot be associated with an implied EDS token.

NULL flits are inserted into the data stream by the physical layer when there are no valid flits available from the link layer. A NULL flit transferred with an implied EDS token ends precisely at the data block boundary; these are variable length flits, up to 528 bits, intended to facilitate transition to ordered set blocks as quickly as possible. A NULL flit is comprised of all zeros payload.

An 8-bit encoding with a hamming distance of four is replicated to create the 16-bit encoding for error protection against bit flips. A correctable protocol ID framing error is logged but no further error handling action is taken if only one 8-bit encoding group looks incorrect; the correct 8-bit encoding group is used for normal processing. If both 8-bit encoding groups are incorrect, an uncorrectable protocol ID framing error is logged, the flit is dropped, and the physical layer enters into recovery to retrain the link.

The physical layer is responsible for dropping any flits it receives with invalid protocol IDs. This includes dropping any flits with unexpected protocol IDs that correspond to Flex Bus defined protocols that have not been enabled during negotiation. When a flit is dropped due to an unexpected protocol ID, the physical layer logs an unexpected protocol ID error in the Flex Bus DVSEC Port Status register.

Please refer to Section 6.2.9 for additional details about protocol ID error detection and handling.

Table 51. Flex Bus.CXL Protocol IDs (Sheet 1 of 2)

Protocol ID[15:0]	Description
0000_0000_0000_0000	Reserved
1111_1111_1111_1111	CXL.io
1101_0010_1101_0010	CXL.io with implied EDS token
0101_0101_0101_0101	CXL.cache/CXL.mem
1000_0111_1000_0111	CXL.cache/CXL.mem with implied EDS token
1001_1001_1001_1001	NULL flit (generated by the Physical Layer)
0100_1011_0100_1011	NULL flit with implied EDS token: Variable length flit containing NULLs that ends precisely at the data block boundary (generated by the Physical Layer)

Table 51. Flex Bus.CXL Protocol IDs (Sheet 2 of 2)

Protocol ID[15:0]	Description
1100_1100_1100_1100	CXL ARB/MUX Link Management Packets (ALMPs)
0001_1110_0001_1110	CXL ARB/MUX Link Management Packets (ALMPs) with implied EDS token
All Others	Reserved

6.2.3 x16 Packet Layout

Figure 88 below shows the x16 packet layout. First, the 16-bits of protocol ID are transferred, split on an 8-bit granularity across consecutive lanes; this is followed by transfer of the 528-bit flit, striped across the lanes on an 8-bit granularity. Depending on the symbol time, as labeled on the leftmost column in the figure, the Protocol ID plus flit transfer may start on lane 0, lane 4, lane 8, or lane 12. The pattern of transfer repeats after every 17 symbol times. The two-bit sync header shown in the figure, inserted after every 128 bits transferred per lane, is not present for the latency optimized mode where sync header bypass is negotiated.

Figure 88. Flex Bus x16 Packet Layout

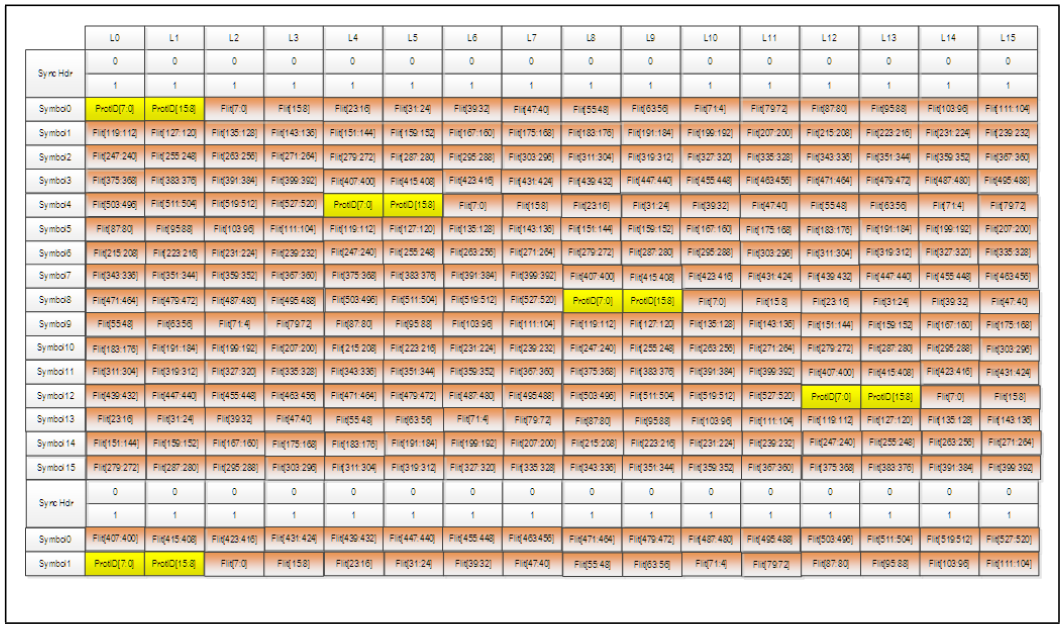
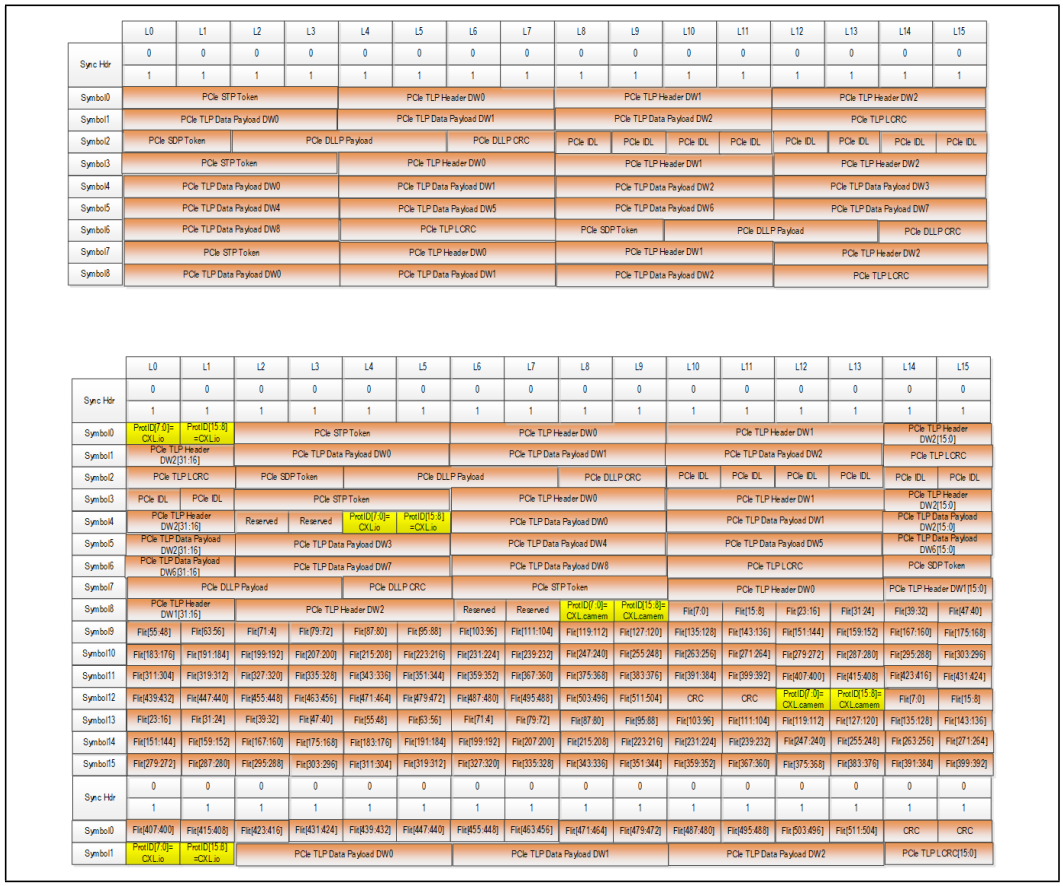


Figure 89 provides an example where CXL.io and CXL.cache/CXL.mem traffic is interleaved with an interleave granularity of two flits on a x16 link. The top figure shows what the CXL.io stream looks like before mapping to the Flex Bus lanes and before interleaving with CXL.cache/CXL.mem traffic; the framing rules follow the x16 framing rules specified in the PCI Express specification, as stated in Section 4.1. The bottom figure shows the final result when the two streams are interleaved on the Flex Bus lanes. For CXL.io flits, after transferring the 16-bit protocol ID, 512 bits are used to transfer CXL.io traffic and 16 bits are unused. For CXL.cachemem flits, after transferring the 16-bit protocol ID, 528 bits are used to transfer a CXL.cachemem flit. Please refer to Chapter 4.0, "Compute Express Link Link Layers" for more details on the

flit format. As this example illustrates, the PCIe TLPs and DLLPs encapsulated within the CXL.io stream may be interrupted by non-related CXL traffic if they cross a flit boundary.

Figure 89. Flex Bus x16 Protocol Interleaving Example



6.2.4 x8 Packet Layout

Figure 90 below shows the x8 packet layout. 16-bits of Protocol ID followed by a 528-bit flit are striped across the lanes on an 8-bit granularity. Depending on the symbol time, the Protocol ID plus flit transfer may start on lane 0 or lane 4. The pattern of transfer repeats after every 17 symbols. The two-bit sync header shown in the figure is not present for the latency optimized mode.

Figure 90. Flex Bus x8 Packet Layout

	L0	L1	L2	L3	L4	L5	L6	L7
Sync Hdr	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1
Symbol0	ProtID[7:0]	ProtID[15:8]	Flit[7:0]	Flit[15:8]	Flit[23:16]	Flit[31:24]	Flit[39:32]	Flit[47:40]
Symbol1	Flit[55:48]	Flit[63:56]	Flit[71:4]	Flit[79:72]	Flit[87:80]	Flit[95:88]	Flit[103:96]	Flit[111:104]
Symbol2	Flit[119:112]	Flit[127:120]	Flit[135:128]	Flit[143:136]	Flit[151:144]	Flit[159:152]	Flit[167:160]	Flit[175:168]
Symbol3	Flit[183:176]	Flit[191:184]	Flit[199:192]	Flit[207:200]	Flit[215:208]	Flit[223:216]	Flit[231:224]	Flit[239:232]
Symbol4	Flit[247:240]	Flit[255:248]	Flit[263:256]	Flit[271:264]	Flit[279:272]	Flit[287:280]	Flit[295:288]	Flit[303:296]
Symbol5	Flit[311:304]	Flit[319:312]	Flit[327:320]	Flit[335:328]	Flit[343:336]	Flit[351:344]	Flit[359:352]	Flit[367:360]
Symbol6	Flit[375:368]	Flit[383:376]	Flit[391:384]	Flit[399:392]	Flit[407:400]	Flit[415:408]	Flit[423:416]	Flit[431:424]
Symbol7	Flit[439:432]	Flit[447:440]	Flit[455:448]	Flit[463:456]	Flit[471:464]	Flit[479:472]	Flit[487:480]	Flit[495:488]
Symbol8	Flit[503:496]	Flit[511:504]	Flit[519:512]	Flit[527:520]	ProtID[7:0]	ProtID[15:8]	Flit[7:0]	Flit[15:8]
Symbol9	Flit[23:16]	Flit[31:24]	Flit[39:32]	Flit[47:40]	Flit[55:48]	Flit[63:56]	Flit[71:4]	Flit[79:72]
Symbol10	Flit[87:80]	Flit[95:88]	Flit[103:96]	Flit[111:104]	Flit[119:112]	Flit[127:120]	Flit[135:128]	Flit[143:136]
Symbol11	Flit[151:144]	Flit[159:152]	Flit[167:160]	Flit[175:168]	Flit[183:176]	Flit[191:184]	Flit[199:192]	Flit[207:200]
Symbol12	Flit[215:208]	Flit[223:216]	Flit[231:224]	Flit[239:232]	Flit[247:240]	Flit[255:248]	Flit[263:256]	Flit[271:264]
Symbol13	Flit[279:272]	Flit[287:280]	Flit[295:288]	Flit[303:296]	Flit[311:304]	Flit[319:312]	Flit[327:320]	Flit[335:328]
Symbol14	Flit[343:336]	Flit[351:344]	Flit[359:352]	Flit[367:360]	Flit[375:368]	Flit[383:376]	Flit[391:384]	Flit[399:392]
Symbol15	Flit[407:400]	Flit[415:408]	Flit[423:416]	Flit[431:424]	Flit[439:432]	Flit[447:440]	Flit[455:448]	Flit[463:456]
Sync Hdr	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1
Symbol0	Flit[471:464]	Flit[479:472]	Flit[487:480]	Flit[495:488]	Flit[503:496]	Flit[511:504]	Flit[519:512]	Flit[527:520]
Symbol1	ProtID[7:0]	ProtID[15:8]	Flit[7:0]	Flit[15:8]	Flit[23:16]	Flit[31:24]	Flit[39:32]	Flit[47:40]

Figure 91 illustrates how CXL.io and CXL.cache/CXL.mem traffic is interleaved on a x8 Flex Bus link. The same traffic from the x16 example in Figure 89 is mapped to a x8 link.

Figure 91. Flex Bus x8 Protocol Interleaving Example

	L0	L1	L2	L3	L4	L5	L6	L7
Sync Hdr	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1
Symbol0	ProtID[7:0]=CXL.io	ProtID[15:8]=CXL.io	PCIe STP Token				PCIe TLP Header DW0[15:0]	
Symbol1	PCIe TLP Header DW0[31:16]		PCIe TLP Header DW1				PCIe TLP Header DW2[15:0]	
Symbol2	PCIe TLP Header DW2[31:16]		PCIe TLP Data Payload DW0				PCIe TLP Data Payload DW1[15:0]	
Symbol3	PCIe TLP Data Payload DW1[31:16]		PCIe TLP Data Payload DW2				PCIe TLP LCRC	
Symbol4	PCIe TLP LCRC		PCIe SDP Token		PCIe DLLP Payload			
Symbol5	PCIe DLLP CRC		PCIe IDL	PCIe IDL	PCIe IDL	PCIe IDL	PCIe IDL	PCIe IDL
Symbol6	PCIe IDL	PCIe IDL	PCIe STP Token				PCIe TLP Header DW0[15:0]	
Symbol7	PCIe TLP Header DW0[31:16]		PCIe TLP Header DW1				PCIe TLP Header DW2[15:0]	
Symbol8	PCIe TLP Header DW2[31:16]		Reserved	Reserved	ProtID[7:0]=CXL.io	ProtID[15:8]=CXL.io	PCIe TLP Data Payload DW0[15:0]	
Symbol9	PCIe TLP Data Payload DW0[31:16]		PCIe TLP Data Payload DW1				PCIe TLP Data Payload DW2[15:0]	
Symbol10	PCIe TLP Data Payload DW2[31:16]		PCIe TLP Data Payload DW3				PCIe TLP Data Payload DW4[15:0]	
Symbol11	PCIe TLP Data Payload DW4[31:16]		PCIe TLP Data Payload DW5				PCIe TLP Data Payload DW6[15:0]	
Symbol12	PCIe TLP Data Payload DW6[31:16]		PCIe TLP Data Payload DW7				PCIe TLP Data Payload DW8[15:0]	
Symbol13	PCIe TLP Data Payload DW8[31:16]		PCIe TLP LCRC				PCIe SDP Token	
Symbol14	PCIe DLLP Payload				PCIe DLLP CRC		PCIe STP Token[15:0]	
Symbol15	PCIe STP Token[31:16]		PCIe TLP Header DW0				PCIe TLP Header DW1[15:0]	
Sync Hdr	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1
Symbol0	PCIe TLP Header DW1[31:16]		PCIe TLP Header DW2				Reserved	Reserved
Symbol1	ProtID[7:0]=CXL.camem	ProtID[15:8]=CXL.camem	Flit[7:0]	Flit[15:8]	Flit[23:16]	Flit[31:24]	Flit[39:32]	Flit[47:40]
Symbol2	Flit[55:48]	Flit[63:56]	Flit[71:4]	Flit[79:72]	Flit[87:80]	Flit[95:88]	Flit[103:96]	Flit[111:104]
Symbol3	Flit[119:112]	Flit[127:120]	Flit[135:128]	Flit[143:136]	Flit[151:144]	Flit[159:152]	Flit[167:160]	Flit[175:168]
Symbol4	Flit[183:176]	Flit[191:184]	Flit[199:192]	Flit[207:200]	Flit[215:208]	Flit[223:216]	Flit[231:224]	Flit[239:232]
Symbol5	Flit[247:240]	Flit[255:248]	Flit[263:256]	Flit[271:264]	Flit[279:272]	Flit[287:280]	Flit[295:288]	Flit[303:296]
Symbol6	Flit[311:304]	Flit[319:312]	Flit[327:320]	Flit[335:328]	Flit[343:336]	Flit[351:344]	Flit[359:352]	Flit[367:360]
Symbol7	Flit[375:368]	Flit[383:376]	Flit[391:384]	Flit[399:392]	Flit[407:400]	Flit[415:408]	Flit[423:416]	Flit[431:424]
Symbol8	Flit[439:432]	Flit[447:440]	Flit[455:448]	Flit[463:456]	Flit[471:464]	Flit[479:472]	Flit[487:480]	Flit[495:488]
Symbol9	Flit[503:496]	Flit[511:504]	CRC	CRC	ProtID[7:0]=CXL.camem	ProtID[15:8]=CXL.camem	Flit[7:0]	Flit[15:8]
Symbol10	Flit[23:16]	Flit[31:24]	Flit[39:32]	Flit[47:40]	Flit[55:48]	Flit[63:56]	Flit[71:4]	Flit[79:72]
Symbol11	Flit[87:80]	Flit[95:88]	Flit[103:96]	Flit[111:104]	Flit[119:112]	Flit[127:120]	Flit[135:128]	Flit[143:136]
Symbol12	Flit[151:144]	Flit[159:152]	Flit[167:160]	Flit[175:168]	Flit[183:176]	Flit[191:184]	Flit[199:192]	Flit[207:200]
Symbol13	Flit[215:208]	Flit[223:216]	Flit[231:224]	Flit[239:232]	Flit[247:240]	Flit[255:248]	Flit[263:256]	Flit[271:264]
Symbol14	Flit[279:272]	Flit[287:280]	Flit[295:288]	Flit[303:296]	Flit[311:304]	Flit[319:312]	Flit[327:320]	Flit[335:328]
Symbol15	Flit[343:336]	Flit[351:344]	Flit[359:352]	Flit[367:360]	Flit[375:368]	Flit[383:376]	Flit[391:384]	Flit[399:392]
Sync Hdr	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1
Symbol0	Flit[407:400]	Flit[415:408]	Flit[423:416]	Flit[431:424]	Flit[439:432]	Flit[447:440]	Flit[455:448]	Flit[463:456]
Symbol1	Flit[471:464]	Flit[479:472]	Flit[487:480]	Flit[495:488]	Flit[503:496]	Flit[511:504]	CRC	CRC
Symbol2	ProtID[7:0]=CXL.io	ProtID[15:8]=CXL.io	PCIe TLP Data Payload DW0				PCIe TLP Data Payload DW1[15:0]	
Symbol3	PCIe TLP Data Payload DW1[31:16]		PCIe TLP Data Payload DW2				PCIe TLP LCRC[15:0]	

6.2.5 x4 Packet Layout

Figure 92 below shows the x4 packet layout. 16-bits of Protocol ID followed by a 528-bit flit are striped across the lanes on an 8-bit granularity. The Protocol ID plus flit transfer always starts on lane 0; the entire transfer takes 17 symbols. The two-bit sync header shown in the figure is not present for the latency optimized mode.

Figure 92. Flex Bus x4 Packet Layout

	L0	L1	L2	L3
Sync Hdr	0	0	0	0
	1	1	1	1
Symbol0	ProtID[7:0]	ProtID[15:8]	Flit[7:0]	Flit[15:8]
Symbol1	Flit[23:16]	Flit[31:24]	Flit[39:32]	Flit[47:40]
Symbol2	Flit[55:48]	Flit[63:56]	Flit[71:4]	Flit[79:72]
Symbol3	Flit[87:80]	Flit[95:88]	Flit[103:96]	Flit[111:104]
Symbol4	Flit[119:112]	Flit[127:120]	Flit[135:128]	Flit[143:136]
Symbol5	Flit[151:144]	Flit[159:152]	Flit[167:160]	Flit[175:168]
Symbol6	Flit[183:176]	Flit[191:184]	Flit[199:192]	Flit[207:200]
Symbol7	Flit[215:208]	Flit[223:216]	Flit[231:224]	Flit[239:232]
Symbol8	Flit[247:240]	Flit[255:248]	Flit[263:256]	Flit[271:264]
Symbol9	Flit[279:272]	Flit[287:280]	Flit[295:288]	Flit[303:296]
Symbol10	Flit[311:304]	Flit[319:312]	Flit[327:320]	Flit[335:328]
Symbol11	Flit[343:336]	Flit[351:344]	Flit[359:352]	Flit[367:360]
Symbol12	Flit[375:368]	Flit[383:376]	Flit[391:384]	Flit[399:392]
Symbol13	Flit[407:400]	Flit[415:408]	Flit[423:416]	Flit[431:424]
Symbol14	Flit[439:432]	Flit[447:440]	Flit[455:448]	Flit[463:456]
Symbol15	Flit[471:464]	Flit[479:472]	Flit[487:480]	Flit[495:488]
Sync Hdr	0	0	0	0
	1	1	1	1
Symbol0	Flit[503:496]	Flit[511:504]	Flit[519:512]	Flit[527:520]
Symbol1	ProtID[7:0]	ProtID[15:8]	Flit[7:0]	Flit[15:8]

6.2.6 x2 Packet Layout

The x2 packet layout looks very similar to the x4 packet layout in that the Protocol ID aligns to lane 0. 16-bits of Protocol ID followed by a 528-bit flit are striped across two lanes on an 8-bit granularity, taking 34 symbols to complete the transfer.

6.2.7 x1 Packet Layout

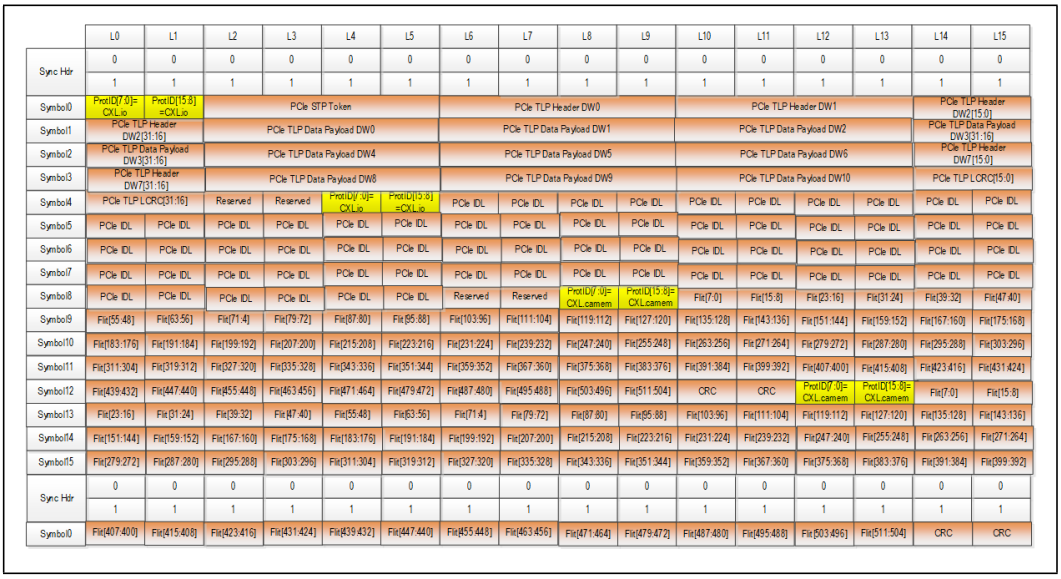
The x1 packet layout is used only in degraded mode. The 16-bits of Protocol ID followed by 528-bit flit are transferred on a single lane, taking 68 symbols to complete the transfer.

6.2.8 Special Case: CXL.io -- When a TLP Ends on a Flit Boundary

For CXL.io traffic, if a TLP ends on a flit boundary and there is no additional CXL.io traffic to send, the receiver still requires a subsequent EDB indication if it is a nullified TLP or all IDLE flit to confirm it is a good TLP before processing the TLP. Figure 93 illustrates a scenario where the first CXL.io flit encapsulates a TLP that ends at the flit boundary, and the transmitter has no more TLPs or DLLPs to send. To ensure that the

transmitted TLP that ended on the flit boundary is processed by the receiver, a subsequent CXL.io flit containing PCIe IDLE tokens is transmitted; this flit is generated by the link layer.

Figure 93. CXL.io TLP Ending on Flit Boundary Example



6.2.9 Framing Errors

The physical layer is responsible for detecting framing errors and, subsequently, initiating entry into recovery to retrain the link.

The following are framing errors detected by the physical layer:

- Sync header errors
- Protocol ID framing errors
- PCIe framing errors located within the 528-bit CXL.io flit

Protocol ID framing errors are described in Section 6.2.2 and summarized below in Table 52. A protocol ID with a value that is defined in the CXL specification is considered a valid protocol ID. A valid protocol ID is either expected or unexpected. An expected protocol ID is one that corresponds to a protocol that was enabled during negotiation. An unexpected protocol ID is one that corresponds to a protocol that was not enabled during negotiation; if only one CXL protocol is enabled, any ARB/MUX protocol IDs are treated as unexpected. A protocol ID with a value that is not defined in the CXL specification is considered an invalid protocol ID. Whenever a flit is dropped by the physical layer due to either an Unexpected Protocol ID Framing Error or an Uncorrectable Protocol ID Framing Error, the physical layer enters LTSSM recovery to retrain the link and notifies the link layers to enter recovery and, if applicable, to initiate link level retry.

Table 52. Protocol ID Framing Errors

Protocol ID[7:0]	Protocol ID[15:8]	Expected Action
Invalid	Valid & Expected	Process normally using Protocol ID[15:8]; Log as CXL_Correctable_Protocol_ID_Framing_Error in DVSEC Flex Bus Port Status register.
Valid & Expected	Invalid	Process normally using Protocol ID[7:0]; Log as CXL_Correctable_Protocol_ID_Framing_Error in DVSEC Flex Bus Port Status register.
Valid & Unexpected	Valid & Unexpected & Equal to Protocol ID[7:0]	Drop flit and log as CXL_Unexpected_Protocol_ID_Dropped in DVSEC Flex Bus Port Status register; enter LTSSM recovery to retrain the link; notify link layers to enter recovery and, if applicable, initiate link level retry
Invalid	Valid & Unexpected	Drop flit and log as CXL_Unexpected_Protocol_ID_Dropped in DVSEC Flex Bus Port Status register; enter LTSSM recovery to retrain the link; notify link layers to enter recovery and, if applicable, initiate link level retry
Valid & Unexpected	Invalid	Drop flit and log as CXL_Unexpected_Protocol_ID_Dropped in DVSEC Flex Bus Port Status register; enter LTSSM recovery to retrain the link; notify link layers to enter recovery and, if applicable, initiate link level retry
Valid	Valid & Not Equal to Protocol ID[7:0]	Drop flit and log as CXL_Uncorrectable_Protocol_ID_Framing_Error in DVSEC Flex Bus Port Status register; enter LTSSM recovery to retrain the link; notify link layers to enter recovery and, if applicable, initiate link level retry
Invalid	Invalid	Drop flit and log as CXL_Uncorrectable_Protocol_ID_Framing_Error in DVSEC Flex Bus Port Status register; enter LTSSM recovery to retrain the link; notify link layers to enter recovery and, if applicable, initiate link level retry

6.3 Link Training

6.3.1 PCIe vs Flex Bus.CXL mode selection

After reset, an Flex Bus link begins training and completes link width negotiation and speed negotiation according to the PCIe LTSSM rules. During link training, the CPU initiates Flex Bus mode negotiation via the PCIe alternate mode negotiation mechanism. Flex Bus mode negotiation is completed before entering L0 at 2.5 GT/s. If sync header bypass is negotiated, sync headers are bypassed as soon as the link has

transitioned to a speed of 8GT/s or higher. The Flex Bus logical PHY transmits NULL flits as soon as it transitions to 8GT/s or higher link speeds if CXL mode was negotiated earlier in the training process. These NULL flits are used in place of PCIe Idle Symbols to facilitate certain LTSSM transitions to L0 as described in Section 6.4. After the link has transitioned to its final speed, it can start sending CXL traffic on behalf of the upper layers after the SDS Ordered Set is transmitted if that was what was negotiated earlier in the training process. For upstream facing ports, the physical layer notifies the upper layers that the link is up and available for transmission only after it has received a flit that was not generated by the physical layer of the partner downstream port (refer to Table 51). To operate in CXL mode, the link speed must be at least 8 GT/s. If the link is unable to transition to a speed of 8 GT/s or greater after committing to CXL mode during link training at 2.5 GT/s, the link may ultimately fail to link up even if the device is PCIe capable.

6.3.1.1 Hardware Autonomous Mode Negotiation

Dynamic hardware negotiation of Flex Bus mode occurs during link training in Configuration before entering L0 at Gen1 speeds using the alternate protocol negotiation mechanism, facilitated by exchanging modified TS1 and TS2 Ordered Sets. The host initiates the negotiation process by sending TS1 Ordered Sets advertising its Flex Bus capabilities. The device responds with a proposal based on its own capabilities and those advertised by the host. The host communicates the final decision of which capabilities to enable by sending modified TS2 Ordered Sets before or during Configuration.Complete.

Please refer to the PCIe 5.0 base specification for details on how the various fields of the modified TS1/TS2 OS are set. Table 53 shows how the modified TS1/TS2 OS is used for Flex Bus mode negotiation. The “Flex Bus Mode Negotiation Usage” column describes the deltas from the PCIe base specification definition that are applicable for Flex Bus mode negotiation. Additional explanation is provided in Table 55. The presence of retimer1 and retimer2 must be programmed into the Flex Bus DVSEC by software before the negotiation begins; if retimers are present the relevant retimer bits in the modified TS1/TS2 OS are used.

Table 53. Modified TS1/TS2 Ordered Set for Flex Bus Mode Negotiation (Sheet 1 of 2)

Symbol Number	PCIe Description	Flex Bus Mode Negotiation Usage
0 thru 4	See PCIe 5.0 Base Spec Symbol	
5	Training Control Bits 0:6: See PCIe 5.0 Base Bit 7: Modified TS1/TS2 supported (see PCIe 5.0 Base Spec for details	Bit 7:6 = 11b
6	For Modified TS1: TS1 Identifier, encoded as D10.2 (4Ah) For Modified TS2: TS2 Identifier, encoded as D5.2 (45h)	TS1 Identifier during Phase 1 of Flex Bus mode negotiation TS2 Identifier during Phase 2 of Flex Bus mode negotiation
7	For Modified TS1: TS1 Identifier, encoded as D10.2 (4Ah) For Modified TS2: TS2 Identifier, encoded as D5.2 (45h)	TS1 Identifier during Phase 1 of Flex Bus mode negotiation TS2 Identifier during Phase 2 of Flex Bus mode negotiation
8-9	Bits 0:2: Usage (see PCIe 5.0 Base Spec) Bits 3:4: Alternate Protocol Negotiation Status if Usage is 010b, Reserved Otherwise (see PCIe 5.0 Base Spec for details) Bits 5:15: Alternate Protocol Details	Bits 2:0 = 010b (indicating alternate protocols) Bits 4:3 = Alternate Protocol Negotiation Status per PCIe spec Bit 7:5 = Alternate Protocol ID (3’d0 = ‘Flex Bus’) Bit 8: Common Clock Bits 15:8: Reserved

Table 53. Modified TS1/TS2 Ordered Set for Flex Bus Mode Negotiation (Sheet 2 of 2)

Symbol Number	PCIe Description	Flex Bus Mode Negotiation Usage
10-11	Alternate Protocol ID/Vendor ID if Usage = 010b See PCIe 5.0 Base Spec for other descriptions applicable to other Usage values	8086h ***Note:*** This may change to include the PCI SIG assigned Vendor ID for CXL.
12-14	See PCIe 5.0 Base Spec If Usage = 010b, Specific proprietary usage	Bits 7:0 = Flex Bus Mode Selection, where Bit 0: PCIe capable/enable Bit 1: CXL.io capable/enable Bit 2: CXL.mem capable/enable Bit 3: CXL.cache capable/enable Bit 7:4: Reserved Bits 23:8 = Flex Bus Additional Info, where Bit 8: Reserved Bit 9: Reserved Bit 10: Sync Header Bypass capable/enable Bit 11: Reserved Bit 12: Retimer1 CXL aware[1] Bit 13: Reserved Bit 14: Retimer2 CXL aware[2] Bits 23:15: Reserved
15	See PCIe 5.0 Base Spec	

Notes:

1. Retimer1 is equivalent to Retimer X or Retimer Z in the PCI Express Specification
2. Retimer2 is equivalent to Retimer Y in the PCI Express Specification

Table 54. Additional Information on Symbols 8-9 of Modifed TS1/TS2 Ordered Set

Bit Field in Symbols 8-9	Description
Alternate Protocol ID[2:0]	This is set to 3'd0 to indicate Flex Bus
Common Clock	The CPU uses this bit to communicate to retimers that there is a common reference clock. Depending on implementation, retimers may use this information to determine what features to enable.

Table 55. Additional Information on Symbols 12-14 of Modified TS1/TS2 Ordered Sets (Sheet 1 of 2)

Bit Field in Symbols 12-14	Description
PCIe capable/enable	The CPU and endpoint advertise their capability in Phase 1. The CPU communicates the results of the negotiation in Phase 2.[1]
CXL.io capable/enable	The CPU and endpoint advertise their capability in Phase 1. The CPU communicates the results of the negotiation in Phase 2.
CXL.mem capable/enable	The CPU and endpoint advertise their capability in Phase 1. The CPU communicates the results of the negotiation in Phase 2.
CXL.cache capable/enable	The CPU and endpoint advertise their capability in Phase 1. The CPU communicates the results of the negotiation in Phase 2.

