eXtensible Host Controller Interface for Universal Serial Bus

(xHCI)

Requirements Specification

November 2017

Revision 1.1

NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. CONTACT INTEL ON FURTHER LICENSING AGREEMENTS AND REQUIREMENTS.

INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Copyright © 2008-2017 Intel Corporation. All rights reserved.

Contents

Figures

Tables

Revision History

Revision	Issue Date	Comments
0.96	5/8/2009	
1.0	5/21/10	Refer to xHCI 0_96 errata files.
1.1	12/20/13	Refer to xHCI 1_0 errata files 1-21.
1.1	11/7/17	xHCI Revision 1.1 with all changes up to errata 4

Please send comments via electronic mail to: xhcisupport@intel.com

Contributors

Randy Aull	Microsoft Corporation
Paul Baleme	Intel Corporation
Marcin Behrendt	Cadence Design Systems, Inc.
Dustin Bingham	Intel Corporation
Martin Borve	Microsoft Corporation
Jeanne Cai	Marvell Technology Group Ltd.
Brent Chartrand	Intel Corporation
Huimin Chen	Intel Corporation
Binuthankakumar Chinnathankam	Larsen & Toubro Infotech Ltd.
Brian Collins	Fresco Logic Inc.
John Couillard	Intel Corporation
Eric DeHaemer	Intel Corporation
Vidyadhari Dharmaraju	Intel Corporation
Paul Diefenbaugh	Intel Corporation
Bob Dunstan	Intel Corporation
Kurt Fankhauser	Fresco Logic Inc.
Nobuo Furuya	NEC Corporation
John Garney	MCCI Corporation
Charlie Guy	Avsys Corporation
Vivek Gupta	Microsoft Corporation
Raul Gutierrez	Intel Corporation
Chip Haldane	Intel Corporation
Dave Harriman	Intel Corporation
Will Harris	Texas Instruments Incorporated

David Hines	Intel Corporation
Yu Hong	Marvell Technology Group Ltd.
Amanda Hosler	Specwerkz LLC
Brad Hosler	Intel Corporation
John S. Howard	Intel Corporation
Ching Lin Hsu	Faraday Technology Corporation
Rahman Ismail	Intel Corporation
Jaya Jeyaseelan	Intel Corporation
Surya Kareenahalli	Intel Corporation
Michael Kentley	High Desert Design Center
Chien-cheng Kuo	Etron Technology, Inc.
Dian Kurniawan	Fresco Logic, Inc.
Piotr Kwidzinski	Intel Corporation
Luke Lai	NVIDIA Corporation
Philip Lantz	Intel Corporation
Hogan Lee	ASMedia Technology Inc.
Brian Lounsbery	Intel Corporation
Baolu Lu	Intel Corporation
Ben Lunt	Forever Young Software
Alberto Martinez	Intel Corporation
Mark Maszak	Microsoft Corporation
Steve McGowan	Intel Corporation
Chintan Mehta	Sibridge Technologies
Chris Meyers	Fresco Logic, Inc.
Lukasz Mielicki	Intel Corporation
Nobuyuki Mizukoshi	NEC Corporation
Saleem Mohammad	Synopsys, Inc.

Anoop Mukker	Intel Corporation
Jie Ni	Fresco Logic, Inc.
Hajime Nozaki	NEC Corporation
Jake Oshins	Microsoft Corporation
Sanket Patel	Microsoft Corporation
Fizal Peermohamed	Microsoft Corporation
Tomaz Pielaszkiewicz	Intel Corporation
Georg Potthast	Independent contractor
CH Ramakrishna	Larsen & Toubro Infotech Ltd.
Diane Rose	Specwerkz LLC
Hiro Sakamoto	NEC Corporation
Nitin Sarangdhar	Intel Corporation
Makoto Sato	NEC Corporation
Joe Scanlon	Advanced Micro Devices
Joe Schaefer	Intel Corporation
Vasudevan Shanmugasundaram	Intel Corporation
Sarah Sharp	Intel Corporation
Wei Sheng	Corigine Inc.
Eyal Skulsky	Qualcomm, Inc.
Glen Slick	Microsoft Corporation
Gary Solomon	Intel Corporation
Raja Subramanian	Larsen & Toubro Infotech Ltd.
Kevin Beow Ee Tan	Intel Corporation
Yuliang Tao	Marvell Technology Group Ltd.
"Peter" Chu Tin Teng	NEC Corporation
Qunzhao Tian	Marvell Technology Group Ltd.
Jay Tseng	VIA Technologies, Inc.

Karthi Vadivelu	Intel Corporation
Cedric Villat	Intel Corporation
Luke Valenty	Intel Corporation
Venkatarama	Larsen & Toubro Infotech Ltd.
Krishnan Venkataraman	Moschip Semiconductor Tech. Ltd.
Sue Vining	Texas Instruments Incorporated
Jim Walsh	Intel Corporation
Jennifer Wang	Intel Corporation
Qiangwen Wang	Synopsys, Inc.
Y.W. Wang	Faraday Technology Corporation
Rafal Wielicki	Intel Corporation
Eric Wittmayer	Fresco Logic, Inc.
Jun Xu	Fresco Logic, Inc.
Steven E. Zawid	Intel Corporation
Hefei Zhu	Marvell Technology Group Ltd.
Kevin Zhenyu Zhu	Intel Corporation

Dedication

The xHCI Specification is dedicated to the memory of Brad Hosler, a good friend and the impact of whose accomplishments have made the Universal Serial Bus one of the most successful technology innovations of the Personal Computer era.

- The xHCI Architecture Team

1 Preface

1.1 Objective of Specification

The eXtensible Host Controller Interface (xHCI) specification describes the register-level host controller interface for Universal Serial Bus (0) Revision 2.0 and above. The specification includes a description of the hardware/software interface between system software and the host controller hardware.

This specification is intended for hardware component designers, system builders and device driver (software) developers. The reader is expected to be familiar with the current Universal Serial Bus Specification revisions. In spite of due diligence, there may exist conflicts between this specification and the USB Specification. The USB Specifications take precedence on all issues of conflict.

1.2 Scope of Document

The specification is primarily targeted to host controller developers and system OEMs, but provides valuable information for platform operating system and BIOS device driver developers, adapter IHVs/ISVs, and platform/adapter controller vendors. This specification can be used for developing new products and associated software.

1.3 Document Organization

This specification presents a view of the overall architecture and detailed description of the operational model requirements of the host controller, using the defined registers and interface data structures.

The architecture (3) and operational (4) sections are followed by two sections of pure structural definitions that detail the register space (5) and interface data structures (6). These definition chapters contain little or no operational requirements or usage models. The final sections describe the xHCI Extended Capabilities (7), and the virtualization operational model (8). The Appendix covers useful information not included elsewhere in the specification.

1.4 References

The following documents are referenced throughout this specification. The *Spec Reference* defines a shorthand mnemonic used in this specification for the respective document listed below.

Spec Reference	Title	Revision	Location
ACPI	Advanced Configuration and Power Interface Specification	3.0b[1] October 10, 2006	www.acpi.info
BCS	Battery Charging Specification	1.1 April 15, 2009	www.usb.org
EHCI	Enhanced Host Controller Interface Specification	1.0 March 12, 2002	www.intel.com/technology/usb
EHCI	Enhanced Host Controller Interface Specification	1.0 March 12, 2002	www.intel.com/technology/usb
EHCI1_1Add	EHCI v1.1 Addendum	1.1 August, 2008	www.intel.com/technology/usb
iASL	iASL - ACPI Source Language Compiler, Table Compiler, and AML Disassembler	20120111-32 January, 11 2012	www.acpica.org/downloads/binary_tools.php
MUCC	Universal Serial Bus Micro-USB Cables and Connectors Specification	1.01 April 4, 2007	www.usb.org
OTG	On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification	2.0 May 8, 2009	www.usb.org
PCI	PCI Local Bus Specification	3.0 February 3, 2004	www.pcisig.com
PCIe	PCI Express Base Specification	3.0 November 10, 2010	www.pcisig.com

[1]Revisions 3.0b and beyond of the ACPI specification define the _UPC (USB3 Connector Type) and _PLD (Group) extensions referenced in Appendix D.1.1. The ACPI extensions required to support USB Power Delivery are going to be defined in an upcoming of the ACPI specification.

PCI PM	PCI Bus Power Management Interface Specification	1.2 March 3, 2004	www.pcisig.com
SR-IOV	PCI Single Root I/O Virtualization	1.0 Sept. 11, 2007	www.pcisig.com
SSIC	Inter-Chip Supplement to the Revision 3.0 Specification	0.9 Dec. 11, 2011	www.usb.org
USB2	Universal Serial Bus Specification	2.0 April 27, 2000	www.usb.org
USB2 LPM	USB 2.0 Link Power Management Addendum and Errata for USB 2.0 ECN: Link Power Management - 7/2007	Final July 16, 2007 5/13/11	www.usb.org
USB3	Universal Serial Bus 3.1 Specification	1.0	www.usb.org
USB PD	Universal Serial Bus Power Delivery Specification	Rev. 1.0, Version 1.2 June 26, 2013	www.usb.org
xHCI	Extensible Host Controller Interface Specification for Universal Serial Bus	1.0 May 21, 2010	www.intel.com/technology/usb/spec.htm

Note: Rather than enumerating the full specification name every time one of the above specs are referenced in this document, the abbreviation listed in the Spec Reference column shall be used.

1.5 Index

This document does not include an index. An effective substitute when viewing with a Adobe® Reader® is to use the **Search** dialog box to locate all references to a specific xHCI feature or field.

To facilitate indexing, all references to register and data fields may be automatically located using their mnemonic, acronym, or name. There was also an effort to maintain consistent naming and phrasing throughout the spec.

- For example: To find all references to the Port Power (PP) field of the PORTSC register, in Reader® open the Search dialog box, and since this field has a acronym, enter the string “PP” in the ‘What word of phrase would you like to search for?’ text box. Check the ‘Whole words only’ and ‘Case-sensitive’ check boxes, and press the ‘Search’ button to list all references to the Port Power flag in this specification in the ‘Results’ window.

- To find all references to the Frame ID field, for which no mnemonic or acronym is defined, simply enter “Frame ID” into the ‘What word of phrase would you like to search for?’ text box.
- Often it is useful to cut and paste a phrase into the ‘What word of phrase would you like to search for?’ text box. If you search on the phrase "advance to the next TD" you will get about 12 hits, "zero length" about 9 hits, etc.

After pressing the ‘Search’ button, the results of the search are displayed in the ‘Results:’ window of the Search dialog box. Clicking on any tree entry in the Results: window will jump you to the selected text in the spec. Using the Up and Down Arrow keys while the Search dialog box has focus will allow you to quickly view all the search results in the document.

The Search dialog box has been supported by Adobe® Reader® and Acrobat® for quite a while, but how it is accessed may vary from one version to another. With most versions of Adobe® products Shift+Ctrl+F will bring up the Search dialog box, or…

In Reader® 8 to open the Search dialog box, select “Edit” then “Search” from the menu.

In Reader® 10 open the Search dialog box, by clicking on the Tool Bar icon that looks like a pair of binoculars. If the Search icon is not visible in the Tool Bar, then right click on the Tool Bar, mouse over ‘Edit’ then click on ‘Advanced Find’.

In Acrobat® 9 open the Search dialog box, next to the Search text box there is a small "down arrow". Click on the arrow and select "Open Full Acrobat Search…". If the Search text box is not visible in the Tool Bar, then right click on the Tool Bar, and click on ‘Find’.

1.6 Terms and Abbreviations

ACK	Handshake packet indicating a positive acknowledgment.
Alternate Interface	An optional Interface setting provided by a USB device. Alternate Interface settings may be used to define a range of payload sizes for USB endpoints.
Async Pipe	A “Best Effort” Pipe defined by a Control or Bulk endpoint.
attached	This specification makes a distinction between the words “attach” and “connect”. A USB2 downstream device is considered to be “attached” to an upstream port if the upstream port has detected either the D+ or D- data line pulled high through a 1.5 kΩ resistor. A USB3 downstream device is considered to be “attached” to an upstream port if the upstream port has detected SuperSpeed far-end receiver terminations.
Aux Power	The xHCI supports split power “wells”; the Core Power well and the Aux Power well (or Auxiliary Power well). The *Aux Power* well is optional. For example, *Aux Power well* voltage may be present whenever AC or battery power is applied to the system. For more information refer to section 4.23.1.
Base	The beginning of the host controller’s MMIO address space is referred to as “Base”.
Best Effort Service Latency (BESL)	BESL indicates the best effort to resumption of service to a device after the initiation of a resume event by a device.
Best Effort Latency Tolerance (BELT)	Best Effort Latency Tolerance (BELT) messages are supported by USB3 devices (excluding hubs) using an optional USB3 “Device Notification (DEV_NOTIFICATION)” Transaction Packet (TP) with a Notification_Type = LATENCY_TOLERANCE_MESSAGE (LTM). This message is also referred to as a Latency Tolerance Message (LTM) TP. This TP contains a specific value known as the Best Effort Latency Tolerance (BELT) value that indicates the current tolerable service latency for that device.
bInterval	Interval value defined by a USB Endpoint Descriptor.

Burst	The transmission of multiple back-to-back data packets on the USB.
Bus Error Counter	The Bus Error Counter is an internal counter that the xHC maintains, which determines the number of consecutive Errors allowed while executing a USB Transaction.
Bus Instance (BI)	A Bus Instance represents a “unit” bus bandwidth at the speed that the BI supports. e.g. A SuperSpeed BI represents 5Gb/s of bandwidth. A High-speed BI represents 480Mb/s of bandwidth, Low-/Full-speed BI represents 12Mb/s of bandwidth. Multiple Root Hub ports may share the bandwidth of a single BI. Note that the bit rates are maximums for the respective buses.
Capability Registers	The Capability Registers specify read-only limits, restrictions and capabilities of the host controller implementation. These values are used as parameters to the host controller driver.
Chip Hardware Reset	A Chip Hardware Reset may be either a PCI reset input or an optional power-on reset input to the xHC, e.g. the initial power-up of the Aux Power well.
clear	When used in reference to a flag or field of a data structure or register, the flag or field shall be cleared to ‘0’.
Composite Device	A USB composite device has only a single USB device address, and exposes multiple interfaces that are controlled independently of each other.
connected	A USB2 downstream device is considered to be “connected” to an upstream port if, 1) device has pulled either the D+ or D- data line high through a 1.5 kΩ resistor, and 2) if the device is high-speed or full-speed it has been reset and the Chirp signaling has determined its speed. A USB3 downstream device is considered to be “connected” to an upstream port if, 1) SuperSpeed far-end receiver terminations have been detected, 2) training was successful, and 3) the Port Capability/Configuration LMP exchanges are successful.

Control Endpoint	As defined by the USB specification, a pair of device endpoints with the same endpoint number that are used by a control pipe. Control endpoints transfer data in both directions and, therefore, use both endpoint directions of a device address and endpoint number combination. Thus, each control endpoint consumes two endpoint addresses.
Core Power	The xHCI supports split power “wells”; the Core Power well and Aux Power well. The xHCI *Core Power* well is required and may be switched on or off to manage xHC power consumption. For more information refer to section 4.23.1.
D0	PCI controller power “On” state. Refer to PCI PM specification.
D1 or D2	PCI controller intermediate power states. Refer to PCI PM specification.
D3	PCI controller power “Off” state. Refer to PCI PM specification.
Default Control Endpoint	The Default Control Endpoint always exists once a USB device is powered, in order to provide access to the device's configuration, status, and control information. The Default Control Endpoint is always endpoint number '0'.
Device Context Base Address Array	The Device Context Base Address Array contains 256 entries and supports up to 255 USB devices or hubs, where each element in the array is a 64-bit pointer to the base address of a Device Context. Entry 0 is reserved.
Device Context	A Device Context is a data structure that describes an individual USB device attached to the host controller. A Device Context is organized as an array of up to 32 context data structures, consisting of 1 Slot Context and up to 31 Endpoint Context data structures.
DCI	The Device Context Index (DCI) is a value used to reference the respective element of the Device Context data structure. Refer to section 4.5.1.

Dequeue Pointer	The Dequeue Pointer is a pointer into a TRB Ring. It references the next TRB in a TRB Ring to be processed by the consumer of TRB Ring work items. The Dequeue Pointer for Transfer and Command Rings is ***NOT*** defined as a physical xHC register. A facsimile of this pointer is maintained internally by the xHC and system software to manage a respective ring.
Device Endpoint	A uniquely addressable portion of a USB device that is the source or sink of information in a communication flow between the host and device. Also see *Endpoint Address*.
Device Resources	Resources provided by USB devices, such as buffer space and endpoints. Also see *Host Resources* and *Universal Serial Bus Resources*.
Device Slot	Device Slot refers to the xHC interface associated with an individual USB device, e.g. the associated Device Context Base Address Array entry, a Doorbell Array register, and its Device Context.
Device Software	Software that is responsible for managing a USB device. This software may or may not also be responsible for configuring the device for use.
Direct-Assignment	Direct-Assignment is a term used with virtualization to describe a hardware device interface that is *Directly Assigned* to a Virtual Machine. Direct-Assigned devices do not suffer from the overhead incurred by device whose hardware register-level interface is emulated in software by a virtual environment.
Doorbell Array	The Doorbell Array is an array of 256 Doorbell Registers, which supports up to 255 USB devices or hubs. Doorbell Register 0 is allocated to the Host Controller, the remaining registers are allocated to individual Device Slots.
Doorbell Register	A Doorbell Register provides system software with a mechanism for notifying the xHC if it has Slot, or Endpoint related work to perform. A *DB Target* field in the Doorbell Register is written with a Reason Code to “ring” the doorbell.