Table 55. Additional Information on Symbols 12-14 of Modified TS1/TS2 Ordered Sets (Sheet 2 of 2)

Bit Field in Symbols 12-14	Description
Sync Header Bypass capable/ enable	The CPU, endpoint, and any retimers advertise their capability in Phase 1. The CPU communicates the results of the negotiation in Phase 2. Note: The Retimer must pass this bit unmodified from its Upstream Pseudo Port to its Downstream Pseudo Port. The retimer clears this bit if it does not support this feature when passing from its Downstream Pseudo Port to its Upstream Pseudo Port but it must never set it (only the endpoint can set this bit in that direction). If the Retimer(s) do not advertise that they are CXL aware, the CPU assumes this feature is not supported by the Retimer(s) regardless of how this bit is set.
Retimer1 CXL aware	Retimer1 advertises whether it is CXL aware in Phase 1. If it is CXL aware, it must use the "Sync Header Bypass capable/enable" bit.[2]
Retimer2 CXL aware	Retimer2 advertises whether it is CXL aware in Phase 1. If it is CXL aware, it must use the "Sync Header Bypass capable/enable" bit.[3]

Notes:
1. PCIe mode and CXL mode are mutually exclusive; when the CPU communicates the results of the negotiation in Phase 2.
2. Retimer1 is equivalent to Retimer X or Retimer Z in the PCI Express Specification
3. Retimer2 is equivalent to Retimer Y in the PCI Express Specification

Hardware autonomous mode negotiation is a two phase process that occurs while in Configuration.Lanenum.Wait, Configuration.Lanenum.Accept, and Configuration.Complete before entering L0 at Gen1 speed:

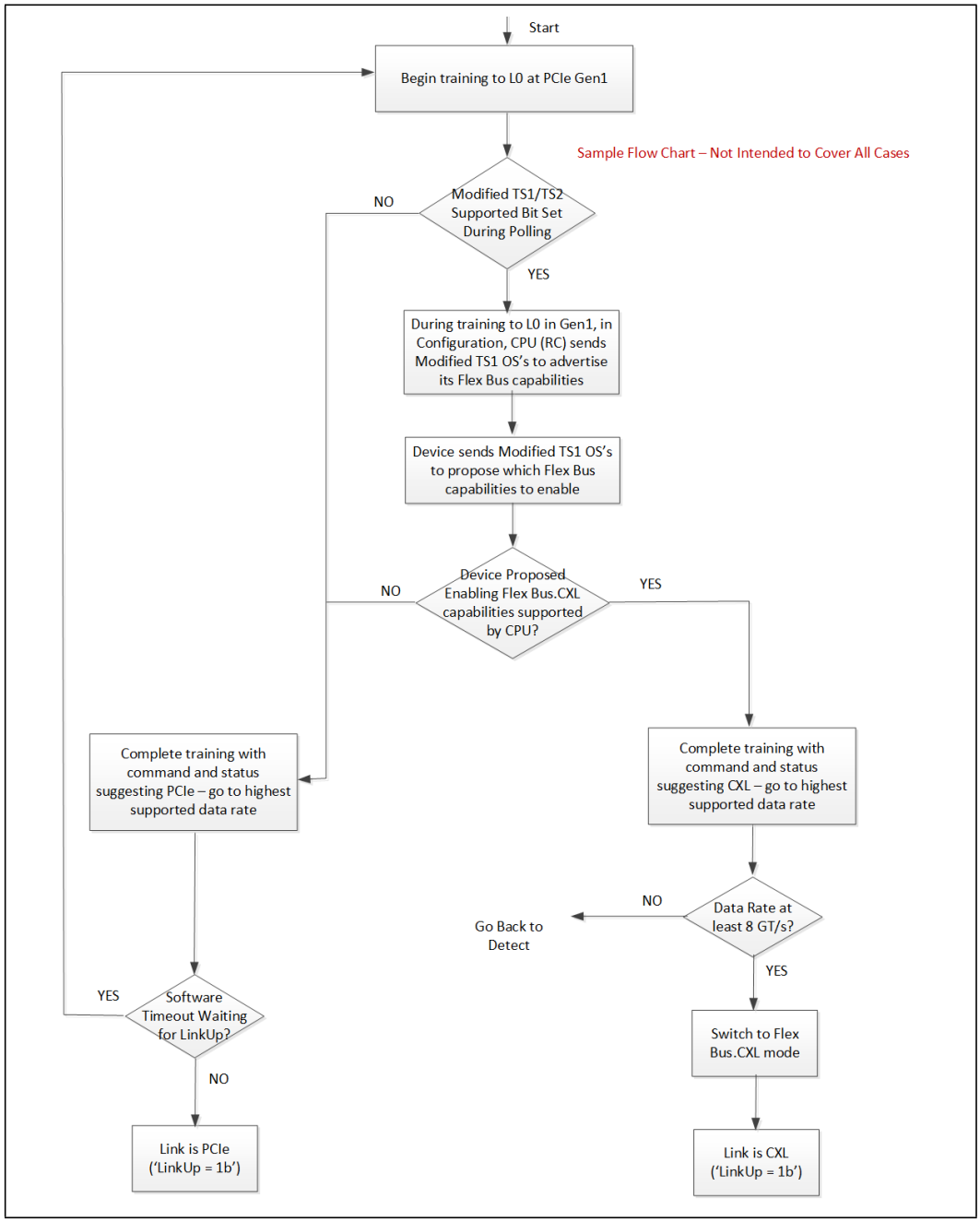
- Phase 1: The root complex sends a stream of modified TS1 Ordered Sets advertising its Flex Bus capabilities; the endpoint device responds by sending a stream of modified TS1 Ordered Sets indicating which Flex Bus capabilities it wishes to enable. This exchange occurs during Configuration.Lanenum.Wait and/or Configuration.Lanenum.Accept. At the end of this phase, the root complex has enough information to make a final selection of which capabilities to enable.
- Phase 2: The root complex sends a stream of modified TS2 Ordered Sets to the endpoint device to indicate whether the link should operate in PCIe mode or in CXL mode; for CXL mode, it also specifies which CXL protocols to enable. The endpoint acknowledges the enable request by sending modified TS2 Ordered Sets with the same Flex Bus enable bits set. This exchange occurs during Configuration.Complete.

The Flex Bus negotiation process is complete before entering L0 at 2.5GT/s. At this point the upper layers may be notified of the decision. If CXL mode is negotiated, the physical layer enables all the negotiated modes and features only after reaching L0 at 8GT/s or higher speed.

Note: If CXL is negotiated but the link does not achieve a speed of at least 8GT/s, the link with fail to link up.

A flow chart describing the mode negotiation process during link training is provided in the figure below. Note, while this flow chart represents the flow for several scenarios, it is not intended to cover all possible scenarios.

Figure 94. Flex Bus Mode Negotiation During Link Training (Sample Flow)



6.3.1.2 Flex Bus.CXL Negotiation with Maximum Supported Link Speed of 8GT/s or 16GT/s

If a Flex Bus physical layer implementation supports Flex Bus.CXL operation only at a maximum speed of 8GT/s or 16GT/s, it must still advertise support of 32GT/s speed during link training at 2.5GT/s to perform alternate protocol negotiation using modified

TS1 and TS2 Ordered Sets. Once the alternate protocol negotiation is complete, the Flex Bus logical PHY can then advertise the true maximum link speed that it supports as per the PCIe Specification.

6.3.1.3 Link Width Degradation and Speed Downgrade

If the link is operating in Flex Bus.CXL and degrades to a lower speed or lower link width that is still compatible with Flex Bus.CXL mode, the link should remain in Flex Bus.CXL mode after exiting recovery without having to go through the process of mode negotiation again. If the link drops to a speed or width not compatible with Flex Bus.CXL, it must go through the Detect state and come up in PCIe mode, if supported.

6.4 Recovery.Idle and Config.Idle Transitions to L0

The PCI Express Specification requires transmission and receipt of a specific number of consecutive Idle data Symbols on configured lanes to transition from Recovery.Idle to L0 or Config.Idle to L0 (see sections 4.2.6.4.5 and 4.2.6.3.6 of the PCI Express Specification, revision 5.0). When the Flex Bus logical PHY is in CXL mode, it looks for NULL flits instead of Idle Symbols to initiate the transition to L0. When in CXL mode and either Recovery.Idle or Config.Idle, the next state is L0 if four consecutive NULL flits are received and eight NULL flits are sent after receiving one NULL flit; all other rules called out in the PCI Express Specification regarding these transitions apply.

6.5 L1 Abort Scenario

Since the CXL ARB/MUX virtualizes the link state seen by the link layers and only requests the physical layer to transition to L1 when the link layers are in agreement, there may be a race condition that results in an L1 abort scenario. In this case, the physical layer may receive an EIOS or detect Electrical Idle when the ARB/MUX is no longer requesting entry to L1. In this scenario, the physical layer is required to initiate recovery on the link to bring it back to L0.

6.6 Exit from Recovery

Upon exit from recovery, the receiver assumes that any partial TLPs that were transmitted prior to recovery entry are terminated and must be retransmitted in full via a link level retry. Partial TLPs include TLPs for which a subsequent EDB, Idle, or valid framing token were not received before entering recovery. The transmitter must satisfy any requirements to enable the receiver to make this assumption.

6.7 Retimers and Low Latency Mode

The Flex Bus specification supports the following features that can be enabled to optimize latency: bypass of sync hdr insertion and use of a drift buffer instead of an elastic buffer. Enablement of sync hdr bypass is negotiated during the Flex Bus mode negotiation process described in Section 6.3.1.1. The CPU, endpoint, and any retimers advertise their sync hdr bypass capability during Phase 1; and the CPU communicates the final decision on whether to enable sync header bypass during Phase 2. Drift buffer mode is decided locally by each component. The rules for enabling each feature are summarized in Table 56; these rules are expected to be enforced by hardware.

Table 56. Rules of Enable Low Latency Mode Features

Feature	Conditions For Enabling	Notes
Sync Hdr Bypass	1) All components support 2) Common reference clock 3) No retimer present or retimer cannot add or delete SKPS (e.g., in low latency bypass mode) 4) Not in loopback mode	
Drift Buffer (instead of elastic buffer)	1) Common reference clock	Each component can enable this independently (i.e., does not have to be coordinated). The physical logs in the Flex Bus DVSEC when this is enabled.

6.7.1 Control SKP Ordered Set Frequency and L1/Recovery Entry

In Flex Bus.CXL mode, if sync header bypass is enabled, the following rules apply:

- After the SDS, the physical layer must schedule a control SKP Ordered Set or SKP Ordered Set after every 340 data blocks, unless it is exiting the data stream. Note: The control SKP OSs are alternated with regular SKP OSs
- When exiting the data stream, the physical layer must replace the scheduled control SKP OS (or SKP OS) with either an EIOS (for L1 entry) or EIEOS (for all other cases including recovery).

When the sync hdr bypass optimization is enabled, retimers rely on the above mechanism to know when L1/recovery entry is occurring. When sync hdr bypass is not enabled, retimers must not rely on the above mechanism.

§ §

7.0 Control and Status Registers

The Compute Express Link device control and status registers are mapped into separate spaces: configuration space and memory mapped space. Configuration space registers are accessed using configuration reads and configuration writes. Memory mapped registers are accessed using memory reads and memory writes. Table 57 has a list of the attributes for the register bits defined in this chapter.

Table 57. Register Attributes

Attribute	Description
RO	Read Only
RO-V	Read-Only-Variant
RW	Read-Write
RWS	Read-Write-Sticky
RWO	Read-Write-Once
RWL	Read-Write-Lockable
RW1CS	Read-Write-One-To-Clear-Sticky

7.1 Configuration Space Registers

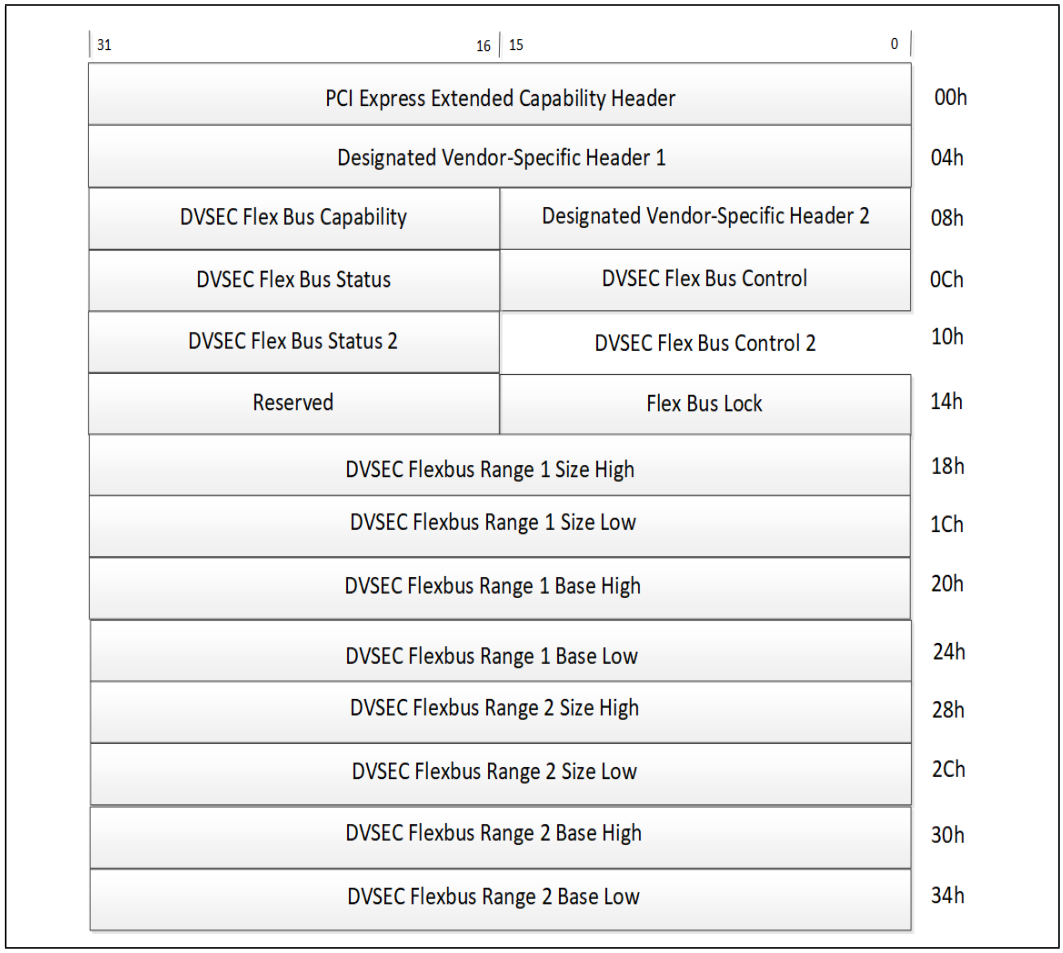
CXL configuration space registers are implemented only by the RCiEP(s) in the downstream device. The CXL upstream and downstream ports do not map any registers into configuration space.

7.1.1 PCI Express Designated Vendor-Specific Extended Capability (DVSEC) for CXL Device

CXL device creates a new PCIe enumeration hierarchy. As such, it spawns a new Root Bus and can expose one or more PCIe device numbers and function numbers at this bus number. These are exposed as Root Complex Integrated Endpoints (RCiEP). The PCIe configuration space of Device 0, Function 0 shall include the CXL PCI Express Designated Vendor-Specific Extended Capability (DVSEC) as shown in the figure below. The capability, status and control fields in Device 0, Function 0 DVSEC control the CXL functionality of the entire CXL device.

Please refer to the PCIe Specification for a description of the standard DVSEC register fields.

Figure 95. PCIe DVSEC for Flex Bus Device



To advertise Flex Bus capability, the standard DVSEC register fields should be set to the values shown in the table below. The DVSEC Length field is set to 16 bytes to accommodate the Flex Bus registers included in the DVSEC. The DVSEC ID is set to 0x0 to advertise that this is an Flex Bus feature capability structure.

Table 58. PCI Express DVSEC Register Settings for Flex Bus Device

Register	Bit Location	Field	Value
Designated Vendor-Specific Header 1 (offset 04h)	15:0	DVSEC Vendor ID	0x8086[1]
Designated Vendor-Specific Header 1 (offset 04h)	19:16	DVSEC Revision	0x0
Designated Vendor-Specific Header 1 (offset 04h)	31:20	DVSEC Length	0x38
Designated Vendor-Specific Header 2 (offset 08h)	15:0	DVSEC ID	0x0

1. Note: This may change to include CXL assigned Vendor ID.

The Flex Bus device specific registers are described in the following subsections.

7.1.1.1 DVSEC Flex Bus Capability (Offset 0Ah)

Bit	Attributes	Description
0	RO	Cache_Capable: If set, indicates CXL.cache protocol support when operating in Flex Bus.AL mode.
1	RO	IO_Capable: If set, indicates CXL.io protocol support when operating in Flex Bus.AL mode. Must be 1.
2	RO	Mem_Capable: If set, indicates CXL.mem protocol support when operating in Flex Bus.AL mode.
3	RO	Mem_HwInit_Mode: If set, indicates this CXL.mem capable device initializes memory with assistance from hardware and firmware located on the device. If clear, indicates memory is initialized by host software such as device driver. This bit should be ignored if CXL.mem Capable=0.
5:4	RO	HDM_Count: Number of HDM ranges implemented by the CXL device and reported through this function. 00 - Zero ranges. This setting is illegal if CXL.mem Capable=1. 01 - One HDM range. 10 - Two HDM ranges 11 - Reserved This field must return 00 if CXL.mem Capable=0.
13:6	N/A	Reserved (RSVD).
14	RO	Viral_Capable: If set, indicates CXL device supports Viral handling.
15	N/A	Reserved (RSVD).

7.1.1.2 DVSEC Flex Bus Control (Offset 0Ch)

Bit	Attributes	Description
0	RWL	Cache_Enable: When set, enables CXL.cache protocol operation when in Flex Bus.AL mode. Locked by CONFIG_LOCK.
1	RO	IO_Enable: When set, enables CXL.io protocol operation when in Flex Bus.AL mode.
2	RWL	Mem_Enable: When set, enables CXL.mem protocol operation when in Flex Bus.AL mode. Locked by CONFIG_LOCK.
7:3	RWL	Cache_SF_Coverage: Performance hint to the device. Locked by CONFIG_LOCK. 0x00: Indicates no Snoop Filter coverage on the Host For all other values of N: Indicates Snoop Filter coverage on the Host of 2^(N+15d) Bytes. For example, if this field contains the value 5, it indicates snoop filter coverage of 1 MB.
10:8	RWL	Cache_SF_Granularity: Performance hint to the device. Locked by CONFIG_LOCK. 000: Indicates 64B granular tracking on the Host 001: Indicates 128B granular tracking on the Host 010: Indicates 256B granular tracking on the Host 011: Indicates 512B granular tracking on the Host 100: Indicates 1KB granular tracking on the Host 101: Indicates 2KB granular tracking on the Host 110: Indicates 4KB granular tracking on the Host 111: Reserved (RSVD)
11	RWL	Cache_Clean_Eviction: Performance hint to the device. Locked by CONFIG_LOCK. 0: Indicates clean evictions from device caches are needed for best performance 1: Indicates clean evictions from device caches are NOT needed for best performance
13:12	N/A	Reserved (RSVD).
14	RWL	Viral_Enable: When set, enables Viral handling in the CXL device. Locked by CONFIG_LOCK.
15	N/A	Reserved (RSVD).

7.1.1.3 DVSEC Flex Bus Status (Offset 0Eh)

Bit	Attributes	Description
13:0	N/A	Reserved (RSVD).
14	RWS	Viral_Status: When set, indicates that the CXL device has entered Viral self-isolation mode. See Section 11.4, "CXL Viral Handling" on page 198 for more details.
15	N/A	Reserved (RSVD).

7.1.1.4 DVSEC Flex Bus Control2 (Offset 10h)

Bit	Attributes	Description
15:0	N/A	Reserved (RSVD).

7.1.1.5 DVSEC Flex Bus Status2 (Offset 12h)

Bit	Attributes	Description
15:0	N/A	Reserved (RSVD).

7.1.1.6 DVSEC Flex Bus Lock (Offset 14h)

Bit	Attributes	Description
0	RWO	CONFIG_LOCK: When set, control register, Memory Base Low and Memory Base High registers become read only.
15:1	N/A	Reserved (RSVD).

7.1.1.7 DVSEC Flex Bus Range registers

DVSEC Flex Bus Range 1 register set must be implemented if CXL.mem Capable=1. DVSEC Flex Bus Range 2 register set must be implemented if (CXL.mem Capable=1 and HDM_Count=10) . Each set contains 4 registers - Size High, Size Low, Base High, Base Low.

7.1.1.7.1 DVSEC Flex Bus Range 1 Size High (Offset 18h)

Bit	Attributes	Description
31:0	RO	Memory_Size_High: Corresponds to bits 63:32 of Flex Bus Range 1 memory size.

7.1.1.7.2 DVSEC Flex Bus Range1 Size Low (Offset 1Ch)

Bit	Attributes	Description
0	RO	Memory_Info_Valid: When set, indicates that the Flex Bus Range 1 Size high and Size Low registers are valid. Must be set within 1 second of deassertion of reset to CXL device.
1	RO	Memory_Active: When set, indicates that the Flex Bus Range 1 memory is fully initialized and available for software use. Must be set within 1 second of deassertion of reset to CXL device if CXL.mem HwInit Mode=1.
4:2	RO	Media_Type: Indicates the memory media characteristics 000 - Volatile memory 001 - Non-volatile memory Other encodings are reserved.
7:5	RO	Memory_Class: Indicates the class of memory 000 - Memory Class (e.g., normal DRAM) 001 - Storage Class (e.g., Intel 3D XPoint)) All other encodings are reserved.
10:8	RO	Desired_Interleave: If a CXL.mem capable device is connected to a single CPU via multiple Flex Bus links, this field represents the memory interleaving desired by the device. BIOS will configure the CPU to interleave accesses to this HDM range across links at this granularity. 00 - No Interleave 01 - 256 Byte Granularity 10 - 4K Interleave all other settings are reserved
27:11	N/A	Reserved (RSVD).
31:28	RO	Memory_Size_Low: Corresponds to bits 31:28 of Flex Bus Range 1 memory size.

7.1.1.7.3 DVSEC Flex Bus Range 1 Base High (Offset 20h)

Bit	Attributes	Description
31:0	RWL	Memory_Base_High: Corresponds to bits 63:32 of Flex Bus Range 1 base in the host address space. Configured by system BIOS.

7.1.1.7.4 DVSEC Flex Bus Range 1 Base Low (Offset 24h)

Bit	Attributes	Description
27:0	N/A	Reserved (RSVD).
31:28	RWL	Memory_Base_Low: Corresponds to bits 31:28 of Flex Bus Range 1 base in the host address space.

A CXL.mem capable device directs host accesses to an address A its local HDM memory if the following two equations are satisfied -

Memory_Base[63:28] <= (A <<28) < Memory_ Base[63:28]+Memory_Size[63:28]

Memory_Active AND Mem_Enable=1

If the address A is not backed by real memory (e.g. a device with less than 256 MB of memory), the device must handle those accesses gracefully i.e. return all 1's on reads and drop writes.

7.1.1.7.5 DVSEC Flex Bus Range 2 Size High (Offset 28h)

Bit	Attributes	Description
31:0	RO	Memory_Size_High: Corresponds to bits 63:32 of Flex Bus Range 2 memory size.

7.1.1.7.6 DVSEC Flex Bus Range 2 Size Low (Offset 2Ch)

Bit	Attributes	Description
0	RO	Memory_Info_Valid: When set, indicates that the Flex Bus Range 2 Size high and Size Low registers are valid. Must be set within 1 second of deassertion of reset to CXL device.
1	RO	Memory_Active: When set, indicates that the Flex Bus Range 2 memory is fully initialized and available for software use. Must be set within 1 second of deassertion of reset to CXL device if CXL.mem HwInit Mode=1.
4:2	RO	Media_Type: Indicates the memory media characteristics 000 - Volatile memory 001 - Non-volatile memory 111 - Not Memory. Other encodings are reserved.
7:5	RO	Memory_Class: Indicates the class of memory 000 - Memory Class (e.g., normal DRAM) 001 - Storage Class (e.g., Intel 3D XPoint) All other encodings are reserved.
10:8	RO	Desired_Interleave: If a CXL.mem capable device is connected to a single CPU via multiple Flex Bus links, this field represents the memory interleaving desired by the device. BIOS will configure the CPU to interleave accesses to this HDM range across links at this granularity. 00 - No Interleave 01 - 256 Byte Granularity 10 - 4K Interleave all other settings are reserved
27:11	N/A	Reserved (RSVD).
31:28	RO	Memory_Size_Low: Corresponds to bits 31:28 of Flex Bus Range 2 memory size.

7.1.1.7.7 DVSEC Flex Bus Range 2 Base High (Offset 30h)

Bit	Attributes	Description
31:0	RWL	Memory_Base_High: Corresponds to bits 63:32 of Flex Bus Range 2 base in the host address space. Configured by system BIOS.

7.1.1.7.8 DVSEC Flex Bus Range 2 Base Low (Offset 34h)

Bit	Attributes	Description
27:0	N/A	Reserved (RSVD).
31:28	RWL	Memory_Base_Low: Corresponds to bits 31:28 of Flex Bus Range 2 base in the host address space.

7.2 Memory Mapped Registers

CXL memory mapped registers are located in four general regions as specified in Table 59. Notably, the CXL downstream port and CXL upstream port are not discoverable through PCIe configuration space. Instead the downstream and upstream port registers are implemented using PCIe root complex registers blocks (RCRBs).

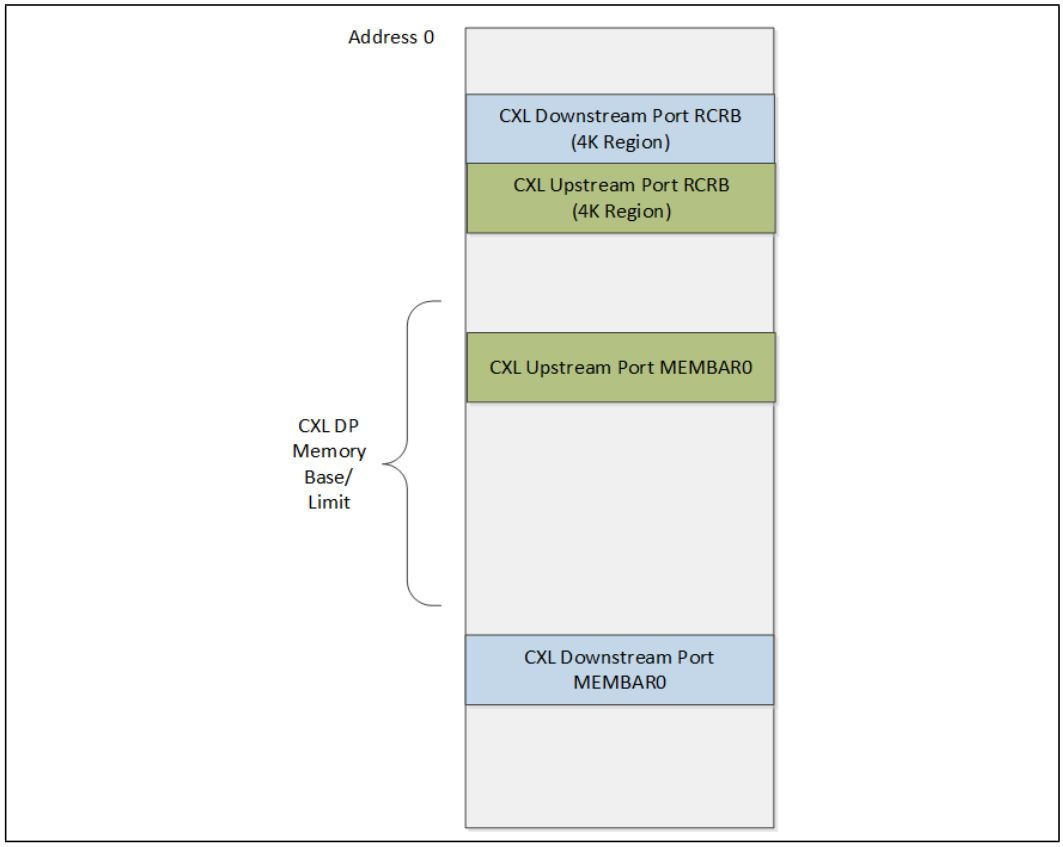
Additionally, the CXL downstream and upstream ports each implement an MEMBAR0 region to host registers for configuring the CXL subsystem components associated with the respective port.

The four memory mapped register regions appear in memory space as shown in Figure 96. Note that the RCRBs do not overlap with the MEMBAR0 regions. Also, note that the upstream port's MEMBAR0 region must fall within the range specified by the downstream port's memory base and limit register. So long as these requirements are satisfied, the details of how the RCRBs are mapped into memory space are implementation specific.

Table 59. CXL Memory Mapped Registers Regions

Memory Mapped Region	Description	Location
CXL Downstream Port RCRB	This is a 4K region with registers based upon PCIe defined registers for a root port with deltas listed in this chapter. Includes registers from PCIe Type 1 Config Header and PCIe capabilities and extended capabilities.	This is a contiguous 4K memory region relocatable via an implementation specific mechanism. This region is located outside of the downstream port's MEMBAR0 region. Note: The combined CXL Downstream and Upstream Port RCRBs are a contiguous 8K region.
CXL Upstream Port RCRB	This is a 4K region with registers based upon PCIe defined registers for an upstream port with deltas listed in this chapter. Includes 64B Config Header and PCIe capabilities and extended capabilities.	This is a contiguous 4K memory region relocatable via an implementation specific mechanism. This region is located outside of the upstream port's MEMBAR0 region. This region may be located within the range specified by the downstream port's memory base/limit registers, but that is not a requirement. Note: The combined CXL Downstream and Upstream Port RCRBs are a contiguous 8K region.
CXL Downstream Port MEMBAR0	This memory region hosts registers that allow software to configure CXL downstream port subsystem components, such as the CXL protocol, link, and physical layers and the CXL ARB/MUX.	The location of this region is specified by a 64-bit MEMBAR0 register located at offset 0x10 and 0x14 of the downstream port's RCRB.
CXL Upstream Port MEMBAR0	This memory region hosts registers that allow software to configure CXL upstream port subsystem components, such as CXL protocol, link, and physical layers and the CXL ARB/MUX.	The location of this region is specified by a 64-bit MEMBAR0 register located at offset 0x10 and 0x14 of the upstream port's RCRB. This MBAR0 region is located within the range specified by the downstream port's memory base/ limit registers.

Figure 96. CXL Memory Mapped Register Regions



7.2.1 Upstream and Downstream Port Registers

7.2.1.1 CXL Downstream Port RCRB

The downstream port RCRB is a 4K memory region that contains registers based upon the PCIe specification defined registers for a root port. Figure 97 illustrates the layout of the CXL RCRB for a downstream port. With the exception of the first DW, the first 64 bytes of the CXL DP RCRB implement the registers from a PCIe Type 1 Configuration Header. The first DW of the RCRB contains a NULL Extended Capability ID with a Version of 0h and a Next Capability Offset pointer. A 64-bit MEMBAR0 is implemented at offset 10h and 14h; this points to a private memory region that hosts registers for configuring downstream port subsystem components as specified in Table 59. The supported PCIe capabilities and extended capabilities are discovered by following the linked lists of pointers. Supported PCIe capabilities are mapped into the offset range from 040h to 0FFh. Supported PCIe extended capabilities are mapped into the offset range from 100h to FFFh. The CXL downstream port supported PCIe capabilities and extended capabilities are listed in Table 60; please refer to the PCIe 5.0 Base Specification for definitions of the associated registers.

Figure 97. CXL Downstream Port RCRB

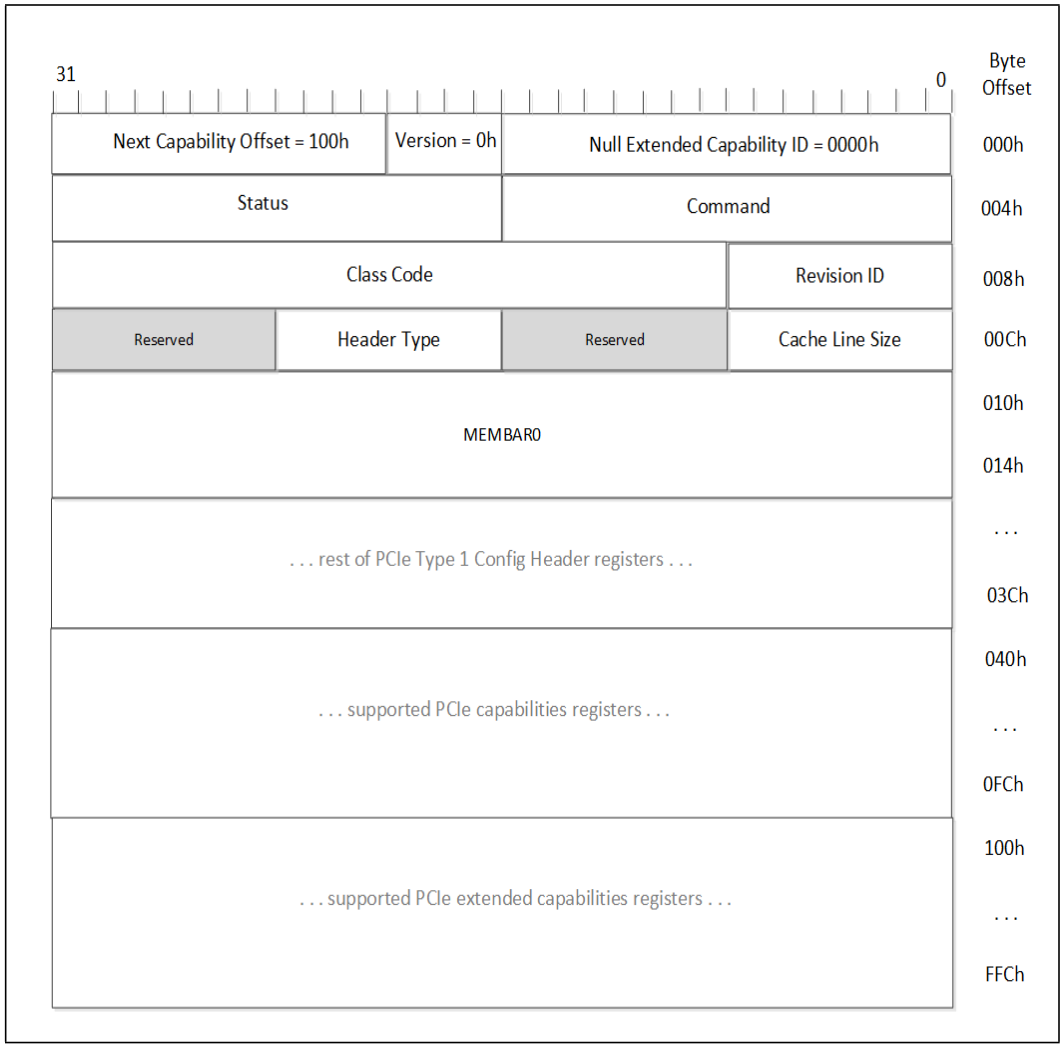


Table 60. CXL Downstream Port Supported PCIe Capabilities and Extended Capabilities (Sheet 1 of 2)

Supported PCIe Capabilities and Extended Capabilities	Exceptions[1]	Notes
PCI Express Capability	Slot Capabilities, Slot Control, Slot Status, Slot Capabilites 2, Slot Control 2, and Slot Status 2 registers are not applicable.	N/A
PCI Power Management Capability	None	N/A
MSI Capability	None	N/A
Advanced Error Reporting Extended Capability	None	Required for CXL despite being optional for PCIe.
ACS Extended Capability	None	N/A

Table 60. CXL Downstream Port Supported PCIe Capabilities and Extended Capabilities (Sheet 2 of 2)

Supported PCIe Capabilities and Extended Capabilities	Exceptions[1]	Notes
Multicast Extended Capability	None	N/A
Downstream Port Containment Extended Capability	None	N/A
Designated Vendor-Specific Extended Capability (DVSEC)	None	Please refer to section Figure 7.2.1.3 for Flex Bus Port DVSEC definition.

1. Note: It is the responsibility of software to be aware of the registers within the capabilities that are not applicable in CXL mode in case designs choose to use a common code base for PCIe and CXL mode.

7.2.1.2 CXL Upstream Port RCRB

The upstream port RCRB is a 4K memory region that contains registers based upon the PCIe specification defined registers. The upstream port captures the upper address bits [63:12] of the first memory access received after link initialization as the base address for the upstream port RCRB. Figure 98 illustrates the layout of the CXL RCRB for an upstream port. With the exception of the first DW, the first 64 bytes of the CXL UP RCRB implement the registers from a PCIe Type 0 Configuration Header. The first DW of the RCRB contains a NULL Extended Capability ID with a Version of 0h and a Next Capability Offset pointer. A 64-bit MEMBAR0 is implemented at offset 10h and 14h; this points to a memory region that hosts registers for configuring upstream port subsystem CXL.mem as specified in Table 59. The supported PCIe capabilities and extended capabilities are discovered by following the linked lists of pointers. Supported PCIe capabilities are mapped into the offset range from 040h to 0FFh. Supported PCIe extended capabilities are mapped into the offset range from 100h to FFFh. The CXL upstream port supported PCIe capabilities and extended capabilities are listed in Table 61; please refer to the PCIe 5.0 Base Specification for definitions of the associated registers.