Downstream	The direction of data flow from the host or away from the host. A downstream port is the port on a hub electrically farthest from the host that generates downstream data traffic from the hub. Downstream ports receive upstream data traffic.
DPH Error	A *DPH Error* may be due to one or more of the following conditions: an incorrect Device Address, the Endpoint Number and Direction does not refer to an endpoint that is part of the current configuration, or the DPH does not have an expected sequence number.
DPP Error	A *DPP Error* may be due to one or more of the following conditions: CRC incorrect, DPP aborted, DPP missing, ACK TP with the *Retry Data Packet* (rty) bit set, or the data length in the DPH does not match the actual data payload length.
DSP	DownStream Port an SSIC term that "refers to the port of a host to which a peripheral is connected".
Dword	A data element that is four bytes (32 bits) in size.
EDTLA	*Event Data Transfer Length Accumulator*. Refer to section 4.11.5.2 for more information.
EHCI	Enhanced Host Controller Interface. Intel defined USB host controller specification for High-speed devices.
Embedded hub	A USB 2.0 or 3.x hub that is located on the system board, and between the xHC device and the system board USB connector or non-removable USB device.
Endpoint	A uniquely addressable portion of a USB device that is the source or sink of information in a communication flow between the host and device.
Endpoint Address	The xHCI defines an Endpoint Address as 5-bit value that is a combination of an Endpoint Number (bits 4-1) and an Endpoint Direction (bit 0). For Control Endpoints, the Direction (bit 0) is set to '1' to form its Endpoint Address. Note that xHCI encoding of an Endpoint Address is not the same as the Endpoint Descriptor *bEndpointAddress* field defined by the USB specification.

Endpoint Context	An Endpoint Context data structure defines a Transfer Ring which is used to manage transfers associated with the respective endpoint. An Endpoint Context exists for each endpoint of a device.
Endpoint Direction	The direction of data transfer on the USB. The direction can be either IN or OUT. IN refers to transfers to the host; OUT refers to transfers from the host. When computing the Endpoint Address an IN endpoint is represented by a '1' and an OUT endpoint is represented by a '0'.
Endpoint ID	Identical to the Device Context Index (DCI). Refer to section 4.5.1.
Endpoint Number	A four-bit value between 0h and Fh, inclusive, associated with an endpoint on a USB device.
Enhanced SuperSpeed	A collection of features or requirements that apply to both USB 3.0 and USB 3.1 bus operation. may also be referred to as "Enhanced SS".
Enqueue Pointer	The Enqueue Pointer is a pointer into a TRB Ring. It references the next TRB location available to producer for scheduling work items to the Ring. The Enqueue Pointer is ***NOT*** defined as a physical xHC register. A facsimile of this pointer is maintained internally by the xHC and system software to manage a respective ring.
ERDY	Handshake acknowledgment packet indicating an Endpoint is Ready to move data.
Endpoint Service Interval Time (ESIT)	The service period of an Interrupt or an Isochronous Endpoint. An ESIT defines a period of one or more microframes.
Event Data TD	A TD that consists of just one Event Data TRB.
Event Data TRB	A Normal Transfer TRB with its *Event Data* (ED) flag equal to '1'. Refer to section 4.11.5.2.
Frame	A 1 millisecond time base established on full-/low-speed USB buses.

Fine-grain scatter/gather	The xHCI TRBs support byte granularity for the *TRB Data Buffer Pointer* and *TRB Transfer Length* fields, which enables “fine-grain” scatter/gather operations.
FS	See *Full-speed*.
Full-speed	USB operation at 12 Mb/s. Also see *low-speed*, *high-speed*, *SuperSpeed* and *SuperSpeedPlus*.
Handshake Packet	A USB packet that acknowledges or rejects a specific condition. For examples, see *ACK* and *NAK*.
High-bandwidth endpoint	A high-speed USB device endpoint that transfers more than 1024 bytes and less than 3073 bytes per microframe.
High-speed	USB operation at 480 Mb/s. Also see *low-speed*, *full-speed*, *SuperSpeed*, and *SuperSpeedPlus*.
High-Touch	High touch registers are referenced regularly during the normal operation of the xHC by system software, e.g. Ringing doorbells to queue work, managing interrupts, etc.
Host	The host computer system where the USB Host Controller is installed. This includes the host hardware platform (CPU, bus, etc.) and the operating system in use.
Host Controller	The host’s USB interface.
Host Controller Driver (xHCD)	This software entity is the interface between the xHC and the USB Driver (USBD). It translates system software requests for USB operations to TRBs scheduled on pipes to USB devices.
Host Controller Driver Enumeration Component (xHCDe)	This software entity is a component of the xHCD that manages the enumeration of USB devices at power up, when they are attached, and when they are detached.
Host Initiated Resume Duration (HIRD)	HIRD is the minimum time the xHC will drive resume signaling on a USB2 port when it initiates an exit from L1.
HS	See *High-speed*.
Hub	A USB device that provides additional connections to the USB.

Hub Tier	One plus the number of USB links in a communication path between the host and a peripheral device.
Input Device Context	The Device Context component (Slot and Endpoint Contexts) of an Input Context. An Input Context data structure pointed to by a Command TRB.
Input Endpoint Context	An Endpoint Context contained in an Input Context. An Input Context data structure pointed to by a Command TRB.
Input Slot Context	A Slot Context contained in an Input Context. An Input Context data structure pointed to by a Command TRB.
Integrated hub	A Tier 2 USB 2.0 hub that is integrated into an xHC device.
Interval	The time delay between scheduling periodic transfers. Intervals are defined in frames (1ms.) for LS/FS devices, microframes (125µs.) for HS and SS devices.
Isoch TD	An Isoch Transfer Descriptor consists of an Isoch TRB chained to 0 or more Normal TRBs, and describes a work item for an isochronous endpoint. Isoch TDs are only found on the Transfer Rings associated with Isoch Endpoints.
Isoch TRB	An Isochronous Transfer Request Block that is always the first TRB of an Isoch TD. They are only found on the Transfer Rings associated with Isoch Endpoints. Refer to section 4.11.2.3.
ISR	The Interrupt Service Routine is the software invoked by an interrupt.
L0	USB2 "On" power state.
L1	USB2 Link Power Managed (LPM) state.
L2	USB2 Suspend state.
L3	USB2 "Off" power state.
Lane	A Lane is a point-to-point serial connection, typically implemented as a differential signal pair.

Latency Tolerance Messaging (LTM)	Latency Tolerance Messaging (LTM) adds the capability for attached devices to provide information that can improve the host platform's ability to select when and how long to sleep. This is accomplished by an attached device sending an LTM, informing the host of its acceptable service latency between accesses, i.e. the device's latency tolerance.
LFPS	Low Frequency Periodic Signal. Refer to USB3 spec.
Link	A USB physical interconnect between two connected ports. A dual-simplex Link consists of a pair of receive and transmit Sublinks. A simplex Link consists of a single bi-directional Sublink. The Sublinks of a dual-simplex Link may be asymmetric in the number of Lanes that they support and Sublink properties for the two directions.
link connection	A “USB3 link connection” refers to the SuperSpeed Rx and Tx signal pairs. A “USB2 link connection” refers to the D+/D- signal pair.
Link Management Packet (LMP)	A type of SuperSpeed header packet used to communicate information between a pair of links.
Link TD	A TD that consists of just one Link TRB.
Link TRB	A Transfer Request Block that is always the last TRB of a TRB Ring Segment. Link TRBs are used to form large, non-contiguous Transfer Rings that cross Page boundaries. Refer to section 4.11.5.1.
Low-speed	USB operation at 1.5 Mb/s. Also see *full-speed*, *high-speed*, *SuperSpeed*, and *SuperSpeedPlus*.
Low-Touch	Low touch registers are referenced infrequently by system software, e.g. only at initialization time, only when a USB device is enumerated, etc.
LS	See *Low-speed*.
Message Pipe	A bi-directional pipe that transfers data using a request/data/status paradigm. The data has an imposed structure that allows requests to be reliably identified and communicated, e.g. a Control endpoint.

Microframe	A 125 microsecond time base established on USB buses by the xHC. Full-speed USB buses utilize an 8 microframe time base.
LTM	See *Latency Tolerance Messaging*.
MMIO	Memory Mapped I/O
MOD	The Modulus function "*dividend* MOD *divisor*" is the remainder of the Euclidean division of the *dividend* by the *divisor*.
MSI	Message Signaled Interrupts. PCI feature that provides vectored interrupts to a single interrupt controller.
MSI-X	Extended Message Signaled Interrupts. PCI feature that provides vectored interrupts to multiple interrupt controllers.
NAK	Handshake packet indicating a negative acknowledgment.
Normal TRB	A Normal Transfer Request Block that is used on transfer Rings to define a single contiguous buffer for a data transfer. Normal TRBs may be “chained” to support scatter/gather or buffer concatenation operations. Refer to section 4.11.2.1.
NRDY	Handshake acknowledgment packet indicating an endpoint is Not Ready to move data.
OHCI	Open Host Controller Interface. Industry defined USB host controller specification for Low-speed and Full-Speed devices.
Optional Normative	If an Optional Normative feature is implemented, it shall comply with the requirements specified for that optional normative feature. The optional normative approach assures interoperability between multiple vendors, by definition, when implementing the same xHCI extensions.
Operational Registers	The Operational Registers specify host controller configuration and runtime modifiable state. And are used by system software to control and monitor the operational state of the host controller.

OSI	An Operating System Instance is the software operating environment that runs in a Virtual Machine. Virtualization allows multiple Operating System Instances to concurrently run within a platform.
Output Device Context	A Device Context data structure pointed to by a Device Context Base Address Array entry.
Output Endpoint Context	An Endpoint Context contained in the Device Context data structure pointed to by a Device Context Base Address Array entry.
Output Slot Context	A Slot Context contained in the Device Context data structure pointed to by a Device Context Base Address Array entry.
Page	A Page refers to the smallest possible size of a block of contiguous physical memory used by a processor architecture that supports paged memory.
0	Peripheral Component Interconnect. Refer to the 0 specification.
PCI Config Space	PCI Configuration Space. A segregated address space that provides a means of identifying and enumerating the host controller by system software.
PCIe	PCI Express. Refer to the PCIe specification.
Periodic Pipe	A “Guaranteed Bandwidth” Pipe defined by an Isoch or Interrupt endpoint.
Pipe	A logical abstraction representing the association between an endpoint on a device and software on the host. A pipe has several attributes; for example, a pipe may transfer data as streams (stream pipe) or messages (message pipe). Throughout this document, term “pipe” is used to generically refer to an endpoint.
Pipe Schedule	An internal xHC construct that identifies the endpoints that currently have work items scheduled for USB.
POST	Power On Self Test - Code executed during a computer's pre-boot sequence.

Power well	Refer to *Aux Power* or *Core Power*.
PSCEG	Port Status Change Event Generation. Refer to section 4.19.3.
Qword	A data element that is eight bytes (64 bits) in size.
Register Space	The Register Space represents the hardware registers presented by the xHC to system software that reside in the Memory Address Space.
Root Hub	A (tier 1) Root Hub is always presented by the xHC. Refer to section 4.19 for more information.
Root Hub Port	The downstream port on a Root Hub.
Scatter/Gather	Scatter/Gather mechanisms are used in Virtual Memory environments to *gather* the non-contiguous physical memory Pages into a contiguous data stream, or to *scatter* a contiguous data stream to non-contiguous physical memory Pages.
Service Interval	The period specified by the bInterval field of the USB Endpoint Descriptor. Service Intervals are always a multiple of microframes (125µs.).
Service Interval Boundary	The point in time defined by the beginning of the first (micro)frame of a Service Interval.
Service Opportunity (SO)	A Service Opportunity is a block of time that the xHC allocates for moving packets on USB, for a specific endpoint. An individual Service Opportunity is limited to the number of packets defined by the Endpoint Context *Max Burst Size* and *Mult* fields, however less packets may be moved in a Service Opportunity.
Service Opportunity Packet Count (SOPC)	The number of packets that the xHC shall schedule during one Service Opportunity. The default value of the SOPC = Endpoint Context (*Max Burst Size* x *Mult*.
set	When used in reference to a flag or field of a data structure or register, a flag shall be set to '1' and field shall be set to a specified value, which may include '0'.

SET_CONFIGURATION	Refers to a standard USB Set Configuration request defined in section 9.4.7 of the USB2 spec.
SET_INTERFACE	Refers to a standard USB Set Interface request defined in section 9.4.10 of the USB2 spec.
Setup Stage TD	A Setup Stage Transfer Descriptor consists of a single Setup Stage TRB. It describes a work item for a control endpoint. Setup Stage TDs are only found on the Transfer Rings associated with Control Endpoints.
Setup Stage TRB	A Setup Stage Transfer Request Block that is always the first TRB of a Setup Stage TD. They are only found on the Transfer Rings associated with Control Endpoints. Refer to section 4.11.2.2.
Slot Context	The Slot Context data structure defines information that applies to the slot, the device as whole, or to all Endpoint Contexts.
Slot ID	Refers to the index of a Device Slot. The Slot Identifier defines a value that is used to index into the Doorbell Array and Device Context Base Address Array. It is a *logical Device Address* that is used for all system software references to a physical USB device attached to the xHC.
SO	See *Service Opportunity*.
SOF	See *Start-of-Frame*.
SOPC	See *Service Opportunity Packet Count*.
SS	See *SuperSpeed*.
SSIC	SuperSpeed Inter-Chip, refer to the SSIC spec.
SSP	See *SuperSpeedPlus*.
Start-of-Frame (SOF)	The first transaction in each USB2 (micro)frame. A SOF allows endpoints to identify the start of the (micro)frame and synchronize internal endpoint clocks to the host.

Stream Pipe	A pipe that transfers data as a stream of samples with no defined USB structure, e.g. an Interrupt, Isoch, or Bulk endpoint.
SR-IOV	PCIe Single Root – I/O Virtualization. Refer to SR-IOV specification.
Sublink	A Sublink consists of one or more Lanes. A simplex Sublink only supports a single bidirectional Lane. A dual-simplex Sublink supports one or more unidirectional Lanes. All Lanes of a multi-lane Sublink are the same speed.
SuperSpeed	USB operation at Gen 1 speed (5 Gb/s). Also see *SuperSpeedPlus*, *low-speed*, *high-speed* and *full-speed*. Refer to the USB3 spec.
SuperSpeedPlus (SSP)	USB operation at Gen 2 speed (10 Gb/s). Also see *SuperSpeed*, *low-speed*, *high-speed* and *full-speed*. However, where specific differences exist between the USB 3.0 and USB 3.1 definition of SuperSpeed features or requirements, those differences will be uniquely identified as SuperSpeedPlus (or SSP) features or requirements.Refer to the USB3 spec.
System Software	A general reference to the software that is responsible for managing the xHCI.
TD	See *Transfer Descriptor*.
TD Transfer Size	The TD Transfer Size is defined by the sum of the *Length* fields in all TRBs that comprise the TD.
Token Packet	A type of packet that identifies what transaction is to be performed on the bus.
Total Available Bandwidth	The Total Available Bandwidth identifies a Bus Instance's ability to move real data. As rule of thumb, the Total Available Bandwidth will be at least 20% lower than the cited bit rate of a Bus Instance, or more depending on the mix of packet sizes. Also note that multiple Root Hub ports may share the bandwidth of a single Bus Instance.
Transaction	The delivery of service to a USB endpoint; consists of a token packet, optional data packet, and optional handshake

	packet. Specific packets are allowed/required based on the transaction type.
Transaction Packet (TP)	Transaction Packets (TPs) are SuperSpeed packets that traverse a path between the host and device. TPs are used to control data flow between devices and the host as well as to manage the end to end connection.
Transaction Translator	A functional component of a USB hub. The Transaction Translator responds to special high-speed transactions and translates them to full/low-speed transactions with full/low-speed devices attached on downstream facing ports.
Transfer	One or more bus transactions to move information between a software client and its function.
Transfer Descriptor (TD)	A Transfer Descriptor defines a single Transfer to a USB device. A TD consists of one or more Transfer Request Blocks. The TRBs of a Multiple-TRB Transfer Descriptor are tied together using the *Chain* flag in the TRB Control component.
Transfer Request Block (TRB)	A TRB is a small, flexible data structure in memory that defines the characteristics of a single DMA operation executed by the xHC.
Transfer Ring	A Transfer Ring is a TRB Ring associated with an Endpoint Context. Each Transfer Ring describes the scheduled work items for a single USB Endpoint.
Transfer Type	Determines the characteristics of the data flow between a software client and its function. Four standard transfer types are defined: control, interrupt, bulk, and isochronous.
TRB	See *Transfer Request Block*.
TRB Ring	A TRB Ring is defined by three parameters: a pointer to the TRB Ring data structure base address, and Enqueue and Dequeue Pointers that define the “active” TRBs in the ring.
TT	See *Transaction Translator*.

U0	Maximum power USB3 link state. The USB3 link is in its full power state and USB 3 device in the “On” power state.
U1, U2	Intermediate USB3 link power state. The link is in an intermediate USB3 Link Power Managed (LPM) state and the USB 3 device in “On” power state.
U3	Lowest USB3 link power state. USB3 device in Suspend state.
UHCI	Universal Host Controller Interface. Intel defined USB host controller specification for Low-speed and Full-Speed devices.
Universal Serial Bus Driver (USBD)	The host resident software entity responsible for providing common services to clients that are manipulating one or more functions on one or more Host Controllers.
Universal Serial Bus Resources	Resources provided by the USB, such as bandwidth and power. Also see *Device Resources* and *Host Resources*.
Upstream	The direction of data flow towards the host. An upstream port is the port on a device electrically closest to the host that generates upstream data traffic from the hub. Upstream ports receive downstream data traffic.
USBD	See *Universal Serial Bus Driver*.
USP	UpStream Port - an SSIC term that “refers to the port of a peripheral to which a host is connected”.
Virtual Intermediary (VI)	A Virtual Intermediaries (VIsUS) describes a mechanism that runs in the VMM, Service VM, or other software entity for sharing devices between virtual platforms. It is assumed that the mechanism shall be invoked and executed on every IO transaction, i.e. generates VM_Enter and VM_Exit events.
Virtualized Environment	A platform software environment that includes a VMM which manages VMs.
VM	Virtual Machine. A Virtual Machine manages a single Operating System Instance (OSI).

VMM	Virtual Machine Manager. A Virtual Machine Manager manages Virtual Machine instances in a virtualized environment.
wMaxPacketSize	Maximum Packet Size value defined by a USB Endpoint Descriptor.
Word	A data element that is two bytes (16 bits) in size.
XactErr	A USB Transaction Error. May be due to a CRC error, a timeout, etc.
xHCI Extended Capabilities	The xHCI Extended Capabilities specify optional features of a xHC implementation, as well as providing the ability to add new capabilities to implementations after the publication of this specification.
xHC instance	A xHC instance is either the physical or virtual version of the xHC presented as a PCIe SR-IOV Physical Function (PF0) or Virtual Function (VF1-n). A xHC implementation that does not support virtualization only presents a single xHC instance to the platform.
Zero-based Value	If a maximum is defined for a range of working values (e.g. 32), a Zero-based Value is a value where the legal range of values is 0 to maximum-1 (e.g. 0 to 31).

1.7 Compliance

Adopters can demonstrate compliance of their product(s) with this specification through the xHCI compliance testing program provided by Intel®. For details on the xHCI compliance testing program, please send email to ssusbcompliance@usb.org.

The xHCI Compliance Test Suite provides an excellent reference of software expectations when communicating with the xHCI and a concise list of the test validation assertions associated with this specification.

1.8 Documentation Conventions

1.8.1 Capitalization

Some terms are capitalized to distinguish their definition in the context of this document from their common English meaning. Words not capitalized have their common English meaning. When terms such as “memory write” or “memory read” appear completely in lower case, they include all transactions of that type.

Register names and the names of fields and bits in registers and headers are presented with the first letter capitalized and the remainder in lower case.

1.8.2 Bold Text

Terms or names in bold text indicate the sentence provides a basic xHCI definition of the respective term/name. All other references to an xHCI defined term/name use the exact same text string as the definition so that you can search on it easily. Refer to section 1.5 for more information on searching.

1.8.3 Italic Text

Italic text is used to identify Capitalized names that are explicitly named xHCI; registers, register fields, or flags in registers.

1.8.4 Numbers and Number Bases

Hexadecimal numbers are written with a lower case “h” suffix, e.g., FFFFh and 80h. Hexadecimal numbers larger than four digits are represented with a space dividing each group of four digits, as in 1E FFFF FFFFh. Binary numbers are written with a lower case “b” suffix, e.g., 1001b and 10b. Binary numbers larger than four digits are written with a space dividing each group of four digits, as in 1000 0101 0010b.

All other numbers are decimal.

1.8.5 Implementation Notes

Implementation Notes should not be considered to be part of this specification. They are included for clarification and illustration only. Implementation Notes within this document are enclosed in a box and set apart from other text.

1.8.6 Word Usage

The word *shall* is used to indicate mandatory requirements strictly to be followed in order to conform to the xHCI specification and from which no deviation is permitted (*shall* equals *is required to*).

The use of the word *must* is deprecated and shall not be used when stating mandatory requirements; *must* is used only to describe unavoidable situations.

The use of the word *will* is deprecated and shall not be used when stating mandatory requirements; *will* is only used in statements of fact.

The word *should* is used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended that*).

The word *may* is used to indicate a course of action permissible within the limits of the standard (*may* equals *is permitted*).

The word *can* is used for statements of possibility and capability, whether material, physical, or causal (*can* equals *is able to*).

The abbreviation *i.e.* is for the Latin phrase *id est* which means *that is*.

The abbreviation *e.g.* is for the Latin phrase *exempli gratia* which means *for example*.

1.8.7 Pseudo Code

Throughout this document pseudo code is used to illustrate operating principals.

Comments are demarcated by the double forward slashes "**//**".

The pseudo code conventions include:

If/else condition statements:

> **If** conditions**:**
>
> // true operations

```
    else
    // false operations
And For loops:
    For conditions:
    // operations
```

1.8.8 Other Notation

The symbol combination “=>” shall be read as “transitions to”. e.g. OCA => ‘1’ means the value of OCA transitions to ‘1’.

§ §

2 Introduction

2.1 Motivation

The development of the eXtensible Host Controller Interface was driven by 3 key factors; *Speed*, *Power Efficiency*, and *Virtualization*.

Speed	The storage capacities of portable devices have been increasing with Moore's Law. Vendors of these devices need high performance interfaces so that these high capacity devices can be loaded in reasonable amounts of time. The SuperSpeed support of the xHCI addresses this need.
Power Efficiency	When USB was originally developed, it was targeted at desktop platforms and performance was the primary objective, which meant that host power consumption was not an important consideration. Since then, mobile platforms have become the platform of choice, and their batteries have made host power consumption and idle time efficiency key considerations. The xHCI elimination of the host memory based transaction schedules and its support for the advanced USB3 power management features are key to providing more power efficient platforms without sacrificing performance.
Virtualization	Virtualization is beginning to play a key role in system architectures and the legacy USB host controller architectures exhibit some serious shortcomings when applied to virtualized environments. Legacy USB host controller interfaces define a data pump; where critical state related to overall bus management (Bandwidth allocation, Address assignment, etc.) reside in the software driver. Trying to apply the standard hardware IO virtualization technique, of replicating IO interface registers, to the legacy USB host controller interface is problematic because critical state that must be managed across Virtual Machines (VMs) is not available to hardware. The xHCI architecture moves the control of this critical state into hardware, enabling USB resource management across VMs. The xHCI virtualization features also provide for: 1) Direct-Assignment of individual USB devices (irrespective of their location in the bus topology) to any VM, 2) minimizing run-time inter-VM communications, and 3) support for native USB device sharing.

The eXtensible Host Controller Interface addresses these factors. In addition, the xHCI architecture provides a new industry standard means for interfacing to USB devices that delivers the extensibility necessary to meet future needs.

2.2 Goals

The goal of xHCI architecture is to define a USB host controller to ultimately replace UHCI/OHCI/EHCI, to provide highly power efficient operation, higher performance, and extensibility to new USB specifications, such as USB3 and beyond. Key xHCI architectural goals are:

- Efficient operation – idle power and performance better than current USB host controller architectures.
- A device level programming model that is fully consistent with the existing USB software model
- Decouple the host controller interface presented to software from the underlying USB protocols
- Minimize host memory accesses, fully eliminating them when USB devices are idle
- Eliminate the “Companion Controller” model
- Enable hardware “fail-over” modes in system resource constrained situations so devices are still accessible, but perhaps at less optimal power/performance point
- Provide the ability for different markets to differentiate hardware capabilities, e.g. target host controller power, performance and cost trade-offs for specific markets
- Define an extensible architecture that provides an easy path for new USB specifications and technologies, such as higher bandwidth interfaces, optical transmission medium, etc., without requiring the definition of yet another USB host controller interface

2.3 Key features

Robust Support for all USB 3.x Features. This specification describes a host controller architecture that is capable of supporting compliant USB 3.x SuperSpeedPlus and SuperSpeed devices. This includes new USB 3.x features such as asynchronous transactions and other extensions to the protocol.

Support for all USB device speeds. The xHCI specification defines support for all USB device speeds including; USB 2.0 Low-, Full-, and High-speed devices, and USB 1.1 Low- and Full-speed.

System Power Management. Current PC architectures are providing ubiquitous support for aggressive power management. The USB3 architecture focuses on power conservation to improve battery life in mobile, battery powered applications. USB2 LPM (Link Power Management) extensions are also supported by the xHCI. Special attention has been paid to minimizing power consumption when the system is Idle. USB is a critical component in delivering a consistent, coherent and robust user experience. If the implementation includes PCI configuration registers, then the host controller is required to implement a PCI Power Management Interface (PCI PM).

Provides simple, robust solutions for legacy USB host controller issues. The xHCI specification enables solutions to a myriad of issues, which have proven to be problematic for USB host controllers. Some of the issues resolved in the xHCI specification include: Memory thrashing, Memory access efficiency, and conflicts with CPU power management. The xHCI architecture provides both new specific features and optimizations to its architecture to solve the legacy issues.

Optimized for Best Memory Access Efficiency. The xHCI's data transfer model eliminates the memory based transaction schedules that existed in previous host controller architectures. It utilizes Transfer level operations to decrease the average number of memory accesses required to execute USB operations.

Minimized Hardware Interface Complexity. The xHCI provides a simple interface for software to provide the host controller with parameterized Transfer Requests that the host controller uses to execute transactions on the USB. The interface allows software to asynchronously add work to the interface while the host controller is executing, without requiring the use of software synchronization primitives.

Support for 32 and 64-bit Addressing. Over the implementation lifetime of this specification, it is expected that xHCI controllers will be used increasingly in architectures that support more than 32-bits of addressable memory space. The xHCI inherently supports up to 64-bits of addressing.

Support for Virtual Memory. All xHCI register and data structures are designed to support the "coarse-grain" Scatter/Gather requirements of page based virtual memory architectures.

Support for "fine-grain" Scatter/Gather. The interface supports a hardware scatter/gather method for all data transfers that may be used for accessing memory. The EHCI scatter/gather mechanism was an example of Coarse Grain scatter/gather. It was tailored specifically to work with page based virtual memory, specifying a Start Offset, a Transfer Length and a list of Page aligned addresses. The xHCI scatter/gather mechanism is not constrained by memory page boundary or size limitations. xHCI scatter/gather lists may be comprised of buffers starting on any byte boundary and any byte length. This feature allows the xHCI scatter/gather mechanism to be used for accessing page aligned data, as well as at the application level to minimize software data copies.

Support for Virtualization. Through use of the PCIe SR-IOV specification, the xHCI provides a Virtual Machine Manager with the ability to enable Virtual xHCs

(VxHCs) controllers, and assign any USB Device to any VxHC instance. Virtualization support is an optional normative xHCI feature.

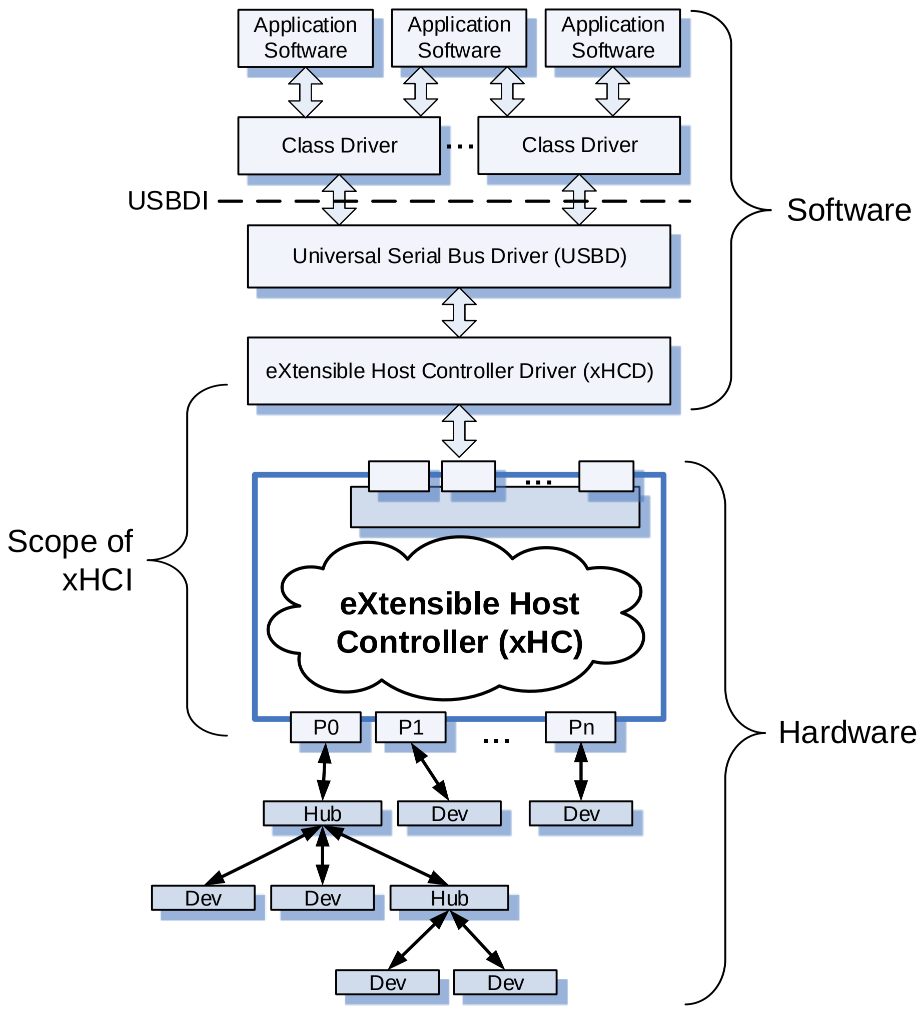
2.4 xHCI Product Compliance

Adopters and Contributors of the eXtensible Host Controller Interface Specification for Universal Serial Bus (xHCI) have signed the eXtensible Host Controller Interface (xHCI) Specification Contributor Agreement in order to be licensed to use and implement this Specification. This Contributors Agreement provides Contributors and Adopters with a reciprocal, royalty-free license to certain intellectual property rights from Intel and other Adopters and Contributors for their products that are compliant with the xHCI specification. Adopters and Contributors can demonstrate compliance with the Specification through the testing program as defined by Intel.

3 Architectural Overview

A USB Host System is composed of a number of hardware and software layers. Figure 3-1 illustrates a conceptual block diagram of the building block layers in a host system that work in concert to support USB 3.x.

Figure 3-1: Universal Serial Bus, Revision 3.x System Block Diagram



The component layers are:

- **Application Software.** This software uses the services provided by one or more USB devices. Application software interfaces with USB devices through standardized interfaces provided by the Class Drivers.

- **Class Driver Software.** This software executes on the host PC corresponding to a particular "class" of USB device (Mass Storage, Human Interface, Audio, etc.). Class Driver software is typically part of the operating system or provided with the USB device.
- **USB Driver (USBD).** The USBD is a system software Bus Driver that abstracts the details of the particular Host Controller Driver for a particular operating system. The generic USB interface presented to the system by USBD is referred to as the *USB Driver Interface* or the **USBDI**.
- **Host Controller Driver (xHCD).** xHCD provides the software layer between the Host Controller hardware and the USBD. The details of the host controller driver depend on the host controller hardware register interface definition.
- **Host Controller (xHC).** The host controller is the specific hardware implementation of the host controller architecture. There is one host controller specification for the USB 3 host controller, which enables support for Low-, Full-, High-, SuperSpeed, and SuperSpeedPlus devices. The interface presented by the xHC to the system is referred to as the *eXtensible Host Controller Interface* or the **xHCI**.
- **USB Device.** This is a hardware device that expands the bus topology (hub) or performs a useful end-user function. Interactions with USB devices flow from the applications through the software and hardware layers to the USB devices.

A key feature of the USB architecture is the **Device Framework** that it presents. The Device Framework defines the interface between a USB device and a Class Driver, which is independent of the particular host controller interface that a system employs to communicate with the USB. This interface consists of a Default Pipe, and zero or more additional class defined Pipes. The Default Pipe (also referred to as the *Default Control Endpoint*) is used to enumerate and manage a USB device. It can also be used to provide access to application specific features of the device. The class defined Pipes provide specialized Quality-of-Service requirements to perform device class specific functions.

The Device Framework allows the USB architecture to separate the details of the "Bus" interface from that of the application specific ("Device") interface, resulting in a split driver model (xHCD/Class Driver). Note that in this context, Device Class refers to the portion of a USB device that performs some useful end user application specific function (e.g. Mass Storage, Audio, Human Interface, etc.).

The USB bus driver (USBD) provides a standard method of interfacing to the transport mechanisms (USB Framework) defined by the USB architecture (Isoch, Interrupt, Control, and Bulk Pipes) and the Device Class driver is where all the application specific knowledge resides. A Class Driver will also include any “value add” that a vendor may provide. As long as the USB Framework presented through the USBDI remains unchanged, the USB Class Drivers do not have to change because the USB bus driver does (e.g. to support the xHCI).

Working groups in the USB-IF have defined several standard USB Device Classes (Mass Storage, Audio, etc.). A USB device vendor may choose to define a proprietary Device Class for their product or utilize part or all of an appropriate USB-IF defined Device Class. The USB-IF defined Device Classes provide a baseline set of features, for their respective class. Several USB Device Classes are supported natively by today’s Operating Systems.

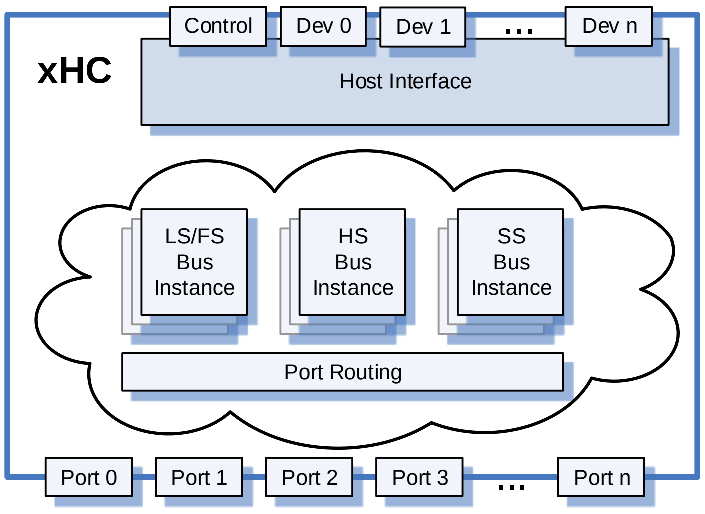
Native OS support for Device Classes allows a compliant device to provide a user with basic functionality if the vendor Device Class drivers are not available, however a vendor can define their own Class Driver to add value. Many commodity USB device vendors (mice, keyboard, etc.) take advantage of those provided by OS vendors and don’t bother to offer their own Class Drivers. If a vendor offers a USB device that does not fall under one of the standard USB defined Device Classes supported by an OS then they shall offer their own Class Driver.

The xHCI is used for all communications to devices connected through the Root Hub ports of the USB 3 host controller.

The xHCI architecture allows the USB 3 host controller to provide USB functionality for all speed devices without requiring, as in previous generations, companion controllers along with the associated software support for their respective drivers. The enhanced features of the xHCI architecture are key to delivering this simplified operating environment.

Note that Figure 3-2 does not imply a particular xHC implementation, however the functional partitioning that it illustrates is useful for this discussion. The Host Interface Logic manages the Registers and DMA associated with the xHC.

Figure 3-2: USB 3 EXtensible Host Controller



The xHC always manages the respective speed USB devices connected to its Root Hub ports. Depending on the implementation, the resources of a USB bus instance (bandwidth, device addressability, etc.) may be presented on each root hub port, shared across multiple root hub ports, or a combination of allocations.

This specification defines the registers and interfaces for the eXtensible Host Controller Interface.