Figure 98. CXL Upstream Port RCRB

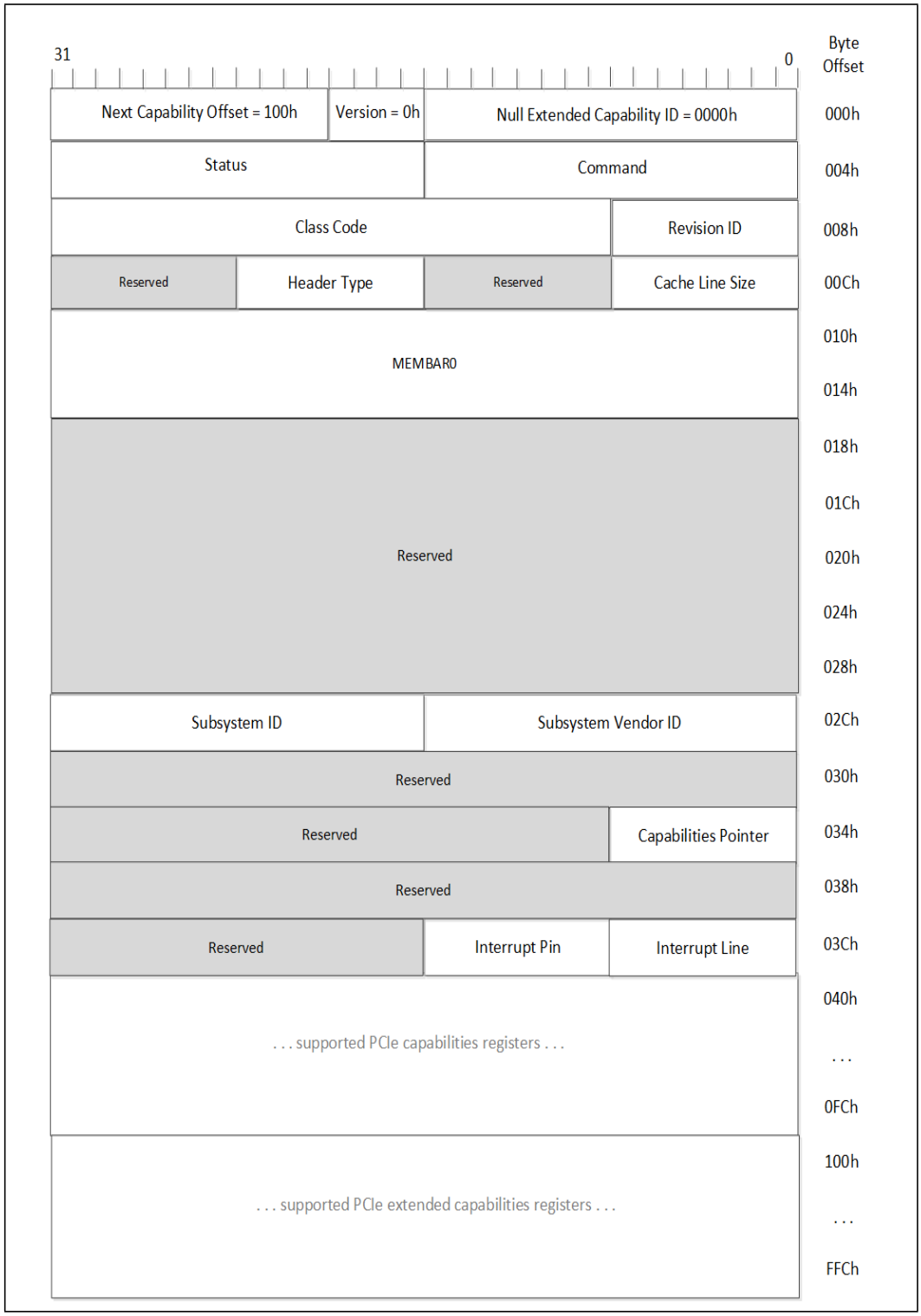


Table 61. CXL Upstream Port Supported PCIe Capabilities and Extended Capabilities

Support PCIe Capabilties and Extended Capabilties	Exceptions[1]	Notes
PCI Express Capability	None	N/A
Advanced Error Reporting Extended Capability	None	Required for CXL despite being optional for PCIe.
Multicast Extended Capability	None	N/A
Virtual Channel Extended Capability	None	VC0 and VC1
Designated Vendor-Specific Extended Capability (DVSEC)	None	Please refer to section Figure 7.2.1.3 for Flex Bus Port DVSEC definition.

1. Note: It is the responsibility of software to be aware of the registers within the capabilities that are not applicable in CXL mode in case designs choose to use a common code base for PCIe and CXL mode.

7.2.1.3 Upstream and Downstream Flex Bus Port DVSEC

The upstream and downstream Flex Bus ports implement a Flex Bus Port DVSEC, which is distinct from that implemented by a CXL device. This DVSEC is located in the RCRBs of the upstream and downstream ports. Figure 99 shows the layout of the Flex Bus Port DVSEC and Table 62 shows how the header1 and header2 registers should be set. The following subsections give details of the registers defined in the Flex Bus Port DVSEC.

Figure 99. PCIe DVSEC for Flex Bus Port

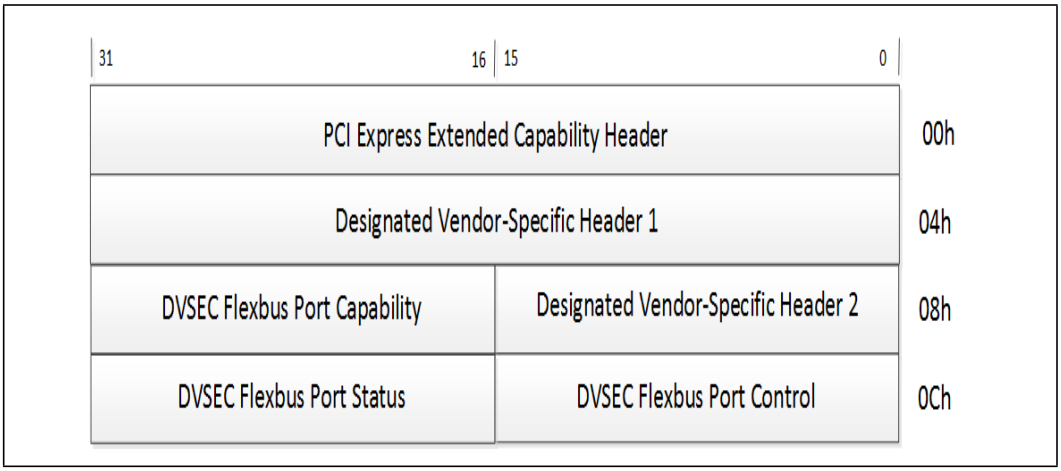


Table 62. PCI Express DVSEC Header Registers Settings for Flex Bus Port

Register	Bit Location	Field	Value
Designated Vendor-Specific Header 1 (Offset 04h)	15:0	DVSEC Vendor ID	0x8086[1]
Designated Vendor-Specific Header 1 (Offset 04h)	19:16	DVSEC Revision	0x0
Designated Vendor-Specific Header 1 (Offset 04h)	31:20	DVSEC Length	0x10
Designated Vendor-Specific Header 2 (Offset 08h)	15:0	DVSEC ID	0x7

1. Note: This may change to include CXL assigned Vendor ID

7.2.1.3.1 DVSEC Flex Bus Port Capability Offset (0Ah)

Note: The Mem_Capable, IO_Capable, and Cache_Capable fields are also present in the Flex Bus DVSEC for the device. This allows for future scalability where multiple devices, each with potentially different capabilities, may be populated behind a single port.

Bit	Attributes	Description
0	RO	Cache_Capable: If set, indicates CXL.cache protocol support when operating in Flex Bus.AL mode.
1	RO	IO_Capable: If set, indicates CXL.io protocol support when operating in Flex Bus.AL mode. Must be 1.
2	RO	Mem_Capable: If set, indicates CXL.mem protocol support when operating in Flex Bus.AL mode.
15:3	N/A	Reserved (RSVD).

7.2.1.3.2 DVSEC Flex Bus Port Control (Offset 0Ch)

Bit	Attributes	Description
0	RW	Cache_Enable: When set, enables CXL.cache protocol operation when in Flex Bus.AL mode.
1	RO	IO_Enable: When set, enables CXL.io protocol operation when in Flex Bus.AL mode. (Must always be set to 1)
2	RW	Mem_Enable: When set, enables CXL.mem protocol operation when in Flex Bus.AL mode.
3	RW	CXL_Sync_Hdr_Bypass_Enable: When set, enables bypass of the 2-bit sync header by the Flex Bus physical layer when operating in Flex Bus.AL mode. This is a performance optimization.
4	RW	Drift_Buffer_Enable: When set, enables drift buffer (instead of elastic buffer) if there is a common reference clock
7:5	N/A	Reserved (RSVD)
8	RW	Retimer1_Present: When set, indicates presence of retimer1. This bit is defined only for a downstream port. This bit is reserved for an upstream port.
9	RW	Retimer2_Present: When set, indicates presence of retimer2. This bit is defined only for a downstream port. This bit is reserved for an upstream port.
15:10	N/A	Reserved (RSVD).

7.2.1.3.3 DVSEC Flex Bus Port Status (Offset 0Eh)

Bit	Attributes	Description
0	RO-V	Cache_Enabled: When set, indicates that CXL.cache protocol operation has been enabled as a result of PCIe alternate protocol negotation for Flex Bus.
1	RO-V	IO_Enabled: When set, indicates that CXL.io protocol operation has been enabled as a result of PCIe alternate protocol negotiation for Flex Bus.
2	RO-V	Mem_Enabled: When set, indicates that CXL.mem protocol operation has been enabled as a result of PCIe alternate protocol negotiation for Flex Bus..
3	RO-V	CXL_Sync_Hdr_Bypass_Enabled: When set, indicates that bypass of the 2-bit sync header by the Flex Bus physical layer has been enabled when operating in Flex Bus.AL mode as a result of PCIe alternate protocol negotiation for Flex Bus..
4	RO-V	Drift_Buffer_Enabled: When set, indicates that the physical layer has enabled its drift bufferinstead of its elastic buffer.
7:5	N/A	Reserved (RSVD)

Bit	Attributes	Description
8	RW1CS	CXL_Correctable_Protocol_ID_Framing_Error: See Section 6.2.2 for more details.
9	RW1CS	CXL_Uncorrectable_Protocol_ID_Framing_Error: See Section 6.2.2 for more details.
10	RW1CS	CXL_Unexpected_Protocol_ID_Dropped: When set, indicates that the physical layer dropped a flit with an unexpected protocol ID that is not due to an Uncorrectable Protocol ID Framing Error. See Section 6.2.2 for more details
15:11	N/A	Reserved (RSVD).

7.2.2 CXL Upstream and Downstream Port Subsystem Component Registers

The CXL upstream and downstream port subsystem components implement registers in memory space allocated via the MEMBAR0 register. In general, these registers are expected to be implementation specific; this section defines the architected registers. Table 63 lists the relevant offset ranges from MEMBAR0 for CXL.io, CXL.cache, CXL.mem, and CXL ARB/MUX registers.

Table 63. CXL Subsystem Component Register Ranges in MEMBAR0

Range	Size	Destination
0000_0000h - 0000_0FFFh	4K	CXL.io registers
0000_1000h - 0000_1FFFh	4K	CXL.cache and CXL.mem registers
0000_2000h - 0000_DFFFh	48K	Implementation specific
0000_E000h - 0000_E3FFh	1K	CXL ARB/MUX registers
0000_E400h - 0000_FFFFh	7K	Reserved

7.2.2.1 CXL.cache and CXL.mem Registers

Within the 4KB region of memory space assigned to CXL.cache and CXL.mem, the location of architecturally specified registers will be described using an array of pointers. The array, described in Table 64, will be located starting at offset 0x0 of this 4KB region. The first element of the array will declare the version of CXL.cache and CXL.mem protocol as well as the size of the array. Each subsequent element will then host the pointers to capability specific register blocks within the 4KB region.

Table 64. CXL.cache and CXL.mem Architectural Registers

Offset	Register Name
0x0	CXL_Capability_Header
0x4	CXL_RAS_Capability_Header
0x8	CXL_Security_Capability_Header
0xC	CXL_Link_Capability_Header

7.2.2.1.1 CXL Capability Header Register (Offset 0x0)

Bit Location	Attributes	Description
15:0	RO	**CXL_Capability_ID:** This defines the nature and format of the CXL_Capability register. For the CXL_Capability_Header register, this field must be 0x1.
19:16	RO	**CXL_Capability_Version:** This defines the version number of the CXL_Capability structure present. For the first generation, this field must be 0x1.
23:20	RO	**CXL_Cache_Mem_Version:** This defines the version of the CXL Cache Mem Protocol supported. For the first generation, this field must be 0x1.
31:24	RO	**Array_Size:** This defines the number of elements present in the CXL_Capability array, not including the CXL_Capability_Header element. Each element is 1 DWORD in size and is located contiguous with previous elements.

7.2.2.1.2 CXL RAS Capability Header (Offset 0x4)

Bit Location	Attributes	Description
15:0	RO	**CXL_Capability_ID:** This defines the nature and format of the CXL_Capability register. For the CXL_RAS_Capability_Pointer register, this field should be 0x2.
19:16	RO	**CXL_Capability_Version:** This defines the version number of the CXL_Capability structure present. For the first generation, this field must be 0x1.
31:20	RO	**CXL_RAS_Capability_Pointer:** This defines the offset of the CXL_Capability relative to beginning of CXL_Capability_Header register. Details in Section 7.2.2.1.4.

7.2.2.1.3 CXL Security Capability Header (Offset 0x8)

Bit Location	Attributes	Description
15:0	RO	**CXL_Capability_ID:** This defines the nature and format of the CXL_Capability register. For the CXL_Security_Capability_Pointer register, this field should be 0x3.
19:16	RO	**CXL_Capability_Version:** This defines the version number of the CXL_Capability structure present. For the first generation, this field must be 0x1.
31:20	RO	**CXL_Security_Capability_Pointer:** This defines the offset of the CXL_Capability relative to beginning of CXL_Capability_Header register. Details in Section 7.2.2.1.13

7.2.2.1.4 CXL Link Capability Header (Offset 0xC)

Bit Location	Attributes	Description
15:0	RO	**CXL_Capability_ID:** This defines the nature and format of the CXL_Capability register. For the CXL_Link_Capability_Pointer register, this field should be 0x4.
19:16	RO	**CXL_Capability_Version:** This defines the version number of the CXL_Capability structure present. For the first generation, this field must be 0x1.
31:20	RO	**CXL_Link_Capability_Pointer:** This defines the offset of the CXL_Capability relative to beginning of CXL_Capability_Header register. Details in Section 7.2.2.1.15

7.2.2.1.5 CXL RAS Capability Structure

Offset	Register Name
0x0	Uncorrectable Error Status Register
0x4	Uncorrectable Error Mask Register
0x8	Uncorrectable Error Severity Register
0xC	Correctable Error Status Register
0x10	Correctable Error Mask Register
0x14	Error Capability and Control Register
0x54 - 0x18	Header Log Registers

7.2.2.1.6 Uncorrectable Error Status Register (Offset 0x0)

Bit Location	Attributes	Description
0	RW1CS	**Cache_Data_Parity:** Internal Data Parity error on CXL.cache. Header Log contains H2D Data Header.
1	RW1CS	**Cache_Address _Parity:** Internal Address Parity error on CXL.cache. Header Log contains H2D Data Header.
2	RW1CS	**Cache_BE_Parity:** Internal Byte Enable Parity error on CXL.cache. Header Log contains H2D Data Header.
3	RW1CS	**Cache_Data_ECC:** Internal Data ECC error on CXL.cache. Header Log contains H2D Data Header.
4	RW1CS	**Mem_Data_Parity:** Internal Data Parity error on CXL.mem. Header Log contains M2S RwD Data Header.
5	RW1CS	**Mem_Address_Parity:** Internal Address Parity error on CXL.mem. If Bit 0 of Header Log is '0, rest of Header Log contains M2S Req. If Bit 0 of Header Log is '1, rest of Header Log contains M2S RwD Data Header.
6	RW1CS	**Mem_BE_Parity**: Internal Byte Enable Parity error on CXL.mem. Header Log contains M2S RwD Data Header.
7	RW1CS	**Mem_Data_ECC:** Internal Data ECC error on CXL.mem. Header Log contains M2S RwD Data Header.

Bit Location	Attributes	Description
8	RW1CS	**REINIT_Threshold:** REINIT Threshold Hit. Header Log not applicable.
9	RW1CS	**Rsvd_Encoding_Violation:** Received unrecognized encoding. Header Log contains the entire flit received.
10	RW1CS	**Poison_Received:** Received Poison from the peer. Header Log contains the entire flit received.
11	RW1CS	**Receiver_Overflow:** First 3b of the Header Log are relevant and should be interpreted as such: 3'b000 --> D2H Req 3'b001 --> D2H Rsp 3'b010 --> D2H Data 3'b100 --> S2M NDR 3'b101 --> S2M DRS The above shows which buffer had the overflow

7.2.2.1.7 Uncorrectable Error Mask Register (Offset 0x4)

Bit Location	Attributes	Description
0	RWS	**Cache_Data_Parity_Mask**
1	RWS	**Cache_Address _Parity_Mask**
2	RWS	**Cache_BE_Parity_Mask**
3	RWS	**Cache_Data_ECC_Mask**
4	RWS	**Mem_Data_Parity_Mask**
5	RWS	**Mem_Address_Parity_Mask**
6	RWS	**Mem_BE_Parity_Mask**
7	RWS	**Mem_Data_ECC_Mask**
8	RWS	**REINIT_Threshold_Mask**
9	RWS	**Rsvd_Encoding_Violation_Mask**
10	RWS	**Poison_Received_Mask**
11	RWS	**Receiver_Overflow_Mask**

7.2.2.1.8 Uncorrectable Error Severity Register (Offset 0x8)

Bit Location	Attributes	Description
0	RWS	**Cache_Data_Parity_Severity**
1	RWS	**Cache_Address _Parity_Severity**
2	RWS	**Cache_BE_Parity_Severity**
3	RWS	**Cache_Data_ECC_Severity**
4	RWS	**Mem_Data_Parity_Severity**
5	RWS	**Mem_Address_Parity_Severity**
6	RWS	**Mem_BE_Parity_Severity**
7	RWS	**Mem_Data_ECC_Severity**
8	RWS	**REINIT_Threshold_Severity**

Bit Location	Attributes	Description
9	RWS	**Rsvd_Encoding_Violation_Severity**
10	RWS	**Poison_Received_Severity**
11	RWS	**Receiver_Overflow_Severity**

7.2.2.1.9 Correctable Error Status Register (Offset 0xC)

Bit Location	Attributes	Description
0	RW1CS	**Cache_Data_ECC:** Internal Data ECC error on CXL.cache.
1	RW1CS	**Mem_Data_ECC:** Internal Data ECC error on CXL.mem.
2	RW1CS	**CRC_Threshold:** CRC Threshold Hit
3	RW1CS	**Retry_Threshold:** Retry Threshold Hit
4	RW1CS	**Cache_Poison_Received:** Received Poison from the peer on CXL.cache.
5	RW1CS	**Mem_Poison_Received:** Received Poison from the peer on CXL.mem.
6	RW1CS	**Physical_Layer_Error:** Received error indication from Physical Layer

7.2.2.1.10 Correctable Error Mask Register (Offset 0x10)

Bit Location	Attributes	Description
0	RWS	**Cache_Data_ECC_Mask**
1	RWS	**Mem_Data_ECC_Mask**
2	RWS	**CRC_Threshold_Mask**
3	RWS	**Retry_Threshold_Mask**
4	RWS	**Cache_Poison_Received_Mask**
5	RWS	**Mem_Poison_Received_Mask**
6	RWS	**Physical_Layer_Error_Mask**

7.2.2.1.11 Error Capabilities and Control Register (Offset 0x14)

Bit Location	Attributes	Description
3:0	ROS-V	**First_Error_Pointer:** This identifies the bit position of the first error reported in the Uncorrectable Error Status register.
9	RO	**Multiple_Header_Recording_Capability:** This indicates if recording more than one error header is supported. For the first generation, his will be set to ‘0.
13	RWS	**Poison_Enabled:** This indicates if poison is supported.

7.2.2.1.12 Header Log Registers (Offset 0x54 - 0x18)

Bit Location	Attributes	Description
511:0	ROS	**Header Log:** The information logged here depends on the type of Uncorrectable Error Status bit recorded as described in Section 7.2.2.1.6

7.2.2.1.13 CXL Security Capability Structure

This capability structure only applies for CXL Downstream Port.

Offset	Register Name
0x0	CXL Security Policy Register

7.2.2.1.14 CXL Security Policy Register (Offset 0x0)

Bit Location	Attributes	Description
1:0	RW	**Device Trust Level:** '0 --> Trusted CXL Device. At this setting, a CXL Device will be able to get access on CXL.cache for both host-attached and device attached memory ranges. The Host can still protect security sensitive memory regions. '1 --> Trusted for Device Attached Memory Range Only. At this setting, a CXL Device will be able to get access on CXL.cache for device attached memory ranges only. Requests on CXL.cache for host-attached memory ranges will be aborted by the Host. '2 --> Untrusted CXL Device. At this setting, all requests on CXL.cache will be aborted by the Host. Please note that these settings only apply to requests on CXL.cache. The device can still source requests on CXL.io regardless of these settings. Protection on CXL.io will be implemented using IOMMU based page tables.

7.2.2.1.15 CXL Link Capability Structure

Offset	Register Name
0x0	CXL Link Layer Capability Register
0x8	CXL Link Control and Status Register
0x10	CXL Link Rx Credit Control Register
0x18	CXL Link Rx Credit Return Status Register
0x20	CXL Link Tx Credit Status Register
0x28	CXL Link Ack Timer Control Register
0x30	CXL Link Defeature

7.2.2.1.16 CXL Link Layer Capability Register (Offset 0x0)

Bit Location	Attributes	Description
3:0	RWS	**CXL Link Version Supported:** Version of AL the port is compliant with. For CXL 1.0, this should be '0001.
7:4	RO-V	**CXL Link Version Received:** Version of AL received from INIT.Param flit. Used for debug.
15:8	RWS	**LLR Wrap Value Supported:** LLR Wrap value supported by this entity. Used for debug.
23:16	RO-V	**LLR Wrap Value Received:** LLR Wrap value received from INIT.Param flit. Used for debug.
28:24	RO-V	**NUM_Retry_Received:** Num_Retry value reflected in the last Retry.Req message received. Used for debug.
33:29	RO-V	**NUM_Phy_Reinit_Received:** Num_Phy_Reinit value reflected in the last Retry.Req message received. Used for debug.
41:34	RO-V	**Wr_Ptr_Received:** Wr_Ptr value reflected in the last Retry.Ack message received
49:42	RO-V	**Echo_Eseq_Received:** Echo_Eseq value reflected in the last Retry.Ack message received
57:50	RO-V	**Num_Free_Buf_Received:** Num_Free_Buf value reflected in the last Retry.Ack message received

7.2.2.1.17 CXL Link Layer Control and Status Register (Offset 0x8)

Bit Location	Attributes	Description
0	RW-V	**LL_Reset:** Re-initialize without resetting values in sticky registers. Write '1 to reset link - this is a destructive reset all link layer state. When link layer reset completes, HW will clear the bit to '0. Entity triggering soft reset should ensure that link is quiesced
1	RWS	**LL_Init_Stall:** If set, link layer stalls the transmission of the LLCTRL-INIT.Param flit until this bit is cleared
2	RWS	**LL_Crd_Stall:** If set, link layer stalls credit initialization until this bit is cleared
4:3	RO-V	**INIT_State:** This field reflects the current initialization status of the Link Layer, including any stall conditions controlled by bits 2:1 '00 --> NOT_RDY_FOR_INIT (stalled or unstalled): LLCTRL-INIT.Param flit not sent '01 --> PARAM_EX: LLCTRL-INIT.Param sent and waiting to receive it '10 --> CRD_RETURN_STALL: Parameter exchanged successfully and Credit return is stalled '11 --> INIT_DONE: Link Initialization Done - LLCTRL-INIT.Param flit sent and received, and initial credit refund not stalled
12:5	RO-V	**LL_Retry_Buffer_Consumed:** Snapshot of link layer retry buffer consumed

7.2.2.1.18 CXL Link Layer Rx Credit Control Register (Offset 0x10)

Bit Location	Attributes	Description
9:0	RWS	**Cache Req Credits:** Credits to advertise for Cache Request channel at init
19:10	RWS	**Cache Rsp Credits:** Credits to advertise for Cache Response channel at init
29:20	RWS	**Cache Data Credits:** Credits to advertise for Cache Data channel at init
39:30	RWS	**Mem Req _Rsp Credits:** Credits to advertise for Mem Request or Response channel at init
49:40	RWS	**Mem Data Credits:** Credits to advertise for Mem Data channel at init

7.2.2.1.19 CXL Link Layer Rx Credit Return Status Register (Offset 0x18)

Bit Location	Attributes	Description
9:0	RO-V	**Cache Req Credits:** Running snapshot of accumulated Cache Request credits to be returned
19:10	RO-V	**Cache Rsp Credits:** Running snapshot of accumulated Cache Response credits to be returned
29:20	RO-V	**Cache Data Credits:** Running snapshot of accumulated Cache Data credits to be returned
39:30	RO-V	**Mem Req _Rsp Credits:** Running snapshot of accumulated Mem Request or Response credits to be returned
49:40	RO-V	**Mem Data Credits:** Running snapshot of accumulated Mem Data credits to be returned

7.2.2.1.20 CXL Link Layer Tx Credit Status Register (Offset 0x20)

Bit Location	Attributes	Description
9:0	RO-V	**Cache Req Credits:** Running snapshot of Cache Request credits for Tx
19:10	RO-V	**Cache Rsp Credits:** Running snapshot of Cache Response credits for Tx
29:20	RO-V	**Cache Data Credits:** Running snapshot of Cache Data credits for Tx
39:30	RO-V	**Mem Req _Rsp Credits:** Running snapshot of Mem Req or Response credits for Tx
49:40	RO-V	**Mem Data Credits:** Running snapshot of Mem Data credits for Tx

7.2.2.1.21 CXL Link Layer Ack Timer Control Register (Offset 0x28)

Bit Location	Attributes	Description
7:0	RWS	**Ack Force Threshold:** This specifies how many Flit Acks the Link Layer should accumulate before injecting a LLCRD
17:8	RWS	**Ack Flush Retimer:** This specifies how many link layer clock cycles the entity should wait in case of idle, before flushing accumulated Acks using a LLCRD

7.2.2.1.22 CXL Link Layer Defeature Register (Offset 0x30)

Bit Location	Attributes	Description
0	RWS	**MDH Disable:** Write '1 to disable MDH. Software needs to ensure it programs this value consistently on the UP & DP. After programming, a warm reset is required for the disable to take effect.

7.2.2.2 CXL ARB/MUX Registers

7.2.2.2.1 ARB/MUX Arbitration Control Register for CXL.io (Offset 0x180)

Bit	Attributes	Description
3:0	N/A	Reserved (RSVD)
7:4	RW	CXL.io Weighted Round Robin Arbitration Weight: This is the weight assigned to CXL.io in the weighted round robin arbitration between CXL protocols.
31:8	N/A	Reserved (RSVD)

7.2.2.2.2 ARB/MUX Arbitration Control Register for CXL.cache and CXL.mem (Offset 0x1C0)

Bit	Attributes	Description
3:0	N/A	Reserved (RSVD)
7:4	RW	CXL.cache and CXL.mem Weighted Round Robin Arbitration Weight: This is the weight assigned to CXL.cache and CXL.mem in the weighted round robin arbitration between CXL protocols.
31:8	N/A	Reserved (RSVD)

7.3 CXL RCRB Base Register

A register is required to communicate to software the memory address location of the CXL RCRB for each CXL port.

Bit	Attributes	Description
0	RW	CXL RCRB Enable: When set, the RCRB region is enabled.
12:1	N/A	Reserved (RSVD).
63:13	RW	CXL_RCRB_Base_Address: This points to an 8K memory region where the lower 4K hosts the downstream port RCRB and the upper 4K hosts the upstream port RCRB.

§ §

8.0 Reset, Initialization, Configuration and Manageability

8.1 Compute Express Link Boot and Reset Overview

8.1.1 General

Boot and Power-up sequencing of Flexbus devices follows the conventions of PCIE-CEM and as such, will not be discussed in detail in this section. However, this section will highlight the differences that exist between CXL and native PCIe for these operations.

Reset and Sx-entry flows require coordinated coherency domain shutdown before the sequence can progress. Therefore, the CXL flow will adhere to the following rules:

- Warnings will be issued to all CXL devices before the above transitions are initiated, including CXL.io.
- To extend the available messages, CXL PM messages will be used to communicate between the host and the device. Devices must respond to these messages with the proper acknowledge, even if no actions are actually performed on the said device.

8.1.2 Comparing CXL and PCIe behavior

The following table summarizes the difference in event sequencing and signaling methods across Reset and Sx flows, for discrete CXL.io/Cache/Cache+Mem and PCIe.

The terms used in the table are as follows

- Warning: an early notification of the upcoming event. Devices with coherent cache or memory are required to complete outstanding transactions, flush internal caches as needed, and place system memory in a Self_refresh state as required. Devices are required to complete all internal actions and then respond with a proper Ack to the processor
- Signaling: Actual initiation of the state transition, using either wires and/or link-layer messaging
-

Table 65. Event Sequencing for Reset and Sx Flows

Case	PCIe	CXL
Cold Reset Entry	Warning: None; Signaling: LTSSM Hot-Reset followed by PERST#	Warning: PM2IP (CXL PM VDM)[2]; Signaling: LTSSM Hot-Reset, followed by PERST#
Warm Reset Entry	Warning: None; Signaling: LTSSM Hot-Reset	Warning: PM2IP (CXL PM VDM)[2]; Signaling: LTSSM Hot-Reset
Surprise Reset Entry	Warning: None; Signaling: LTSSM detect-entry	Warning: None; Signaling: LTSSM detect-entry
Sx Entry	Warning: PME-Turn_off/Ack; Signaling: PERST# (Power will go down)	Warning: PM2IP (CXL PM VDM)[2]; PME-Turn_off/Ack; Signaling: PERST# (Power will go down)

Notes:
1. All CXL profiles support CXL PM VDMs and use end-end (PM - PM controller) sequences where possible
2. CXL PM VDM with different encodings for different events. If CXL.io devices do not respond to the CXL PM VDM, the host will still end up in the correct state due to timeouts
3. Flex Bus Physical Layer link states across cold reset, warm reset, surprise reset, and Sx entry match PCIe Physical Layer link states.

8.2 Compute Express Link Device Boot Flow

CXL devices will follow with PCIe CEM spec defined boot flows.

8.3 Compute Express Link Device Warm Reset Entry Flow

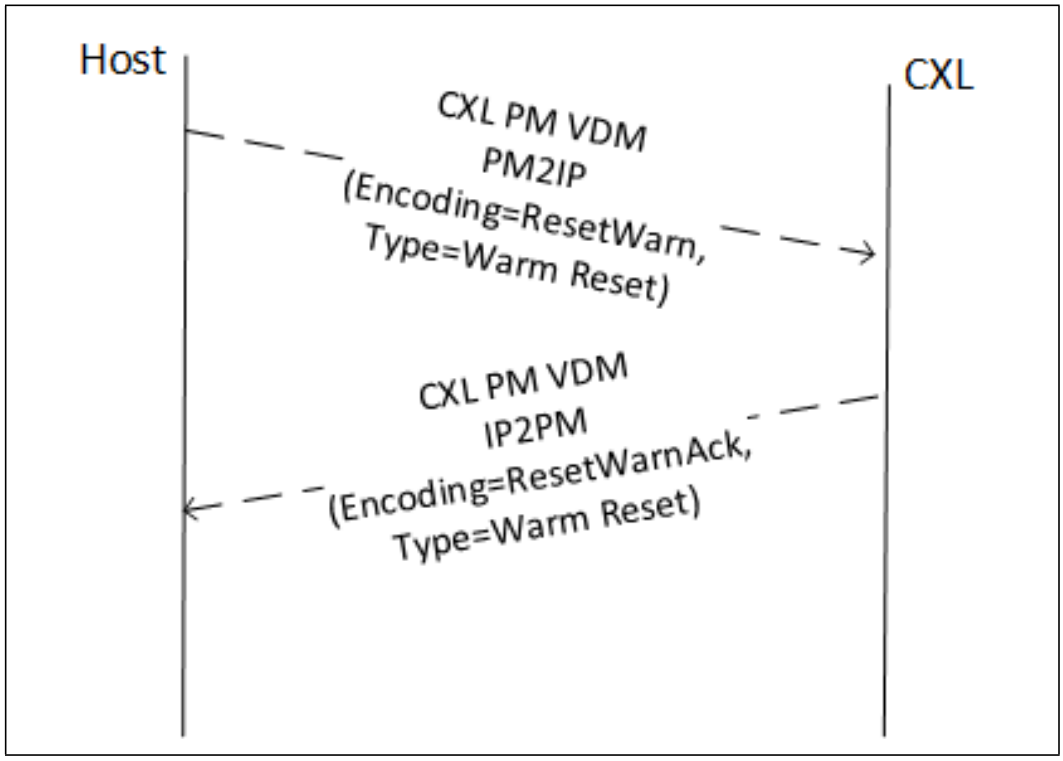
Note: In an OS orchestrated warm reset flow, it is expected that the CXL devices are already in D3 state with their contexts flushed to the system memory before the platform warm reset flow is triggered.

Note: In a platform triggered warm reset flow (due to unexpected CATERR etc.), a CXL.io device can be in a D3 or D0.

Host issues a CXL PM VDM defined as ResetPrep (ResetType = Warm Reset; PrepType = General Prep) to the CXL device as specified in Table 3. CXL device flushes any relevant context to the host (if any), cleans up the data serving the host and puts any CXL device connected memory into self-refresh. CXL device takes any additional steps for the CXL host to enter LTSSM Hot-Reset. After all the Warm reset preparation is completed, the CXL device will issue a CXL PM VDM defined as ResetPrepAck (ResetType = Warm Reset; PrepType = General Prep) to the CXL device as specified in Table 3.

Note: CXL device may or may not have PERST# asserted after warm reset handshake. If PERST# is asserted, CXL device should clear any sticky content internal to the device.

Figure 100. CXL Device Warm Reset Entry Flow



8.4 Compute Express Link Device Cold Reset Entry Flow

Note: In an OS orchestrated cold reset flow, it is expected that the CXL devices are already in D3 state with their contexts flushed to the system memory before the platform warm reset flow is triggered.