3.1 Interface Architecture

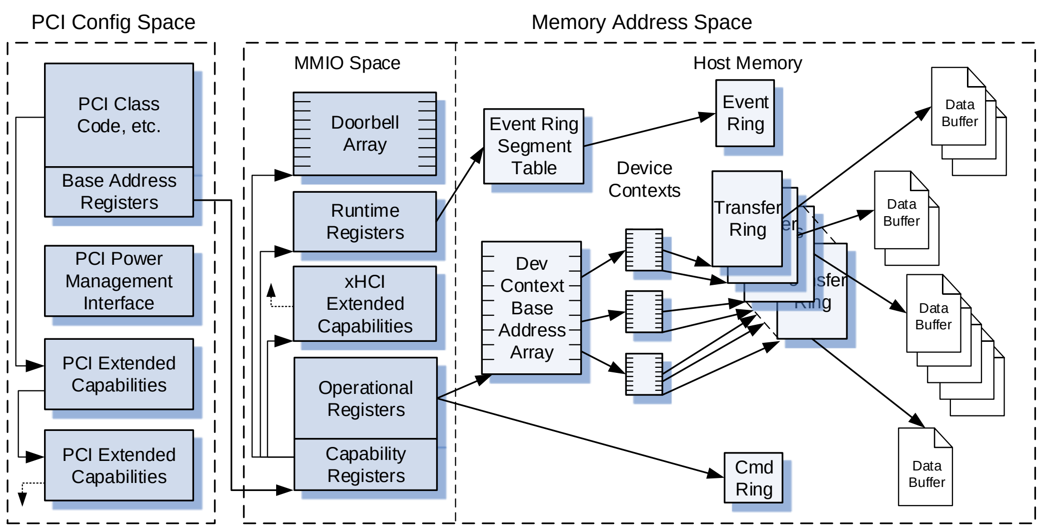
The xHCI interface defines three interface spaces (refer to Figure 3-3):

- **Host Configuration Space.** Every xHC implementation shall include a means of identifying and enumerating the host controller by system software. This specification provides a PCI example of the Host Configuration Space, which is referred to as **PCI Config Space**. The PCI Config Space definition provides a working example of configuration space use for system xHC enumeration and resource (interrupt, power, virtualization, etc.) management.

- **MMIO Space.** The Register Space represents the hardware registers presented by the xHC to system software that reside in the Memory Address Space. The Register Space provides for the implementation-specific parameters defined in the xHCI normal and Extended Capabilities registers, the Operational and Runtime control and status registers, and the Doorbell Array used to flag accesses to individual USB devices. This space, normally referred to as I/O space, is implemented as Memory-Mapped I/O (**MMIO**) space.

- **Host Memory.** This space is defined by the control data structures (Device Context Base Address Array, Device Contexts, Transfer Rings, etc.) and data buffers that are allocated and managed by the xHC Driver to enable the endpoint traffic of individual devices. This space is allocated in the Kernel and User areas of the Memory Address Space.

Figure 3-3: General Architecture of the eXtensible Host Controller Interface



The xHCI provides support for two categories of USB transfer types: asynchronous and periodic. Isochronous and Interrupt transfers are Periodic transfer types. Asynchronous transfer types include Control and Bulk. Figure 3-3 illustrates that the xHCI provides a homogeneous mechanism (Transfer Rings) for each category of transfer type.

The USB Base Address Register (BAR) in the PCI Config Space points to the base address of the xHC register interface. The xHC register interface consists of 4 major components: Capability Registers, Operational Registers, Runtime Registers, and the Doorbell Array. The Operational and Capability Registers are concatenated in MMIO space. The Runtime Registers are actually just an extension of the Operational Registers. Their partitioning allows the xHC to better support virtualization, by allowing the Runtime Registers to reside on a separate page boundary. A xHCI Capabilities Pointer mechanism (similar to that defined by PCI) is presented in the Capability Registers to point to new or optional capabilities of an xHC implementation.

The ***Capability Registers*** specify read-only limits, restrictions and capabilities of the host controller implementation. These values are used as parameters to the host controller driver.

The ***Runtime*** and ***Operational Registers*** specify host controller configuration and runtime modifiable state, and are used by system software to control and monitor the operational state of the host controller. These registers are partitioned as a function of those that are heavily accessed during runtime and those that are accessed only at initialization time or only lightly during runtime to better support virtualization of the xHCI.

The ***xHCI Extended Capabilities*** specify optional features of an xHC implementation, as well as providing the ability to add new capabilities to implementations after the publication of this specification.

The ***Doorbell Array*** is an array of up to 256 Doorbell Registers, which supports up to 255 USB devices or hubs. Each ***Doorbell Register*** provides system software with a mechanism for notifying the xHC if it has Slot or Endpoint related work to perform. A *DB Target* field in the Doorbell Register is written with a value that identifies the reason for **"ringing"** the doorbell. Doorbell Register 0 is allocated to the Host Controller for Command Ring management.

The term ***Device Slot*** is used as a generic reference to a set of xHCI data structures associated with an individual USB device. Each device is represented by an entry in the *Device Context Base Address Array*, a register in the *Doorbell Array* register, and a device's *Device Context*. The term ***Slot ID*** refers to the index used to identify a specific *Device Slot*. For example the value of *Slot ID* will be used as an index to identify a specific entry in the *Device Context Base Address Array*.

The ***Device Context Base Address Array*** supports up to 255 USB devices or hubs, where each element in the array is a pointer to a *Device Context* data structure.

The **Command Ring** is used by software to pass device and host controller related commands to the xHC. The *Command Ring* shall be treated as read-only by the xHC. Refer to section 4.9.3 for a discussion of Command Ring Management.

The **Event Ring** is used by the xHC to pass command completion and asynchronous events to software. The *Event Ring* shall be treated as read-only by system software. Refer to section 4.9.4 for a discussion of Event Ring Management.

A ***Transfer Ring*** is used by software to schedule work items for a single USB Endpoint. A Transfer Ring is organized as a circular queue of ***Transfer Descriptor*** (TD) data structures, where each *Transfer Descriptor* defines one or more Data Buffers that will be moved to or from the USB. *Transfer Rings* are treated as read-only by the xHC. Refer to section 4.9.2 for a discussion of Transfer Ring Management.

All three types of rings support the ability for system software to grow or shrink them while they are active. Special TDs written to the Transfer and Command rings allow software to change their size, however since the Event Ring is read-only to software, the **Event Ring Segment Table** is provided so that software may modify its size.

3.2 xHCI Data Structures

The xHC is expected to run in virtual memory environments where the size of a contiguous block of physical memory will be limited by the Page size of the system. The data structures that the xHC uses to manage devices and endpoints are designed to accommodate this limitation, by either keeping the data structure under 4K Bytes (the minimum Page size supported), or providing mechanisms to link non-contiguous blocks of physical memory to form larger, logically contiguous data structures, e.g. circular queues of data structures that point to the data buffers used for transferring USB data to or from the host. The data buffers referenced by these data structures may be byte aligned and reference from 1 to 64K bytes of contiguous physical data.

3.2.1 Device Context Base Address Array

The *Device Context Base Address Array* (DCBAA) provides the xHC with a Slot ID based lookup table for accessing the Device Context data structure associated with each slot. This data structure consists of an array of pointers to Device Context data structures. When a device attach is detected: system software initializes a Device Context data structure, requests a Slot ID from the xHC, and inserts a pointer to the newly created Device Context into the DCBAA at the location indicated by the Slot ID.

Note that the first entry (Slot ID = '0') in the Device Context Base Address Array is utilized by the xHCI Scratchpad mechanism. Refer to section 4.20 for more information.

3.2.2 Device Context

The Device Context data structure is managed by the xHC and used to report device configuration and state information to system software. The Device Context data structure consists of an array of 32 data structures. The first context data structure (index = '0') is a *Slot Context* data structure (6.2.2). The remaining context data structures (indices 1-31) are *Endpoint Context* data structures (6.2.3).

As part of the process of enumerating a USB device, system software allocates a *Device Context* data structure for the device in host memory and initializes it to '0'. Ownership of the data structure is then passed to the xHC with an *Address Device Command*. The xHC maintains ownership of the *Device Context* until the device slot is disabled with a *Disable Slot Command*. The *Device Context* data structure shall be treated as read-only by system software while it is owned by the xHC.

3.2.3 Slot Context

The Slot Context data structure contains information that relates to the device as a whole, or affects all endpoints of a USB device. This data structure is defined as a member of the *Device Context* and *Input Context* data structures. Refer to section 3.2.5 for information on the *Input Context* data structure.

The information provided by the *Slot Context* includes; control, state, addressing, and power management. The *Slot States* reported by the xHC identify the current state of a device and map closely to the USB Device States described in the USB specification. The addressing information is used for a variety of purposes; The *USB Device Address*, assigned by the xHC, is available for developers to trace device related USB activity with a bus analyzer. The *Route String* is used by the xHC to target SuperSpeed packets. And the *Speed*, *TT Port Number*, and *TT Hub Slot ID* fields allow the xHC to execute the split transactions necessary to address low- and full-speed devices attached to high-speed hubs. The power management information includes the *Max Exit Latency*, used by the xHC to determine the scheduling of Isoch packets on the bus.

As a *Device Context* member, the *Slot Context* data structure is used by the xHC to report the current values of device parameters to system software. The *Slot Context* data structure of a *Device Context* is also referred to as "Output *Slot Context*".

As an *Input Context* member, the *Slot Context* data structure is used by system software to pass command parameters to the host controller. The *Slot Context* data structure of an *Input Context* is also referred to as "Input *Slot Context*". If a command targeted at a Device Slot is successful, the xHC will update the Output *Slot Context* to reflect the parameter values that it is actively using to manage the device prior to generating a *Command Completion Event*.

An *xHCI Reserved* area of the *Slot Context* is available as an xHC implementation defined scratchpad.

All *Reserved* fields in the Slot Context are for the exclusive use of the xHC and shall not be modified by system software except when the Slot is in the *Disabled* state.

3.2.4 Endpoint Context

The Endpoint Context data structure defines the configuration and state of a specific USB endpoint. This data structure is defined as a member of the *Device Context* and *Input Context* data structures. Refer to section 3.2.5 for information on the *Input Context* data structure.

Most of the fields of the *Endpoint Context* contain endpoint related type, control, state, and bandwidth information, that correspond to the information in the associated endpoint related descriptors reported by the device. An Endpoint Context also defines a *TR Dequeue Pointer* field, which normally provides a pointer to the *Transfer Ring* associated with the pipe. There is a special case for USB3 Bulk endpoints where *Streams* may be associated with an endpoint. **Streams** allow the data stream of an endpoint to be multiplexed between Transfer Rings by the device (refer to section 4.12 for more information on Streams). In this case, a level of indirection is introduced to access the *Transfer Rings* associated with the endpoint, and the Endpoint Context *TR Dequeue Pointer* field contains a pointer to a *Stream Context Array* data structure (commonly referred to as a **Stream Array**), where each *Stream Context* data structure in the array may contain a NULL pointer (if the Stream ID is not assigned) or point to the *Transfer Ring* or another *Stream Context Array* associated with the respective Stream.

Note that the *Device Context* and *Input Context* data structures provide for all possible (31) endpoints that can be declared by a USB device. Most devices declare only a small number of endpoints, which means that many of the *Endpoint Context* data structures in a *Device Context* or *Input Context* may be unused.

The *Endpoint Context* also contains some fields that are helpful in debugging the transfer operations associated with the pipe. An *Error Counter* (CErr) may be used to force unlimited retries of USB transactions.

As a *Device Context* member, the *Endpoint Context* data structure is used by the xHC to report the current values of endpoint related parameters to system software. In this document the *Endpoint Context* data structure of a *Device Context* is also referred to as "Output *Endpoint Context*".

As an *Input Context* member, the *Endpoint Context* data structure is used by system software to pass endpoint related command parameters to the host controller. In this document the *Endpoint Context* data structure of an *Input Context* is also referred to as "Input *Endpoint Context*". If a command referencing an Input Context is successful, the xHC will update the Output *Endpoint Context* to reflect the parameter values that it is actively using to manage the endpoint prior to generating a *Command Completion Event*.

An *xHCI Reserved* area of the *Endpoint Context* is available as an xHC implementation defined scratchpad.

3.2.4.1 Stream Context Array

A *Stream Context Array* is employed to define the Transfer Rings of a USB3 endpoint that supports Streams. A *Stream Context Array* consists of *Stream Context* data structures. The number of *Stream Context* data structures in a

Primary Stream Context Array and its location are defined by fields in the parent *Endpoint Context*.

Figure 4-20 illustrates how a *Stream Context Array* may be used to extend the number of Transfer Rings that are supported by an endpoint.

3.2.4.1.1 Stream Context

The Stream Context data structure provides a pointer to the Stream's Transfer Ring and provides some opaque (scratchpad) space for the xHC.

3.2.5 Input Context

The *Input Context* data structure is used by system software to define device configuration and state information that will be passed to the xHC by an *Address Device*, *Configure Endpoint*, or *Evaluate Context Command*. It consists of an *Input Control Context* data structure, followed by a *Slot Context*, and 1-31 *Endpoint Context* data structures. The *Input Control Context* data structure qualifies which of the remaining contexts are affected by the command. After a command is complete, software may reuse or free the *Input Context* data structure.

Throughout this document *Slot Context* or *Endpoint Context*s contained in an *Input Context* are also referred to as “Input” Slot or Endpoint Contexts.

Refer to section 6.2.5 for more information on the *Input Context*.

3.2.5.1 Input Control Context

The *Input Control Context* data structure contains two groups of flags (*Drop* and *Add*) organized as bit vectors. The interpretation of these flags is command dependent, but generally they are used to indicate which endpoints are affected by the command and how.

For example: to set up the xHC to support a particular USB device configuration, software will initialize the *Endpoint Context* data structures of an *Input Context* with the target endpoint configuration information, insert a *Configure Endpoint Command* on the Command Ring that points to the Input Context, and ring the Host Controller Doorbell. The Input *Endpoint Context* information would include: type, Max Packet Size, Interval, etc. The *Add* flags in the *Input Control Context* indicate which endpoints software wants to be added to the xHC's list of valid endpoints, i.e. which Input *Endpoint Context*s are valid. If the command is successful, the endpoint information in the *Input Context* is copied by the xHC to the respective contexts in the *Device Context* and the xHC will set the state of those endpoints to *Running* and begin listening to their doorbells.

Refer to section 6.2.5.1 for more information on the *Input Control Context*.

3.2.6 Rings

A Ring is a circular queue of data structures. Three types of Rings are used by the xHC to communicate and execute USB operations:

- Command Ring
    - One for the xHC
- Event Ring
    - One for each Interrupter (refer to section 4.17)
- Transfer Ring
    - One for each Endpoint or Stream

The Command Ring is used by system software to issue commands to the xHC.

The Event Ring is used by the xHC to return status and results of commands and transfers to system software.

Transfer Rings are used to move data between system memory buffers and device endpoints.

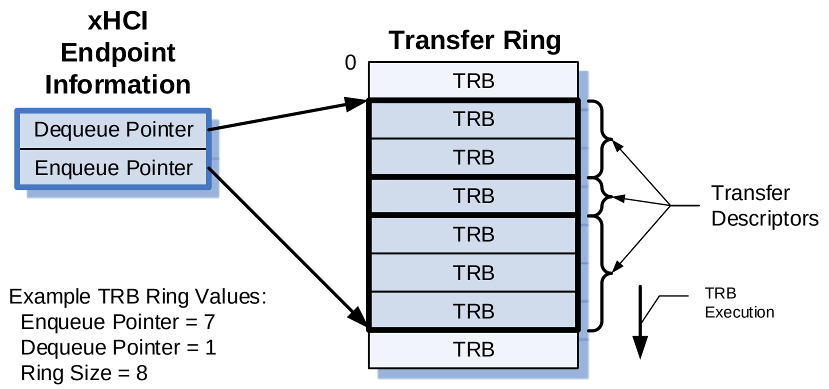
Below is a description of the operation of a Transfer Ring. All ring types employ the same basic mechanisms to transfer information between the xHC and host memory.

3.2.6.1 Transfer Ring Example

Transfers to and from the Endpoint of a USB device are defined using a **Transfer Descriptor** (TD), which consists of one or more **Transfer Request Blocks** (TRBs, refer to sections 4.11 and 6.4). Transfer Descriptors are managed through **Transfer Rings** that reside in host memory. A *Chain* flag in the TRB is used to identify the TRBs that comprise a TD. Therefore, a TD refers to a consecutive set of TRB data structures on a Transfer Ring, where the *Chain* flag is set in all but the last TRB of a TD. Note that a TD may consist of a single TRB, whose *Chain* flag shall not be set.

A Transfer Ring exists for each active endpoint or Stream declared by a USB device. Transfer Rings contain “Transfer” specific TRBs. Section 4.11.2 for more information on Transfer TRBs.

Figure 3-4: Transfer Ring[2]



In the simplest case, software defines a Transfer Ring by allocating and initializing a memory buffer for it, then setting the Enqueue and Dequeue Pointers to the address of this memory buffer and writing it into the TR Dequeue Pointer field of the associated Endpoint or Stream Context. Each memory buffer that comprises a Transfer Ring is called a Segment. Multiple Segments may be linked together to form large rings, and Segments may be added or removed from a ring during runtime. A Transfer Ring is empty when the Enqueue Pointer equals the Dequeue Pointer.

Note: The Transfer Ring *Enqueue* and *Dequeue Pointers* are *not* accessible through physical xHC registers. They are logical entities, maintained internally by both system software and the xHC. Refer to section 4.9.2 for more information on *Enqueue* and *Dequeue Pointers*.

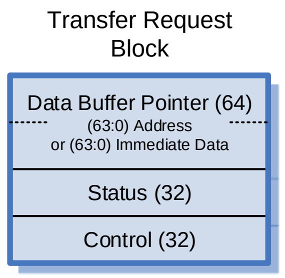
After a Transfer Ring is initialized Transfer Descriptors (comprised of one or more TRBs) may be placed on it.

A "ring" is formed by the placement of a special Link TRB at the end of a Transfer Ring which jumps the TRB execution back to its beginning.

3.2.7 Transfer Request Block

[2]When the *Dequeue* and *Enqueue Pointers* are equal the Transfer Ring is empty. The *Dequeue Pointer* identifies the address of the next TRB to be executed by the xHC. The *Enqueue Pointer* identifies the address of the next TRB location available to software for queuing a TD. TRBs between the *Dequeue* and *Enqueue Pointers* are owned by the xHC.