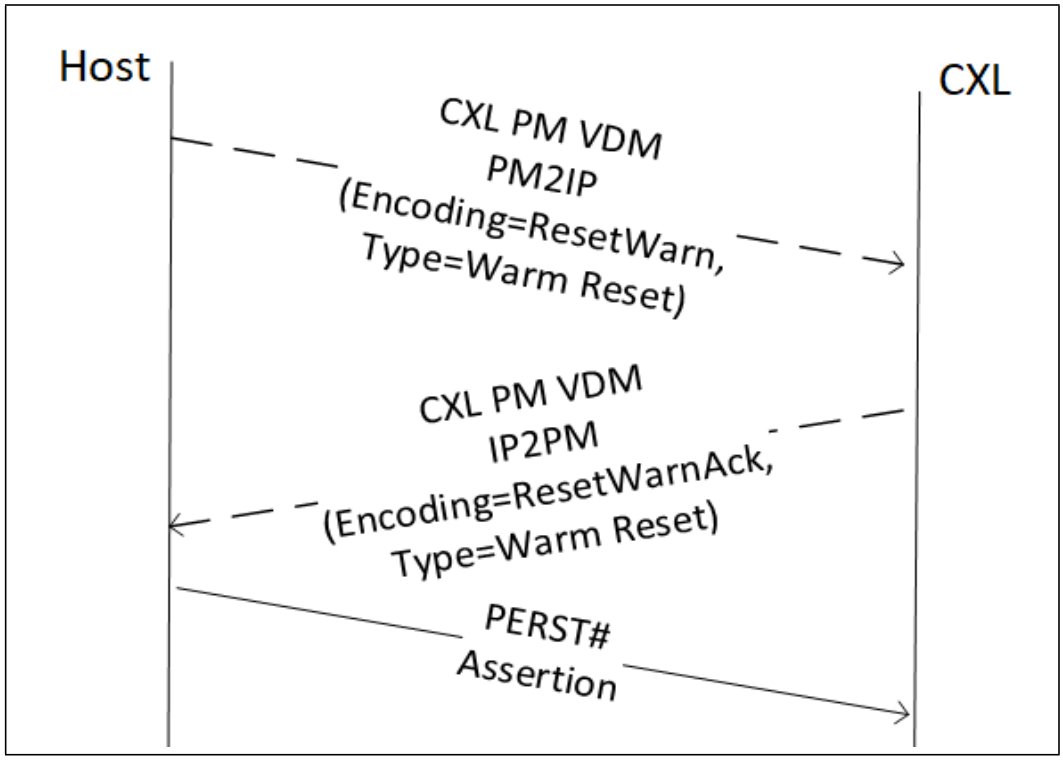
Note: In a platform triggered cold reset flow (due to unexpected CATERR etc.), a CXL device can be in a D3 or D0.

Note: Host cannot differentiate between a platform warm vs cold reset. The host issues a CXL PM VDM defined as ResetPrep (ResetType = Warm Reset; PrepType = General Prep) to the CXL device as specified in Table 3.

CXL device flushes any relevant context to the host (if any), cleans up the data serving the host and puts any CXL device connected memory into self-refresh. CXL device takes any additional steps for the CXL host to enter LTSSM hot reset. After all the Warm reset preparation is completed, CXL device will issue a CXL PM VDM defined as ResetPrepAck (ResetType = Warm Reset; PrepType = General Prep) to the CXL device as specified in Table 3.

CXL device will have PERST# asserted after warm reset handshake on a Cold Reset. On PERST# assertion, CXL device should clear any sticky content internal to the device.

Figure 101. CXL Device Cold Reset Entry Flow



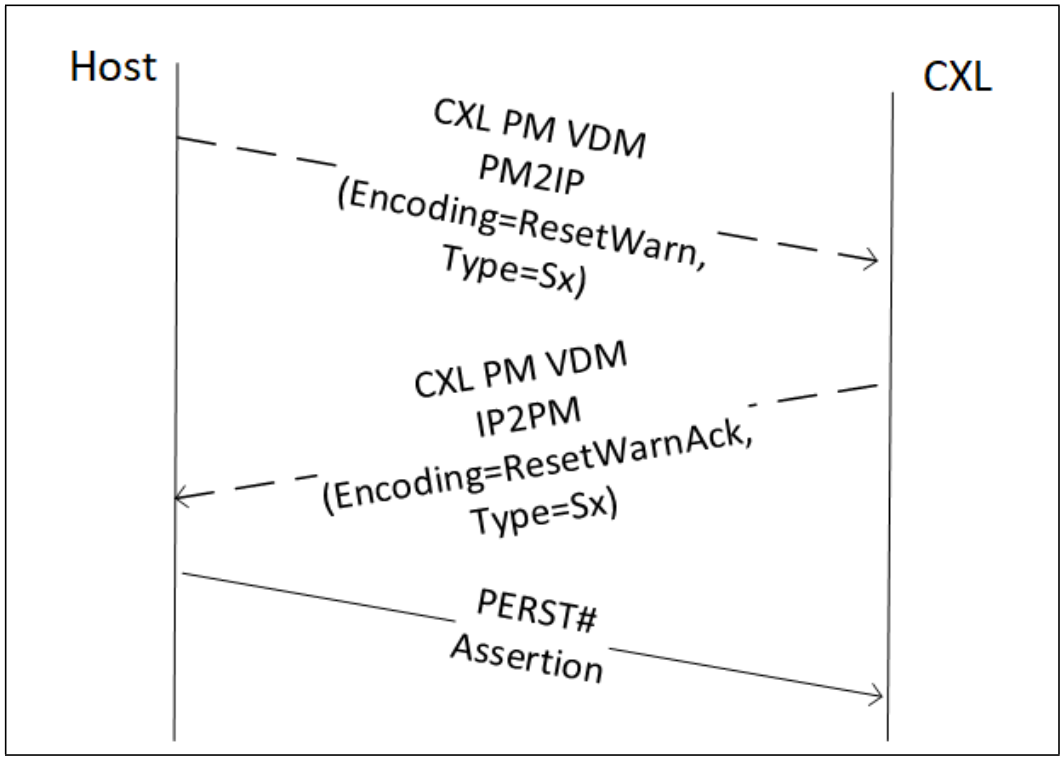
8.5 Compute Express Link Device Sleep State Entry Flow

Note: Since OS is the orchestrator of Sx flows always, it is expected that the CXL devices are already in D3 state with their contexts flushed to the CPU-attached or CXL-attached memory before the platform Sx flow is triggered.

Note: The host issues a CXL PM VDM defined as ResetPrep (ResetType = S3/S4/S5; PrepType = General Prep) to the CXL device as specified in Table 3. CXL device flushes any relevant context to the host (if any), cleans up the data serving the host and puts any CXL device connected memory into self-refresh. CXL device takes any additional steps for the CXL host to initiate a L23 flow. After all the Warm reset preparation is completed, the CXL device will issue a CXL PM VDM defined as ResetPrepAck (ResetType = S3/S4/S5; PrepType = General Prep) to the CXL device as specified in Table 3. PERST# to the CXL device can be asserted any time after this handshake is completed. On PERST# assertion, CXL device should clear any sticky content internal to the device.

Note: PERST# will always be asserted for CXL Sx Entry flows.

Figure 102. CXL Device Sleep State Entry Flow



8.6 Function Level Reset (FLR)

PCIe FLR mechanism enables software to quiesce and reset Endpoint hardware with Function-level granularity. CXL devices expose one or more PCIe functions to host software. These functions can expose FLR capability and existing PCIe compatible software can issue FLR to these functions. PCIe specification provides specific guidelines on impact of FLR on PCIe function level state and control registers. For compatibility with existing PCIe software, CXL PCIe functions should follow those guidelines if they support FLR. For example, any software readable state that potentially includes secret information associated with any preceding use of the Function must be cleared by FLR.

FLR has no effect on CXL.cache and CXL.mem protocol. Any CXL.cache and CXL.mem related control registers and state held by the CXL device is not affected by FLR. The memory controller hosting HDM is not reset by FLR. Upon FLR, certain cache lines in a CXL.cache device side cache may be flushed, but cache coherency must be maintained.

In some cases, system software uses FLR to attempt error recovery. In the context of CXL devices, errors in CXL.mem and CXL.cache logic cannot be recovered by FLR. FLR may succeed in recovering from CXL.io domain errors.

8.7 Hotplug

None of the current usage models for Flex Bus require hotplug support. Additionally, surprise hot remove is not supported.

8.8 Software Enumeration

8.8.1 Software Model

CXL device is exposed to the host software as one or more PCI express endpoints. PCIe is the most widely used device model by various OSs. In addition to leveraging the SW infrastructure and device driver writer expertise, this choice also enables us to readily use PCIe extensions like SR-IOV and PASID.

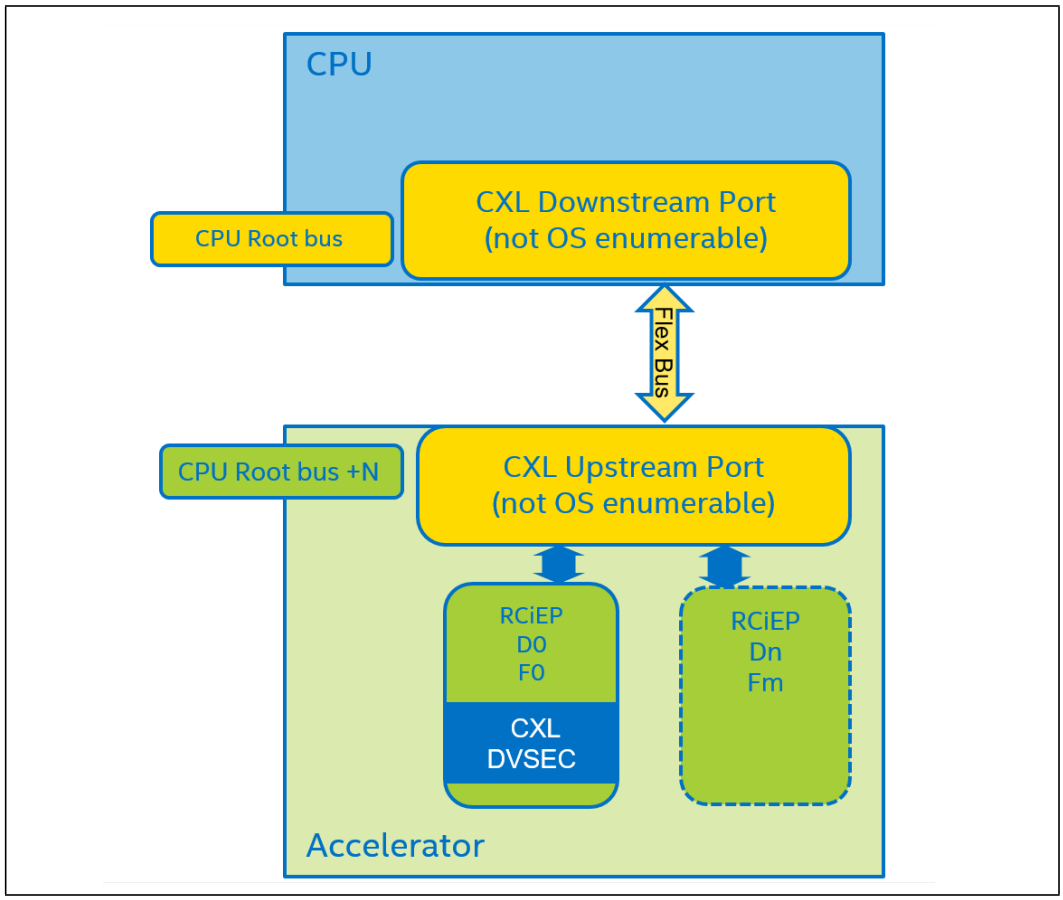
The link itself is not exposed to the Operating System. This is different from PCIe model where PCIe bus driver in OS is able to manage the PCIe link. Hiding CXL link ensures 100% compatibility with legacy PCIe software.

Since the link is not exposed to OS, each CXL device creates a new PCIe enumeration hierarchy in the form of an ACPI defined PCIe Root Bridge (PNP ID PNP0A08). CXL endpoints appear as Root Complex Integrated Endpoints (RCiEP).

CXL endpoints report “PCIe” interface errors to OS via Root Complex Event Collector (RCEC) implemented in the host. This is enabled via an extension to the RCEC (Root Complex Event Collector Bus Number Association ECN) to PCIe specification.

8.8.2 PCIe Software View of the Hierarchy

Figure 103. PCIe Software View



Since the CXL link is not exposed to OS, the BIOS view of the hierarchy is different than that of the OS.

8.8.2.1 BIOS View

The functionality of the CXL downstream port and the CXL upstream port can be accessed via memory mapped registers. These will not show up in standard PCI bus scan by existing Operating Systems. The base addresses of these registers are set up by BIOS and BIOS can use that knowledge to configure CXL.

BIOS configures the downstream port to decode the memory resource needs of the CXL device as expressed by PCIe BAR registers and upstream port BAR(s). PCIe BARs do not decode any HDM associated with the CXL device.

8.8.2.2 OS View

The CXL device instantiates one or more ACPI root bridges. The _BBN method for this root bridge matches the bus number that hosts CXL RCiEPs.

This ACPI root bridge spawns a legal PCIe hierarchy. All PCI/PCIe endpoints located in the CXL device are children of this ACPI root bridge. These endpoints may appear directly on the Root bus number or may appear behind a root port located on the Root bus.

The _CRS method for PCIe rootbridge returns bus and memory resources claimed by the CXL Endpoints. _CRS response does not include HDM on CXL.mem capable device. Nor does it comprehend any Upstream Port BARs (hidden from OS).

CXL devices cannot claim IO resources.

8.8.3 BIOS Enumeration Flow

CXL device discovery

- Parse configuration space of device 0, function 0 on the Secondary bus # and discover CXL specific attributes. These are exposed via Flex Bus DVSEC Capability structures. See Section 7.0.

If the device supports CXL.cache, configure the CPU coherent bridge. Set Cache Enable.

If the device supports CXL.mem, check Mem_HwInit_Mode.

If Mem_HwInit_Mode =1

- The device must set Memory_Info_Valid and Memory_Active within 1 second of reset deassertion.
- When Memory_Info_Valid and Memory_Active are 1, BIOS reads Memory_Size_High and Memory_Size_Low fields for each HDM range.
- BIOS computes the size of the HDM range and maps those in system address space.
- BIOS programs Memory_Base_Low and Memory_Base_High for each HDM range.
- BIOS programs ARB/MUX arbitration control registers.
- BIOS sets CXL.mem Enable. Any subsequent accesses to HDM are decoded and routed to the local memory by the device.
- Each HDM range is exposed as a separate, memory-only NUMA node via ACPI SRAT table.

- BIOS obtains latency and bandwidth information from the UEFI device driver and uses this information during construction of ACPI memory map and ACPI HMAT. The latency information reported by UEFI driver is measured from the point of ingress and must be adjusted to accommodate other hops.

If Mem_HwInit_Mode =0

- The device must set Memory_Info_Valid within 1 second of reset deassertion.
- When Memory_Info_Valid is 1, BIOS reads Memory_Size_High and Memory_Size_Low fields for each HDM range.
- BIOS computes the size of the HDM range and maps those in system address space.
- BIOS programs Memory_Base_Low and Memory_Base_High for each HDM range.
- BIOS programs ARB/MUX arbitration control registers.
- BIOS sets CXL.mem Enable. Any subsequent accesses to the HDM ranges are decoded and completed by the device. The reads shall return all 1's and the writes will be dropped.
- Each HDM range is exposed as a separate, memory-only NUMA node via ACPI SRAT table.
- If the memory is initialized prior to OS boot by UEFI device driver,
  - The UEFI driver is responsible for setting Memory_Active.
  - Once Memory_Active is set, any subsequent accesses to the HDM range are decoded and routed to the local memory by the device.
  - The UEFI device driver is responsible for reporting presence of memory to BIOS via UEFI APIs. UEFI driver reports the latency and bandwidth information associated with HDM to BIOS.
  - BIOS uses the information supplied by UEFI driver during construction of ACPI memory map and ACPI HMAT. The latency information reported by UEFI driver is measured from the point of ingress and must be adjusted to accommodate other hops.
- If the memory is initialized by an OS device driver post OS boot,
  - UEFI driver reports the latency and bandwidth information associated with each HDM range to BIOS.
  - BIOS uses the information supplied by UEFI driver during construction of ACPI memory map and ACPI HMAT. The latency information reported by UEFI driver is measured from the point of ingress and must be adjusted to accommodate other hops.
  - At OS hand-off, BIOS reports that the size of memory associated with HDM NUMA node is zero.
  - The OS device driver is responsible for setting Memory_Active after memory initialization is complete. Any subsequent accesses to the HDM memory are decoded and routed to the local memory by the device.
  - Availability of memory is signaled to the OS via capacity add flow.

CXL.io resource needs are discovered as part of PCIe enumeration. PCIe Root Complex registers including Downstream Port registers are appropriately configured to decode these resources. CXL Downstream Port and Upstream Port requires MMIO resources.

BIOS programs the memory base and limit registers in the downstream port to decode CXL Endpoint MMIO BARs, CXL Downstream Port MMIO BARs, CXL Upstream Port MMIO BARs.

If an accelerator supports CXL.mem and Mem_HwInit_Mode=0, system BIOS will unconditionally bind the accelerator to the appropriate UEFI driver by calling Start() function

8.8.4 Software View of CXL.cache

Legacy OS or legacy PCIe bus driver is not made aware of CXL.cache capability. The device driver is aware of this CXL.cache capability and manages the CXL cache. As shown in the table below, software cannot assume that lines in device cache that map to HDM will be flushed by CPU cache flush instructions. Software must use device specific mechanism to flush these lines.

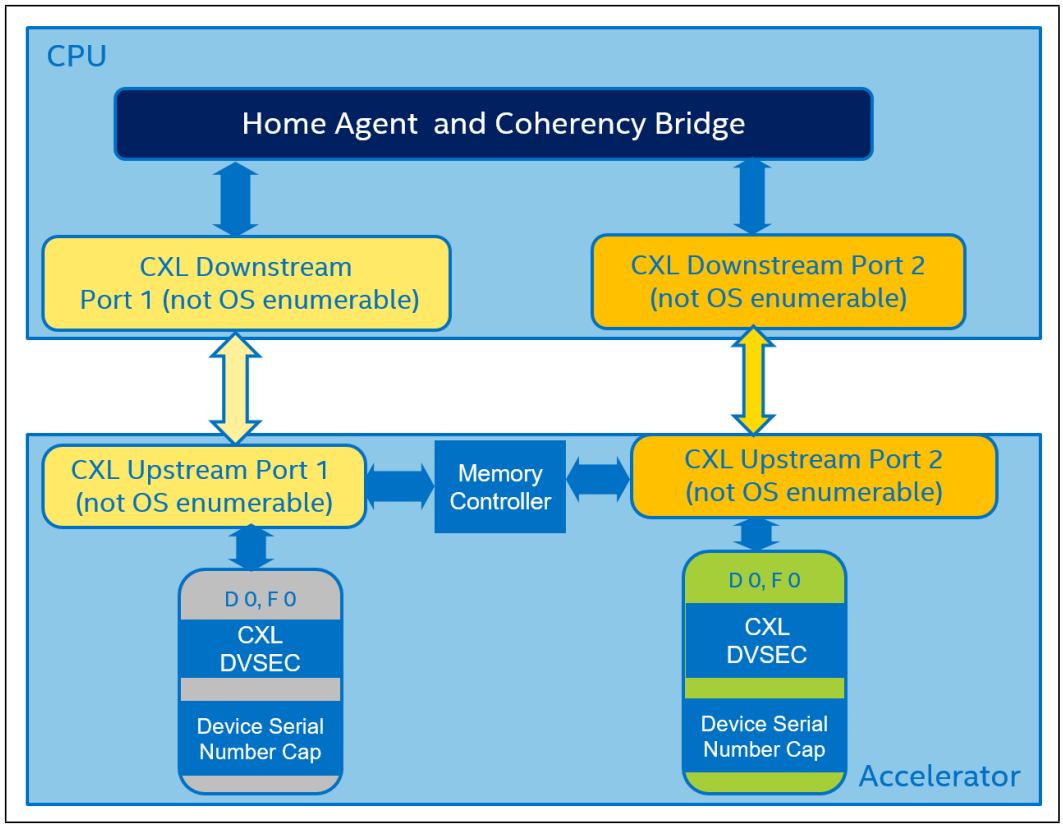
Table 66. Interaction Between CPU Cache Flush Instructions and CXL.cache

	Lines held in CPU cache	Lines held in device cache
Lines mapped to CPU attached memory	Behavior specified in Intel Software Developers Manual (SDM)	Behavior specified in Intel Software Developers Manual (SDM)
Lines mapped to device attached memory (HDM)	Behavior specified in Intel Software Developers Manual (SDM)	Implementation specific.

8.9 Accelerators with Multiple Flex Bus Links

8.9.1 Single CPU Topology

Figure 104. One CPU Connected to One Accelerator Via Two Flex Bus Links



In this configuration, BIOS shall report two PCI root bridges to the Operating system, one that hosts the left Device 0, Function 0 and the second one that hosts the Device 0, function 0 on the right. Both Device 0, function 0 instances implement Flex Bus DVSEC and a Device Serial Number PCIe Extended Capability. A vendor ID and serial number match indicates that the two links are connected to a single accelerator and this enables BIOS to perform certain optimizations.

In some cases, the accelerator may expose a single accelerator function that is managed by the accelerator device driver, whereas the other Device 0/function 0 represents a dummy device. In this configuration, application software submits the work to the single accelerator device instance. However, the accelerator hardware is free to use both links for traffic and snoops as long as the programming model is not violated.

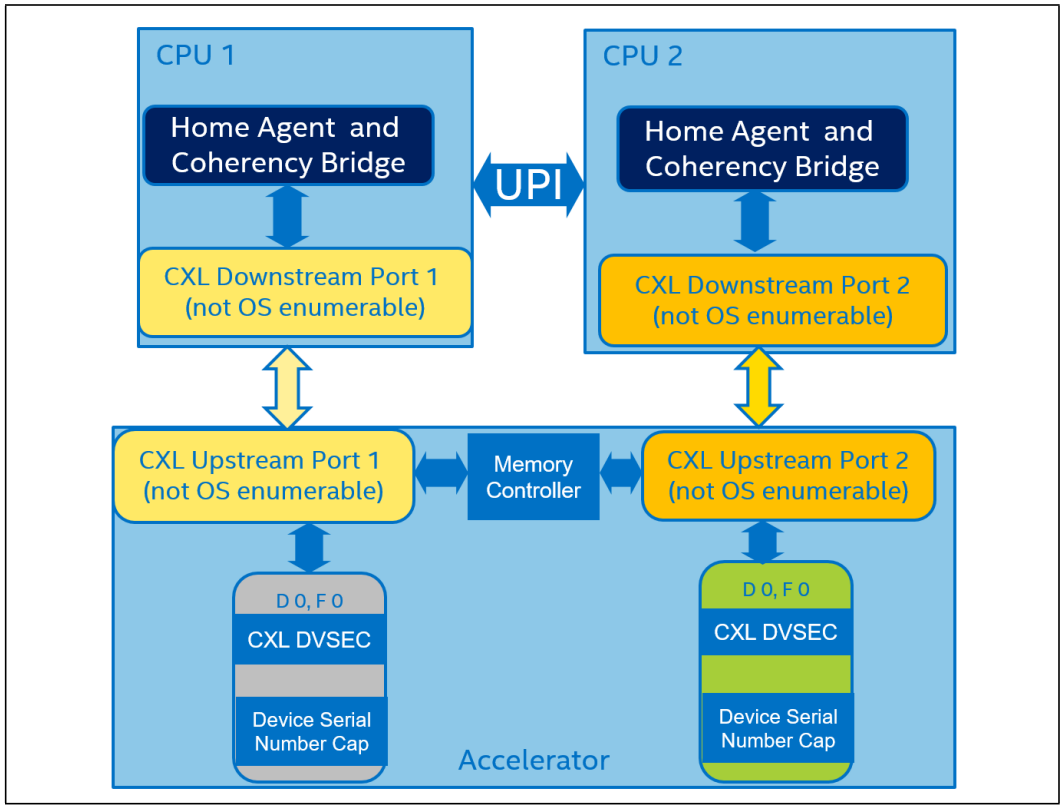
The BIOS maps the HDM into system address space using the following rules.

Table 67. Memory Decode rules in presence of one CPU/two Flex Bus links

Left D0, F0 Mem_Capable	Left D0, F0 Mem_Size	Right D0, F0 Mem_ Capable	Right D0, F0 Mem_Size	BIOS requirements
0	NA	0	NA	No HDM
1	M	0	NA	Range of size M decoded by Left Flex Bus link. Right Flex Bus link does not receive CXL.mem traffic.
0	NA	1	N	Range of size N decoded by Right Flex Bus link. Left Flex Bus link does not receive CXL.mem traffic.
1	M	1	N	Two ranges set up, Range of size M decoded by Left Flex Bus link, Range of size N decoded by right Flex Bus link
1	M	1	0	Single range of size M. CXL.mem traffic is interleaved across two links at a cache line granularity
1	0	1	N	Single range of size N. CXL.mem traffic is interleaved across two links at a cache line granularity

8.9.2 Multiple CPU Topology

Figure 105. Two CPUs Connected to One Accelerator Via Two Flex Bus Links



In this configuration, BIOS shall report two PCI root bridges to the Operating system, one that hosts the left Device 0, Function 0 and the second one that host the Device 0, function 0 on the right. Both Device 0, function 0 instances implement Flex Bus DVSEC and a Device Serial Number PCIe Extended Capability. A vendor ID and serial number match indicates that the two links are connected to a single accelerator and this enables BIOS to perform certain optimizations.

In some cases, the accelerator may choose to expose a single accelerator function that is managed by the accelerator device driver and handles all work requests. This may be necessary if the accelerator framework or applications do not support distributing work across multiple accelerator instances. Even in this case, both links should spawn a legal PCIe root bridge hierarchy with at least one PCIe function. However, the accelerator hardware is free to use both links for traffic and snoops as long as the programming model is not violated. To minimize the snoop penalty, the accelerator needs to be able to distinguish between the system memory range decoded by CPU 1 versus CPU 2. The device driver can obtain this information via ACPI SRAT table and communicate it to the accelerator using device specific mechanisms.

The BIOS maps the HDM into system address space using the following rules. Unlike the single CPU case, the BIOS shall never interleave the memory range at a cache line granularity across the two Flex Bus links.

Table 68. Memory Decode rules in presence of two CPU/two Flex Bus links

Left D0, F0 Mem_Capable	Left D0, F0 Mem_Size	Right D0, F0 Mem_ Capable	Right D0, F0 Mem_Size	BIOS requirements
0	NA	0	NA	No HDM
1	M	0	NA	Range of size M decoded by Left Flex Bus link. Right Flex Bus link does not receive CXL.mem traffic.
1	M	1	0	
0	NA	1	N	Range of size N decoded by Right Flex Bus link. Left Flex Bus link does not receive CXL.mem traffic.
1	0	1	N	
1	M	1	N	Two ranges set up, Range of size M decoded by Left Flex Bus link, Range of size N decoded by right Flex Bus link

8.10 Software View of HDM

HDM is exposed to OS/VMM as normal memory. However, HDM likely has different performance/latency attributes compared to host attached memory. Therefore, a system with CXL.mem devices can be considered as a heterogeneous memory system.

ACPI HMAT table was introduced for such systems and can report memory latency and bandwidth characteristics associated with different memory ranges. ACPI Specification version 6.2 carries the definition of revision 1 of HMAT. As of August 2018, ACPI WG has decided to deprecate revision 1 of HMAT table because it had a number of shortcomings. As a result, the subsequent discussion refers to revision 2 of HMAT table.In addition, ACPI has introduced a new type of Affinity structure called Generic Affinity (GI) Structure. GI structure is useful for describing execution engines such as accelerators that are not processors. Existing software ignores GI entries in SRAT, but newer software can take advantage of it. As a result, CXL.mem accelerators will result in two entries in SRAT - One GI entry to represent the accelerator cores and one memory entry to represent the attached HDM. GI entry is especially useful when describing CXL.cache accelerator. Previous to introduction of GI, CXL.cache accelerator could not be described as a separate entity in SRAT/HMAT and had to be combined with the attached CPU. With this specification change, CXL.cache accelerator can be described as a separate proximity domain. _PXM method can be used to associate the proximity domain associated with the PCI device . Since Legacy OSs do not understand

GI, BIOS is required to return the processor domain that is most closely associated with the IO device when running such an OS. ASL code can use bit 17 of Platform-Wide _OSC Capabilities DWORD 2 to detect whether the OS supports GI or not.

BIOS must construct and report HMAT table to OS in systems with CXL.mem devices and CXL.cache devices. Since system BIOS is not aware of HDM properties, that information must come from the UEFI driver for the CXL device in the format of HMAT Fragment Table. The format of this table is described below.

BIOS combines the information it has about the host and CXL connectivity with the HMAT Fragment Tables during construction of HMAT tables.

8.10.1 Accelerator HMAT Fragment Table Format

The fragment table is published by the accelerator UEFI driver and contains one or more memory range entries. These entries, when combined together, must cover the entire HDM range defined by CXL DVSEV base and limit registers.

Each memory range entry contains five fields:

- Memory base in system address space.
- Memory size -
- Memory Type – Represents whether the memory is available for host use or reserved (may be dedicated for accelerator local use).
- Local Memory Latency – follows HMAT convention
- Local Bandwidth – Follows HMAT convention

8.11 Manageability Model for CXL Devices Matches PCIe

Manageability is the set of capabilities that a managed entity exposes to a management entity. In the context of CXL, CXL device is the managed entity. These capabilities are generally classified in sensors and effectors. Performance counter is an example of a sensor, whereas ability to update the IA device firmware is an example of an effector. Sensors and effectors can either be accessed in-band, i.e., by OS/VMM resident software or out of band, i.e., by firmware running on a management controller that is OS independent.

In band software can access CXL device's manageability capabilities by issuing PCIe configuration read/write or MMIO read/write transactions. These accesses are generally mediated by CXL device driver. This is consistent with how PCIe cards are managed.

Out of band manageability in S0 state can leverage MCTP over PCI express infrastructure. This assumes CXL.io path will decode and forward MCTP over PCIe VDMs in both directions. Flex Bus slot definition includes two SMBUS pins (clock and data). The SMBUS path can be used for out of band manageability in Sx state or link down case. This is consistent with PCIe. The exact set of sensors and effectors exposed by the CXL card over SMBUS interface or PCIe are outside the scope of this specification. These can either be found in class specific specifications such as NVMe-MI specification.

§ §

9.0 Power Management

9.1 Statement of Requirements

All CXL implementations are required to support the Physical Layer Power management as defined in this chapter. CXL Power management is divided into protocol specific Link Power management and CXL Physical layer power management. The Arb&Mux layer is also responsible for managing protocol specific Link Power Management between the Protocols on both sides of the link. The Arb&Mux co-ordinates the Power Managed states between Multiple Protocols on both sides of the links, consolidates the Power states and drives the Physical Layer Power Management.

9.2 Policy based Runtime Control - Idle Power - Protocol Flow

9.2.1 General

For CXL connected devices, there is a desire to optimize power management of the whole system, with the device included.

As such, a hierarchical power management architecture is proposed, where the discrete device is viewed as a single autonomous entity, with thermal and power management executed locally, but in coordination with the processor socket. State transitions are coordinated with the processor die using Vendor Defined Messages over CXL. The coordination between primary power management controller and the device is best accomplished via PM2IP and IP2PM messages that are encoded as VDMs.

Since native support of PCIe is also required, support of more simplified protocols is also possible. The following table highlights the required and recommended handling method for Idle transitions.

Table 69. Runtime-Control - CXL Versus PCIe Control Methodologies

Case	PCIe	CXL[1]
Pkg-C Entry/Exit	Devices that do not share coherency with CPU can work with the PCIe profile: 1. LTR-notifications from Device; 2. Allow-L1 signaling from CPU on Pkg_C entry	Optimized handshake protocol, for all non-PCIe CXL profiles 1. LTR-notifications from Device; 2. PMreq/Rsp (VDM) signaling between CPU and device on Pkg_C entry and exit

Notes:
1. All CXL profiles support VDMs and use end-end (PM - PM controller) sequences where possible
2. PM2IP: VDM carrying messages associated with different Reset/PM flows

9.2.2 Package-Level Idle (C-state) Entry and Exit Coordination

At a high level, a discrete CXL device, that is coherent with the processor, is treated like another processor socket. Expectation is that there is coordination and agreement between the processor and discrete device before the platform can enter idle power

state. Neither device nor processor can enter a low power state individually as long as its memory resources are needed by the other die. As an example, in a case where the device may contain shared High-BW memory (HBM) on it, while the processor controls the system's DDR, if the device wants to be able to go into a low power state, it must take into account the processor's need for accessing the HBM memory. Likewise, if processor wants to go into a low power state, it must take into account, among other things, the need for the device to access DDR. These requirements are encapsulated in the LTR requirements that are provided by entities that need QOS for access to memory. In this case, we would have a notion of LTR for DDR access and LTR for HBM access. We would expect the device to inform the processor about its LTR wrt DDR, and processor to inform the device about its LTR wrt HBM

Managing latency requirements can be done in two methods.

- CXL devices that do not share coherency with CPU (either a shared coherent memory or a coherent cache), can notify the processor on changes in its latency tolerance via the PMReq() and PMRsp() messages. When appropriate latency is supported and processor execution has stopped, the processor will enter an Idle state and proceed to transition the Link to L1 (see Link-Layer section, Section 9.4, “Compute Express Link Power Management” on page 186).
- CXL devices that include a coherent cache or memory device are required to coordinate their state transitions using the CXL optimized VDM based protocol, which includes the ResetPrep(), PMReq(), PMRsp() and PMGo() messages, to prevent loss of memory coherency.

9.2.3 PkgC Entry flows

Figure 106. PkgC Entry Flows

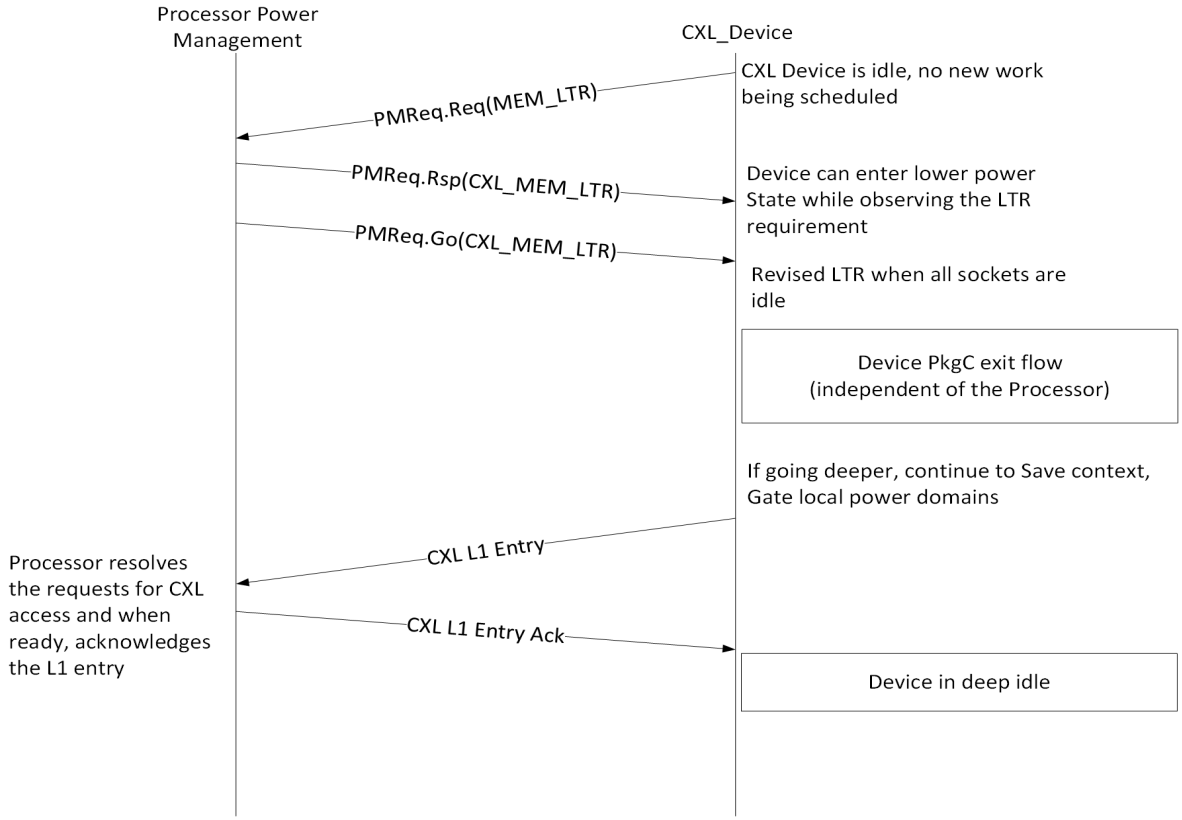


Figure 106 illustrates the PkgC entry flow. A device (or the processor) when wishing to enter a higher-latency Idle state, in which CPU is not active, will issue a PMReq() with LTR field marking the memory access tolerance of the entity.