Figure 3-5: Transfer Request Block



A *Transfer Request Block* (**TRB**) is a data structure constructed in memory by software to transfer a single physically contiguous block of data between host memory and the xHC. TRBs contain a single Data Buffer Pointer, the size of the buffer, and some additional control information.

3.2.7.1 Operation

For small, single buffer operations (of which many are required in the USB protocol) a TD will be composed of a single TRB. For large multi-buffer operations (e.g. Scatter/Gather), TRBs can be chained to form a complex TD. The small size of the TRB data structure allows up to 256 individual buffers to be defined in a 4K Segment (page of memory).

The longer a system is running, the harder it is to find contiguous pages in physical memory. If due to runtime changes in workload demands, hot-plug events, etc., the host needs to increase the size of an existing Transfer Ring or allocate a multi-page Transfer Ring, then a special *Link TRB* may be used to extend a ring to include additional non-physically contiguous Segments.

The **Data Buffer Pointer** field of a TRB provides byte granularity for data addressing.

The **Length** field, which resides in the Status Dword, identifies the size of the buffer referenced by the Data Buffer Pointer. The maximum value the Length field may contain is 64K. When *Length* bytes are transferred, the next TRB in the ring is automatically accessed by the xHC. It is system software's responsibility to ensure that the *Length* field is consistent with any Page crossings that may be encountered.

The **Control** Dword in the TRB shall contain a *TRB Type* field and may contain one or more of the following fields: *Chain* (CH), *Interrupt On Completion* (IOC), *Immediate Data* (IDT), *No-Snoop* (NS), *Interrupt-on Short Packet* (ISP), *Start Isoch ASAP* (SIA), and *Frame ID*. Refer to section 6.4.1 for more information on the contents and use of the Transfer TRB **Control** Dword.

Figure 3-6: Simple Transfer Example

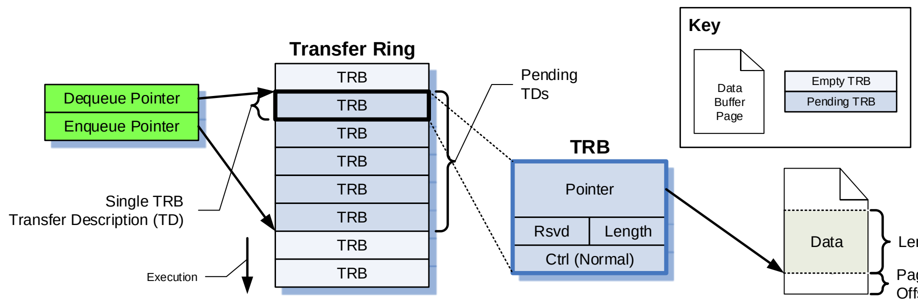


Figure 3-6 illustrates a Transfer *TRB Ring* with multiple pending TDs. The ***Enqueue Pointer*** identifies the next TRB location available to system software for scheduling work (TDs) to the Ring. The ***Dequeue Pointer*** identifies the next TRB in the Transfer Ring to be executed by the xHC. Upon completion of a Transfer TRB, the *Length* and *Status* of the transfer may optionally be reported in a *Transfer Event TRB*. Refer to section 6.4.2.1 for more information on the *Transfer Event TRB*.

Note: A Transfer Ring may include an *Event Data TRB*. Rather than pointing to a Data buffer this TRB contains a 64-bit value which software may use to tag a TD and generate a special Transfer Event to pass that tag back to software when the TD is complete. Refer to section 4.11.5.2 for more information.

3.2.7.2 Other Rings

In addition to the Transfer Ring, the xHCI utilizes a Command and Event Rings. These rings are described later in this document. All xHCI ring types support the ability of software to grow or shrink them while the xHC is actively using them.

3.2.8 Scatter/Gather Transfers

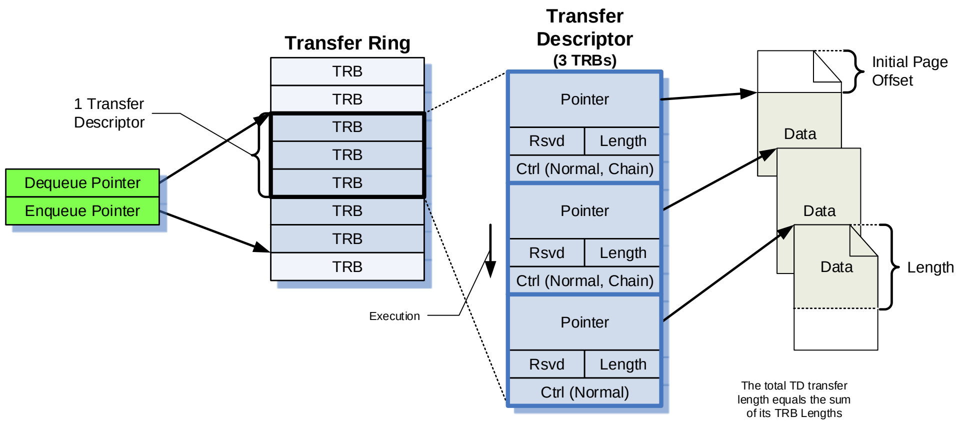
Virtual Memory environments divide physical memory into Pages, and use Page Tables to make non-contiguous physical memory appear contiguous in User "virtual" address space. Scatter/Gather mechanisms are typically used to concatenate the non-contiguous physical memory Pages into a contiguous data stream to present to a device. In this case, the host builds a Multi-TRB TD to define the contiguous virtual memory seen by the User. Because the block of User memory to be transferred often does not start on a Page boundary, the *Data Buffer Pointer* of the first TRB of a Multi-TRB TD may not point to a Page boundary (and the *Length* field of that TRB will be less than a Page Size). Subsequent TRBs of the TD will point to Page boundaries and be Page Size in length, respectively, defining full Pages of data, except for the last TRB, whose *Data Buffer Pointer* will point to a Page boundary but may have a *Length* value less than the Page Size.

Transfers that are comprised of non-contiguous data (e.g. cross memory Page boundaries) are referred to as *Scatter/Gather Transfers*. Chained TRBs are used to provide the additional pointers that are required to define a Scatter/Gather Transfer. A sequence of “chained” TRBs form a *Multi-TRB Transfer Descriptor*. The *Chained* bit in the TRB *Control* word is set in all TRBs, except the last one of a **Multi-TRB TD**. Chained TRBs are always contiguous in a Transfer Ring.

Software shall never update the Enqueue Pointer (that is, toggle the Cycle bit of a TRB) until all TRBs between the previous and the new Enqueue Pointer location are fully formed. It is the responsibility of system software to ensure that the TDs are correctly formed, i.e. the TRBs of a TD are contiguous in the Transfer Ring and correctly chained.

The size of a Scatter/Gather Transfer is equal to the sum of the *Length* fields all the TRBs of a TD.

Figure 3-7: Scatter/Gather Transfer Example



In the figure above note that the *Chain* bit (CH) is set in all but the last TRB of the Multi-TRB TD. The xHC parses the TRBs in the Multi-TRB TD from the Dequeue Pointer towards the Enqueue Pointer (top to bottom in this figure) to form a concatenated data buffer from separate buffers that reside in memory. If the Transfer Ring was associated with an OUT Endpoint then the concatenated data buffer would be sent to the USB Device as single transfer.

Note that no constraints are placed on the TRB *Length* fields in a Scatter/Gather list. Classically all the buffers pointed to by a scatter gather list were required to be “page size” in length except for the first and last (as illustrated by the example above). The xHCI does not require this constraint. Any buffer pointed to by a Normal, Data Stage, or Isoch TRB in a TD may be any size between 0 and 64K bytes in size. For instance, if when an OS translates a virtual memory buffer into a list of physical pages, some of the entries in the list reference multiple contiguous pages, the flexible Length fields of TRBs allow a 1:1 mapping of list entries to TRBs, i.e. a multi-page list entry does not need to be defined as multiple page sized TRBs.

3.2.9 Control Transfers

Several features of a *Control Endpoint* require that it be handled differently than other USB endpoint types. In particular a Control Endpoint defines a *Message Pipe*, while all other endpoint types are *Stream Pipe*s.

A USB *Message Pipe* is bidirectional and transfers data using the USB setup/data/status stage paradigm. The data has an imposed structure that allows requests to be reliably identified and communicated. A USB *Stream Pipe* (Isoch, Interrupt, and Bulk endpoint) transfers data as a stream of samples with no defined USB structure.

USB Control transfers minimally require two transaction stages on the bus: Setup and Status. A control transfer may optionally contain a Data stage between the Setup and Status stages. The xHCI defines three types of TDs: *Setup Stage*, *Data Stage*, and *Status Stage* TDs, which correspond to respective USB control transfer stages, to support control transfers. Software “constructs” a control transfer by placing either two (Setup Stage and Status Stage), or three (Setup Stage, Data Stage, and Status Stage) TDs on the Transfer Ring before ringing the doorbell.

A *Setup Stage TD* generates a USB SETUP transaction, which is used to transmit information to the control endpoint of a USB device. A *Setup Stage TD* always consists of a single *Setup Stage TRB* which contains the 8 byte *Setup Data* described in section 9.3 of the USB2 spec.

Software is responsible for the amount of data that is transferred with a *Data Stage TD* and its direction are consistent with the length and direction specified by the *Setup Data* in the *Setup Stage TRB*. A *Data Stage TD* consists of a *Data Stage TRB* followed by zero or more *Normal TRBs*. If the data is not physically contiguous, Normal TRBs may be chained to the *Data Stage TRB.* All the TRBs in the *Data Stage TD* transfer data in the same direction (i.e., all INs or all OUTs), as defined by the *Data Stage TRB*.

A *Status Stage TD* is required to complete a control transfer by retrieving the completion status of the USB SETUP transaction from the USB device. The *Status Stage TD* is always the last TD in a control transfer sequence. A *Status Stage TD* always consists of a single *Status Stage TRB* and may include an *Event Data TRB*. Refer to section 8.5.3.1 of the USB2 specification and section 8.12.2.1 of the USB3 specification for more information on status reporting.

Figure 3-8: Control Transfer Descriptor Example

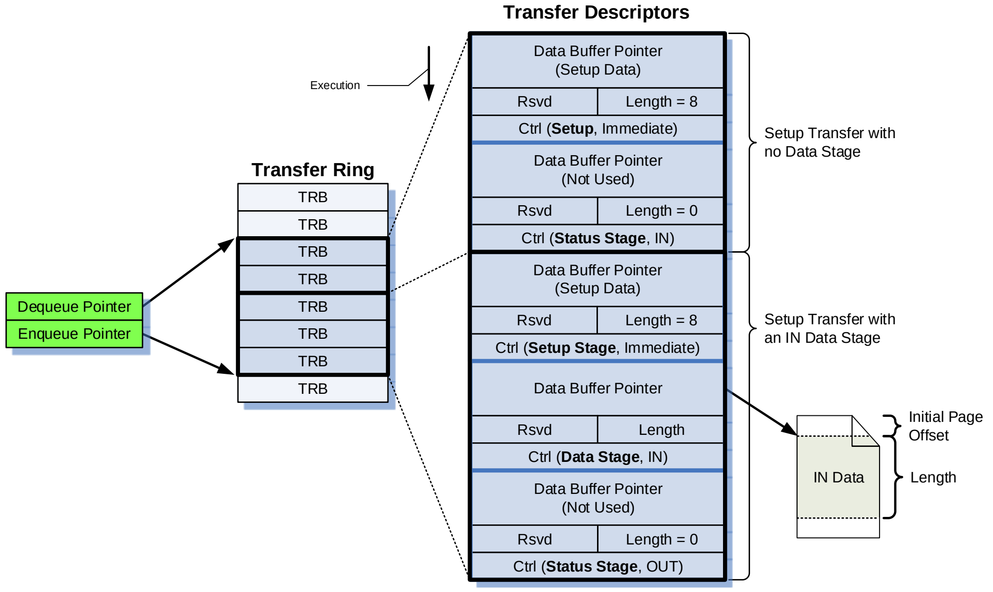


Figure 3-8 is an example of the contents of a Control Endpoint Transfer Ring. This example illustrates two control transfers: 1) a Setup stage transfer with no Data stage (top TD) is followed by 2) a Setup stage transfer with an IN Data stage. Note that the *Status Stage TRBs* define '0' length transfers, and that the direction of the Data Stage and Status Stage TRBs depends on the Control transfer direction identified in the Setup Stage TRB, and whether a Data Stage is required. Refer to section 4.11.2.2 for more information on Setup Stage transfers.

3.2.10 Bulk and Interrupt Transfers

Bulk and Interrupt Transfer Descriptors use *Normal TRBs* and depending on the data buffering requirements can use one or more chained *Normal TRBs* to form a TD. Multi-TRB Bulk or Interrupt TDs may define a Scatter/Gather operation as described in section 3.2.8.

3.2.11 Isoch Transfers

The Transfer Ring associated with an Isochronous Endpoint works as follows:

- Each Isoch Transfer Descriptor (TD) consists of an *Isoch TRB* chained to zero or more *Normal TRBs*.
- The *TRB Type* field in the *Control* field of the first TRB of an Isoch TD is set to ***Isoch TRB***.
- One *Isoch TD* is "consumed" every Interval (defined by bInterval in the USB Endpoint Descriptor).

- If the data required by an Isoch TD is not physically contiguous (e.g. crosses a page boundary), then one or more additional *Normal TRBs* shall be chained to the *Isoch TRB* by the host.
- The size of an Isoch Transfer in bytes shall be limited to either *Max Packet Size * Max Burst Size * Mult* (defined in the Endpoint Context), or the sum of the *Length* fields defined by the *Isoch TRB* and all *Normal TRBs* chained to it.
- For Isoch Out transfers, the xHC shall generate a *Ring Underrun* Transfer Event if the Transfer Ring is empty when an active interval boundary is reached.
- For Isoch IN transfers, the xHC shall generate a *Ring Overrun* Transfer Event if the Transfer Ring is empty when an active interval boundary is reached.

IMPLEMENTATION NOTE

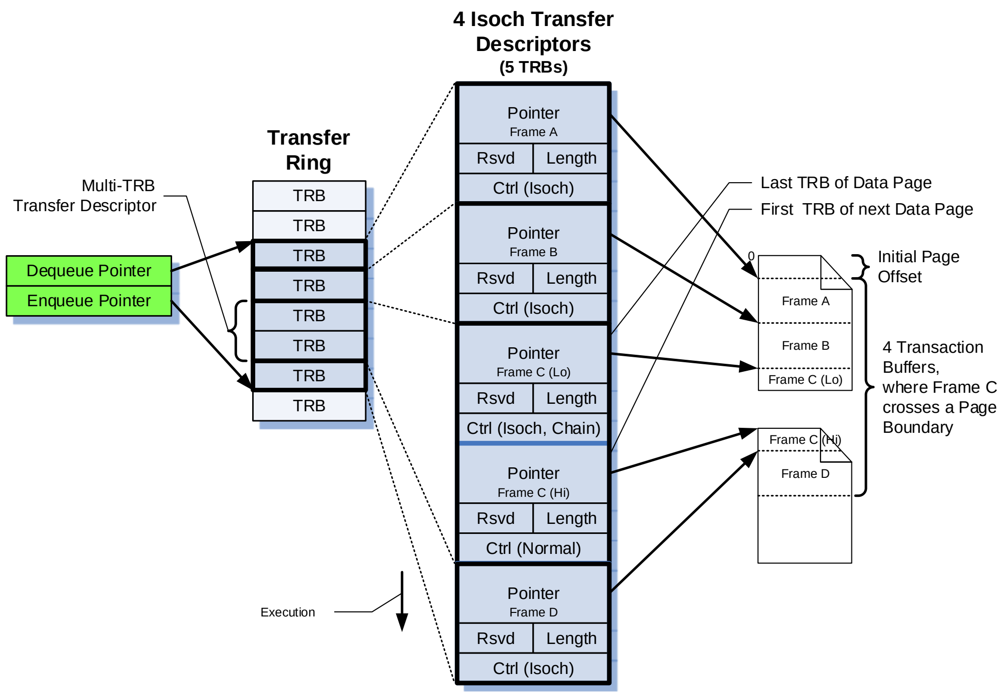
Fractional Isoch Transfers

To relax the real-time demands on the system, an Isoch Transfer scheduled by an application may define the data for many frames[3]. Also in order to hit a precise data rate the size of the Isoch transfers may have to vary from frame to frame. For instance, system software may define 10ms. of 44.1 KHz 16-bit stereo data to be transferred to a set of USB headphones. To minimize latency and the buffering requirements of the USB headphones, the driver will schedule the minimum amount of data to be sent every millisecond. That is, 176 bytes (44 4-byte/sample (16-bits/channel)) are moved every millisecond for 9ms. and 180 bytes are moved in the 10th ms. (to cover the ".1"). Assuming that the 10ms. of audio data is stored contiguously on a single page in memory, then a set of 10 TDs shall be posted to the Transfer Ring each containing a single Isoch TRB, with the *Length* of the last TRB being 4 bytes larger than the rest.

If the audio data buffer is not physically contiguous (e.g. crosses a Page boundary), then an additional Normal TRB will be chained to the Isoch TD that crossed the Page boundary.

[3]The period between isochronous transfers is often referred to as a "Frame", however strictly speaking the period is defined by the Endpoint Descriptor bInterval field. The value of bInterval is in Frames (1ms.) or Microframes (125µs.) depending on whether the device is LS/FS or HS/SS. In this document, references to "frame" or "interval" in isochronous discussions should be interpreted as "the period between isochronous transfers".

Figure 3-9: Isochronous Transfer Example



In Figure 3-9 above note:

- Four Isoch TDs are defined, representing the Isoch data scheduled for 4 consecutive frames.
- The Isoch data transferred in Frames A, B and D are all contiguous blocks (i.e. no page boundary crossings).
- The Isoch data to be transferred in Frame C crosses a Page boundary. The Pointer of the Isoch TRB (Frame C Lo) is used to access the first bytes of Isoch data in memory. A *Normal TRB* is chained to the Frame C *Isoch TRB*, and the Pointer of the Normal TRB (Frame C Hi) is used to access the remaining Isoch data for the frame on the next Page of memory.
- The number of bytes that will be transmitted in single USB Frame is defined by sum of the *Length* fields of all TRBs in an Isoch TD.

This example illustrates a case where the Isoch data buffers for multiple Intervals are physically contiguous. The xHCI Isoch mechanism also supports cases where multiple data buffers are transferred in a single Isoch Interval. In this latter case, one or more *Normal TRBs* may be chained to the initial *Isoch TRB*. It is the responsibility of system software to ensure that the *Length* and

Pointer fields of all TRBs in an Isoch TD are correct. An Isoch TD is terminated by a TRB with the Chain flag cleared to '0'.

3.3 Command Interface

To manage the xHC and the devices attached to it, the xHC provides an independent Command Ring interface. A work item on a Command Ring is called a *Command Descriptor* (CD). Command Ring operation is very similar to that of Transfer Rings, software issues a command to the xHC by placing a CD on the Command Ring then rings the Host Controller doorbell. The size of the Command Ring can be modified using the same Link TRB mechanism that Transfer Rings use.

All commands result in a *Command Completion Event* being placed on the Event Ring, which reports the completion status of the command.

Commands are executed by the xHC in the order that they are placed on the Command Ring. System software may add CDs to the Command Ring while it is running, however the execution of CDs should be stopped if software wants to delete or reorder (i.e. raise the priority of) scheduled CDs. Special Command Ring controls allow commands to be stopped or aborted.

The table below provides a summary of the xHCI command set. The remainder of this section provides a high level description of each of the commands.

Table 3-1: Command TRB Summary

Name	Description
No Op	Tests TRB Ring mechanism
Enable Slot	Returns a Device Slot ID and transitions the Device Slot from the Disabled to the *Default* state.
Disable Slot	Transitions the selected Device Slot from any state to the *Disabled* state. Any pending transfers are terminated and the slot is made available again.
Address Device	Enables the Default Control Endpoint, optionally issues a SET_ADDRESS request to the USB device, and transitions the Device Slot to the *Addressed* state.
Configure Endpoint	Enables and/or Disables selected endpoints for the device.

Evaluate Context	Informs xHC that software has modified selected Context parameters.
Reset Endpoint	Resets selected Endpoint. This command is used to recover from a halted endpoint.
Stop Endpoint	Stops or aborts operation on selected Endpoint.
Set TR Dequeue Pointer	Updates the Transfer Ring Dequeue Pointer of an enabled endpoint.
Reset Device	Resets selected Device Slot. This command is used to synchronize the state of a Device Slot when resetting a USB device.
Force Event	Used with virtualization by a VMM to force a TRB on to an Event Ring owned by a VM.
Negotiate Bandwidth	Initiates Bandwidth Request Events.
Set Latency Tolerance	Used by software to set the Best Effort Latency Tolerance (BELT) value for the xHC.
Get Port Bandwidth	Provides a means for software to identify the periodic bandwidth available on xHC Root Hub Ports.
Force Header	Allows software to generate SS LMPs or TPs to a Root Hub Port.

Refer to Table 6-86 for the TRB Type IDs associated with Commands.

3.3.1 No Op

The *No Op Command* may be issued by software to exercise the TRB Ring mechanism of the xHC without affecting any xHC or USB Device state, or to report the current value of the Command Ring Dequeue Pointer.

Refer to section 4.6.2 for more information on the *No Op Command*.

3.3.2 Enable Slot

The *Enable Slot Command* is issued by software to obtain an ID for an available Device Slot. System software uses the *Slot ID* returned by the command as an index into the *Device Context Base Address Array* to link a *Device Context* data structure for the USB device to a xHC Device Slot.

Refer to section 4.9.3 for more information on the *Enable Slot Command*.

3.3.3 Disable Slot

The *Disable Slot Command* is issued by software to inform the xHCI that a Device Slot is no longer needed, and that any resources assigned to the slot can be released. This command would be issued when a device is detached from the USB. A disabled Device Slot is available for assignment by the *Enable Slot Command*.

Refer to section 4.9.4 for more information on the *Disable Slot Command*.

3.3.4 Address Device

This xHCI command replaces the USB SET_ADDRESS request normally generated by a system enumerator when enumerating USB devices through the xHC. All USB devices use the default address ('0') after the device has been reset. Execution of the *Address Device Command* (*BSR* = '0') causes the xHC to issue a SET_ADDRESS request to the USB device, assigning a unique address to it. This operation causes a USB device that is in the Default state to transition to the Address state.

This command, which is issued immediately after an *Enable Slot Command*, also informs the xHC that the pointer in the *Device Context Base Address Array* references a *Device Context* data structure.

The *Address Device Command TRB* points to an *Input Context* data structure. The Input *Slot Context* and *Endpoint 0 Context* define the information needed by the xHC to communicate with the control endpoint of the device. If the SET_ADDRESS request issued by the xHC is successful, the contents of the Input *Slot* and *Endpoint 0 Context* data structures are copied to the respective *Device Context* data structures, and the *Transfer Ring* associated with endpoint 0 is set to the *Running* state.

Note that the xHC, not software, selects the address that is assigned to the USB device. This approach ensures that addresses will not be overloaded when assigned in virtualized environments.

This command is issued as part of the USB device enumeration process after a USB device attachment or reset. Once a successful *Address Device Command* has completed, system software can complete the standard USB device enumeration process, i.e. issuing GET_DESCRIPTOR requests through the Default Control Endpoint to retrieve the USB Device, Configuration, etc. descriptors from the USB device. Using the information in these descriptors system software may then determine which Class Driver(s) to associate with the USB device.

Refer to section 4.6.5 for more information on the *Address Device Command*.

3.3.5 Configure Endpoint

When system software issues a SET_CONFIGURATION request to a USB Device, it enables a specific set of endpoints (**pipes**) in the device, which are defined by the respective Configuration Descriptor. To simplify the xHC hardware implementation, the xHC does not read descriptors from a device or monitor SET_CONFIGURATION (or SET_INTERFACE) requests to a device. Instead, the xHC depends on system software to coordinate the pipes configured in the xHC with those configured in the device. System software uses the *Configure Endpoint Command* to explicitly identify to the xHC the pipes that would be enabled by a target configuration and the characteristics of those pipes. Not only does the *Configure Endpoint Command* inform the xHC of the target USB Device configuration, but it also gives the xHC an opportunity to reject a configuration if the necessary USB bandwidth or xHC internal resources are not available.

The *Configure Endpoint Command* points to an *Input Context* data structure, which defines the target configuration parameters for the xHC. For proper operation of the xHC, every endpoint that will be enabled by a target device configuration *shall* be defined in a respective *Endpoint Context* data structure of the *Input Context*, and the parameters of the *Endpoint Contexts* shall correlate target endpoint settings (Endpoint Type, Max Packet Size, Burst Size, etc.). *xHC and device behavior will be undefined if there are any mismatches.* This also means that if the *Configure Endpoint Command* does not complete successfully, software *shall not* issue a SET_CONFIGURATION request to the device.

System software also uses the *Configure Endpoint Command* to inform the xHC of pipe changes due to selecting an Alternate Interface on a device. Typically an Alternate Interface setting is used to modify the payload size or bandwidth requirement of a pipe, however it may also be used to disable or enable one or more pipes. The *Input Control Context* data structure of the *Input Context* allows software to explicitly identify which pipes are enabled, disabled, or modified by a target Alternate Interface setting. The parameters of the Input *Endpoint Contexts* for enabled or modified pipes shall correlate target pipe settings (Endpoint Type, Max Packet Size, etc.). If the *Configure Endpoint Command* does not complete successfully, software *shall not* issue a SET_INTERFACE request to the device.

Prior to issuing this command, software constructs a set of data structures based on the *Input Context* in host memory that fully describe the target configuration (or Alternate Interface setting). The *Input Control Context* identifies which endpoints are affected by the command. The *Endpoint Contexts* of endpoints that are either enabled or modified shall be fully specified. The *Endpoint Contexts* of endpoints disabled by the command or not referenced in the *Input Control Context* are ignored by the xHC. If Streams are enabled for an endpoint, then the Endpoint Context shall point to a Primary Stream Context Array, otherwise it points to a Transfer Ring. If declared, each Stream Context in a Primary Stream Context Array may point to a Secondary Stream Context Array or a Transfer Ring. Stream Contexts in a Secondary Stream Context Array shall point to a Transfer Ring or Null.

If the *Configure Endpoint Command* is successful, the contents of the Input *Endpoint Context* data structures enabled or modified by the command are copied to the respective Output *Endpoint Context* data structures in the *Device Context*. And any Transfer Rings or Stream Contexts referenced by the Input Endpoint Contexts will be used by the xHC to manage the respective pipes. In this case, software may free the Input Context data structure, but any Stream Context or Transfer Ring referenced by it shall remain allocated for use by the xHC.

If the *Configure Endpoint Command* fails, the previous configuration defined in the Device Context is maintained.

Refer to section 4.6.6 for more information on the *Configure Endpoint Command*.

3.3.6 Evaluate Context

The *Evaluate Context Command* is issued by software to inform the xHC that specific fields should be modified in the Device Context. There are several cases during the enumeration process of a USB device where an incomplete Context is used to communicate with the device. For instance, the default Max Packet Size for a FS device is 8 bytes. Software will initialize the *Max Packet Size* field of the Default Control Endpoint Context to ‘8’. Then use the endpoint to issue a GET_DESCRIPTOR(Device) request to the device, retrieving the first 8 bytes of the Device Descriptor. Byte 7 of the Device Descriptor defines the actual *Max Packet Size* for the Default Control Endpoint. This command would then be used to update the Max Packet Size field of the Default Control Endpoint to its true value. Other fields that may need to be updated late in the enumeration process are the Slot Context *Hub* and *Max Exit Latency*.

The command passes a pointer to an *Input Context* data structure to the xHC. The xHC evaluates specific fields of the Input Context and updates the Device Context. The specific fields affected by the command are identified in the respective context descriptions in section 6.2.

Upon successful completion of an *Evaluate Context Command*, the xHC shall begin executing with the updated context parameters.

Refer to section 4.6.7 for more information on the *Evaluate Context Command*.

3.3.7 Reset Endpoint

The *Reset Endpoint Command* is issued by software to recover from a halted condition on an endpoint.

Refer to section 4.6.8 for more information on the *Reset Endpoint Command*.

3.3.8 Stop Endpoint

The *Stop Endpoint Command* is used by system software to manage a Transfer Ring. This command allows software to abort, reprioritize, or temporarily stop the execution of TDs on a ring.

Refer to section 4.6.9 for more information on the *Stop Endpoint Command*.

3.3.9 Set TR Dequeue Pointer

The *Set TR Dequeue Pointer Command* complements the *Stop Endpoint Command*, allowing software to modify the xHC Dequeue Pointer associated with a pipe, and redirect the execution of TDs on its Transfer Ring.

Refer to section 4.6.10 for more information on the *Set TR Dequeue Pointer Command*.

3.3.10 Reset Device

The *Reset Device Command* is used by software to inform the xHC that the USB Device associated with a Device Slot has been Reset. In the Slot Context of the selected device slot, the reset operation sets the *Slot State* field to the *Default* state and the *USB Device Address* field to ‘0’. The reset operation also disables all endpoints of the slot except for the Default Control Endpoint by setting the Endpoint Context *Slot State* field to *Disabled* in all enabled Endpoint Contexts.

Refer to section 4.6.11 for more information on the *Reset Device Command*.

3.3.11 Force Event

The *Force Event Command* is an Optional Normative command of the xHCI, that is only used when the virtualization features of the xHC are enabled. This command, combined with other xHC mechanisms, allows a Virtual Machine Manager (VMM) to emulate a USB device to a Virtual Machine. Specifically this command is used by a VMM to insert an Event TRB on an Event Ring of a target VM. Refer to section 8 for more details on the xHC virtualization support.

Refer to section 4.6.12 for detailed information on the use of the *Force Event Command*.

3.3.12 Negotiate Bandwidth

The *Negotiate Bandwidth Command* is an Optional Normative command of the xHCI, that is used to recover USB bandwidth in a running system. Refer to section 4.16 for more information on how xHC bandwidth management works.

3.3.13 Set Latency Tolerance Value

The *Set Latency Tolerance Value Command* may be issued by software to provide a software defined Best Effort Latency Tolerance (BELT) value for the xHC.

Refer to section 4.6.14 for more information on the *Set Latency Tolerance Value Command*.

3.3.14 Get Port Bandwidth

The *Get Port Bandwidth Command* is issued by software to retrieve the percentage of periodic bandwidth available on each Root Hub Port of the xHC. This information can be used by system software to recommend topology changes to the user if they were unable to enumerate a device due to a *Bandwidth Error*.

Refer to section 4.6.15 for more information on the *Get Port Bandwidth Command*.

3.3.15 Force Header

The *Force Header Command* may be issued by software to send a Link Management (LMP) or Transaction Packet (TP) to a USB device, through a selected Root Hub Port. For instance, it may be used to send a PING TP or a Vendor Device Test LMP.

Refer to section 4.6.16 for more information on the *Force Header Command*.

3.4 General Information

The xHC manages all transfer types using a simple *TRB Ring* data structure. The *TRB Ring* provides automatic, in-order streaming of data transfers. Software can asynchronously add *TRBs* (data buffers) to a *TRB Ring* and maintain streaming, without having to invoke locking schemes.

USB-defined Short Packet semantics are fully supported on all processing boundary conditions without software intervention.

Hub TT Split transactions are automatically managed by the xHC without software intervention.

Isochronous transfers are managed using *Isoch TRBs*. These data structures are optimized for the variability per data payload and time-oriented characteristics of the isochronous transfer type.

3.5 Root Hub Management

The host controller of a USB bus is required to implement Root Hub functionality. The Operational Register space contains port registers that provide the hardware status and control needed to manage each port within the USB Specification. An xHC Root Hub may provide USB 2.0 and USB 3.x Root hub ports[4] to support Low-, Full-, or High-Speed as well as SuperSpeed devices. The host controller traverses the Transfer Rings and encounters work items that result in the host controller executing USB transactions. These transactions are routed to the Root Hub port associated with the attached downstream USB device.

The port registers provide system software with the control and status information required to manipulate the port in accordance with the USB Specification. The supported features include: detecting device connects, disconnects, performing device resets, manipulating port power and managing port power management capabilities.

System software should provide an abstraction to the USB system software stack that allows the Root Hub ports to be manipulated by the system as if they were ports on an external hub. Refer to section 5.4.8 for more information on Root Hub Port Status and Control Registers.

3.6 xHCI Device Enumeration

Under normal operating conditions (assuming all xHCI drivers are loaded and operational), the typical port enumeration sequence is described in section 4.3.

[4]Section 10.1 of the USB3 spec describes a USB 3.x hub as a "logical combination of 2 hubs: a USB 2.0 hub and an Enhanced SuperSpeed hub". Each logical hub has its own set of addressable ports for supporting the respective protocol. Each downstream (A) connector of a hub connects to one port of each logical hub. This allows Low-, Full-, or High-Speed as well as Enhanced SuperSpeed devices to be attached to any connector. The xHCI follows this model by providing separate USB2.0 and USB3.x Root Hub ports. Refer to section 4.19.7 for details.

4 Operational Model

This section describes the general operational model for the eXtensible Host Controller Interface (xHCI) hardware and eXtensible Host Controller Driver (xHCD) (generally referred to as system software). Each significant operational feature of the eXtensible Host Controller (xHC) is discussed in a separate section. Each section presents the operational model requirements for the xHC hardware. Where appropriate, recommended system software operational models for features are also presented.

4.1 Command Operation

There is only one Command Ring that is used for issuing xHC specific commands or commands related to Device Slots. The *Command Ring Control Register* is defined in the Operational Register space (refer to section 5.4.5).

All xHC commands are issued by placing the desired Command TRB(s) (6.4.3) on the Command Ring, then ringing the xHC command Doorbell register, i.e. writing the *Host Controller Command* code to the *DB Target* field of Doorbell register 0 (refer to section 5.6).

All commands result in the generation of a *Command Completion Event TRB* (6.4.3) on the Event Ring. Refer to section 4.11.3 for a discussion of Event TRBs.

4.2 Host Controller Initialization

When the system boots, the host controller is enumerated, assigned a base address for the xHC register space, and the system software sets the *Frame Length Adjustment* (FLADJ) register to a system-specific value.

Refer to section 4.23.1 for a discussion of the affect of Power Wells on register state after power-on and light resets.

Following are a review of the operations that system software would perform in order to initialize the xHC using MSI-X as the interrupt mechanism[5]:

- Initialize the system I/O memory maps, if supported.
- After Chip Hardware Reset[6] wait until the *Controller Not Ready* (CNR) flag in the USBSTS is '0' before writing any xHC Operational or Runtime registers.

[5]Refer to the PCI spec for the initialization and use of MSI or PIN interrupt mechanisms

[6]A **Chip Hardware Reset** may be either a PCI reset input or an optional power-on reset input to the xHC.

Note: This text does not imply a specific order for the following operations, however these operations shall be completed before setting the USBCMD register *Run/Stop* (R/S) bit to '1'.

- Program the Max Device Slots Enabled (MaxSlotsEn) field in the CONFIG register (5.4.7) to enable the device slots that system software is going to use.
- Program the Device Context Base Address Array Pointer (DCBAAP) register (5.4.6) with a 64-bit address pointing to where the Device Context Base Address Array is located.
- Define the Command Ring Dequeue Pointer by programming the Command Ring Control Register (5.4.5) with a 64-bit address pointing to the starting address of the first TRB of the Command Ring.
- Initialize interrupts[7] by:
    - Allocate and initialize the MSI-X Message Table (5.2.8.3), setting the Message Address and Message Data, and enable the vectors. At a minimum, table vector entry 0 shall be initialized and enabled. Refer to the PCI specification for more details.
    - Allocate and initialize the MSI-X Pending Bit Array (PBA, 5.2.8.4).
    - Point the *Table Offset* and *PBA Offsets* in the *MSI-X Capability Structure* to the MSI-X Message Control Table and Pending Bit Array, respectively.
    - Initialize the Message Control register (5.2.8.3) of the *MSI-X Capability Structure*.
    - Initialize each active interrupter by:
        - Defining the Event Ring: (refer to section 4.9.4 for a discussion of Event Ring Management.)
            - Allocate and initialize the Event Ring Segment(s).
            - Allocate the *Event Ring Segment Table* (ERST) (section 6.5). Initialize ERST table entries to point to and to define the size (in TRBs) of the respective Event Ring Segment.
            - Program the Interrupter *Event Ring Segment Table Size* (ERSTSZ) register (5.5.2.3.1) with the number of segments described by the Event Ring Segment Table.
            - Program the Interrupter *Event Ring Dequeue Pointer* (ERDP) register (5.5.2.3.3) with the starting address of the first segment described by the Event Ring Segment Table.
            - Program the Interrupter *Event Ring Segment Table Base Address* (ERSTBA) register (5.5.2.3.2) with a 64-bit address pointer to where the Event Ring Segment Table is located.