If Idle state is allowed, the peer entity will respond with a matching PMRsp() message, with the negotiated allowable latency tolerance LTR. Both entities can independently enter an Idle state without coordination, as long as the shared resources remain accessible.

For a full package C entry, both entities need to negotiate as to the depth/latency tolerance, by responding with a PMRsp() message with the agreeable latency tolerance. Once the master power mgmt. agent has coordinated LTR across all the agents in the system, it will send a PMGo() with the proper Latency field set, indicating local idle power actions can be taken subject to the communicated latency tolerance value.

In case of a transition into deep-idle states (client systems mostly), device will initiate a CXL transition into L1.

9.2.4 PkgC Exit Flows

Figure 107. PkgC Exit Flows - Triggered by device access to system memory

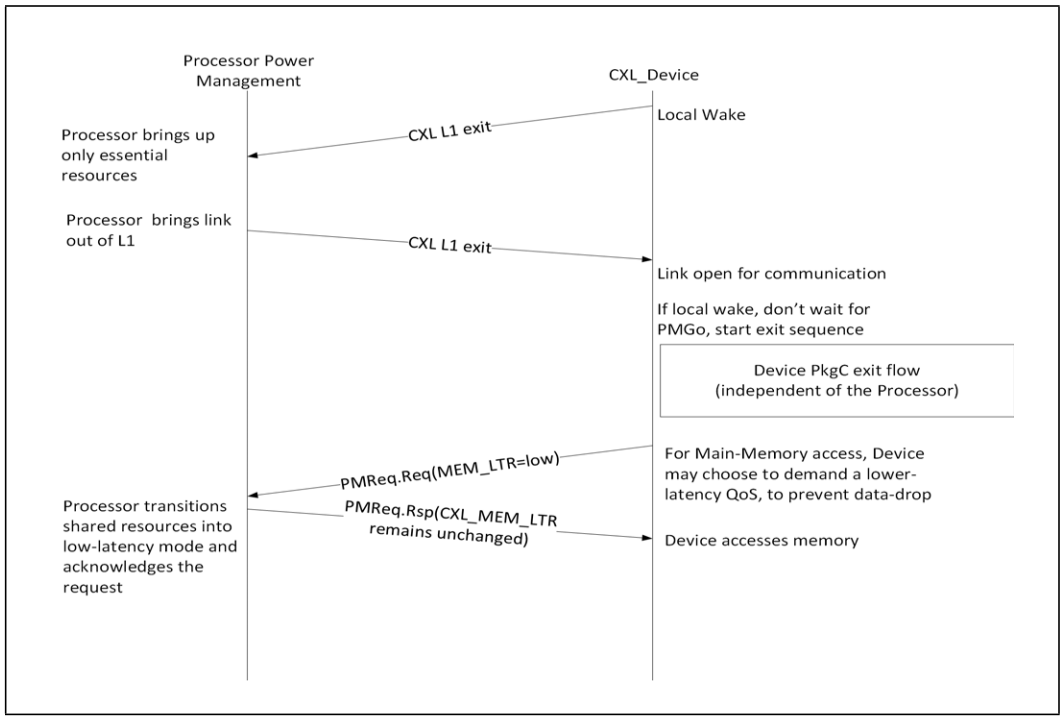


Figure 107 illustrates the PkgC exit flow initiated by the device. Link state during Idle may be in one of the select L1.x states, during Deep-Idle (as depicted here). In-band wake signaling will be used to transition the link back to L0. For more, see Section 9.4, "Compute Express Link Power Management" on page 186.

Once CXL is out of L1, signaling can be used to transfer the device into a Package-C2 state, in which shared resources are available across CXL.

Processor will bring the shared resources out of Idle and acknowledge with a PMRsp() to indicate low-latency QoS has been achieved.

Figure 108. PkgC Exit Flows - Execution Required by Processor

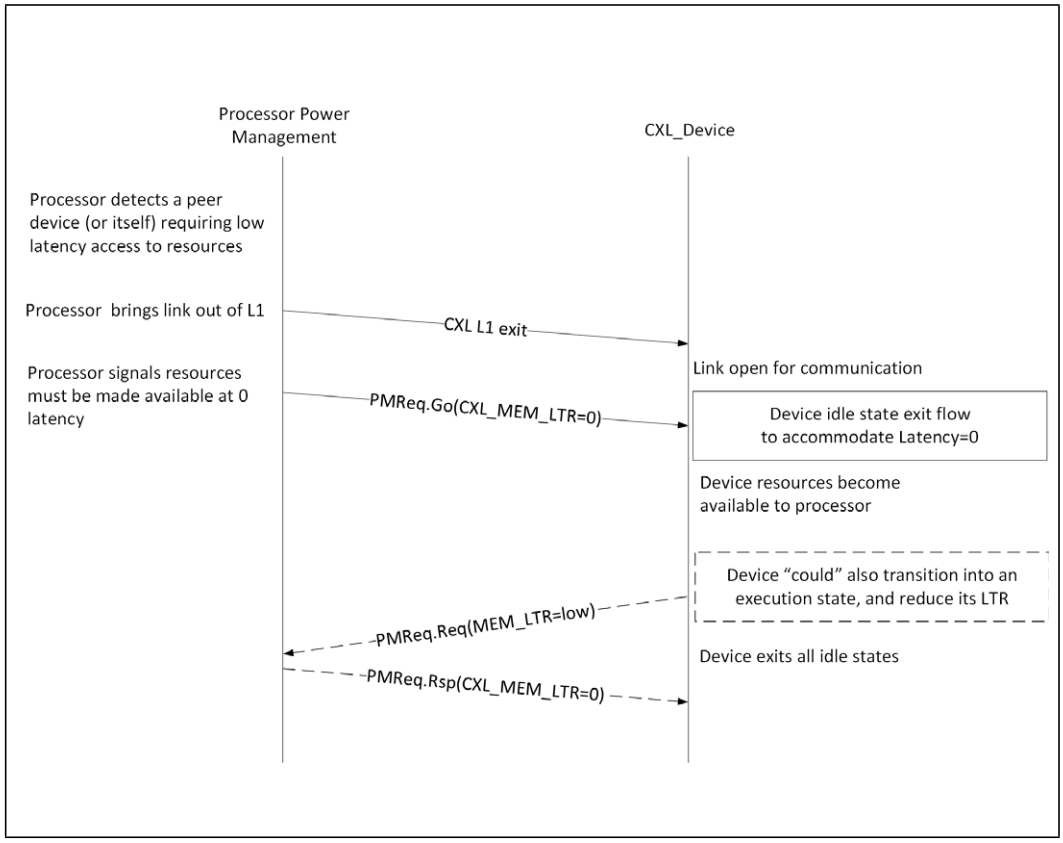


Figure 108 illustrates the PkgC exit flow initiated by the processor. In the case where the processor, or one of the peer devices connected to it requires to have coherent low latency access to system memory, the processor will initiate a Link L1 exit towards the device.

Once the link is running, the processor will follow with a PMGo(Latency=0), indicating some device in the platform requires low latency access to coherent memory and resources. A device receiving PMG0 with latency 0 must ensure that further low power actions that might impede access to memory are not taken.

9.3 Compute Express Link Physical Layer Power Management States

CXL Physical layer supports L1 and L2 states as defined in PCI Express Base Specification. CXL Physical layer does not support L0s. The entry and exit conditions from these states are as defined in the PCI Express Base Specification. The notable difference is that for CXL Physical Layer the entry and exit from Physical Layer Power Managed states is directed by CXL ARB&MUX.

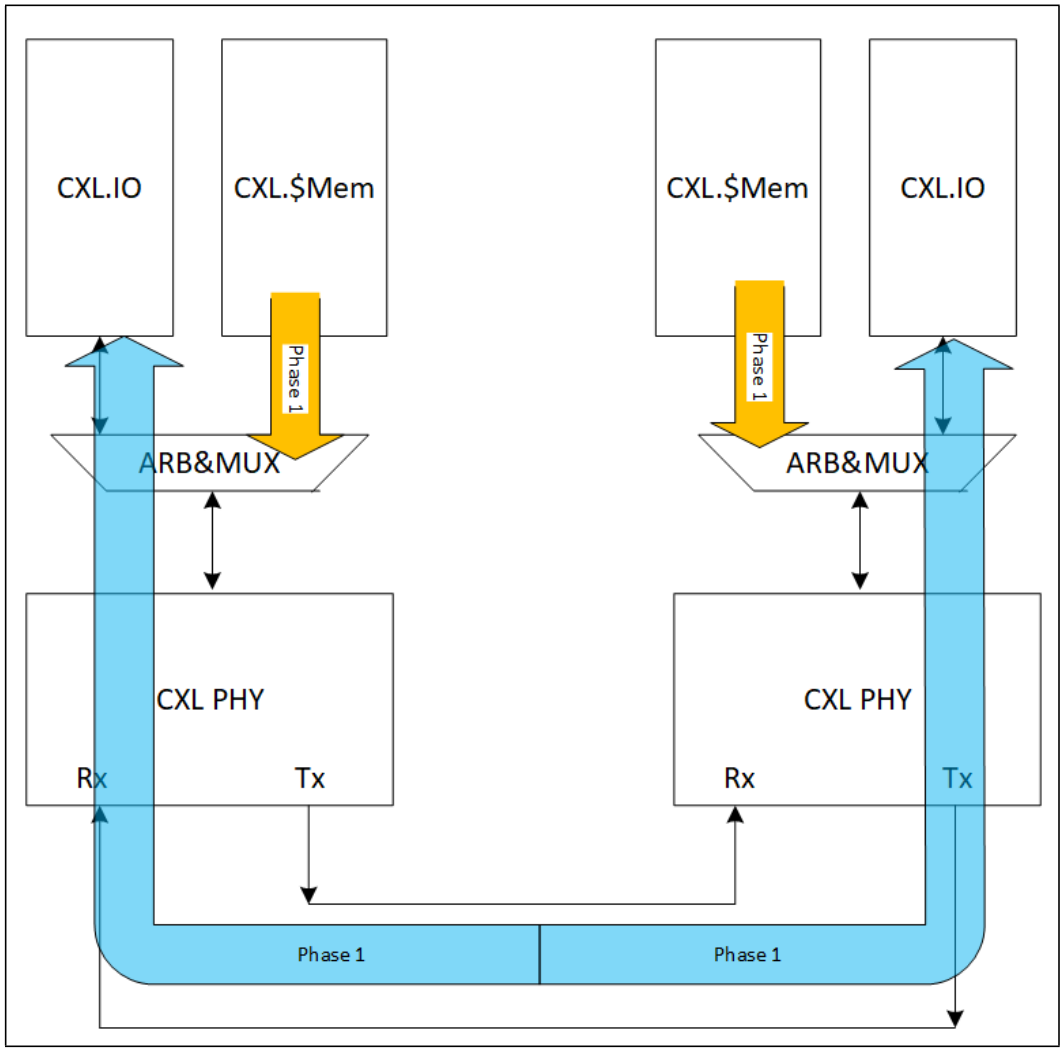
9.4 Compute Express Link Power Management

CXL Link Power Management supports Active Link State Power Management and L1 and L2 are the only 2 Power states supported. The PM Entry/Exit process is further divided into 3 phases as described below.

9.4.1 Compute Express Link PM Entry Phase 1

The CXL PM Entry phase 1 involves protocol specific mechanisms to negotiate entry into PM state. Once the conditions to enter PM state as defined in the protocol section are satisfied, transaction layer is ready for Phase 2 entry and directs the ARB&MUX to enter PM State.

Figure 109. CXL Link PM Phase 1

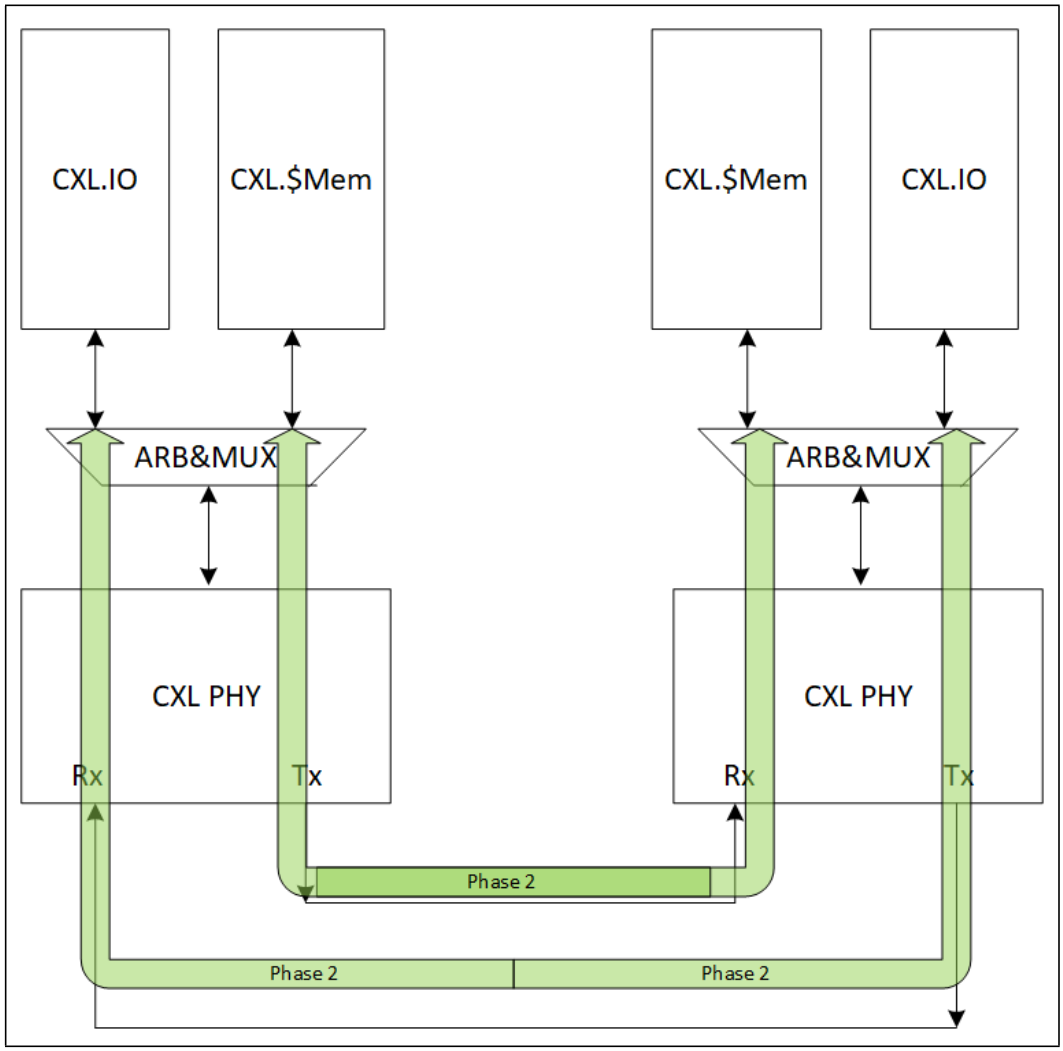


9.4.2 Compute Express Link PM Entry Phase 2

When directed by the transaction layer to enter PM, the Phase 2 entry process is initiated by ARB&MUX. The second Phase of PM entry consists of bringing the ARB&MUX interface of both sides of the Link into PM state. This entry into PM state is coordinated

using ALMPs as described below. The Phase 2 entry is independently managed for each protocol. The Physical Layer continues to be in L0 until all the transaction layers enter Phase 2 state.

Figure 110. CXL Link PM Phase 2



Rules for Phase 2 entry into ASPM are as follows:

1. The Phase 2 Entry into PM State is always initiated by ARB&MUX on the Downstream Component.
2. When directed by the transaction layer the ARB&MUX on the Downstream Component must transmit ALMP request to enter Virtual LSM state PM.
3. When the ARB&MUX on the Upstream Component is directed to enter L1 and receives ALMP request from the Downstream Component, the Upstream Component responds with an ALMP response indicating acceptance of entry into L1 state. The transaction layer on the Upstream Component must also be notified that the ARB&MUX port has accepted entry into PM state.
4. The Upstream Component ARB&MUX port does not respond with an ALMP response if not directed by protocol on the Upstream Component to enter PM.

5. When the ARB&MUX on the Downstream Component is directed to enter L1 and receives ALMP response from the Upstream Component, it notifies acceptance of entry into PM state to the transaction layer on the Downstream component.
6. The Downstream Component ARB&MUX port must wait for <TBD> amount of time for a response from the Upstream Component. If no response is received from the Upstream component then the Downstream Component is permitted to abort the PM entry or retry entry into PM again.
7. L2 entry is an exception to rule number 6. Protocol must ensure that Upstream component is directed to enter L2 before setting up the conditions for the Downstream Component to request entry into L2 state. This ensures that L2 abort or L2 Retry conditions do not exist.
8. Transaction layer on either side of the Link is permitted to direct exit from L1 state once the ARB&MUX interface reaches L1 state.

9.4.3 Compute Express Link PM Entry Phase 3

The third Phase is a conditional phase of PM entry and is executed only when all Protocol interfaces of ARB&MUX have entered the same virtual PM state. The phase consists of bringing the Tx lanes to electrical Idle and is always initiated by the Downstream Component.

Figure 111. CXL PM Phase 3

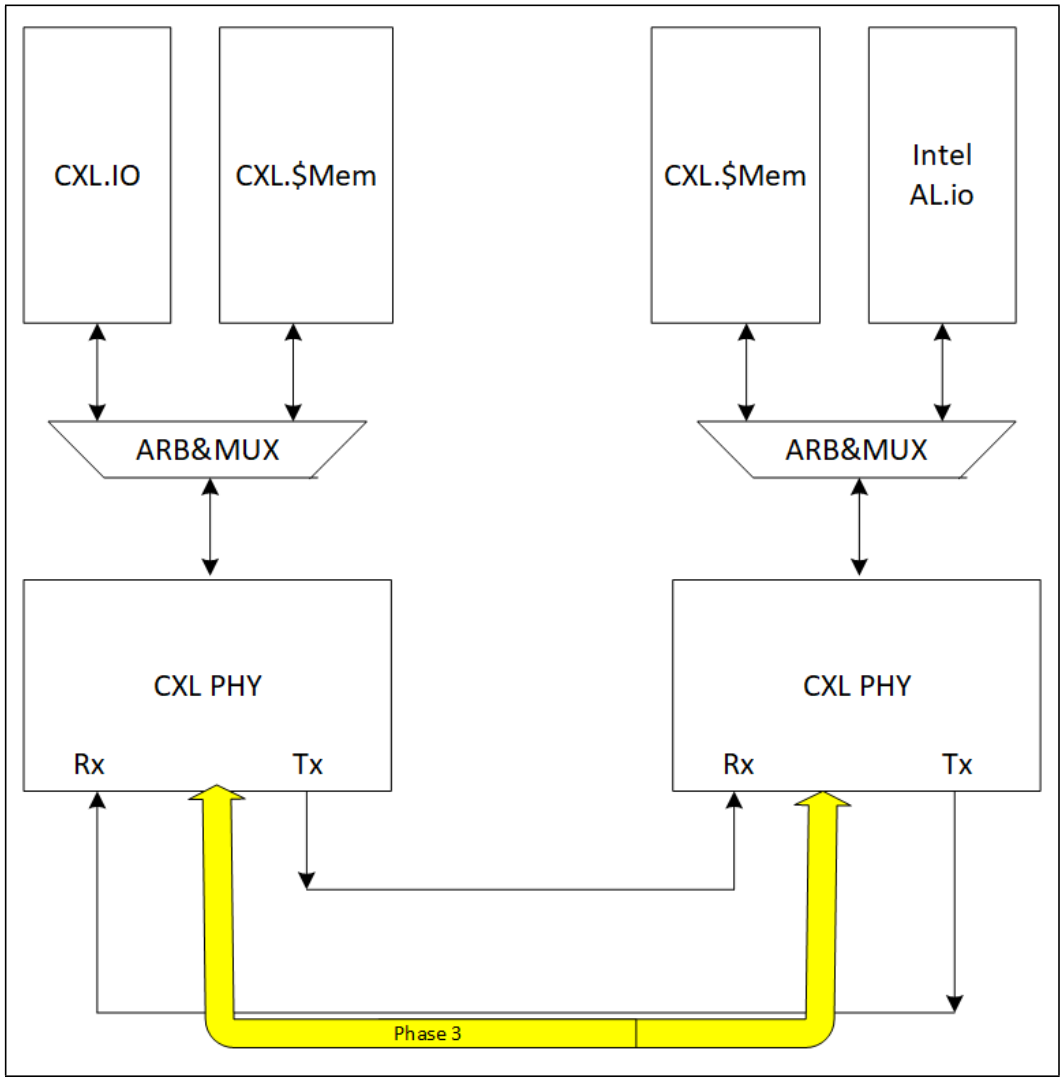
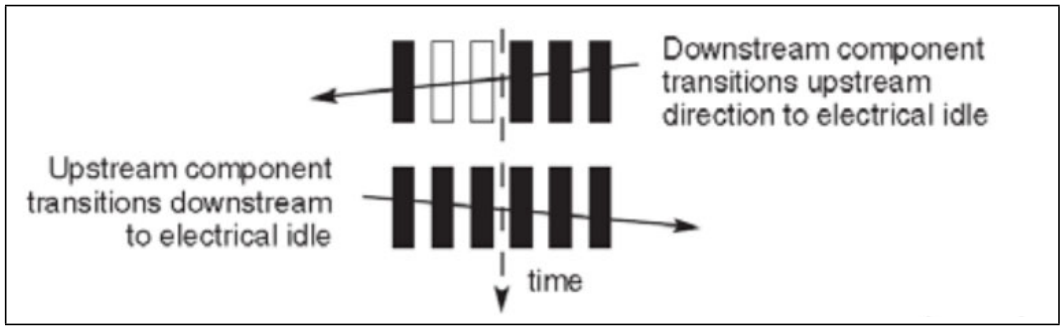


Figure 112. Electrical Idle



9.4.4 Compute Express Link Exit from ASPM L1

Components on either end of the Link may initiate exit from the L1 Link State. The ASMP L1 exit depends on whether the exit is from phase 3 or phase 2 of L1. The exit is hierarchical and phase 3 must exit before phase 2.

Phase 3 exit is initiated when directed by the ARB&Mux from either end of the link. The ARB&MUX layer initiates exit from Phase 3 when there is an exit requested on any one of its primary protocol interfaces. The phase 3 ASPM L1 exit is the same as exit from L1 state as defined in PCI Express Base Specification. The steps are followed until the LTSSM reaches L0 state. Protocol level information is not permitted to be exchanged until the virtual LSM on the ARB&MUX interface has exited L1 state.

Phase 2 exit involves bringing the protocol interface at the ARB&MUX out of L1 state independently. The transaction layer directs the ARB&MUX state to exit virtual LSM state. If the PHY is in Phase 3 L1 then the ARB&MUX waits for the PHY LTSSM to reach L0 state. Once the PHY is in L0 state, the following rules apply.

The ARB&MUX on the protocol side that is triggering an exit transmits a ALMP requesting entry into Active state.

Any ARB&MUX interface that receives the ALMP request to enter Active State must transmit an ALMP acknowledge response on behalf of that interface. The ALMP acknowledge response is an indication that the corresponding protocol side is ready to process received packets.

Any ARB&MUX interface that receives the ALMP request to enter Active State must also transmit an ALMP Active State request on behalf of that interface if not sent already.

Protocol level transmission must be permitted by the ARB&MUX after an Active State Status ALMP is transmitted and received. This guarantees that the receiving protocol is ready to process packets.

9.5 CXL.io Link Power Management

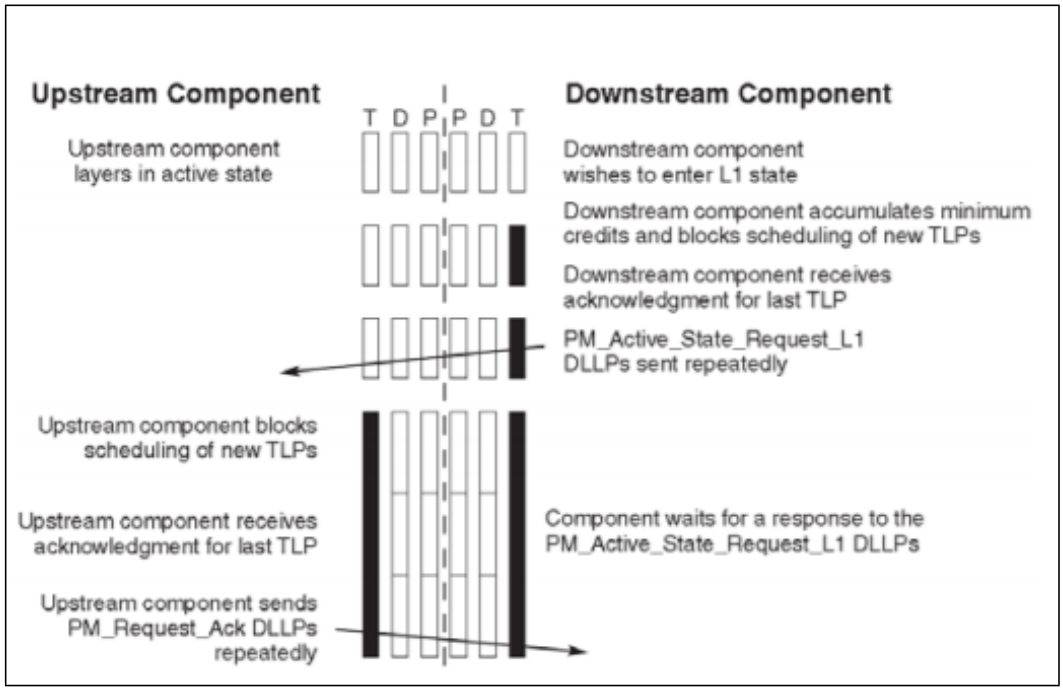
CXL.io Link Power Management is as defined in PCIe Express Base Specification with the following notable differences.

- Only ASPM L1 is supported
- L0s state is not supported
- PCI-PM is not supported

9.5.1 CXL.io ASPM Phase L1 Entry

- The first phase consists of completing the ASPM L1 negotiation rules as defined in the PCI Express Base Specification with the following notable exceptions for the rules in case of acceptance of ASPM L1 Entry. All rules upto the completion of the ASPM L1 handshake are maintained, however the process of bringing the Transmit Lanes into Electrical Idle state are divided into 2 additional phases described above. Phase 1 flow is described below.

Figure 113. ASPM L1 Entry Phase 1



9.5.2 CXL.io ASPM Phase 2 Entry

The following conditions apply for Phase 2 Entry for CXL.io

Phase 2: The second Phase of L1 entry consists of bringing the CXL.io ARB&MUX interface of both sides of the Link into L1 state. This entry into L1 state is coordinated using ALMPs as described below.

Rules for Phase 2 entry into ASPM L1.

1. CXL.io on the Upstream Component must direct the ARB&MUX to be ready to enter L1 before returning the PM_Request_Ack DLLPs as shown above in Phase 1.
2. When the PM_Request_Ack DLLPs are successfully received by the CXL.io on the Downstream Component, it must direct the ARB&MUX on the Downstream Component to transmit ALMP request to enter Virtual LSM state L1.
3. When the ARB&MUX on the Upstream Component is directed to enter L1 and receives ALMP request from the Downstream Component, it notifies the CXL.io that the interface has received ALMP request to enter L1 state and has entered L1 state
4. When the Upstream Component is notified entry into virtual LSM it ceases sending PM_Request_Ack DLLP
5. When the ARB&MUX on the Downstream Component is directed to enter L1 and receives ALMP Status from the Upstream Component, it notifies the CXL.io that the interface has entered L1 state

9.5.3 CXL.io ASPM Phase 3 Entry

The Phase 3 entry is dependent on the virtual LSM state of multiple protocols and is managed by the ARB&MUX as described in the section on Phase 3 entry above.

9.6 CXL.cache + CXL.mem Link Power Management

CXL.cache and CXL.mem support Active Link State Power Management only, unlike CXL.io there is no PM Entry handshake defined between the Link Layers. Each side independently requests to the ARB&MUX to enter L1. The ARB&MUX layers on both sides of the Link co-ordinate the entry into PM state using ALMPs. CXL.cache + CXL.mem Link Power Management follows the process for PM entry and exit as defined in section Compute Express Link Power Management.

§ §

10.0 Security

Security requirements are product specific and thus outside of the scope of this specification.

§ §

11.0 Reliability, Availability and Serviceability

CXL RAS is defined to work with Client Hosts as well as Servers. Therefore RAS features intended for server use must also consider the impact for client space or provide means for disabling.

11.1 Supported RAS Features

The table below lists the RAS features supported by CXL and their applicability to CXL.io vs. CXL.cache and CXL.mem.

Table 70. CXL RAS Features

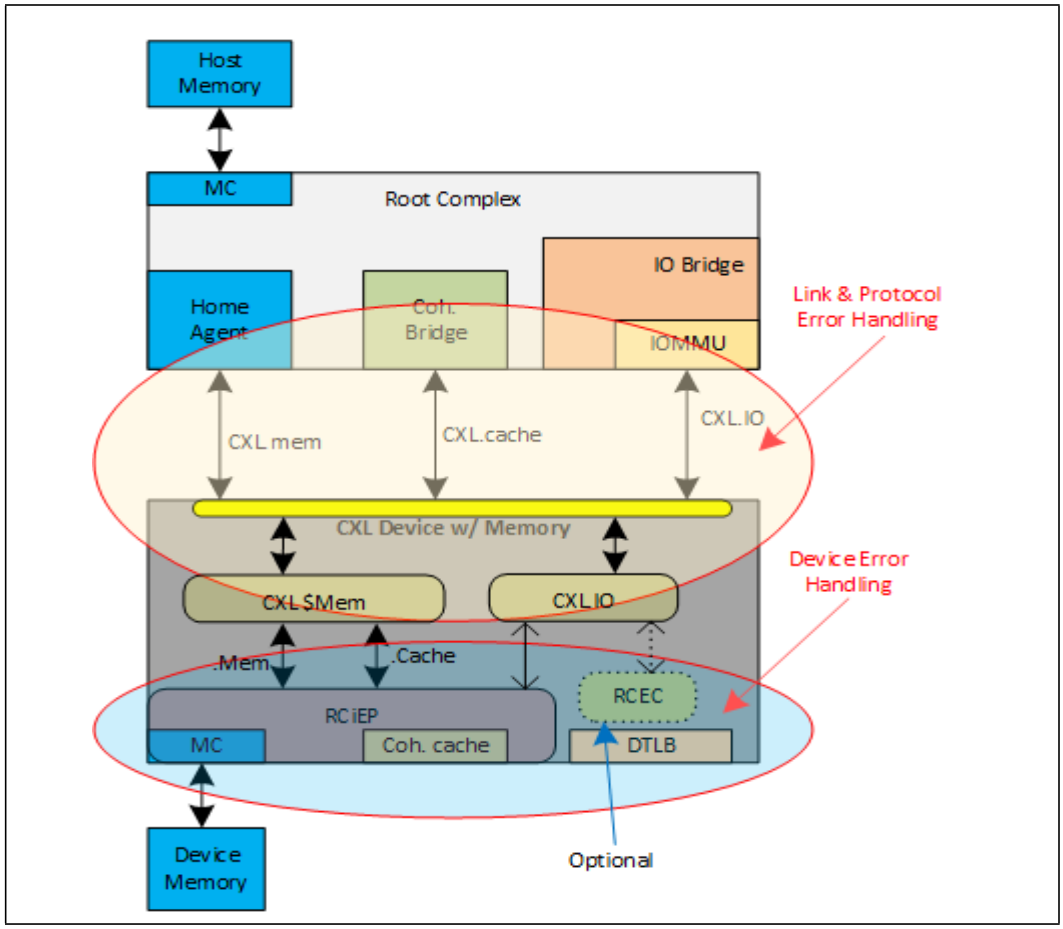
Feature	CXL.io	CXL.cache and CXL.mem
Link CRC and Retry	Required	Required
Link Retraining and Recovery	Required	Required
eDPC	Not Supported	N/A
ECRC	Optional	N/A
Hot-Plug	Not Supported	Not Supported
Corrected Error Count Information	Requied	Required
Data Poisoning	Required	Required
Viral	N/A	Required

11.2 CXL Error Handling

As shown in Figure 114, CXL can simultaneously carry three protocols: CXL.io, CXL.cache and CXL.mem. CXL.io carries PCIe like semantics and must be supported by all CXL endpoints. All RAS capabilities must address all of these protocols and usages. For details of CXL architecture and all protocols, please refer to the other sections in this document.

Figure 114 below is an illustration of CXL and the focus areas for CXL RAS. Namely, Link & protocol RAS, which applies to the Host-CXL communication mechanism and Device RAS which applies exclusively to the device itself. All errors are reflected to the OS via PCIe AER mechanisms as “Correctable Internal Error” (CIE) or “Uncorrectable Internal Error” (UIE). Errors may also be reflected to Platform SW if so configured.

Figure 114. CXL Error Handling



Referring to Figure 114, the Host/Root Complex is located on the north side and contains all the usual error handling mechanisms such as MCA, PCIe AER, RCEC and other platform level error reporting and handling mechanisms. CXL.mem and CXL.cache errors encountered by the device are communicated to the CPU across CXL.io. to be logged in PCIe AER registers. The following sections will focus on the link layer and transaction layer error handling mechanisms as well as CXL device error handling.

11.2.1 Protocol and Link Layer Error Reporting

Protocol and Link errors are detected and communicated to the Host where they can be exposed and handled. There are no error pins connecting CXL devices to the Host. Errors are communicated between the Host and the CXL device via messages over CXL.io.

11.2.1.1 CXL Downstream Port (DP) Detected Errors

Errors detected by the CXL are escalated and reported via the Root Complex error reporting mechanisms as UIE/CIE.

To handle the error, the OS would inspect the RCEC AER and handle as appropriate. For Platform SW Error handling, the SW would interrogate the Platform specific error logs.

11.2.2 CXL Device Error Handling

CXL connected devices are required to support data poisoning and will use the Data Poisoning mechanism to communicate uncorrectable data errors whenever possible (all flavors of CXL.cache, CXL.mem, CXL.io support poison communication).

The Host may send poisoned data to the CXL connected device. How the CXL device responds to Poison is device specific but must follow PCIe guidelines. The device must consciously make a decision about what to make of poisoned data. In some cases, simply ignoring poisoned data may lead to SDC (Silent Data Corruption).

Any device errors that cannot be handled with Poison indication will be signaled by the device back to the Host as messages since there are no error pins. It's highly desirable to have the same nomenclature and reporting scheme. To that end, Table 71 below shows a summary of the error types, their mappings and error reporting guidelines.