[7]Interrupts are optional. The xHC may be managed by polling Event Rings.

        - Note that writing the *ERSTBA* enables the Event Ring. Refer to section 4.9.4 for more information on the Event Ring registers and their initialization.
    - Defining the interrupts:
        - Enable the MSI-X interrupt mechanism by setting the MSI-X Enable flag in the MSI-X Capability Structure Message Control register (5.2.8.3).
        - Initializing the Interval field of the Interrupt Moderation register (5.5.2.2) with the target interrupt moderation rate.
        - Enable system bus interrupt generation by writing a '1' to the Interrupter Enable (INTE) flag of the USBCMD register (5.4.1).
        - Enable the Interrupter by writing a '1' to the Interrupt Enable (IE) field of the Interrupter Management register (5.5.2.1).
- Write the USBCMD (5.4.1) to turn the host controller ON via setting the Run/Stop (R/S) bit to '1'. This operation allows the xHC to begin accepting doorbell references.

At this point, the host controller is up and running and the Root Hub ports (5.4.8) will begin reporting device connects, etc., and system software may begin enumerating devices. System software may follow the procedures described in section 4.3, to enumerate attached devices.

USB2 (LS/FS/HS) devices require the port reset process to advance the port to the **Enabled** state. Once USB2 ports are Enabled, the port is active with SOFs occurring on the port, but the Pipe Schedules have not yet been enabled.

SS ports automatically advance to the Enabled state if a successful device attach is detected.

4.3 USB Device Initialization

This section describes the process of detecting and initializing a USB device attached to an xHC Root Hub port.

The USB device initialization process is the same, whether the device attached to the port is a Function or a Hub. Once the Pipes associated with an external hub are set up, the Hub Driver will enumerate the devices attached to the external hub's ports using standard Hub Class command sequences. This section focuses on the device initialization process when a device is attached to a Root Hub port.

After a Chip Hardware Reset, HCRST, or commanded to the *PLS* = RxDetect state, all Root Hub ports shall be in **Disconnected** state, i.e. the port is powered on (PP

= ‘1’) and waiting for a device connect. Refer to section 4.19.1 for more information on xHCI Root Hub port states.

If a USB device is attached to a port when it is in the **Disconnected** state:

- USB3 protocol ports shall:
    - Advance to the Polling state (refer to Figure 4-30):
        - If polling is successful, the port shall advance to the **Enabled** state, and the *Current Connect Status* (CCS) and *Connect Status Change* (CSC) flags are set to ‘1’.
        - If polling is unsuccessful, the port shall advance to the **Disconnected** state.
- USB2 protocol ports shall:
    - Advance to the Disabled state (refer to Figure 4-25) and set the Current Connect Status (CCS) and Connect Status Change (CSC) flags to ‘1’.

Note: The “**Disabled**” Root Hub port state represents different conditions when referring to USB3 or USB 2 protocol ports. For USB3 ports, the **Disabled** state indicates that the port is in the *DSPORT.Disabled* state (refer to Figure 10-9 in the USB3 spec.). For USB2 ports, the **Disabled** state indicates that the port is in the *Disabled* state (refer to Figure 11-10 in the USB2 spec).

The following steps describe a typical USB Device initialization process:

1. When the xHC detects a device attach, it shall set the *Current Connect Status* (CCS) and *Connect Status Change* (CSC) flags to ‘1’. If the assertion of *CSC* results in a ‘0’ to ‘1’ transition of *Port Status Change Event Generation* (PSCEG, section 4.19.2), the xHC shall generate a *Port Status Change Event*.

2. Upon receipt of a *Port Status Change Event* system software evaluates the *Port ID* field to determine the port that generated the event.

3. System software then reads the PORTSC register of the port that generated the event.
   CSC = ‘1’ if the event was due to an attach (*CCS* = ‘1’) or detach (*CCS* = ‘0’). Assuming the event was due to an attach:

    a. A USB3 protocol port attempts to automatically advance to the Enabled state as part of the attach process.

       If successful, the port shall transition to the Enabled state, i.e. the Port Enabled/Disabled (PED) flag shall be set to ‘1’, and the Port Reset (PR) flag and Port Link State (PLS) field shall be ‘0’. The attached USB device shall be in the Default state.

       If unsuccessful, the port shall transition to the Disconnected state, i.e. the PED and PR flags shall be cleared to ‘0’ and Port Link State

(PLS) field shall be set to RxDetect (‘5’). The attached USB device shall remain powered.

b. A USB2 protocol port requires software to reset the port to advance the port to the Enabled state and a USB device from the Powered state to the Default state. After an attach event, the PED and PR flags shall be ‘0’ and the PLS field shall be ‘7’ (Polling) in the PORTSC register.

   System software shall enable the port by resetting the port (writing a '1' to the PORTSC PR bit) then waiting for a Port Status Change Event due to the assertion of Port Reset Change (PRC) flag. Refer to section 4.3.1 for an overview of the Root Hub port reset activities.

   The completion of the port reset shall cause the PORTSC register PRC and PED flags to be set (‘1’), the PR flag to be cleared (‘0’), and the PLS field to be U0 (‘0’). If the assertion of PRC results in a ‘0’ to ‘1’ transition of PSCEG (4.19.2), the xHC shall generate a Port Status Change Event as a result of the transition of PRC. The reset operation sets the USB2 device into the Default state, preparing it for a SET_ADDRESS request.

4. After the port successfully reaches the Enabled state, system software shall obtain a Device Slot for the newly attached device using an Enable Slot Command, as described in section 4.3.2.

5. After successfully obtaining a Device Slot, system software shall initialize the data structures associated with the slot as described in section 4.3.3.

6. Once the slot related data structures are initialized, system software shall use an Address Device Command to assign an address to the device and enable its Default Control Endpoint, as described in section 4.3.4.

7. For LS, HS, and SS devices; 8, 64, and 512 bytes, respectively, are the only packet sizes allowed for the Default Control Endpoint, so step a may be skipped.

   For FS devices, system software should initially read the first 8 bytes of the USB Device Descriptor to retrieve the value of the bMaxPacketSize0 field and determine the actual Max Packet Size for the Default Control Endpoint, by issuing a USB GET_DESCRIPTOR request to the device, update the Default Control Endpoint Context with the actual Max Packet Size and inform the xHC of the context change. Step a describes this operation.

   a. The USB GET_DESCRIPTOR request requires a Data Stage, so the Setup Stage TD shall be followed by a Data Stage TD, then a Status Stage TD. To do this software shall:

i) Allocate an 8 byte buffer to receive the Device Descriptor.

ii) Initialize the *Setup Stage TD* (a single *Setup Stage TRB*) on the Endpoint 0 Transfer Ring.

- *TRB Type* = Setup Stage TRB.
- *Transfer Type* (TRT) = *IN Data Stage* (3).
- *TRB Transfer Length* = 8.
- *Interrupt On Completion* (IOC) = 0.
- *Immediate Data* (IDT) = 1.
- *bmRequestType* = 80h. (Dir = Device-to-Host, Type = Standard, Recipient = Device)
- *bRequest* = 6 (GET_DESCRIPTOR).
- *wValue* = 0100h. Low byte = 0 (Descriptor Index), High Byte = 1 (Descriptor type).
- *wIndex* = 0.
- *wLength* = 8.
- *Cycle* bit = Current Producer Cycle State.

iii) Advance the Endpoint 0 Transfer Ring Enqueue Pointer

iv) Initialize the *Data Stage TD* (a single *Data Stage TRB*) on the Endpoint 0 Transfer Ring.

- *TRB Type* = Data Stage TRB.
- *Direction* (DIR) = ‘1’.
- *TRB Transfer Length* = 8.
- *Chain* bit (CH) = 0.
- *Interrupt On Completion* (IOC) = 0.
- *Immediate Data* (IDT) = 0.
- *Data Buffer Pointer* = The address of the Device Descriptor receive buffer.
- Cycle bit = Current Producer Cycle State.

v) Advance the Endpoint 0 Transfer Ring Enqueue Pointer

vi) Initialize the *Status Stage TD* (a *Status Stage TRB*) on the Endpoint 0 Transfer Ring.

- *TRB Type* = Status Stage TRB.
- *Direction* (DIR) = ‘0’.
- *TRB Transfer Length* = 0.
- *Chain* bit (CH) = 0.
- *Interrupt On Completion* (IOC) = 1.
- *Immediate Data* (IDT) = 0.
- *Data Buffer Pointer* = 0.
- *Cycle* bit = Current Producer Cycle State.

vii) Advance the Endpoint 0 Transfer Ring Enqueue Pointer

viii) Ring the Device Slots' Doorbell with *DB Target = Control EP 0 Enqueue Pointer Update*.

ix) When a successful Transfer Event is returned for the GET_DESCRIPTOR Status Stage TRB system software shall update the Endpoint 0 Context *Max Packet Size* with *wMaxPacketSize* value returned in the Device Descriptor buffer, if the *wMaxPacketSize* value is different.

x) Software shall then issue an *Evaluate Context Command* with *Add Context* bit 1 (A1) set to '1' to inform the xHC of the change to the Default Control endpoint's *Max Packet Size* parameter. After successfully executing the *Evaluate Context Command* the xHC will use the updated *Max Packet Size* for all subsequent Default Control Endpoint transfers.

8. Now that the Default Control Endpoint is fully operational, system software may read the complete USB Device Descriptor and possibly the Configuration Descriptors so that it can hand the device off to the appropriate Class Driver(s). To read the USB descriptors, software will issue USB GET_DESCRIPTOR requests through the devices' Default Control Endpoint.

9. After reading the Configuration Descriptors software may issue an Evaluate Context Command with Add Context bit 0 (A0) set to '1' to inform the xHC of the value of the Max Exit Latency parameter. Note that the value of the Output Slot Context Interrupter Target field may also be modified by this command.

10. The Class Driver may then configure the Device Slot using a Configure Endpoint Command as described in section 4.3.5, and configure the USB Device itself by issuing a USB SET_CONFIGURATION request through the devices' Default Control Endpoint. The successful completion of both operations is required to advance the state of the USB device from Address to Configured and xHC Device Slot from Addressed to Configured.

11. If required, system software may configure Alternate Interfaces. For each Alternate Interface set the alternate interface as described in section 4.3.6.

12. The pipe interfaces to the USB device are now fully operational.

Note: To ensure proper operation software shall fully initialize the hubs and TTs of each tier of the USB topology before proceeding to the next tier, starting at the Root Hub. Failure to meet this requirement may result in undefined xHC behavior.

4.3.1 Resetting a Root Hub Port

Resetting a Root Hub port, resets the attached USB device, and if successful, the port logic reports the speed of the attached device and sets the port to the **Enabled** state. Whether successful or not, the *Port Reset Change* (PRC) flag is set to '1'. If the assertion of *PRC* results in a '0' to '1' transition of PSCEG (4.19.2), a *Port Status Change Event* shall be generated.

To reset a USB device attached to a Root Hub port, system software shall perform the following operations:

1. Write the PORTSC register with the Port Reset (PR) bit set to '1'.
2. Wait for a successful Port Status Change Event for the port, where the Port Reset Change (PRC) bit in the PORTSC field is set to '1'.

Section 4.19.5 describes the port reset operations performed by the xHC.

The next step requires system software to obtain a Device Slot (section 4.3.2), then associate the newly attached device with the Device Slot and enable its Default Control Endpoint.

Note: After a Root Hub port is successfully reset, the PORTSC *Port Speed* field shall indicate the speed of the attached device.

4.3.2 Device Slot Assignment

The first operation that software shall perform after detecting a device attach event and resetting the port is to obtain a Device Slot for the device by issuing an *Enable Slot Command* to the xHC through the Command Ring. The *Enable Slot Command* returns a *Slot ID* that is selected by the host controller. Refer to section 4.6.3 for a detailed description of the *Enable Slot* command.

System software executes the Slot Assignment process by successfully completing an *Enable Slot Command* as described in section 4.11.4.2.

System software shall wait for the Command Completion Event associated with the *Enable Slot Command* before issuing any more commands to the slot. If the command was successful, software may proceed to the Device Slot Initialization phase (section 4.3.3).

Successful completion of the *Enable Slot Command* shall transition the Device Slot to the *Enabled* state. Refer to section 4.5.3 for more information on Device Slot states.

4.3.3 Device Slot Initialization

Once an xHC Device Slot ID has been obtained for a USB device, software shall initialize the data structures associated with the slot. The following steps shall be performed by system software:

1. Allocate an Input Context data structure (6.2.5) and initialize all fields to '0'.

2. Initialize the Input Control Context (6.2.5.1) of the Input Context by setting the A0 and A1 flags to '1'. These flags indicate that the Slot Context and the Endpoint 0 Context of the Input Context are affected by the command.

3. Initialize the Input Slot Context data structure (6.2.2).

   - *Root Hub Port Number* = Topology defined.
   - *Route String* = Topology defined[8]. Refer to section 8.9 in the USB3 spec. Note that the *Route String* does not include the *Root Hub Port Number*.
   - *Context Entries* = 1.

4. Allocate and initialize the Transfer Ring for the Default Control Endpoint. Refer to section 4.9 for TRB Ring initialization requirements and to section 6.4 for the formats of TRBs.

5. Initialize the Input default control Endpoint 0 Context (6.2.3).

   - *EP Type* = Control.
   - *Max Packet Size* = The default maximum packet size for the Default Control Endpoint, as function of the PORTSC *Port Speed* field.
   - *Max Burst Size* = 0.
   - *TR Dequeue Pointer* = Start address of first segment of the Default Control Endpoint Transfer Ring.
   - *Dequeue Cycle State* (DCS) = 1. Reflects Cycle bit state for valid TRBs written by software.
   - *Interval* = 0.
   - *Max Primary Streams* (MaxPStreams) = 0.
   - *Mult* = 0.
   - *Error Count* (CErr) = 3.

6. Allocate the Output Device Context data structure (6.2.1) and initialize it to '0'.

[8]e.g. To access a device attached directly to a Root Hub port, the *Route String* shall equal '0', and the *Root Hub Port Number* shall indicate the specific Root Hub port to use.

7. Load the appropriate (Device Slot ID) entry in the Device Context Base Address Array (5.4.6) with a pointer to the Output Device Context data structure (6.2.1).

8. Issue an Address Device Command for the Device Slot, where the command points to the Input Context data structure described above. Refer to sections 3.3.4 and 6.4.3.4 for more information on the Address Device Command.

4.3.4 Address Assignment

Typically the first operation that software performs on a USB device is to assign an address to it, which transitions the USB device from the Default to the Address state. To assign an address to a USB device attached to the xHC, system software shall issue an *Address Device Command* with the *Block Set Address Request* (BSR) flag cleared to '0' to the xHC through the Command Ring. Refer to section 4.6.5 for a detailed description of the *Address Device* command.

System software executes the Address Assignment process by successfully completing an *Address Device Command* as described in section 4.6.5.

System software shall wait for *Address Device Command* completion event on the Event Ring before issuing any more commands to the slot. If successful, software proceeds to the Device Configuration phase (section 4.3.5).

Note: For some legacy USB devices it may be necessary to communicate with the device when it is in the Default state, before transitioning it to the Address state. To accomplish this system software shall issue an *Address Device Command* with the *BSR* flag set to '1'. Setting the *BSR* flag enables the operation of the Default Control Endpoint for the Device Slot but blocks the xHC from issuing a SET_ADDRESS request to the device, which would transition it to the Address state.

Successful completion of the *Address Device Command* with *BSR* = '0' shall transition the Device Slot from the *Enabled* to the *Addressed* state. Successful completion of the *Address Device Command* with *BSR* = '1' shall transition the Device Slot from the *Enabled* to the *Default* state. Refer to section 4.5.3 for more information on Device Slot states.

4.3.5 Device Configuration

As part of the initialization process of a USB device, the system software shall select a configuration. A USB device presents one or more configurations to choose from. The USB Framework requires that a SET_CONFIGURATION request is issued to a device to set a specific configuration. Refer to section 9.4.7 of the USB2 spec for more information on the USB SET_CONFIGURATION request.

For software to successfully "configure" a USB device, the state of both the USB Device and the xHC Device Slot assigned to the device must be synchronized. Software shall successfully complete a SET_CONFIGURATION request (with a Setup Stage TD on the device's Default Control Endpoint) to select a specific configuration, and a *Configure Endpoint Command* for the slot with the matching Endpoint Context configuration information, to transition the USB device and the xHC Device Slot to the Configured state. Refer to section 4.11.4.5 for more information on the *Configure Endpoint Command*.

A USB device may declare multiple alternate interfaces, each with different periodic bandwidth and resource requirements. If a *Configure Endpoint Command* for a particular configuration is unsuccessful, software may issue additional *Configure Endpoint Commands* with other interface settings in an attempt to successfully configure the slot. If all interface settings have been exhausted (i.e. none have been accepted by the xHC), only the Default Control Endpoint will remain enabled.

If system software was unable to successfully complete a *Configure Endpoint Command* due to a *Bandwidth Error*, it may optionally use the *Negotiate Bandwidth Command* to cause the xHC to request bandwidth with other devices. Refer to section 4.16.1 for more information on bandwidth negotiation.

System software executes the xHCI portion of the device configuration process by successfully completing a *Configure Endpoint Command* as described in section 4.11.4.5.

System software shall wait for the *Command Completion Event* associated with the *Configure Endpoint Command* before issuing any more commands to the slot.

After the *Configure Endpoint Command* and SET_CONFIGURATION request complete successfully, software may schedule TDs on any enabled endpoint Transfer Ring.

If the *Configure Endpoint Command* is not successful, undefined behavior will result if software issues a SET_CONFIGURATION request to the device.

Successful completion of the *Configure Endpoint Command* with the *Deconfigure* (DC) flag = '0' shall transition the Device Slot from the *Addressed* to the *Configured* state. Refer to section 4.5.3 for more information on how the *Configure Endpoint Command* affects Device Slot states.