Table 71. Device Specific Error Reporting and Nomenclature Guidelines

Error Severity	Definition/ Example	Signaling Options (SW picks one)	Logging	Host HW/FW/SW Response
Corrected	Memory single bit error corrected via ECC	MSI to Device driver	Device specific registers	Device specific flow in Device driver
Uncorrected Recoverable	UC errors that device can recover from, with minimal or no SW help (e.g., error localized to single computation)	MSI to driver	Device specific registers	Device specific flow in driver (e.g., discard results of suspect computation)
Uncorrected NonFatal	Equivalent to PCIe UCNF, contained by the device (e.g., write failed, memory error that affects many computations)	MSI to Device Driver	Device specific registers	Device specific (e.g., reset affected device) flow in driver. Driver can escalate through SW.
		PCIe AER Internal Error	Device specific registers + PCIe AER	System FW/SW AER flow, ends in reset.
Uncorrected Fatal	Equivalent to PCIe UCF, poses containment risk (e.g., command/ parity error, pUnit ROM error)	PCIe AER Internal error	Device specific registers + PCIe AER	System FW/SW AER flow, ends in reset.
		AER + Viral		System FW/SW Viral flow

In keeping with the standard error logging requirements, all error logs must be sticky across warm reset.

11.2.2.1 CXL.mem and CXL.cache Errors

If demand accesses to memory result in an uncorrected data error, the CXL device must return data with poison. The requester (processor core or a peer device) is responsible for dealing with the poison indication. The CXL device should not signal an uncorrected error along with the poison. If the processor core consumes the poison, the error will be logged and signaled by the Host.

It is recommended that the CXL device keep track of any poison data it receives while it stores the data in its local storage including memory. It can optionally notify the device driver of such an event. If the CXL device cannot keep track of poison, Device logic shall ensure containment by signaling an uncorrected non-fatal error (AER_NONFATAL).

Any non-demand uncorrected errors (e.g., memory scrub logic in CXL device memory controller) will be signaled to the device driver via device MSI. Any corrected memory errors will be signaled to the device driver via device MSI. The driver may choose to deallocate memory pages with repeated errors. Neither the platform firmware nor the OS directly deal with these errors.

11.2.2.2 CXL Device Error Handling Flows

Device errors maybe sourced from a Root Port (RP) or Endpoint (RCiEP). For the purpose of differentiation RCiEP sourced errors shall use tag value of zero whereas RP sourced errors shall use tag of non-zero value. Errors detected by the CXL device shall be communicated to the host via PCIe Error messages across the CXL.io link.

Errors that are not related to any specific Function within the device (Non-Function errors) are reported to the Host via PCIe error messages where they can be escalated to the platform. Non-Function errors are logged in the Upstream Port RCRB in the PCIe AER Registers. In addition, the UP reports non-function errors to all RCiEPs where they are logged. Each RCiEP reports the non-function specific errors to the host via error messages. At most one error message of a given severity is generated for a multi-function device. The error message must include the Requester ID of a function that is enabled to send the error message. Error messages with the same Requester ID may be merged for different errors with the same severity. No error message is sent if no function is enabled to do so. If different functions are enabled to send error messages of different severity, at most one error of each severity level is sent. If a Root Complex Error Collector is implemented, errors may optionally be sent to the corresponding RCEC. Each RCiEP must be associated with no more than one RCEC.

11.3 CXL Link Down Handling

There is no expectation of a graceful Link Down. A Link Down condition will most likely result in a timeout in the Host since it is quite possible that there are transactions headed to or from the CXL device that will end up not making progress.

11.4 CXL Viral Handling

CXL link and CXL devices are expected to be Viral compliant. Viral is an error containment mechanism. A platform must choose to enable Viral at boot time. The Host implementation of Viral allows the platform to opt-in by writing into a register that enables the Viral feature. Similarly, a BIOS accessible control register on the device will be required to enable Viral behavior (both receiving and sending) on the device. Viral support capability and control for enabling are reflected in DVSEC.

When enabled, a Viral indication is generated whenever an Uncorrected_Fatal error is detected. Viral is not a replacement for existing error reporting mechanisms. Instead, its purpose is an additional error containment mechanism. The detector of the error is responsible for reporting the error through AER and generating a Viral indication. Any entity that is capable of reporting Uncorrected_Fatal errors must also be capable of generating a Viral indication.

CXL.mem and CXL.cache come enabled with the Viral concept. Viral needs to be communicated in both directions. When Viral is enabled and the Host runs into a Viral condition, it will communicate Viral across CXL.mem and/or CXL.cache. The Viral indication must beat any data that may have been affected by the error (general Viral requirement).

The device's reaction to Viral is going to be device specific but the device is expected to take error containment actions consistent with Viral requirements. Chiefly, it must prevent bad data from being committed to permanent storage. Meaning, if the device is connected to any permanent storage or an external interface that may be connected to permanent storage, then the device is required to self-isolate in order to be Viral compliant. This means that the device has to take containment actions without depending on help from the Host.

The self-isolation actions taken by the device must not prevent the Host from making forward progress. This is important for diagnostic purposes as well as error pollution (e.g., withholding data for read transactions to device memory may cause cascading timeouts in the Hosts). Therefore, on Viral detection, in addition to the containment requirements, the device must:

- Drop writes to permanent storage on the device or connected to the device.
- Keep responding to snoops
- Complete pending writes to Host memory
- Complete all reads and writes to Device volatile memory.

When the device itself runs into a Viral condition and Viral is enabled, it must:

- Set the Viral Status bit to indicate that a Viral condition has occurred
- Self-Isolate – i.e., take steps to contain the error within the device as per Viral requirements (i.e., ensure that Viral signaling beats any data affected by the error)
- Communicate the Viral condition back up CXL.{Mem,Cache} towards the Host.
  - In reaction to this the CXL Downstream Port will trigger Viral on the host.
- Report the error as UIE via AER.

Viral Control and Status bits are defined in DVSEC (please refer to Section 7.0, “Control and Status Registers” on page 145 for details).

When a CXL.AL device goes into Viral, the upstream CXL.io shall perform the following:

- Master Abort Upstream Requests
- Completer Abort Upstream Completions
- Signal Failed Response for Downstream Completions

11.5 CXL Error Injection

The major aim of error injection mechanisms is to allow system validation and system FW/SW development …etc. the means to create error scenarios and error handling flows. To this end, CXL UP and DP are recommended to implement the following error injection hooks to a specified address (where applicable):

- One type of CXL.io UC error (optional - similar to PCIe).
  - CXL.io is always present in any CXL connection
- One type of CXL.mem UC error (if applicable)
- One type of CXL.cache UC error (if applicable)
- Link Correctable errors
  - Transient mode and
  - Persistent mode
- Returning Poison on a read to a specified address (CXL.mem only)

CXL devices themselves might need error injection mechanisms for developing device driver flows. But error injection into CXL devices is device specific and out of the scope of this document.

12.0 Platform Architecture

12.1 Flex Bus connector definition

12.1.1 Connector Type

The current direction for x16 Flex Bus connector is to be the same as the standard x16 PCIe gen5 connector as specified in PCI_Express_CEM_r5.0 specification. This connector is expected to scale up to 32GTs transfer rate being supported on the Flex Bus interface.

12.1.2 Pin Count

The x16 Flex Bus connector will have the same pin count and pin assignment as the standard x16 PCIe gen4 connector as in PCI_Express_CEM_r5.0 specification. The expectation is that all supported Flex Bus cards will not require additional signals (main band, sideband, power, etc.) beyond what is provided by the standard x16 PCIe gen5 connector.

Note: The standard x16 PCIe gen4 connector does have 5 "RSVD" pins, but customers might have used these pins for some unique implementations on their platform and Flex Bus cards should not plan to use these "RSVD" pins.

The figure below shows the standard x16 PCIe connector pin list for reference purpose.

Figure 115. Standard x16 PCIe Connector Pin List - For Reference Purpose Only

	Side B		Side A	
Pin #	Name	Description	Name	Description
1	+12V	+12 V power	PRSNT1#	Hot-Plug presence detect
2	+12V	+12 V power	+12V	+12 V power
3	+12V	+12 V power	+12V	+12 V power
4	GND	Ground	GND	Ground
5	SMCLK	SMBus (System ManagementBus) clock	JTAG2	TCK (Test Clock), clock input forJTAG interfa
6	SMDAT	SMBus (System ManagementBus) data	JTAG3	TDI (Test Data Input)
7	GND	Ground	JTAG4	TDO (Test Data Output)
8	+3.3V	+3.3 V power	JTAG5	TMS (Test Mode Select)
9	JTAG1	TRST# (Test Reset) resets theJTAG interface	+3.3V	+3.3 V power
10	+3.3Vaux	+3.3 V auxiliary power	+3.3V	+3.3 V power
11	WAKE#	Signal for Link reactivation	PERST#	Fundamental reset
Mechanical Key				
12	CLKREQ#	Clock Request Signal	GND	Ground
13	GND	Ground	REFCLK+	Reference clock (differential pair)
14	PETp0	Transmitter differential pair,Lane 0	REFCLK-	
15	PETn0		GND	Ground
16	GND	Ground	PERp0	Receiver differential pair, Lane 0
17	PRSNT2#	Hot-Plug presence detect	PERn0	
18	GND	Ground	GND	Ground
End of x1 conn				
19	PETp1	Transmitter differential pair,Lane 1	RSVD	
20	PETn1		GND	Ground
21	GND	Ground	PERp1	Receiver differential pair, Lane 1
22	GND	Ground	PERn1	
23	PETp2	Transmitter differential pair,Lane 2	GND	Ground
24	PETn2		GND	Ground
25	GND	Ground	PERp2	Receiver differential pair, Lane 2
26	GND	Ground	PERn2	
27	PETp3	Transmitter differential pair,Lane 3	GND	Ground
28	PETn3		GND	Ground
29	GND	Ground	PERp3	Receiver differential pair, Lane 3
30	PWRBRK#	Emergency Power Reduction	PERn3	
31	PRSNT2#	Hot-Plug presence detect	GND	Ground
32	GND	Ground	RSVD	Reserved
End of x4 conn				
33	PETp4	Transmitter differential pair,Lane 4	RSVD	Reserved
34	PETn4		GND	Ground
35	GND	Ground	PERp4	Receiver differential pair, Lane 4
36	GND	Ground	PERn4	
37	PETp5	Transmitter differential pair,Lane 5	GND	Ground
38	PETn5		GND	Ground
39	GND	Ground	PERp5	Receiver differential pair, Lane 5
40	GND	Ground	PERn5	
41	PETp6	Transmitter differential pair,Lane 6	GND	Ground
42	PETn6		GND	Ground
43	GND	Ground	PERp6	Receiver differential pair, Lane 6
44	GND	Ground	PERn6	
45	PETp7	Transmitter differential pair,Lane 7	GND	Ground
46	PETn7		GND	Ground
47	GND	Ground	PERp7	Receiver differential pair, Lane 7
48	PRSNT2#	Hot-Plug presence detect	PERn7	
49	GND	Ground	GND	Ground
End of x8 conn				
50	PETp8	Transmitter differential pair,Lane 8	RSVD	Reserved
51	PETn8		GND	Ground
52	GND	Ground	PERp8	Receiver differential pair, Lane 8
53	GND	Ground	PERn8	
54	PETp9	Transmitter differential pair,Lane 9	GND	Ground
55	PETn9		GND	Ground
56	GND	Ground	PERp9	Receiver differential pair, Lane 9
57	GND	Ground	PERn9	
58	PETp10	Transmitter differential pair,Lane 10	GND	Ground
59	PETn10		GND	Ground
60	GND	Ground	PERp10	Receiver differential pair, Lane 10
61	GND	Ground	PERn10	
62	PETp11	Transmitter differential pair,Lane 11	GND	Ground
63	PETn11		GND	Ground
64	GND	Ground	PERp11	Receiver differential pair, Lane 11
65	GND	Ground	PERn11	
66	PETp12	Transmitter differential pair,Lane 12	GND	Ground
67	PETn12		GND	Ground
68	GND	Ground	PERp12	Receiver differential pair, Lane 12
69	GND	Ground	PERn12	
70	PETp13	Transmitter differential pair,Lane 13	GND	Ground
71	PETn13		GND	Ground
72	GND	Ground	PERp13	Receiver differential pair, Lane 13
73	GND	Ground	PERn13	
74	PETp14	Transmitter differential pair,Lane 14	GND	Ground
75	PETn14		GND	Ground
76	GND	Ground	PERp14	Receiver differential pair, Lane 14
77	GND	Ground	PERn14	
78	PETp15	Transmitter differential pair,Lane 15	GND	Ground
79	PETn15		GND	Ground
80	GND	Ground	PERp15	Receiver differential pair, Lane 15
81	PRSNT2#	Hot-Plug presence detect	PERn15	
82	RSVD	Reserved	GND	Ground
End of x16 conn				

12.2 Topologies

Since Flex Bus utilizes PCIe Gen4/5 electrical interface and PCIe gen5 connector, its topologies will closely follow those of PCIe gen4/5 as well. Similar PCIe gen4/5 platform enablers (lower loss PCB material, re-timer, etc.) are also applicable for Flex Bus in order to achieve more challenging platform topologies (longer length, multiple connectors).

Note: Flex Bus re-timer is essentially the same as PCIe gen4/5 retimer with the exception that it requires a much lower latency on the retimer. Refer to Section 6.7, “Retimers and Low Latency Mode” on page 143 and Section 1.4.2, “Flex Bus” on page 16 for more details on Flex Bus re-timer support and requirements.

12.3 Protocol Detection

Since Flex Bus or PCIe card can be installed in the same PCIe slot, platform will need to be able to detect which card type is being installed in order to configure the link to the correct protocol. Current direction on protocol detection (Flex Bus vs. PCIe) is "in-band" methodology during link training. Refer to section Section 6.3.1, “PCIe vs Flex Bus.CXL mode selection” on page 138 for more detail on detection mechanism during booting up.

12.4 AIC Form Factor

Flex Bus card form factor will follow the standard PCI_Express_CEM_r5.0 specification.

12.5 AIC Power Envelope

Flex bus card power envelope will follow the standard PCI_Express_CEM_r4.0 specification, which supports up to 300W.

The x16 PCIe conn only supports up to 75W card. Auxiliary power connector will be required to support >75W Flex Bus card (per CEM spec).

12.6 Flexbus Slot Auxiliary Power

For system with S3 power state support (e.g. Workstation platform), Flexbus slot is required to support up to 375mA on the “+3.3Aux” pin. This is to accommodate CXL cards with HDM. If a CXL card requires more than 375mA in S3 state, platform will needs to supply additional aux power to the Add-card (platform implementation dependent).

§ §

13.0 Performance Considerations

Compute Express Link (CXL) provides a low-latency, high-bandwidth path for an accelerator to access the system. Performance on CXL is dependent on a variety of factors. The following table captures the key performance attributes of CXL.

Characteristic	Compute Express Link via Flex Bus (if Gen 4)	Compute Express Link via Flex Bus (if Gen 5)
Width	16 Lanes	16 Lanes
Link Speed	16 GT/s	32 GT/s
Total BW per link[1]	32 GB/s	64 GB/s

1. Achieved bandwidth depends on protocol and payload size. Expect 60-90% efficiency on CXL.cache and CXL.mem. Efficiency similar to PCIe on CXL.io.

In general, it is expected that the downstream-facing port and the upstream-facing ports are rate-matched. However, if the implementations are not rate-matched, it would require the faster of the implementations to limit the rate of its protocol traffic to match the slower (including bursts), whenever there is no explicit flow-control loop.

CXL allows accelerators/devices to coherently access host memory and allows memory attached to an accelerator/device to be mapped into the system address map and accessed directly by the host as writeback memory. In order to support this, it supports a Bias-based Coherency model as described in section Section 2.2.1. There are specific performance considerations to take into account for selecting the method for mode management. This is addressed in section Section 2.2.1.3.

Note: On CXL.cache, in order to ensure system performance is not negatively impacted, it is recommended that the maximum latency for a snoop-miss is 50ns from H2D snoop request seen on the CXL pins to a D2H snoop-response back at the CXL pins. Similarly, the maximum latency for a H2D Wr_Pull response to D2H Data response is 40ns.

Note: On CXL.mem, in order to ensure system performance is not negatively impacted, it is recommended that the maximum latency for a memory read is 80ns from M2S Req seen on the CXL pins to a S2M DRS back at the CXL pins. Similarly, the maximum latency for a M2S RwD to S2M NDR is 40ns. The latency budgets mentioned here are for HBM or DDR type memory technologies. If a slower memory technology is used, and the above targets cannot be met, the device and Host may need to provision for special QoS in order to ensure that the rest of the system is not negatively affected. These QoS mechanisms are outside the scope of this specification.

§ §

14.0 CXL Compliance Testing

14.1 Applicable Devices Under Test (DUTs)

The tests outlined in this chapter are applicable to all devices that support alternate protocol negotiation and are capable of CXL only or CXL and PCIe protocols. The tests are broken into the different categories corresponding to the different chapters of CXL specification, starting with Chapter 3.0.

14.2 Starting Configuration/Topology (Common for All Tests)

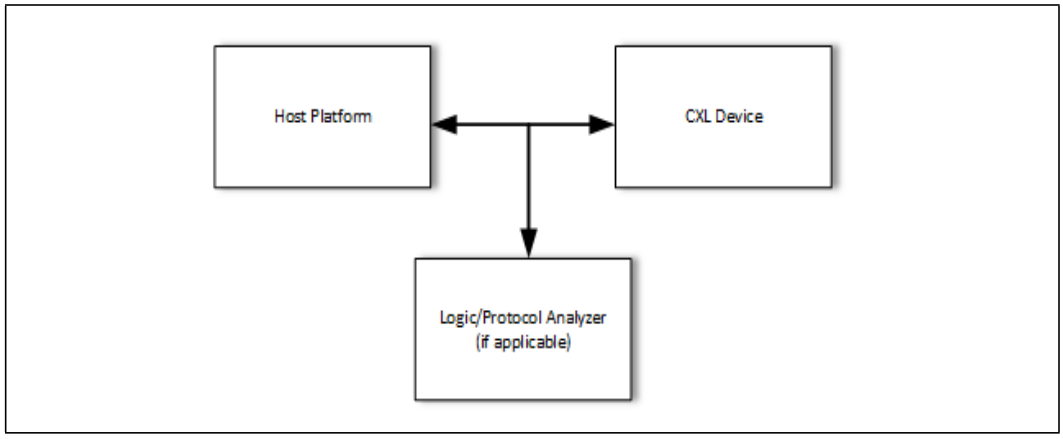
In most tests, the initial conditions assumed are as follows (deviations from these conditions are pointed out in specific tests, if applicable):

System is powered on, running in test environment OS, device specific drivers have loaded on device, and link has trained to supported CXL modes. All error status registers should be clear on the device under test.

Some tests make assumptions about only one CXL device present in the system – this is called out in relevant tests. If nothing is mentioned, there is no limit on the number of CXL devices present in the system, however, the number of DUTs is limited to what the test software can support.

Certain tests may also require the presence of a protocol analyzer to monitor flits on the physical link for determining Pass or Fail results.

Figure 116. Example Test Topology



Each category of tests has certain device capability requirements in order to exercise the test patterns. The associated registers and programming is defined in this chapter as well.

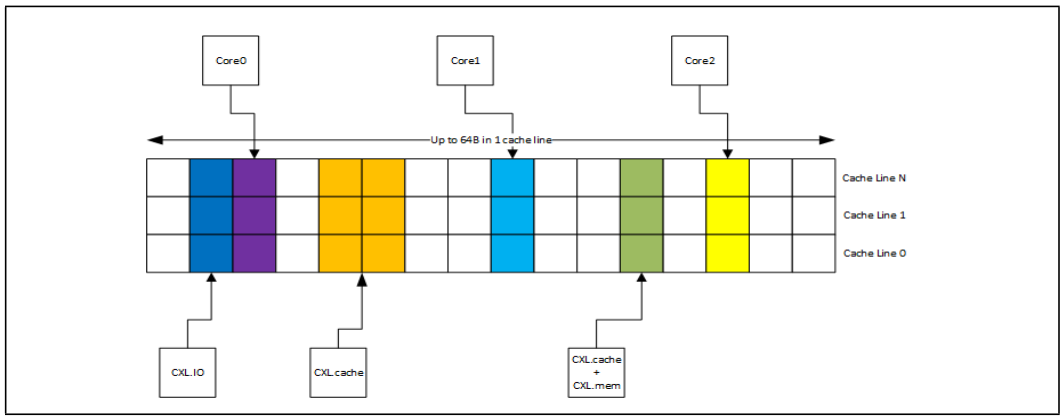
Refer to Section 14.11, “Device Capability and Test Configuration Control” on page 239 for registers applicable to tests in the following sections.

14.3 CXL.cache and CXL.io Application Layer / Transaction Layer Testing

14.3.1 General Testing Overview

Standard practices of testing coherency rely on “false sharing” of cache lines. Different agents in the system (cores, I/O etc.) are assigned one or more fixed byte locations within a shared set of cache lines. Each agent continuously executes an assigned Algorithm independently. Since multiple agents are sharing the same cache line, stressful conflict scenarios can be exercised. Figure 117 illustrates the concept of false sharing. This can be used for CXL.io (Load/Store semantics) or CXL.cache (caching semantics) or (CXL.cache + CXL.mem) devices (Type 2 devices).

Figure 117. Representation of False Sharing Between Cores (on Host) and CXL Devices



This document outlines three Algorithms that enable stressing the system with false sharing tests. In addition, this document specifies the required device capabilities to execute, verify and debug runs for the Algorithms. All of the Algorithms are applicable for CXL.io and CXL.cache (protocols that originate requests to the host). Devices are permitted to be self-checking. Self-checking devices must have a way to disable the checking Algorithm independent of executing the Algorithm. All devices must support the non-self-checking flow in the Algorithms outlined below.

14.3.2 Algorithms

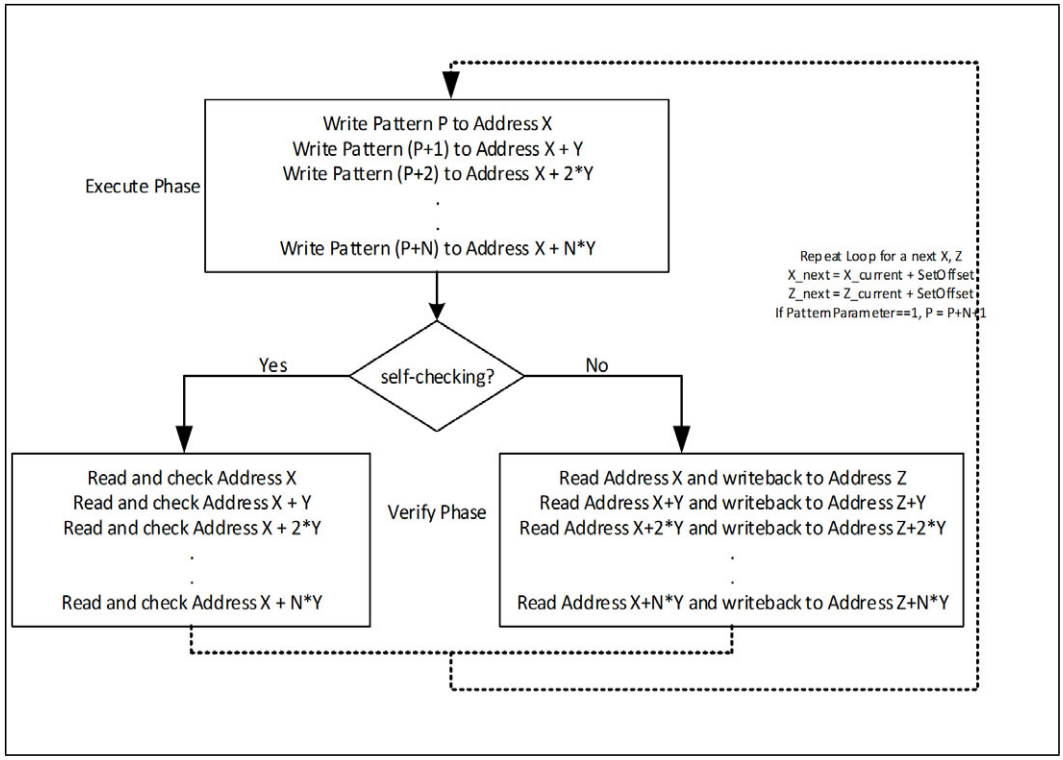
14.3.3 Algorithm 1a: Multiple Write Streaming

In this Algorithm, the device is setup to stream an incrementing pattern of writes to different sets of cache lines. Each set of cache line is defined by a base address “X”, and an increment address “Y”. Increments are in multiples of 64B. The number of increments “N” dictates the size of the set beginning from base address X. The base address includes the byte offset within the cache line. A pattern P (of variable length in bytes) determines the starting pattern to be written. Subsequent writes in the same set increment P. A device is required to provide a byte mask configuration capability that can be programmed to replicate pattern P in different parts of the cache line. The programmed byte masks must be consistent with the base address.

Different sets of cache lines are defined by different base addresses (so a device may support a set like “X_1, X_2, X_3”). “X_1” is programmed by software in the base address register, X_2 is obtained by adding a fixed offset to X_1 (offset is programmed by software in a different register).X_3 is obtained by adding the same offset to X_2 and so on. Minimum support of 2 sets is required by the device. Figure 118 illustrates the flow of

this Algorithm as implemented on the device. Address Z is the write back address where system software can poll to verify the expected pattern associated with this device, in cases where self-checking on the device is disabled. There is 1:1 correspondence between X and Z. It is the responsibility of the device to ensure that the writes in the execute phase are globally observable before beginning the verify phase. Depending on the write semantics used, this may imply additional fencing mechanism on the device to make sure the writes are visible globally before the verify phase can begin. When beginning a new set iteration, devices must also give an option to use "P" again for the new set, or continue incrementing "P" for the next set. The select is programmed by software in "PatternParameter" field described in the register section.

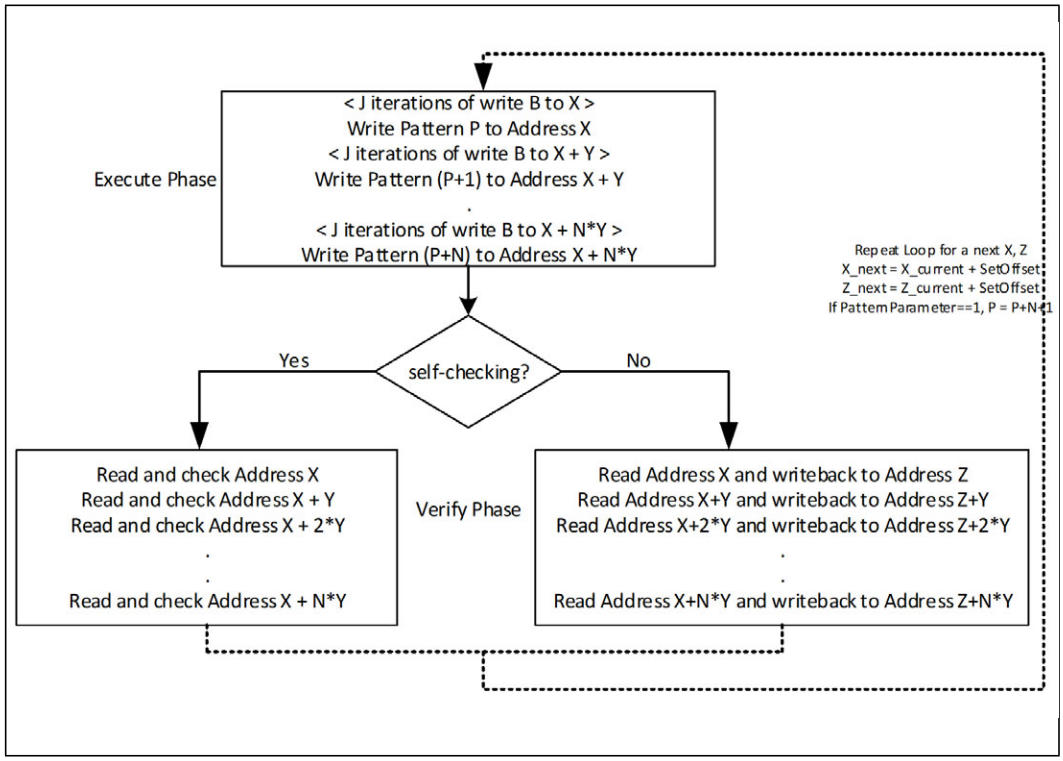
Figure 118. Flow Chart of Algorithm 1a



14.3.4 Algorithm 1b: Multiple Write Streaming with Bogus Writes

This Algorithm is a variation on Algorithm 1a, except that before writing the expected pattern to an address, the device does "J" iterations of writing a bogus pattern "B" to that address. Figure 119 illustrates this Algorithm. In this case, if ever a pattern "B" is seen in the cache line during the Verify phase, it is a Fail condition. The bogus writes help give a longer duration of conflicts in the system. It is the responsibility of the device to ensure that the writes in the execute phase are globally observable before beginning the verify phase. Depending on the write semantics used, this may imply additional fencing mechanism on the device to make sure the writes are visible globally before the verify phase can begin. When beginning a new set iteration, devices must also give an option to use "P" again for the new set, or continue incrementing "P" for the next set. The select is programmed by software in "PatternParameter" field described in the register section.

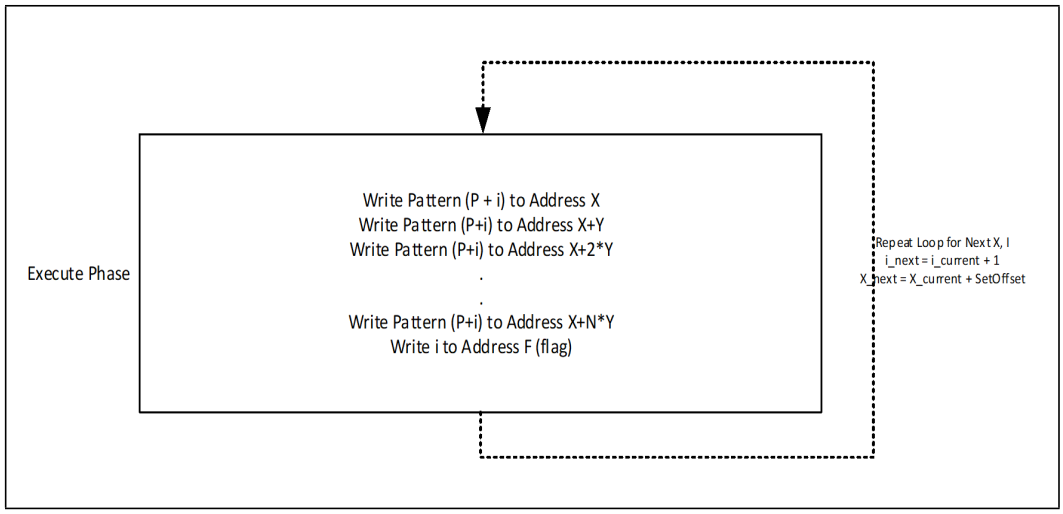
Figure 119. Flow Chart of Algorithm 1b



14.3.5 Algorithm 2: Producer Consumer Test

This Algorithm aims to test the scenario where a Device is a producer and the CPU is a consumer. Device simply executes a pre-determined Algorithm of writing known patterns to a data location followed by a flag update write. Threads on the CPU poll the flag followed by reading the data patterns, followed by polling the flag again. This is a simple way of making sure the required ordering rules of producer consumer workloads are being followed through the stack. Device only participates in the execute phase of this Algorithm. Figure 120 illustrates the device execute phase. The Verify phase is run on the CPU, software reads addresses in the following order [F, X, (X+Y)…(X+N*Y), F]. Knowing the value of the flag at two ends, the checker knows the range in which [X, (X+Y)…(X+N*Y)] have to be in. For example, if P=0, the first read of F returns a value of 3 and the next read of F returns a value of 4, then checker knows that all intermediate values have to be either 3 or 4. Moreover, if the device is using strongly ordered semantics, then the checker should never see a transition of values from 3 to 4 (implying monotonically decreasing values for the non-flag addresses). If using CXL.cache protocol, device must ensure global observability of previous “data” writes before updating the flag. When using strongly ordered semantics, each update must be globally visible before the next one. Depending on the flow used for dirty evicts, this can be implementation specific. It is the responsibility of the device to ensure that the writes in the execute phase are globally observable before updating the flag “F”. The “PatternParameter” field is not relevant for this Algorithm.

Figure 120. Execute Phase for Algorithm 2



14.3.6 Test Descriptions

14.3.6.1 Application Layer/Transaction Layer Tests

The Transaction Layer Tests implicitly give coverage for Link Layer functionality. Specific error injection cases for the Link Layer are covered in the RAS section.

14.3.6.1.1 CXL.io Load/Store Test

For CXL.io, this test and associated capabilities are optional but strongly recommended. This test sets up the device to execute Algorithm 1a, 1b and 2 in succession in order to stress data path for CXL.io transactions. Configuration details are determined by the host platform testing the device. Refer to Section 14.11 for the configuration registers and device capabilities. Each run includes execute/verify phases as described in section Section 14.3.1.

Test Steps:

1. Host software will setup Device for Algorithm 1a: Multiple Write Streaming
2. If the device supports self-checking, enable it
3. Host software decides test run time and runs test for that period of time (The software details of this are host platform specific, but will be compliant with the flows mentioned in Section 14.3.1 and follow configurations outlined in Section 14.11).
4. Setup Device for Algorithm 1b: Multiple Write Streaming with Bogus writes
5. If the device supports self-checking, enable it
6. Host software decides test run time and runs test for that period of time
7. Setup Device for Algorithm 2: Producer Consumer Test
8. Host software decides test run time and runs test for that period of time

Required Device Capability:

Hardware and configuration support for Algorithms 1a, 1b and 2 described in Section 14.3.1 and Section 14.11. If a device supports self-checking, it must escalate fatal

system error if Verify phase fails. Refer to Section 11.2 for specific error escalation mechanisms. Device is permitted to log failing address, iteration number and/or expected vs received data.

Pass Criteria:
No data corruptions or system errors reported

Fail Criteria:
Data corruptions or system errors reported

14.3.6.1.2 CXL.cache Coherency Test

This test sets up the device to execute Algorithm 1a, 1b and 2 in succession in order to stress data path for CXL.cache transactions. This test should only be run if the device supports CXL.cache or CXL.cache + CXL.mem protocols. Configuration details are determined by the host platform testing the device. Refer to Section 14.11 for the configuration registers and device capabilities. Each run includes execute/verify phases as described in section Section 14.3.1. ATS capabilities of the device can also be exercised in this test (see "AddressIsVirtual" field in Table 87).

Test Steps:

1. Host software will setup Device for Algorithm 1a: Multiple Write Streaming
2. If the device supports self-checking, enable it
3. Host software decides test run time and runs test for that period of time (The software details of this are host platform specific, but will be compliant with the flows mentioned in Section 14.3.1 and follow configurations outlined in Section 14.11)
4. Setup Device for Algorithm 1b: Multiple Write Streaming with Bogus writes
5. If the device supports self-checking, enable it
6. Host software decides test run time and runs test for that period of time.
7. Setup Device for Algorithm 2: Producer Consumer Test
8. Host software decides test run time and runs test for that period of time

Required Device Capability:
Hardware and configuration support for Algorithms 1a, 1b and 2 described in Section 14.3.1 and Section 14.11. If a device supports self-checking, it must escalate fatal system error if Verify phase fails. Refer to Section 11.2 for specific error escalation mechanisms. Device is permitted to log failing address, iteration number and/or expected vs received data.

Pass Criteria:
No data corruptions or system errors reported

Fail Criteria:
Data corruptions or system errors reported

14.3.6.1.3 CXL Test for Receiving Go_ERR

This test is only applicable for devices that support CXL.cache protocols. This test sets up the device to execute Algorithm 1a, while mapping one of the sets of the address to a memory range not accessible by the device. Test system software and configuration details are determined by the host platform and are system specific.

Test Steps:

1. Configure device for Algorithm 1a, and setup one of the base addresses to be an address not accessible by the device under test
2. Disable self-checking in the device under test
3. Host software decides test run time and runs test for that period of time

Required Device Capability:

Support for Algorithm 1a

Pass Criteria:

1. No data corruptions or system errors reported
2. No fatal device errors on receiving Go-ERR
3. Inaccessible memory range has not been modified by the device

Fail Criteria:

1. Data corruptions or system errors reported
2. Fatal device errors on receiving Go-ERR
3. Inaccessible memory range modified by the device (Host Error)

14.3.6.1.4 CXL.mem Test

This test sets up the **Host** to execute Algorithm 1a, 1b and 2 in succession in order to stress data path for CXL.mem transactions. Test system software and configuration details are determined by the host platform and are system specific.

Test Steps:

1. Map device attached memory to a test memory range accessible by the Host
2. Run equivalent of Algorithm 1a, 1b and 2 on the Host targeting device attached memory

Required Device Capability:

Support for CXL.mem protocol

Pass Criteria:

No data corruptions or system errors reported

Fail Criteria:

Data corruptions or system errors reported

14.4 ARB/MUX

14.4.1 Reset to Active Transition

The initial conditions for this test do not assume that the CXL link is up and device drivers have been loaded.

Test Steps:

1. With the link in Reset state, Link layer sends a Request to enter Active
2. ARB/MUX waits to receive indication of Active from Physical Layer

Pass criteria:

- ALMP Status sync exchange completes before ALMP Request{Active} sent by Local ARB/MUX
- Local ARB/MUX sends ALMP Request{Active} to the remote ARB/MUX
- Local ARB/MUX waits for ALMP Status{Active} and ALMP Request{Active} from remote ARB/MUX
- Local ARB/MUX sends ALMP Status{Active} in response to Request.
- Once ALMP handshake is complete, link transitions to Active
- Link successfully enters Active state with no errors

Fail criteria:

- Link hangs and does not enter Active state
- Any error occurs before transition to Active

14.4.2 ARB/MUX Multiplexing (Requires Protocol Analyzer)

Test Requirements:

Host generated traffic or device generated traffic and support for Algorithm 1a, 1b or 2

Test Steps:

1. Bring the link up into multi-protocol mode with CXL.io and CXL.cache and/or CXL.mem enabled
2. Ensure the arbitration weight is a non-zero value for both interfaces
3. Send continuous traffic on both CXL.io and CXL.cache and/or CXL.mem using Algorithm 1a, 1b or 2
4. Allow time for traffic transmission while snooping the bus

Pass criteria:

- Data from both CXL.io and CXL.cache and/or CXL.mem are sent across the link by the ARB/MUX

Fail criteria:

- Data on the link is only CXL.io
- Data on the link is only CXL.cache or CXL.mem (cache and mem share a single protocol ID, see Table 51)

14.4.3 Active to L1.x Transition (If Applicable)

Test Requirements:

Support for ASPM L1

Test Steps:

1. Force the remote and local link layer to send a request to the ARB/MUX for L1.x state

Pass criteria:

- UP ARB/MUX sends ALMP Request{L1.x}
- DP ARB/MUX sends ALMP Status{L1.x} in response

- Once ALMP Status is received by local ARB/MUX, L1.x is entered
- State transition doesn't occur until ALMP handshake is completed
- LogPHY enters L1 ONLY after both protocol enter L1 (applies to multi-protocol mode only)

Fail criteria:

- Error in ALMP handshake
- Protocol layer packets sent after ALMP L1.x handshake is complete (Requires Protocol Analyzer)
- State transition occurs before ALMP handshake completed

14.4.4 L1.x State Resolution (If Applicable)

Test Requirements:

Support for ASPM L1

Test Steps:

1. Force the remote and local link layer to send a request to the ARB/MUX for **different** L1.x states.

Pass criteria:

- UP ARB/MUX sends ALMP Request{L1.x} according to what the link layer requested
- DP ARB/MUX sends ALMP Status{L1.x} response. The state in the Status ALMP is the more shallow L1.x state.
- Once ALMP Status is received by local ARB/MUX, L1.x is entered
- State transition doesn't occur until ALMP handshake is completed
- LogPHY enters L1 ONLY after both protocol enter L1 (applies to multi-protocol mode only)

Fail criteria:

- Error in ALMP handshake
- Protocol layer packets sent after ALMP L1.x handshake is complete (Requires Protocol Analyzer)
- State transition occurs before ALMP handshake completed

14.4.5 Active to L2 Transition

Test Steps:

1. Force the remote and local link layer to send a request to the ARB/MUX for L2 state

Pass criteria:

- UP ARB/MUX sends ALMP Request{L2} to the remote vLSM
- DP ARB/MUX waits for ALMP Status{L2} from the remote vLSM
- Once ALMP Status is received by local ARB/MUX, L2 is entered
- If there are multiple link layers, repeat steps 1-4 for all link layers
- Physical link enters L2

- vLSM and physical link state transitions don't occur until ALMP handshake is completed

Fail criteria:

- Error in ALMP handshake
- Protocol layer packets sent after ALMP L1.x handshake is complete (Requires Protocol Analyzer)
- State transition occurs before ALMP handshake completed

14.4.6 L1 to Active Transition (If Applicable)

Test Requirements:

Support for ASPM L1

Test Steps:

1. Bring the link into L1 State
2. Force the link layer to send a request to the ARB/MUX to exit L1

Pass criteria:

- Local ARB/MUX sends Retrain notification to the Physical Layer
- Link exits L1
- Link enters L0 correctly
- Status synchronization handshake completes before request to enter L0

Fail criteria:

- State transition does not occur

14.4.7 Reset Entry

Test Steps:

1. Initiate warm reset flow

Pass criteria:

- Link sees hot reset and transitions to Detect state

Fail criteria:

- Link does not enter Detect

14.4.8 Entry into L0 Synchronization (Requires Protocol Analyzer)

Test Steps:

1. Put the link into Retrain state
2. After exit from Retrain, check Status ALMPs to synchronize interfaces across the link

Pass criteria:

- State contained in the Status ALMP is the same state the link was in before entry to retrain

Fail criteria:

- No Status ALMPs sent after exit from Retrain
- State in Status ALMPs different from the state that the link was in before the link went into Retrain
- Other communication occurred on the link after Retrain before the Status ALMP handshake for synchronization completed

14.4.9 ARB/MUX Tests Requiring Injection Capabilities

The tests in this section are optional but strongly recommended. The test configuration control registers for the tests in this section are implementation specific.

14.4.9.1 ARB/MUX Bypass (Requires Protocol Analyzer)

Test Requirements:

Device capability to force a request ALMP for any state

Test Steps:

1. Put the Link into PCIe mode or single protocol mode
2. Trigger entry to Retrain State
3. Snoop the bus and check for ALMPs

Pass criteria:

- No ALMPs generated by the ARB/MUX

Fail criteria:

- ALMP seen on the bus when checked

14.4.9.2 Repeated ALMP Request

Test Requirements:

Device capability to force a request ALMP for any state

Test Steps:

1. Wait for the Device ARB/MUX to transition to a state other than Active and send a Status ALMP indicating the state
2. Force the Device ARB/MUX to send a Request ALMP for the same state that it is currently in

Pass criteria:

- Error occurs and link enters retrain

Fail criteria:

- No Error, regular operation continues.

14.4.9.3 PM State Request Rejection (Requires Protocol Analyzer)

Test Requirements:

Host capability to put the host into a state where it will reject any PM request ALMP

Test Steps:

1. Device sends PM state Request ALMP
2. Wait for an ALMP Request for entry to a PM State
3. Host rejects the request by not responding to the Request ALMP

Pass criteria:

- Device continues operation despite no Status received and initiates an Active Request

Fail criteria:

- Any system error

14.4.9.4 Unexpected Status ALMP

Test Requirements:

Device Capability to force the ARB/MUX to send a Status ALMP at any time

Test Steps:

1. While link is in Active, force the ARB/MUX to send a Status ALMP without first receiving a Request ALMP

Pass criteria:

- Link enters Retrain without any errors reported

Fail criteria:

- No error on the link and normal operation continues OR
- System errors are observed

14.4.9.5 ALMP Error

Test Requirements:

Device capability that allows the device to inject errors into a flit

Test Steps:

1. Inject a single bit error into the lower 16 bytes of a 528-bit flit
2. Send data across the link
3. ARB/MUX detects error and enters Retrain
4. Repeat Steps 1-3 with a double bit error

Pass criteria:

- Error is logged
- Link enters retrain

Fail criteria:

- No error detected

14.4.9.6 Recovery Re-entry

Test Requirements:

Device capability that allows the device to ignore ALMP State Requests

Test Steps:

1. Place the link into Active state
2. Request link to go to Retrain State
3. Prevent the Local ARB/MUX from entering Retrain
4. Remote ARB/MUX enters Retrain state
5. Remote ARB/MUX exits Retrain state and sends ALMP Status{Active} to synchronize
6. Local ARB/MUX receives Status ALMP for synchronization but does not send
7. Local ARB/MUX triggers re-entry to Retrain

Pass criteria:

- Link successfully enters Retrain on re-entry attempt

Fail criteria:

- Link continues operation without proper synchronization

14.5 Physical Layer

14.5.1 Protocol ID Checks (Requires Protocol Analyzer)

Test Steps:

1. Bring the link up to the Active state
2. Send one or more flits from the CXL.io interface, check for correct Protocol ID
3. If applicable, send one or more flits from the CXL.cache and/or CXL.mem interface, check for correct Protocol ID
4. Send one or more flits from the ARB/MUX, check for correct Protocol ID

Pass criteria:

- All Protocol IDs are correct

Fail criteria:

- Errors during test
- No communication

14.5.2 NULL Flit (Requires Protocol Analyzer)

Test Steps:

1. Bring the link up to the Active state
2. Delay flits from the Link Layer
3. Check for NULL flits from the Physical Layer
4. Check that NULL flits have correct Protocol ID

Pass criteria:

- NULL flits seen on the bus when Link Layer delayed
- NULL flits have correct Protocol ID
- NULL flits contain all zero data

Fail criteria:

- No NULL flits sent from Physical Layer
- Errors logged during tests

14.5.3 EDS Token (Requires Protocol Analyzer)

Test Steps:

1. Bring the link up to the Active state
2. Send a flit with an implied EDS token, check the following:

Pass criteria:

- EDS token is the last flit in the data block
- EDS token does not cross the data block boundary
- Next block after EDS token is an ordered set
- EDS token is the last flit in the data block and does not cross block boundary
- OS block follows EDS token

Fail criteria:

- Errors logged during test

14.5.4 Correctable Framing Error

This test is optional but strongly recommended.

Test Requirements:

Protocol ID error perception in the Device Log PHY (Device can forcibly react as though there was an error even if the protocol ID is correct)

Test Steps:

1. Bring the link up to the Active state
2. Create a correctable framing error by injecting an error into one 8-bit encoding group of the Protocol ID
3. Check that an error was logged and normal processing continues

Pass criteria:

- Error correctly logged
- Correct 8-bit encoding group used for normal operation

Fail criteria:

- No error logged
- Flit with error dropped
- Error causes retrain

- Normal operation does not resume after error

14.5.5 Uncorrectable Framing Error

This test is optional but strongly recommended.

Test Requirements:

Protocol ID error perception in the Device Log PHY (Device can forcibly react as though there was an error even if the protocol ID is correct)

Test Steps:

1. Bring the link up to the Active state
2. Create a uncorrectable framing error by injecting an error into both 8-bit encoding groups of the Protocol ID
3. Check that an error was logged and flit is dropped
4. Link goes into Retrain

Pass criteria:

- Error correctly logged
- Link enters Retrain

Fail criteria:

- No error log
- Error corrected

14.5.6 Unexpected Protocol ID

This test is optional but strongly recommended.

Test Requirements:

Protocol ID error perception in the Device Log PHY (Device can forcibly react as though there was an error even if the protocol ID is correct)

Test Steps:

1. Bring the link up to the Active state
2. Send a flit with an invalid protocol ID
3. Check that an error is logged and the flit is dropped

Pass criteria:

- Error logged
- Flit is dropped

Fail criteria:

- No Error logged
- Flit is processed normally

14.5.7 Sync Header Bypass (Requires Protocol Analyzer) (If Applicable)

Test Requirements:

Support for Sync Header Bypass

Test Steps:

1. Negotiate for sync header bypass during PCIe alternate mode negotiation
2. Link trains to 2.5GT/s speed
3. Transition to 8GT/s speed
4. Check for Sync Headers

Pass criteria:

- No Sync Headers observed after 8GT/s transition

Fail criteria:

- Link training not complete
- Sync headers at 8GT/s speed

14.5.8 Link Speed Advertisement (Requires Protocol Analyzer)

Test Steps:

1. Enter CXL link training at 2.5GT/s
2. Check speed advertisement before multi-protocol negotiations have completed

Pass criteria:

- CXL speed advertisement contains 32, 16 and 8GT/s speeds regardless of capabilities

Fail criteria:

- Speed advertisement does not contain all 3 speeds

14.5.9 Idle Transition to L0 (Requires Protocol Analyzer)

Test Steps:

1. Bring the link up in CXL mode to the Config.Idle or Recovery.Idle state
2. Wait for NULL flit to be received by DUT
3. Check that DUT sends NULL flits after receiving NULL flits

Pass criteria:

- LTSSM transitions to L0 after 8 NULL flits are sent and at least 4 NULL flits are received

Fail criteria:

- LTSSM stays in IDLE
- LTSSM transitions before the exchange of NULL flits is completed

14.5.10 Drift Buffer (If Applicable)

Test Requirements:

Support Drift Buffer

Test Steps:

1. Enable the Drift buffer

Pass criteria:

- Drift buffer is logged in the Flex Bus DVSEC

Fail criteria:

- No log in the Flex Bus DVSEC

14.5.11 SKP OS Scheduling / Alternation (Requires Protocol Analyzer) (If Applicable)

Test Requirements:

Support Sync Header Bypass

Test Steps:

1. Bring the link up in CXL mode with sync header bypass enabled
2. Check for SKP OS

Pass criteria:

- Physical Layer schedules SKP OS every 340 data blocks
- Control SKP OS and regular SKP OS alternate at 16GT/s or higher speed
- Regular SKP OS used only at 8GT/s

Fail criteria:

- No SKP OS observed
- SKP OS observed at interval other than 340 data blocks

14.5.12 SKP OS Exiting the Data Stream (Requires Protocol Analyzer) (If Applicable)

Test Requirements:

Support Sync Header Bypass

Test Steps:

1. Bring the link up in CXL mode with sync header bypass enabled
2. Exit Active mode

Pass criteria:

- Physical Layer replaces SKP OS with EIOS or EIEOS

Fail criteria:

- SKP OS not replaced by Physical Layer

14.5.13 Link Speed Degradation - CXL Mode

Test Steps:

1. Train the CXL link up to the highest speed possible (At least 8GT/s)
2. Degrade the link down to a lower CXL mode speed

Pass criteria:

- Link degrades to slower speed without going through mode negotiation

Fail criteria:

- Link leaves CXL mode

14.5.14 Link Speed Degradation Below 8GT/s

Test Steps:

1. Train the CXL link up to the highest speed possible (At least 8GT/s)
2. Degrade the link down to a speed below CXL mode operation
3. Link goes to detect state

Pass criteria:

- Link degrades to slower speed
- Link enter Detect

Fail criteria:

- Link stays in CXL mode
- Link does not change speed

14.5.15 Tests Requiring Injection Capabilities

The tests in this section are optional but strongly recommended. The test configuration control registers for the tests in this section are implementation specific.

14.5.15.1 TLP Ends On Flit Boundary (Requires Protocol Analyzer)

Test Steps:

1. Bring the link up to the Active state
2. CXL.io sends a TLP that ends on a flit boundary
3. Check that next flit sent by link layer contains IDLE tokens, EDB or more data

Pass criteria:

- TLP that ends on flit boundary not processed until subsequent flit is transmitted
- IDLE tokens, EDB or more data observed after TLP that ends on flit boundary

Fail criteria:

- Errors logged
- No IDLE, EDB or data observed after TLP flit

14.5.15.2 Failed CXL Mode Link Up

Test Steps:

1. Negotiate for CXL during PCIe alternate mode negotiation
2. Hold the link at 2.5GT/s
3. Link transitions back to detect

Pass Criteria:

- Link transitions back to detect after not able to reach 8GT/s speed
- Link training does not complete

Fail criteria:

- Link does not transition to detect

14.6 Configuration Register Tests

Configuration space register cover the registers defined in Chapter 7.0, "Control and Status Registers". These tests are run on the device under test, and require no additional hardware to complete. Tests must be run with Root/Administrator privileges. Test makes the assumption that there is one and only one CXL device in the system, and it is the DUT. This test Section has granularity down to the CXL Device.

14.6.1 Device Presence.

Test Steps:

1. Read the PCI Device hierarchy and filter for RCiEP devices.
2. Locate RCiEP Device with VID of 8086 and type of 0.
3. Save this RCiEP Device location for subsequent tests. This will be referred to in subsequent tests as DUT

Pass criteria:

- One RCiEP device found.

Fail criteria:

- NO RCiEP Devices found
- More than 1 RCiEP Device Found

14.6.2 Flex Bus Device DVSEC Capability Header

Test Steps:

1. Read the Configuration space for **DUT**. Offset 0x04, Length 4 bytes.
2. Decode this into:

Bits	Variable
15:0	VID
19:16	REV
31:20	LEN

3. Verify:

Variable		Value	Condition
VID	=	0x8086	Always
REV	=	0	Always
LEN	=	0x38	Always

4. Read the Configuration space for **DUT**, Offset 0x08, Length 2 bytes,
5. Decode this into:

Bits	Variable
15:0	ID

6. Verify:

Variable		Value	Condition
ID	=	0	Always

Pass criteria:

- Test 14.6.1 Passed
- Verify Conditions met

Fail criteria:

- Verify Conditions Failed

14.6.3 DVSEC Capability Structure

Test Steps:

1. Read the Configuration space for DUT, Offset 0x0A, Length 2.
2. Decode this into:

Bits	Variable
0:0	Cache_Capable
1:1	IO_Capable
2:2	Mem_Capable
3:3	Mem_HW_Init_Mode
5:4	HDM_Count
14:14	Viral Capable

3. Verify:

Variable		Value	Condition
IO_Capable	=	b1	Always
HDM_Count	!=	b11	Always
HDM_Count	!=	b00	Mem_Capable = 1
HDM_Count	=	b00	Mem_Capable = 0

Pass criteria:

- Test 14.6.2 Passed
- Verify Conditions Met

Fail criteria:

- Verify Conditions Failed

14.6.4 DVSEC Control Structure

Test Steps:

1. Read the Configuration space for DUT, Offset 0x0C, Length 2.
2. Decode this into:

Bits	Variable
0:0	Cache_Enable
1:1	IO_Enable
2:2	Mem_Enable
7:3	Cache_SF_Coverage
10:8	Cache_SF_Granularity
11:11	Cache_Clean_Eviction
14:14	Viral_Enable

3. Verify:

Variable		Value	Condition
Cache_SF_Granularity	!=	b111	Always

Pass criteria:

- Test 14.6.2 Passed
- Verify Conditions Met

Fail criteria:

- Verify Conditions Failed

14.6.5 DVSEC Control Lock

Test Steps:

1. Read Configuration Space for DUT, Offset 0x14, length 2
2. Decode this into:

Bits	Variable
0:0	CONFIG_LOCK

3. Read Configuration Space for DUT, Offset 0x0C, Length 2
4. Store this into Variable **R1**
5. Invert **R1** and store in **W1** masking RSVD Fields 13:12 and 15:15 from inversion
6. Write Configurations Space for DUT, Offset 0x0C, length 2 with variable **W1**
7. Read Configurations Space for DUT, Offset 0x0C, Length 2
8. Store this into Variable **R2**

9. Verify:

Variable		Value	Condition
R1	=	R2	CONFIG_LOCK = 1
R1	!=	R2	CONFIG_LOCK = 0

Pass criteria:

- Test 14.6.2 Passed
- Verify Conditions Met

Fail criteria:

- Verify Conditions Failed

14.7 Memory Device Tests

This section covers tests applicable to devices supporting CXL.mem protocol

14.7.1 Flex Bus Range 1

Necessary Conditions:

- Device is CXL.mem capable

Inputs:

Type Volatile or Non-Volatile

Class Memory or Storage

Test Steps:

1. Read Configuration Space for DUT, Offset 0x1C Length 4
2. Decode this into:

Bits	Variable
0:0	Memory_Info_Valid
1:1	Memory_Active
4:2	Media_Type
7:5	Memory_Class
10:8	Desired_Interleave
31:20	Memory_Size_Low

3. Verify:

Variable		Value	Condition
Media_Type	=	b000 or b001	
Media_Type	=	b000	Type = Volatile
Media_Type	=	b001	Type = Non-Volatile
Memory_Class	=	b000 or b001	
Memory_Class	=	b000	Class = Memory
Memory_Class	=	b001	Class = Storage

Desired_Interleave	=	b00 or b01 or b10

Pass criteria:

- Test 14.6.2 Passed
- Verify Conditions Met

Fail criteria:

- Verify Conditions Failed

14.7.2 Flex Bus Range 2

Necessary Conditions:

Device is CXL.mem capable

HDM_Count = b10

Inputs:

Type Volatile or Non-Volatile

Class Memory or Storage

Test Steps:

1. Read Configuration Space for DUT, Offset 0x2C Length 4
2. Decode this into:

Bits	Variable
0:0	Memory_Info_Valid
1:1	Memory_Active
4:2	Media_Type
7:5	Memory_Class
10:8	Desired_Interleave
31:20	Memory_Size_Low

3. Verify:

Variable		Value	Condition
Media_Type	=	b000 or b001	
Media_Type	=	b000	Type = Volatile
Media_Type	=	b001	Type = Non-Volatile
Memory_Class	=	b000 or b001	
Memory_Class	=	b000	Class = Memory
Memory_Class	=	b001	Class = Storage
Desired_Interleave	=	b00 or b01 or b10	

Pass criteria:

- Test 14.6.2 Passed
- Verify Conditions Met

Fail criteria:

- Verify Conditions Failed

14.8 Memory Mapped Registers

14.8.1 RCRB MEMBAR0 location

Test Steps:

1. Read Downstream port MEMBAR0 address Store in **MB0D**
2. Read Upstream port MEMBAR0 address store in **MB0U**
3. Determine end of RCRB Upstream region and store in **RCRB**
4. Verify:
   — **MB0D** and **RCRB** memory regions do not overlap
   — **MB0U** and **RCRB** memory regions do not overlap
   — **MB0U** and **MB0D** form a continuous 8k memory region
   — **MB0U** and **MB0D** set to valid address in MMIO region

Pass criteria:

- Verify Conditions Met

Fail criteria:

- Verify Conditions Failed

14.9 Reset and Initialization Tests

14.9.1 Warm Reset Test

DUT must be in D3 state with context flushed

Test Steps:

1. Host issues CXL PM VDM, Reset Prep (ResetType= Warm Reset; PrepType=General Prep)
2. Host waits for CXL device to respond with CXL PM VDM ResetPrepAck

Pass criteria:

- DUT responds with an ACK

Fail criteria:

- DUT fails to respond to ACK

14.9.2 Cold Reset Test

DUT must be in D3 state with context flushed

Test Steps:

1. Host issues CXL PM VDM, Reset Prep (ResetType= Warm Reset; PrepType=General Prep)
2. Host waits for CXL device to respond with CXL PM VDM ResetPrepAck

Pass criteria:

- DUT responds with an ACK

Fail criteria:

- DUT fails to respond to ACK

14.9.3 Sleep State Test

DUT must be in D3 state with context flushed

Test Steps:

1. Host issues CXL PM VDM, Reset Prep (ResetType= S3; PrepType=General Prep)
2. Host waits for CXL device to respond with CXL PM VDM ResetPrepAck

Pass criteria:

- DUT responds with an ACK

Fail criteria:

- DUT fails to respond to ACK

14.9.4 Function Level Reset Test

Necessary Conditions:

- Device supports Function Level Reset.

Function Level Reset has the requirement that the CXL device maintain Cache Coherency. This test is accomplished by running the Application Layer tests as described in Section 14.3.6.1, and issuing a Function level reset in the middle of it.

Required Device Capability

Hardware configuration support for Algorithm 1a described in Section 14.3.1. If the device supports self-checking it must escalate a fatal system error. Device is permitted to log failing information.

Test Steps:

1. Determine test run time T based on the amount of time available or allocated for this testing.
2. Host software sets up Cache Coherency test for Algorithm 1a: Multiple Write Streaming
3. If the devices supports self-checking, enable it.
4. At a time between 1/3 and 2/3 of T and with at least 200 ms of test time remaining, Host initiates Host initiates FLR by writing to the Initiate Function Level Reset bit.

Pass criteria:

- System does not elevate a fatal system error, and no errors are logged

Fail Criteria:

- System error reported, Logged failures exist.

14.9.5 Flex Bus Range Setup Time

Necessary Conditions:

- Device is CXL.mem capable
- Ability to monitor the device reset

Test Steps:

1. Reset the system, Monitor Reset until clear
2. Wait for 1 second
3. Read Configuration Space for DUT, Offset 0x1C Length 4
4. Decode this into:

Bits	Variable
0:0	Memory_Info_Valid
1:1	Memory_Active

5. Verify:

Variable		Value	Condition
Memory_Info_Valid	=	1	
Memory_Active	=	1	**Mem_HW_Init_Mode = 1**

Pass criteria:

- Test 14.6.2 Passed
- Verify Conditions Met

Fail criteria:

- Verify Conditions Failed

14.9.6 FLR Memory

This test ensures that FLR does not affect data in device attached memory.

Necessary Conditions:

- Device is CXL.mem capable

Test Steps:

1. Write a known pattern to a known location within HDM
2. Host performs a FLR as defined in steps of Section 14.9.4.
3. Host Reads HDM memory location
4. Verify: that read data matches previously written data.

Pass criteria:

- HDM retains information following FLR

Fail criteria:

- HDM memory is reset.

14.10 Reliability, Availability, and Serviceability

RAS Testing is dependent on being able to inject and correctly detect the injected errors. For this testing in is required that the host and device support error injection capabilities.

Certain Device/Host capabilities of error injection are required to enable the RAS tests. First, the required capabilities and configurations are provided. Then, the actual test procedures are laid out. Since these capabilities may only be firmware accessible, currently these are implementation specific. However, future revisions of this specification may define these under an additional capability structure.

The following register describes the required functionalities.

Table 72. Register 1: CXL.cache/CXL.mem LinkLayerErrorInjection

Bit	Attribute	Description
0	RWL	**CachePoisonInjectionStart**: Software writes 0x1 to this bit to trigger a single poison injection on a CXL.cache message in the Tx direction. Hardware must override the poison field in the data header slot of the corresponding message (D2H if device, H2D if Host). This bit is required only if CXL.cache protocol is supported.
1	RO-V	**CachePoisonInjectionBusy**: Hardware loads 1'b1 to this bit when the Start bit is written. Hardware must clear this bit to indicate that it has indeed finished poisoning a packet. Software is permitted to poll on this bit to find out when hardware has finished poison injection. This bit is required only if CXL.cache protocol is supported.
2	RWL	**MemPoisonInjectionStart**: Software writes 0x1 to this bit to trigger a single poison injection on a CXL.mem message in the Tx direction. Hardware must override the poison field in the data header slot of the corresponding. This bit is required only if CXL.mem protocol is supported.
3	RO-V	**MemPoisonInjectionBusy**: Hardware loads 1'b1 to this bit when the Start bit is written. Hardware must clear this bit to indicate that it has indeed finished poisoning a packet. Software is permitted to poll on this bit to find out when hardware has finished poison injection. This bit is required only if CXL.mem protocol is supported.
4	RWL	**IOPoisonInjectionStart**: Software writes 0x1 to this bit to trigger a single poison injection on a CXL.io message in the Tx direction. Hardware must override the poison field in the data header slot of the corresponding message.
5	RO-V	**IOPoisonInjectionBusy**: Hardware loads 1'b1 to this bit when the Start bit is written. Hardware must clear this bit to indicate that it has indeed finished poisoning a packet. Software is permitted to poll on this bit to find out when hardware has finished poison injection.

Table 72. Register 1: CXL.cache/CXL.mem LinkLayerErrorInjection

7:6	RWL	**CacheMemCRCInjection**: Software writes to these bits to trigger CRC error injections. The number of CRC bits flipped is given as follows: 2'b00 – Disable. No CRC errors are injected 2'b01 – Single bit flipped in the CRC field for "n" subsequent Tx flits, where n is the value in CacheMemCRCInjectionCount. 2'b10 – 2 bits flipped in the CRC field for "n" subsequent Tx flits, where n is the value in CacheMemCRCInjectionCount. 2'b11 – 3 bits flipped in the CRC field for "n" subsequent Tx flits, where n is the value in CacheMemCRCInjectionCount. The specific bit positions that are flipped are implementation specific. This field is required if any of CXL.cache or CXL.mem protocols are supported.
9:8	RWL	**CacheMemCRCInjectionCount**: Software writes to these bits to program the number of CRC injections. This field must be programmed by software before OR at the same time as CacheMemCRCInjection field. The number of flits where CRC bits are flipped is given as follows: 2'b00 – Disable. No CRC errors are injected 2'b01 – CRC injection is only for 1 flit. CacheMemCRCInjectionBusy bit is cleared after 1 injection. 2'b10 – CRC injection is for 2 flits in succession. CacheMemCRCInjectionBusy bit is cleared after 2 injections. 2'b11 – CRC injection is for 3 flits in succession. CacheMemCRCInjectionBusy bit is cleared after 3 injections. This field is required if any of CXL.cache or CXL.mem protocols are supported.
10	RO-V	**CacheMemCRCInjectionBusy**: Hardware loads 1'b1 to this bit when the Start bit is written. Hardware must clear this bit to indicate that it has indeed finished CRC injections. Software is permitted to poll on this bit to find out when hardware has finished CRC injection. This bit is required if any of CXL.cache or CXL.mem protocols are supported.

Table 73. Register 2: CXL.io LinkLayer Error injection

Bit	Attribute	Description
0	RWL	**IOPoisonInjectionStart**: Software writes 0x1 to this bit to trigger a single poison injection on a CXL.io message in the Tx direction. Hardware must override the poison field in the data header slot of the corresponding message.
1	RO-V	**IOPoisonInjectionBusy**: Hardware loads 1'b1 to this bit when the Start bit is written. Hardware must clear this bit to indicate that it has indeed finished poisoning a packet. Software is permitted to poll on this bit to find out when hardware has finished poison injection.
2	RWL	**FlowControErrorInjection:** Software writes 0x1 to this bit to trigger a Flow Control error on CXL.io only. Hardware must override the Flow Control DLLP.
3	RO-V	**FlowControlInjectionBusy:** Hardware loads 1'b1 to this bit when the Start bit is written. Hardware must clear this bit to indicate that it has indeed finished Flow Control error injections. Software is permitted to poll on this bit to find out when hardware has finished Flow Control error injection.

Table 74. Register 3: Flex Bus LogPHY Error injections

Bit	Attribute	Description
0	RWL	**CorrectableProtocolIDErrorInjection**: Software writes 0x1 to this bit to trigger a correctable protocol ID error on any CXL flit issued by the FlexBus LogPHY. Hardware must override the Protocol ID field in the flit.
1	RWL	**UncorrectableProtocolIDErrorInjection**: Software writes 0x1 to this bit to trigger an uncorrectable protocol ID error on any CXL flit issued by the FlexBus LogPHY. Hardware must override the Protocol ID field in the flit.
2	RWL	**UnexpectedProtocolIDErrorInjection**: Software writes 0x1 to this bit to trigger an unexpected protocol ID error on any CXL flit issued by the FlexBus LogPHY. Hardware must override the Protocol ID field in the flit.
3	RO-V	**ProtocolIDInjectionBusy:** Hardware loads 1'b1 to this bit when the Start bit is written. Hardware must clear this bit to indicate that it has indeed finished Protocol ID error injections. Software is permitted to poll on this bit to find out when hardware has finished Protocol ID error injection. Software should only program one of the bits between correctable, uncorrectable and unexpected protocol ID error injection bits.

14.10.1 RAS Configuration

14.10.1.1 AER Support

CXL spec calls out for errors to be reported via PCI AER mechanism. AER is listed as an optional Extended Capability.

Test Steps:

1. Read through each Extended Capability (EC) Structure for the RCiRP, and locate EC structure for type.

Pass criteria:

- AER Extended Capability Structure exists.

Fail criteria:

- AER Extended Capability Structure does not exist.

14.10.1.2 CXL.io Poison Injection from Device to Host

Test Steps:

1. Write a pre-determined pattern to Cache line aligned Address A1 (example pattern – all 1s – {64{8'hFF}}).

2. Setup CXL.io device for Algorithm 1a (multiple write streaming) with the following parameters
   a. StartAddress1::StartAddress1 = A1
   b. WriteBackAddress1::WriteBackAddress1 = A2 (separate location from A1)
   c. AddressIncrement::AddressIncrement = 0x0
   d. Pattern1::Pattern1 = 0xAA [this can be any pattern that is different from the values programmed in step 1]
   e. ByteMask::ByteMask = 0xFFFFFFFFFFFFFFFF (write to all bytes)
   f. ByteMask::PatternSize = 0x1 (use only 1 byte of Pattern1)
   g. AlgorithmConfiguration::SelfChecking = 0x0
   h. AlgorithmConfiguration::NumberOfAddrIncrements = 0x0
   i. AlgorithmConfiguration::NumberOfSets = 0x0
   j. AlgorithmConfiguration::NumberOfLoops = 0x1
   k. AlgorithmConfiguration::AddressIsVirtual = 0x0 (use physical address for this test)
   l. AlgorithmConfiguration::Protocol = 0x1
3. Setup Poison Injection from CXL.io device
   a. LinkLayerErrorInjection::IOPoisonInjectionStart = 0x1
4. Start the Algorithm. AlgorithmConfiguration::Algorithm = 0x1

Required Device Capabilities:

- The CXL device must support Algorithm 1a, and Link Layer Error Injection capabilities for CXL.io.

Pass Criteria:

- Receiver logs poisoned received error.
- Test software is permitted to read address A1 to observe written pattern.

Fail Criteria:

- Receiver does not log poison received error.

14.10.1.3 CXL.cache Poison Injection

Device to Host Poison Injection

Test Steps:

1. Write a pre-determined pattern to Cache line aligned Address A1 (example pattern – all 1s – {64{8'hFF}}). A1 should belong to Host attached memory.
2. Setup CXL.cache device for Algorithm 1a (multiple write streaming) with the following parameters
   a. StartAddress1::StartAddress1 = A1
   b. WriteBackAddress1::WriteBackAddress1 = A2 (separate location from A1)
   c. AddressIncrement::AddressIncrement = 0x0
   d. Pattern1::Pattern1 = 0xAA [this can be any pattern that is different from the values programmed in step 1]
   e. ByteMask::ByteMask = 0xFFFFFFFFFFFFFFFF (write to all bytes)
   f. ByteMask::PatternSize = 0x1 (use only 1 byte of Pattern1)

g. AlgorithmConfiguration::SelfChecking = 0x0
h. AlgorithmConfiguration::NumberOfAddrIncrements = 0x0
i. AlgorithmConfiguration::NumberOfSets = 0x0
j. AlgorithmConfiguration::NumberOfLoops = 0x1
k. AlgorithmConfiguration::AddressIsVirtual = 0x0 (use physical address for this test)
l. AlgorithmConfiguration::WriteSemanticsCache = 0x7
m. AlgorithmConfiguration::ExecuteReadSemanticsCache = 0x4
n. AlgorithmConfiguration::Protocol = 0x1

3. Setup Poison Injection from CXL.cache device
   a. LinkLayerErrorInjection::CachePoisonInjectionStart = 0x1
4. Start the Algorithm. AlgorithmConfiguration::Algorithm = 0x1

Required Device Capabilities:

- The CXL device must support Algorithm 1a, and Link Layer Error Injection capabilities for CXL.Cache

Pass Criteria:

- Receiver (Host) logs poisoned received error.
- Test software is permitted to read address A1 to observe written pattern

Fail Criteria:

- Receiver does not log poison received error.

Host to Device Poison Injection

This test aims to ensure that if a CXL.cache device receives poison for data received from the Host, it returns the poison indication in the write-back phase. Receiver on the CXL device must also log and escalate poison received error.

Test Steps:

1. Write a pre-determined pattern to Cache line aligned Address A1 (example pattern – all 1s – {64{8'hFF}}). A1 should belong to Host attached memory.
2. Setup CXL.Cache device for Algorithm 1a with the following parameters
   a. StartAddress1::StartAddress1 = A1 [A1 should map to host attached memory]
   b. WriteBackAddress1::WriteBackAddress1 = A2 (separate location from A1)
   c. AddressIncrement::AddressIncrement = 0x0
   d. Pattern1::Pattern1 = 0xAA [this can be any pattern that is different from the values programmed in step 1]
   e. ByteMask::ByteMask = 0x1 (write to single byte, so that device has to read)
   f. ByteMask::PatternSize = 0x1 (use only 1 byte of Pattern1)
   g. AlgorithmConfiguration::SelfChecking = 0x0
   h. AlgorithmConfiguration::NumberOfAddrIncrements = 0x0
   i. AlgorithmConfiguration::NumberOfSets = 0x0
   j. AlgorithmConfiguration::NumberOfLoops = 0x1

k. AlgorithmConfiguration::AddressIsVirtual = 0x0 (use physical address for this test)

l. AlgorithmConfiguration::WriteSemanticsCache = 0x2 (use DirtyEvict)

m. AlgorithmConfiguration::ExecuteReadSemanticsCache = 0x0 (use RdOwn, so device reads from host)

n. AlgorithmConfiguration::Protocol = 0x1

3. Setup Poison injection on the **Host** CXL.cache Link Layer (through Link Layer Error Injection register)
4. AlgorithmConfiguration::Algorithm = 0x1 (start the test)
5. Read Address A1 from the Host and check if it matches the pattern {64{8'hFF}} or {63{8'hFF},8'hAA}

Required Device Capabilities:

- The CXL device must support Algorithm 1a with DirtyEvict and RdOwn semantics

Pass Criteria:

- Receiver (Device) logs poisoned received error.
- Test software is permitted to read address A1 to observe written pattern

Fail Criteria:

- Receiver does not log poison received error.

14.10.1.4 CXL.cache CRC Injection (Protocol Analyzer Required)

Device to Host CRC injection

Test Steps:

1. Setup is same as Test 14.3.6.1.2.
2. While test is running, software will periodically perform the following steps to Device registers

   a. Write LinkLayerErrorInjection::CacheMemCRCInjectionCount = 0x3

   b. Write LinkLayerErrorInjection::CacheMemCRCInjection = 0x2

   c. Poll on LinkLayerErrorInjection::CacheMemCRCInjectionBusy

   — If 0, Write LinkLayerErrorInjection::CacheMemCRCInjection = 0x0

   — Write LinkLayerErrorInjection::CacheMemCRCInjection = 0x2

   — Return to (c) to Poll

Required Device Capabilities:

- The CXL device must support Algorithm 1a, and Link Layer Error Injection capabilities for CXL.Cache

Pass Criteria:

- Same as Test 14.3.6.1.2
- Monitor and Verify that CRC errors are injected (using the Protocol Analyzer), and that Retries are triggered as a result.

Fail Criteria:

- Same as Test 14.3.6.1.2

Host to Device CRC injection

Test Steps:

1. Setup is same as Test 14.3.6.1.2.
2. While test is running, software will periodically perform the following steps to **Host** registers
   a. Write LinkLayerErrorInjection::CacheMemCRCInjectionCount = 0x3
   b. Write LinkLayerErrorInjection::CacheMemCRCInjection = 0x2
   c. Poll on LinkLayerErrorInjection::CacheMemCRCInjectionBusy
   — If 0, Write LinkLayerErrorInjection::CacheMemCRCInjection = 0x0
   — Write LinkLayerErrorInjection::CacheMemCRCInjection = 0x2
   — Return to (c)

Required Device Capabilities:

- The CXL device must support Algorithm 1a

Pass Criteria:

- Same as Test 14.3.6.1.2
- Monitor and Verify that CRC errors are injected (using the Protocol Analyzer), and that Retries are triggered as a result.

Fail Criteria:

- Same as Test 14.3.6.1.2

14.10.1.5 CXL.mem Poison Injection

This test is only applicable if a device supports CXL.mem

Host to Device Poison Injection

Test Steps:

1. Write {64{8'hFF}} to address B1 from Host. B1 must belong to Device Attached memory.
2. Setup **Host** Link Layer for poison injection
   a. LinkLayerErrorInjection::MemPoisonInjectionStart = 0x1
3. Write {64{8'hAA}} to address B1 from Host

Required Device Capabilities:

- Device should be CXL.mem capable

Pass Criteria:

- Receiver (Device) logs poisoned received error.
- Test software is permitted to read address B1 to observe written pattern

Fail Criteria:

- Receiver does not log poison received error.

14.10.1.6 CXL.mem CRC Injection (Protocol Analyzer Required)

Host to Device CRC injection

Test Steps:

1. Write {64{8'hFF}} to address B1 from Host (B1 must belong to Device Attached memory)
2. Setup **Host** Link Layer for CRC injection
   a. Write LinkLayerErrorInjection::CacheMemCRCInjectionCount = 0x1
   b. Write LinkLayerErrorInjection::CacheMemCRCInjection = 0x2
3. Write {64{8'hAA}} to address B1 from Host
4. Read address B1 from Host, and compare to {64{8'hAA}}

Required Device Capabilities:

- Device should support CXL.mem

Pass Criteria:

- Read data == {64{8'hAA}}
- CRC error and Retry observed on Link (Protocol Analyzer used for observation)

Fail Criteria:

- Read data != {64{8'hAA}}

14.10.1.7 Flow Control Injection

This is an optional but strongly recommended test only applicable for CXL.io

Device to Host Flow Control injection

Test Steps:

1. Setup is same as Test 14.3.6.1.1.
2. While test is running, software will periodically perform the following steps to Device registers
   a. Write LinkLayerErrorInjection::FlowControlInjection = 0x1
   b. Poll on LinkLayerErrorInjection::FlowControlInjectionBusy
      — If 0, Write LinkLayerErrorInjection::FlowControlInjection = 0x0
      — Write LinkLayerErrorInjection::FlowControlInjection = 0x2
      — Return to (c) to Poll

Required Device Capabilities:

- The CXL device must support Algorithm 1a, and Link Layer Error Injection capabilities

Pass Criteria:

- Same as Test 14.3.6.1.1

Fail Criteria:

- Same as Test 14.3.6.1.1

Host to Device Flow Control injection

Test Steps:

1. Setup is same as Test 14.3.6.1.1.
2. While test is running, software will periodically perform the following steps to Host registers
   a. Write LinkLayerErrorInjection::FlowControlInjection = 0x1
   b. Poll on LinkLayerErrorInjection::FlowControlInjectionBusy
   — If 0, Write LinkLayerErrorInjection::FlowControlInjection = 0x0
   — Write LinkLayerErrorInjection::FlowControlInjection = 0x2
   — Return to (c) to Poll

Required Device Capabilities:

- The CXL device must support Algorithm 1a

Pass Criteria:

- Same as Test 14.3.6.1.1

Fail Criteria:

- Same as Test 14.3.6.1.1

14.10.1.8 Unexpected Completion Injection

This is an optional but strongly recommended test that is only applicable for CXL.io

Device to Host Unexpected Completion injection

Test Steps:

1. Setup is same as Test 14.3.6.1.1, except that Self-checking should be disabled.
2. While test is running, software will periodically perform the following steps to Device registers
   a. Write DeviceErrorInjection::UnexpectedCompletionInjection = 0x1

Required Device Capabilities:

- The CXL device must support Algorithm 1a, and Device Error Injection capabilities

Pass Criteria:

- Unexpected completion error logged

Fail Criteria:

- No errors logged

14.10.1.9 Completion Timeout

This is an optional but strongly recommended test. It is only applicable for CXL.io

Device to Host Completion Timeout

Test Steps:

1. Setup is same as Test 14.3.6.1.1.

2. While test is running, perform the following to Device registers
   a. Write DeviceErrorInjection::CompleterTimeoutInjection = 0x1

Required Device Capabilities:

- The CXL device must support Algorithm 1a, and Device Error Injection capabilities

Pass Criteria:

- Completion timeout logged and escalated to error manager

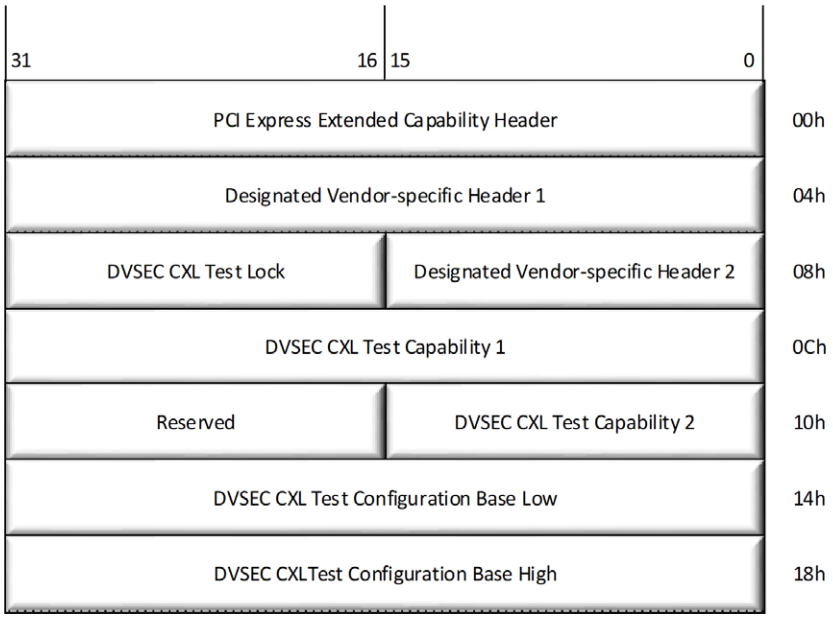
Fail Criteria:

- No errors logged and data corruption seen

14.11 Device Capability and Test Configuration Control

14.11.1 CXL Device Test Capability Advertisement

Figure 121. PCIe DVSEC for Test Capability:



To advertise Test capabilities, the standard DVSEC register fields should be set as below:

Table 75. DVSEC Registers (Sheet 1 of 2)

Register	Bit Location	Field	Value
Designated Vendor-Specific Header 1 (offset 04h)	15:0	DVSEC Vendor ID	0x8086

Table 75. DVSEC Registers (Sheet 2 of 2)

Designated Vendor-Specific Header 1 (offset 04h)	19:16	DVSEC Revision	0x0
Designated Vendor-Specific Header 1 (offset 04h)	31:20	DVSEC Length	0x22
Designated Vendor-Specific Header 2 (offset 08h)	15:0	DVSEC ID	0x0A

Table 76. DVSEC CXL Test Lock (offset 0Ah)

Bit	Attribute	Description
0	RWO	**TestLock**: Software writes 1'b1 to lock the relevant test configuration registers
15:1	N/A	Reserved

Table 77. DVSEC CXL Test Capability1 (offset 0Ch)

Bit	Attribute	Description
0	RO	**SelfChecking**: Set to 1 if Device supports Self Checking
1	RO	**Algorithm1a**: Set to 1 if Device supports hardware for test Algorithm 1a
2	RO	**Algorithm1b**: Set to 1 if Device supports hardware for test Algorithm 1b
3	RO	**Algorithm2**: Set to 1 if Device supports hardware for test Algorithm 2
4	RO	**RdCurr**: Set to 1 if Device supports CXL.cache and RdCurr opcodes as requester.
5	RO	**RdOwn**: Set to 1 if Device supports CXL.cache and RdOwn opcodes as requester.
6	RO	**RdShared**: Set to 1 if Device supports CXL.cache and RdShared opcodes as requester.
7	RO	**RdAny**: Set to 1 if Device supports CXL.cache and RdAny opcodes as requester.
8	RO	**RdOwnNoData**: Set to 1 if Device supports CXL.cache and RdOwnNoData opcodes as requester.
9	RO	**ItoMWr**: Set to 1 if Device supports CXL.cache and ItoMWr opcodes as requester.
10	RO	**MemWr**: Set to 1 if Device supports CXL.cache and MemWr opcodes as requester.
11	RO	**CLFlush**: Set to 1 if Device supports CXL.cache and CLFlush opcodes as requester.
12	RO	**CleanEvict**: Set to 1 if Device supports CXL.cache and CleanEvict opcodes as requester.
13	RO	**DirtyEvict**: Set to 1 if Device supports CXL.cache and DirtyEvict opcodes as requester.
14	RO	**CleanEvictNoData**: Set to 1 if Device supports CXL.cache and CleanEvictNoData opcodes as requester.
15	RO	**WOWrInv**: Set to 1 if Device supports CXL.cache and WOWrInv opcodes as requester.
16	RO	**WOWrInvF**: Set to 1 if Device supports CXL.cache and WOWrInvF opcodes as requester.
17	RO	**WrInv**: Set to 1 if Device supports CXL.cache and WrInv opcodes as requester.
18	RO	**CacheFlushed**: Set to 1 if Device supports CXL.cache and CacheFlushed opcodes as requester.
19	RO	**UnexpectedCompletion**: Device supports sending an unexpected completion on CXL.io
20	RO	**CompletionTimeoutInjection**: Device supports dropping a read in the completion timeout scenario
23:21	N/A	Reserved
31:24	RO	**ConfigurationSize**: Size in Bytes of Test configuration control registers.

Table 78. Device CXL Test Capability2 (Offset 10h)

Bit	Attribute	Description
13:0	RO	**CacheSize**: Cache size supported by the device.
15:14	RO	**CacheSizeUnits**: Units of advertised cache size in CacheSize field 2'b00 : Bytes 2'b01 : KiloBytes (KB) 2'b10 : MegaBytes (MB) Reserved.

Table 79. DVSEC CXL Test Configuration Base Low (Offset 14h)

Bit	Attribute	Description
0	RO	**MemorySpaceIndicator**: The test configuration registers are in memory space. Device must hardwire this to 1'b0
2:1	RO	**Type**: 2'b00 – Base register is 32 bit wide and can be mapped anywhere in the 32 bit address space. 2'b01 – Reserved 2'b10 – Base register is 64 bit wide and can be mapped anywhere in the 64 bit address space. 2'b11 – Reserved
3	RO	**Reserved**: Device must hardwire this bit to 1'b0
31:4	RW	**BaseLow**: bits [31:4] of the base address where the test configuration registers exist.

Table 80. DVSEC CXL Test Configuration Base High (Offset 18h)

Bit	Attribute	Description
31:0	RW	**BaseHigh**: Bits [63:32] of the base address where the test configuration registers exist.

14.11.2 Device Capabilities to Support the Test Algorithms

This section lays out the configuration registers required in the application layer of the device that enable execute/verify/debug of the above Algorithms. These registers are memory mapped and the base is given by the capability structure defined in previous sections. Default value of all register bits must be 0.

Table 81. Register 1: StartAddress1 (Offset 00h)

Bit	Attribute	Description
63:0	RW	**StartAddress1**: Indicates the start address "X1" of the corresponding set in Algorithms 1a,1b, and 2. This could be Host attached memory, device attached memory (if applicable), or an invalid address to test Go-Err support.

Table 82. Register 2: WriteBackAddress1 (Offset 08h)

Bit	Attribute	Description
63:0	RW	**WriteBackAddress1**: Indicates the start address "Z1" of the corresponding set in Algorithms 1a and 1b. This register is only used if device is NOT self-checking, or if self-checking is disabled on the device. This address should map to Host attached memory.

Table 83. Register 3: Increment (Offset 10h)

Bit	Attribute	Description
31:0	RW	**AddressIncrement**: Indicates the increment for address "Y" in Algorithms 1a,1b and 2. The value in this register should be left shifted by 6 bits before using as address increment. Example, a value of 1'b1 implies increment granularity of 7'b1000000 (cache line increments)
63:32	RW	**SetOffset**: Indicates the set offset increment for address "X" and "Z" in Algorithms 1a,1b and 2. The value in this register should be left shifted by 6 bits before using as address increment. Example, a value of 1'b1 implies increment granularity of 7'b1000000 (cache line increments)

Table 84. Register 4: Pattern (Offset 18h)

Bit	Attribute	Description
31:0	RW	**Pattern1**: Indicates the pattern "P" as defined in Algorithms 1a,1b, and 2.
63:32	RW	**Pattern2**: Indicates the pattern "B" as defined in Algorithm 1b.

Table 85. Register 5: ByteMask (Offset 20h)

Bit	Attribute	Description
63:0	RW	**ByteMask**: 1 bit per byte of the cache line to indicate which bytes of the cache line are modified by the device in Algorithms 1a, 1b and 2. This will be programmed consistently with the StartAddress1 register.

Table 86. Register 6: PatternConfiguration (Offset 28h)

Bit	Attribute	Description
2:0	RW	**PatternSize**: Defines what size (in bytes) of "P" or "B" to use starting from least significant byte. As an example, if this is programmed to 3'b011, only the lower 3 bytes of "P" and "B" registers will be used as the pattern. This will be programmed consistently with the ByteMask field, for example, in the given example, the ByteMask would always be in sets of three consecutive bytes.
3	RW	**PatternParameter**: If this field is programmed to 1'b1, device hardware must continue to use the incremented value of patterns (P+N+1) as the base pattern of the next set iteration. If this field is programmed to 1'b0, device hardware must use the original pattern "P" for every new set iteration.
63:4	N/A	Reserved

Table 87. Register 7: AlgorithmConfiguration (Offset 30h) (Sheet 1 of 2)

Bit	Attribute	Description
2:0	RWL	**Algorithm**: 3'b000 – Disabled – serves as a way to stop test. 3'b001 – Algorithm 1a: Multiple Write Streaming 3'b010 – Algorithm 1b: Multiple Write Streaming with Bogus writes 3'b100 – Algorithm 2: Producer Consumer Test Rest are reserved. Implementation Notes: Software will setup all of the other registers (address, patterns, byte-masks etc.) before it writes to this field to start the test. A value of 3'b001, 3'b010, 3'b100 in this field starts the corresponding Algorithm on the device from iteration 0, set 0. No action must be taken by device hardware if a reserved value is programmed. While a test is running, software can write to this field to stop the test. If this happens, device must gracefully complete the current execute and verification loop and then stop the hardware from issuing any more requests to the Host. If software subsequently programs it to any of the other valid values, device hardware must execute the corresponding Algorithm from a fresh loop (iteration 0 on set 0).
3	RW	**SelfChecking**: 1'b0 – device is not going to perform self-checking. 1'b1 – Device is going to perform self-checking in the Verify phase for Algorithms 1 and 2.
7:4	RW	Reserved
15:8	RW	**NumberOfAddrIncrements**: Sets the value of "N" for all 3 Algorithms. A value of 0 implies only the first write (base address) is going to be issued by device.
23:16	RW	**NumberOfSets**: A value of 0 implies that only the first write is going to be issued by the device. If both NumberOfAddIncrements and NumberOfSets is zero, only a single transaction (to the base address) should be issued by the device [NumberOfLoops should be set to 1 for this case]. For Algorithm 1a and 1b: Bits 19:16 gives the number of sets. Bits 23:20 give the number of bogus writes "J" in Algorithm 1b. For Algorithm 2: Bits 23:16 gives the number of iterations "i"
31:24	RW	**NumberOfLoops**: If set to 0, device continues looping across address and set increments indefinitely. Otherwise, it indicates the number of loops to run through for address and set increments.
32	RW	**AddressIsVirtual**: If set to 1, indicates that all programmed addresses are virtual and need to be translated by the device (via ATS). Useful for testing virtualization/device TLBs
35:33	RW	**Protocol**: 3'b000 - PCIe mode 3'b001 - CXL.io only 3'b010 - CXL.cache only 3'b100 - CXL.cache and CXL.io [support is optional and device is free to interleave writes at iteration or set granularity]
39:36	RW	**WriteSemanticsCache**: Only applicable when **Protocol**==3'b010 or 3'b100. In the encodings below, dirty writes can mean evictions or flush depending on device behavior. 4'b0000 - Dirty Writes use ItoMWr, Clean Writes use CleanEvict [Clean writes will occur if PatternSize==0] 4'b0001 - Dirty Writes use MemWr, Clean Writes use CleanEvictNoData 4'b0010 - Dirty Writes use DirtyEvict 4'b0011 - Dirty Writes use WOWrInv 4'b0100 - Dirty Writes use WOWrInvF [only programmed by test software if Device is expected to own/ modify the full cache line] 4'b0101 - Dirty Writes use WrInv 4'b0110 - Dirty Writes use ClFlush 4'b0111 - Dirty Writes/Clean Writes can use any of CXL.cache supported opcodes. Device implementation specific. All other encodings are reserved; and device hardware should not take any actions if this has been programmed to a reserved value.

Table 87. Register 7: AlgorithmConfiguration (Offset 30h) (Sheet 2 of 2)

40	RWL	**FlushCache**: Test software can program this value at runtime to trigger a cache flush from Device and issue CacheFlush opcode. Execute/Verify loops must stop after completing the current loop and CacheFlushed has been issued, until software changes this value back to 1'b0 – after which, device hardware should resume execute/verify loops from the next iteration/set [it must remember the iteration and set value where execution stopped].
43:41	RW	**ExecuteReadSemanticsCache**: Only applicable when **Protocol**==3'b010 or 3'b100. 3'b000 : Ownership reads use RdOwn 3'b001 : Ownership reads use RdAny 3'b010 : Ownership reads use RdOwnNoData [only programmed by test software if device is expected to modify the entire cache line] 3'b100 : Device can use any of the CXL.cache supported opcodes All other encodings are reserved, and should not start execute/verify loops if programmed.
46:44	RW	**VerifyReadSemanticsCache**: Read opcodes used when device is in Verify phase. 3'b000 : RdCurr 3'b001 : RdShared 3'b010 : RdOwn 3'b100 : RdAny All other encodings are reserved, and should not start execute/verify loops if programmed.
63:47	N/A	Reserved

Table 88. Register 8: DeviceErrorInjection (Offset 38h)

Bit	Attribute	Description
0	RWL	**UnexpectedCompletionInjection**: Software writes 0x1 to this bit to trigger a completion injection on a message in the Tx direction. Hardware must inject an unexpected completion by sending the same completion twice.
1	RO-V	**UnexpectedCompletionInjectionBusy**: Hardware loads 1'b1 to this bit when the Start bit is written. Hardware must clear this bit to indicate that it has indeed finished error injections. Software is permitted to poll on this bit to find out when hardware has finished error injection.
2	RWL	**CompleterTimeout** Software writes 0x1 to this bit to trigger a completer timeout injection on a message in the Tx direction. Hardware must suppress the transmission of completion packet.
3	RO-V	**CompleterTimeoutInjectionBusy**: Hardware loads 1'b1 to this bit when the Start bit is written. Hardware must clear this bit to indicate that it has indeed finished error injections. Software is permitted to poll on this bit to find out when hardware has finished error injection.
31:4	N/A	Reserved

14.11.3 Debug Capabilities in Device

14.11.3.1 Error Logging

The following capabilities in a device are strongly recommended to support ease of verification and compliance testing.

A device that supports self-checking must include an error status and header log register with the following fields:

Table 89. Register 9: ErrorLog1 (Offset 40h)

Bit	Attribute	Description
31:0	RW	**ExpectedPattern**: Expected data pattern as per device hardware.
63:32	RW	**ObservedPattern**: Observed data pattern as per device hardware.

Table 90. Register 10: ErrorLog2 (Offset 48h)

Bit	Attribute	Description
31:0	RW	**ExpectedPattern**: Expected data pattern as per device hardware.
63:32	RW	**ObservedPattern**: Observed data pattern as per device hardware.

Table 91. Register 11: ErrorLog3 (Offset 50h)

Bit	Attribute	Description
7:0	RW	**ByteOffset**: First byte offset within the cache line where the data mismatch was observed.
15:8	RW	**LoopNum**: Loop number where data mismatch was observed.
16	RW1C	**ErrorStatus**: Set to 1 by device if data miscompare was observed

14.11.3.2 Event Monitors

It is strongly recommended that a device advertise at least 2 event monitors, which can be used to count device-defined events. An event monitor consists of two 64 bit registers:

a. An event controller: EventCtrl
b. An event counter: EventCount

The usage model is for software to program EventCtrl to count an event of interest, and then read the EventCount to determine how many times the event has occurred. At a minimum, a device must implement the ClockTicks event. When the ClockTicks event is selected via the event controller, the event counter will increment every clock cycle, based on the application layer's clock. Further suggested events may be published in the future. Examples of other events that a device may choose to implement are:

a. Number of times a particular opcode is sent or received
b. Number of retries or CRC errors
c. Number of credit returns sent or received
d. Device-specific events that may help visibility on the platform or with statistical computation of performance

Below are the formats of the EventCtrl and EventCount registers.

Table 92. Register 12: EventCtrl (Offset 60h)

Bit	Attribute	Description
7:0	RW	**EventSelect**: Field to select which of the available events should be counted in the paired EventCount register.
15:8	RW	**SubEventSelect**: Field to select which sub-conditions of an event should be counted in the paired EventCount register. This field is a bit-mask, where each bit represents a different condition. The EventCount should increment if any of the selected sub-conditions occurs. For example, an event might be "transactions received", with three sub-conditions of "read", "write", and "completion".
16	N/A	Reserved

Table 92. Register 12: EventCtrl (Offset 60h)

17	RW	**Reset:** When set to 1, the paired EventCount register will be cleared to 0. Writing a 0 to this bit has no effect.
18	RW	**EdgeDetect**: When this bit is 0, the counter will increment in each cycle that the event has occurred. When set to 1, the counter will increment when a 0 to 1 transition (i.e., rising edge) is detect.
63:19	N/A	Reserved

Table 93. Register 13: EventCount (Offset 68h)

Bit	Attribute	Description
63:0	RO	**EventCount**: Hardware load register which is updated with a running count of the occurrences of the event programmed in the EventCtrl register. It is monotonically increasing, unless software explicitly writes it to a lower value or writes to the “Reset” field of the paired EventCtrl register.

§ §

Appendix A Taxonomy

A.1 Accelerator Usage Taxonomy

Table 94. Accelerator Usage Taxonomy (Sheet 1 of 2)

Accelerator Type	Description	Challenges & Opportunities	CXL Support
Producer-Consumer Accelerators that don't execute against "Memory" w/o special needs	Work on data streams or large contiguous data objects. Little interaction with host Standard P/C ordering model works well.	Efficient work submission Efficient exchange of meta-data (flow control)	Basic PCIe + AiA CXL.io
Producer-Consumer Plus Accelerators that don't execute against "Memory" w/ special needs	Same as above, but… P/C ordering model doesn't work well Need special data operations such as atomics	Device Coherency can be used to implement varied ordering models and special data operations	CXL.cache on CXL w/ baseline snoop filter support CXL.io CXL.cache
SW Assisted SVM Memory Accelerators that execute against "Memory" w/ software supportable data management	Local memory is often needed for BW or latency predictability Little interaction with the host Data management easily implemented in SW, e.g., few and simple data buffers	Host SW should be able to interact directly with accelerator memory (SVM, Google) Reduce copies, replication, pinning Optimizing coherency impact on performance is a challenge SW can provide best optimization of coherency impact	CXL Bias model with SW managed bias. CXL.io CXL.cache CXL.mem

Table 94. Accelerator Usage Taxonomy (Sheet 2 of 2)

Accelerator Type	Description	Challenges & Opportunities	CXL Support
Autonomous SVM Memory Accelerators that execute against "Memory" where software supported data management is impractical	Local memory often needed for BW or latency predictability Interaction with the host is common Data movement very difficult to manage in SW, e.g., sparse data structures, pointer based data structures, etc.	Host SW should be able to interact directly with accelerator memory (SVM, Google) Reduce copies, replication, pinning Optimizing coherency impact on performance is a challenge Cannot count on SW for bias management	CXL Bias model with HW managed bias. CXL.io CXL.cache CXL.mem
Giant Cache Accelerators that execute against "Memory" where local memory and caching is required.	Local memory needed for BW or latency predictability Data footprint is larger than local memory Interaction with the host is common Data must be cycled through accelerator memory in small blocks Data movement very difficult to manage in SW	Accelerator memory needs to work like a cache (not SVM/system memory) Ideally cache misses detected in HW, but cache replacements can be managed in SW	CXL.cache on CXL w/ "Enhanced Directory" snoop filter support CXL.io CXL.cache
Disaggregated Memory Controller Typically for memory controllers with remote persistent memory, which may be in 2LM or App Direct mode	PCIe semantics needed for device enumeration, driver support and device management Most operational flows rely on being able to communicate directly with a Home Device or Near Memory Controller on the Host	Device needs high BW and low latency path from memory controller to Home Device in the CPU	CXL.mem on CXL CXL.io CXL.mem

A.2 Bias Model Flow Example – From CPU

- Start with pages in Device Bias
  - Pages guaranteed not to be cached in host cache hierarchy
- Software allocates pages from device memory
  - Software pushes operands to allocated pages from peer CPU core:
  - Software uses, e.g., OCL API to flip operand pages to Host Bias
  - No data copies or cache flushes required
  - Host CPUs generate operand data in target pages – data ends up in some arbitrary location in the host cache hierarchy.
- Device uses operands to generate results
  - Software uses, e.g., OCL API to flip operand pages back to Device Bias
  - API call causes work descriptor submission to device; descriptor asks the device to flush operand pages from host cache.
  - Cache flush executed using CLFLUSH on CXL CXL.cache protocol.
  - When Device Bias flip is compete, software submits work to the accelerator
  - Accelerator executes with no host related coherency overhead
  - Accelerator dumps data to results pages.

- Software pulls results from the allocated pages:
  - Software uses, e.g., OCL API to flip results pages to Host Bias.
  - This action causes some bias state to be changed but does not cause any coherency or cache flushing actions.
  - Host CPUs can access, cache and share results data as needed.
- Software releases the allocated pages.

A.3 CPU Support for Bias Modes

There are two envisaged models of support that the CPU would provide for Bias Modes. These are described below.

A.3.1 Remote Snoop Filter

- Remote socket owned lines belonging to accelerator attached memory are tracked by a Remote SF located in the C-CHA. Remote SF does not track lines belonging to Host memory. The above obviates the need for directory in device memory. Please note this is only possible in host bias mode since in device bias mode, local/remote sockets can't cache lines belonging to device memory.
- Local socket owned lines belonging to accelerator attached memory will be tracked by local SF in the C-CHA. Please note this is only possible in host bias mode since in device bias mode, local/remote sockets can't cache lines belonging to device memory.
- Device owned lines belonging to accelerator attached memory (in host bias mode) will NOT be tracked by local SF in the C-CHA. These will be tracked by the Device Coherency Engine (DCOH) using a device specific mechanism (device SF). In device bias mode, SF in the C-CHA does not even see the requests.
- Device owned lines belonging to host memory (in either mode) WILL be tracked by local SF in the C-CHA. This may cause the device to receive snoops through CXL (CXL.cache) for such lines.

14.11.4 Directory in Accelerator Attached Memory

- Remote socket owned lines belonging to device memory are tracked by directory in device memory. C-CHA may choose to do OSB for some cases.
- Local socket owned lines belonging to device memory will be tracked by local SF in the C-CHA. For access by device, local socket owned lines belonging to device memory will also update directory.
- Device owned lines belonging to device memory will NOT be tracked by local SF in the C-CHA. These will be tracked by the Device Coherency Engine (DCOH) using a device specific mechanism (device SF).
- Device owned lines belonging to host memory (in either mode) WILL be tracked by local SF in the C-CHA. This may cause the device to receive snoops through CXL (CXL.cache) for such lines.
- Bias Table is located in stolen memory in the device memory and is accessed through the DCOH.

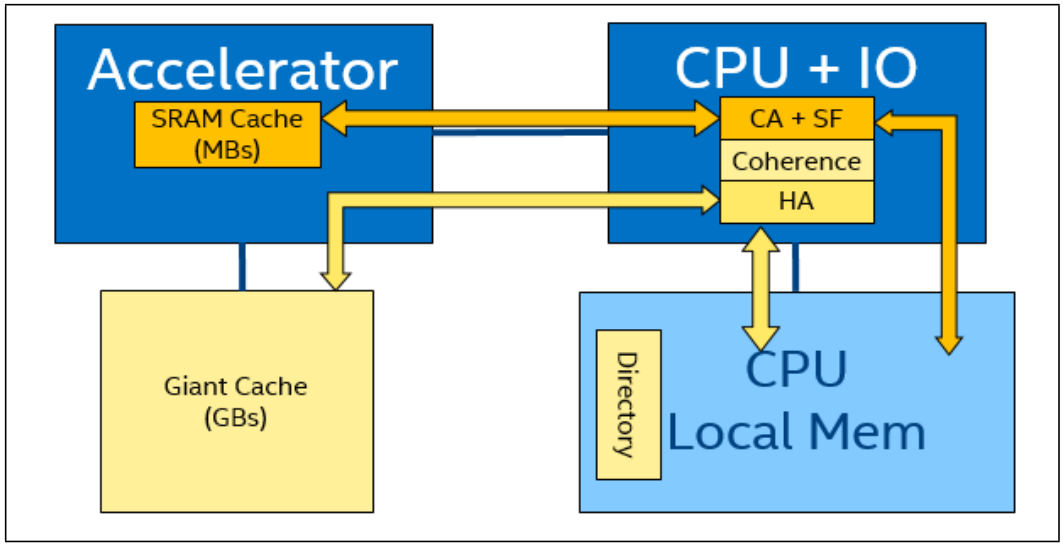
A.4 Giant Cache Model

For problems whose datasets exceed the size of device attached memory, the memory attached to the accelerator really wants to be a cache, not memory:

- Typically the full dataset will live in processor attached memory.

- Subsets of this larger data set are cycled through the accelerator memory as the computation proceeds.
- For such use cases, caching is the right solution:
  - Accelerator memory is not mapped into system address map – data set is built up in host memory
  - Single page table entry per page in data set – no page table manipulation as pages are cycled through accelerator memory
  - Copies of data can be created under driver and/or hardware control with no OS intervention

Figure 122. Profile D - Giant Cache Model



Critical issues with a Giant Cache:

- Cache is too big for tracking in the Host on-die snoop filter
- Snoop latency for a Giant Cache is likely to be much higher than standard on-die cache snoop latency.

CXL recommended solution:

- Implements snoop filter in processor's coherency directory (stored in DRAM ECC bits) which essentially becomes a highly scalable snoop filter
- Minimizes impact to processor operations unrelated to accelerators
- Allows accelerator to access data over CXL.cache as a caching Device.
- Provides support on CXL.cache to allow an accelerator to explicitly request directory snoop filtering for giant cache.
- Processor infrastructure differentiates between low latency and high latency requester types
- Support for simultaneous use of a small, low latency cache, associated with the on-die snoop filter, will come for free.

§ §