4.3.6 Setting Alternate Interfaces

The USB SET_INTERFACE request allows the host to select an Alternate Setting for a specified interface in a USB device. A SET_INTERFACE request may disable or modify the operation of currently enabled endpoints, or it may enable previously unused endpoints. A SET_INTERFACE request does not affect endpoints owned by another interface. Refer to section 9.4.10 of the USB2 spec. for more information on the USB SET_INTERFACE request.

A SET_INTERFACE request provides the "Number" of the Interface that is affected and the Alternate Setting that it will be set to. A SET_INTERFACE request does not explicitly identify which endpoints of a device are affected or how. This information is available in the Configuration Descriptor retrieved from the device, hence known to host software and the device at their respective ends.

The xHC does not keep track of relationships between USB interfaces and endpoints, so it is system software's responsibility to explicitly "Disable" any endpoints that are affected in the current configuration by a USB SET_INTERFACE request, and then explicitly "Enable" any endpoints identified in the new Alternate Interface Setting. An xHCI endpoint (i.e. Endpoint Context) is "Disabled" by stopping it if it is in the *Running* state with a *Stop Endpoint Command* and freeing its Transfer Ring.

Setting an Alternate Interface is accomplished by the successful completion of a *Configure Endpoint* command (refer to section 4.6.6), and a USB SET_INTERFACE request to the USB device (with a Setup Stage TD on the Default Control Endpoint).

Below is an example of the sequence of events that would be employed to successfully set an alternate interface in a USB device.

System software shall wait for the *Command Completion Event* associated with the *Configure Endpoint Command* before issuing further commands to the slot.

Prior to issuing a *Configure Endpoint Command* to change an Alternate Interface setting system software should perform the following operations:

1. Stop any Running Transfer Rings affected by the Alternate Interface setting.
2. Free[9] Transfer Rings of all endpoints that will be affected by the Alternate Interface setting.
3. Clear all the Endpoint Context fields of each endpoint that will be disabled by the Alternate Interface setting, to '0'.
4. For each endpoint enabled by the Configure Endpoint Command:
    a. Allocate a Transfer Ring[9].

[9]If just the parameters of a currently defined endpoint are being changed by the Alternate Interface setting then software may chose to reuse the Transfer Ring for the new interface setting and not free it. In this case, software does not need to allocate a new Transfer Ring as described in step 4a).

b. Initialize the Transfer Ring Segment(s) by clearing all fields of all TRBs to '0'.10

c. Initialize the Endpoint Context data structure:
   - *EP Type* = Derived from the Endpoint Descriptor:bmAttributes:Transfer Type and Endpoint Descriptor:bEndpointAddress:Direction. Refer to Table 6-9 for the encoding.
   - *Max Packet Size* = Endpoint Descriptor:wMaxPacketSize & 07FFh.
   - *Interval* = Refer to section 6.2.3.6 for the computation of the *Interval* value.
   - *Max Burst Size* = SuperSpeed Endpoint Companion Descriptor:bMaxBurst or (Endpoint Descriptor: wMaxPacketSize & 1800h) >> 11.
   - *Mult* = '0' or SuperSpeed Endpoint Companion Descriptor:bmAttributes Mult field.
   - *CErr* = 3, or 0 for an Isoch endpoint.
   - If Streams are supported by the endpoint (i.e. SuperSpeed Endpoint Companion Descriptor:bmAttributes MaxStreams field > 0):
     - Select a *Max Primary Streams* (MaxPStreams) value > 0 and <= SuperSpeed Endpoint Companion Descriptor:bmAttributes MaxStreams
     - Update *MaxPStreams*.
     - Allocate and clear Primary Stream Array.
     - *MaxPStreams* = Size of Primary Stream Array.
     - *TR Dequeue Pointer* = Start address of Primary Stream Array.
   - else
     - *MaxPStreams* = '0'.
     - *TR Dequeue Pointer* = Start address of the first segment of the previously allocated Transfer Ring.
     - *Dequeue Cycle State* (DCS) = 1. Assuming that all TRBs in the segment referenced by the TR Dequeue Pointer have been initialized to '0', this field reflects Cycle bit state for valid TRBs written by software.

5. Issue and successfully complete a Configure Endpoint Command as described in section 4.11.4.5.

[10]The *Cycle bit* (C) of all TRBs in a TR Segment shall be initialized to the *inverse* of the value that the *Dequeue Cycle State* (DCS) field is initialized to. This pseudo code recommends initializing the all bytes in a TR Segment to '0', which also initializes the Cycle bit to '0' in all TRBs of the TR Segment, and the *DCS* flag of the pointer that references the TR Segment to '1', however software may initialize the Cycle bit to '1' in all TRBs of a newly allocated TR Segment and the *DCS* flag of the pointer that references it to '0'. Refer to section 4.9.2 for more information on *Cycle bit* (C) initialization.

System software shall wait for the *Command Completion Event* associated with the *Configure Endpoint Command* before issuing any more commands to the slot.

Note: A *Configure Endpoint Command* is not necessary prior to a SET_INTERFACE request, if the SET_INTERFACE request does not change any endpoint parameters.

4.3.7 Low-Speed/Full-Speed Device Support

Special provisions shall be made to generate the Split Transactions required for a Low- or Full-speed device connected through a High-speed hub. A Split Transaction token targets the downstream facing port of the hub that isolates the High-speed signaling environment from the Full/Low-speed signaling environment for this device. To generate the Split Transaction token, the xHC requires parameters associated with the target hub for which this full-/low-speed transaction is destined. This information shall be provided by system software in the *Multi-TT* (MTT), *TT Hub Slot ID* and *TT Port Number* fields of the device's Slot Context.

The xHC uses the *TT Hub Slot ID* to obtain the hub's address from the *USB Device Address* field of the hub's Slot Context.

The xHC also checks that the *Hub* flag in the hub's Slot Context equals '1', to verify that the *TT Hub Slot ID* references a hub. A *Parameter Error* shall be generated for the offending TD if the *Hub* flag = '0'.

If the device is not Low- or Full-speed or if the device is attached to a Root Hub port, then the *TT Hub Slot ID*, *Multi-TT* (MTT), and the *TT Port Number* fields shall be cleared to '0'.

Refer to section 8.4.2 of the USB2 spec. for more information on Split Transaction tokens, and section 11.14 for Transaction Translator information.

4.3.8 Bandwidth Management

When a device cannot be configured because of bandwidth constraints Bandwidth Negotiation may be performed. Refer to section 4.16.1 for more details.

4.4 Device Detach

When the device is detached from a Root Hub port, the PORTSC *Current Connection Status* (CCS) bit shall be cleared to '0' and the *Connect Status Change* (CSC) bit shall be set to '1'. If a '0' to '1' transition of PSCEG (4.19.2), the xHC shall report the change through a *Port Status Change Event*. After the detection of a detach, system software shall disable the Device Slot associated with the port by issuing a *Disable Slot Command* for the affected slot. Refer to section 4.6.4 for a description of the *Disable Slot* command.

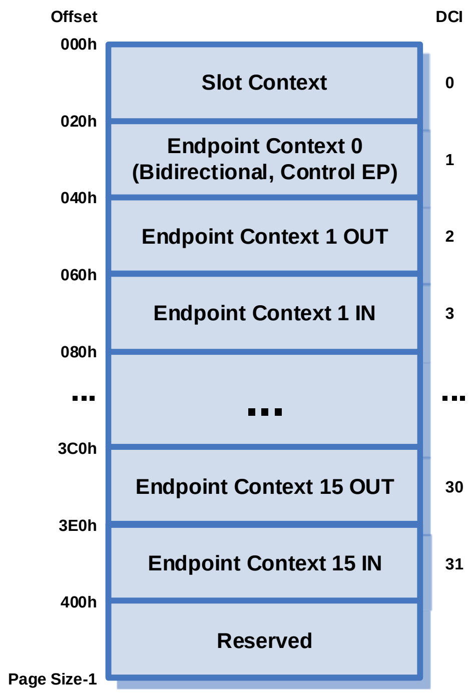
4.5 Device Slot Management

The xHCI supports up to 255 USB devices, where each USB device is assigned to a *Device Slot*. Each xHC Device Slot is comprised of 3 major components: an entry in the *Device Context Base Address Array*, a *Device Context* data structure, and a Doorbell Register in the *Doorbell Array*.

The ***Device Context Base Address Array*** supports up to 255[11] USB devices or hubs, where each element in the array is a 64-bit pointer to the base address of a *Device Context* data structure.

The Slot ID is the index that software uses when accessing the *Device Context Base Address Array* to retrieve a pointer to the Device Context data structure or to access the *Doorbell Register associated with a device*.

Figure 4-1: Device Context



A ***Device Context*** data structure describes the characteristics and current state of an individual USB device attached to the host controller. The *Device Context*

[11]The total number of USB devices supported by the xHCI architecture is less than 256 (the number of Device Context slots) because some of the Device Context slots are reserved by the xHCI for special purposes and are not available for enumerating USB devices. e.g. If virtualization is enabled, slots allocated to one VF will appear to be “reserved” to another VF.

is organized as an array of 32 context data structures, consisting of 1 *Slot Context* and 31 *Endpoint Context* data structures. Figure 4-1 illustrates the *Device Context* layout. Refer to section 6.2.1 for data structure details.

When software allocates a *Device Context* data structure all fields in all entries shall be initialized to '0'.

The ***Slot ID*** is the index that system software uses when accessing a specific Device Slot in the *Device Context Base Address Array* and the *Doorbell Array*.

The ***Slot Context*** data structure defines information that applies to the slot, the device as whole, or to all Endpoint Contexts.

Each ***Endpoint Context*** data structure defines the characteristics of the endpoint; type, direction, bandwidth requirements, etc., and points to a ***Transfer Ring*** or a ***Stream Context Array***. An *Endpoint Context* exists for each endpoint of a device. The "enabled[12]" Endpoint Contexts depend on the Configuration selected by the Device's Class Driver. Note that *Endpoint Context 0* is always associated with the Default Control Endpoint of the device.

A 32-bit *Doorbell Register* exists in the *Doorbell Array* for each *Device Slot* and is indexed by the *Slot ID*. The *DB Target* and *DB Stream ID* fields in the *Doorbell Register* indicates the purpose of "ringing" the doorbell.

Ringing the *Host Controller Doorbell* (Doorbell Register 0) with the *DB Target = Host Controller Command*, indicates to the xHC that software has defined a command in the *Command Ring* that it wants executed.

Ringing the Device Slot's *Doorbell Register,* indicates to the xHC that software has added work to be executed on the Transfer Ring (pipe) defined by the *DB Target* and *DB Stream ID* field values. Refer to section 5.2.

4.5.1 Device Context Index

The term ***Device Context Index*** (DCI) is used throughout this document to reference an individual context data structure in the *Device Context*. The range of *DCI* values is 0 to 31.

The *DCI* of the *Slot Context* is 0.

For *Device Context Indices* 1-31, the following rules apply:

1. For Isoch, Interrupt, or Bulk type endpoints the DCI is calculated from the Endpoint Number and Direction with the following formula;

[12]An Endpoint Context is "enabled" if it is not in the *Disabled* state.

DCI = (Endpoint Number * 2) + Direction,
where Direction = ‘0’ for OUT endpoints and ‘1’ for IN endpoints.

2. For Control type endpoints:
   DCI = (Endpoint Number * 2) + 1.

4.5.2 Slot Context Initialization

All fields of an Input Slot Context data structure (including the Reserved fields) shall be initialized to ‘0’ with the following exceptions:

For Address Device Command:

- *Route String* = Topology defined.
- *Root Hub Port Number* = Topology defined.
- *Context Entries* = ‘1’. Only the Default Control Endpoint is enabled.
- *Interrupter Target* = System defined.
- *Speed* = Defined by downstream facing port attached to the device.
- If the device is a Low-/Full-speed function or hub accessed through a High-speed hub, then the following values are derived from the “parent” High-speed hub whose downstream facing port isolates the High-speed signaling environment from the Low-/Full-speed signaling environment:
    - *MTT* = '1' if the Multi-TT Interface of the hub has been enabled with a Set Interface request, otherwise '0'. Software shall issue a Set Interface request to select the Multi-TT interface of the hub prior to issuing any transactions to devices attached to the hub.
    - *TT Port Number* = The number of the downstream facing port in the parent High-speed hub that the device is accessed through.
    - *TT Hub Slot ID* = The Slot ID of the parent High-speed hub.

For Evaluate Context Command:

- *Max Exit Latency* = Topology Defined. Refer to section 4.23.5.2.
- *Interrupter Target* = System defined.

For Configure Endpoint Command:

- *Context Entries* = Maximum DCI+1 of configured Endpoint Contexts.
- If the device is a hub:
    - *Hub* = ‘1’.
    - *Number of Ports* = *bNbrPorts* from the USB Hub Descriptor.
    - If the device *Speed* = *High-Speed* (‘3’):
        - *TT Think Time* (TTT) = Value of the TT Think Time sub-field (USB2 spec,

Table 11-13) in the Hub Descriptor:wHubCharacteristics field.

- *Multi-TT* (MTT) = '1' if the Multi-TT Interface of the hub has been enabled with a Set Interface request, otherwise '0'.

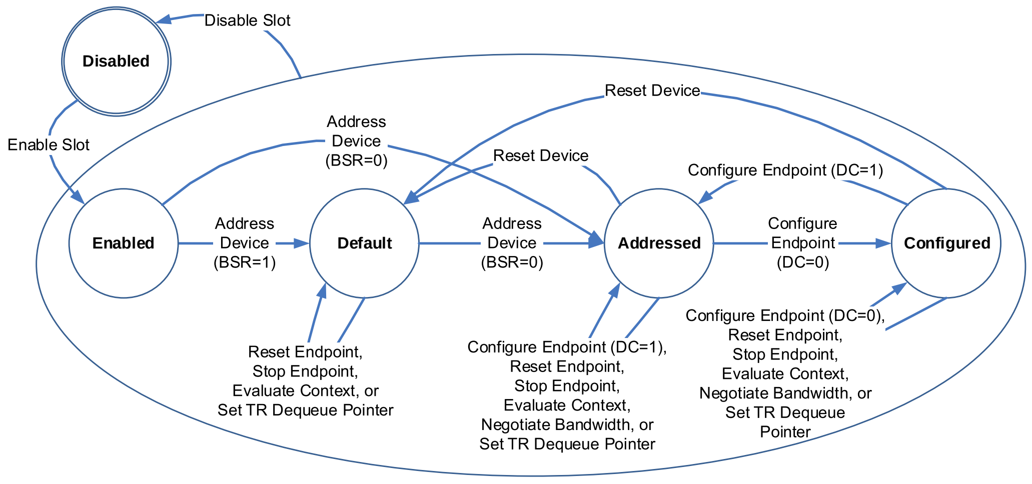
Note: The values of the *Route String* and *Root Hub Port Number* fields shall be initialized by the first *Address Device Command* issued to a Device Slot, and shall not be modified by any other command. The *Interrupter Target* field may be modified by an *Address Device Command* or *Evaluate Context Command*.

Note: After entering the *Addressed* state for the first time from the *Enabled* or *Default* states, the values of the Output Slot Context hub related fields (*Hub*, *TTT*, *MTT*, and *Number of Ports*) shall be initialized by the xHC by the first *Configure Endpoint Command* to transition the Slot from the *Addressed* to the *Configured* state. To change the Output Slot Context hub related fields, a Slot must first be transitioned through the *Enabled* or *Default* state.

4.5.3 Slot States

The current state of a *Device Slot* is identified by the **Slot State**. A subset of the possible Slot States are recorded in the *Slot State* field in the *Slot Context* data structure. The xHCI commands referenced in Figure 4-2 cause a Device Slot to transition from one state to another. Table 4-1 defines the Slot State codes.

Figure 4-2: Slot State Diagram


Disabled
Disable Slot
Enable Slot
Enabled
Address Device (BSR=0)
Reset Device
Reset Device
Configure Endpoint (DC=1)
Address Device (BSR=1)
Default
Address Device (BSR=0)
Addressed
Configure Endpoint (DC=0)
Configured
Reset Endpoint, Stop Endpoint, Evaluate Context, or Set TR Dequeue Pointer
Configure Endpoint (DC=1), Reset Endpoint, Stop Endpoint, Evaluate Context, Negotiate Bandwidth, or Set TR Dequeue Pointer
Configure Endpoint (DC=0), Reset Endpoint, Stop Endpoint, Evaluate Context, Negotiate Bandwidth, or Set TR Dequeue Pointer

Refer to Appendix E for state machine notation.

Note: The *Enabled*, *Default*, *Addressed*, and *Configured* states may transition to the *Disabled* state due to a *Disable Slot Command*, as noted by the large bubble.

Note: A Device Slot may be referred to as “enabled” if it is not in the *Disabled* state.

Note: Software shall not transition more than one Device Slot to the *Default* State at a time.

Note: When system software initially allocates and initializes the Output *Slot Context* data structure, it shall set the *Slot State* field to *Disabled* ('0'). All subsequent updates of the *Slot State* field shall be performed by the xHC.

Note: Unless otherwise stated, the unsuccessful completion of a command will not cause a state transition.

4.5.3.1 Device Slot State Codes

The following *Slot States* are maintained by the Host Controller. Refer to section 9.1 of the USB2 specification for information on the USB Device States.

Table 4-1: Device Slot State Code Definitions

Definition	USB Device State	Default Control EP State	Other EP State	USB Device Address	DCBAA Pointer	Slot Context *Slot State* value
Disabled	N/A	Disabled	Disabled	N/A	Not valid	Disabled
Enabled	Default	Disabled	Disabled	0	Not valid	Disabled
Default	Default	Not Disabled	Disabled	0	Valid	Default
Addressed	Address	Not Disabled	Disabled	Assigned	Valid	Addressed
Configured	Configured	Not Disabled	Any[13]	Assigned	Valid	Configured

Refer to Table 6-7 for the numeric encoding of Slot States.

Note: The *Slot State* field of the Slot Context data structure is used to convey a *subset* of the possible Slot States maintained by the xHC. The following sections identify the use of the *Slot State* field. Refer to section 6.2.2 for more information on the *Slot Context* data structure.

[13]Whether a non-Default Control endpoint is Disabled or not is determined by the *Configure Endpoint Command*.

4.5.3.2 Disabled

In this slot state the *Device Slot* is disabled, i.e. the slot's Doorbell register is disabled and the pointer to the slot's Output Device Context in the *Device Context Base Address Array* is invalid. The only command that software is allowed to issue for the slot in this state is the *Enable Slot Command*.

If the Output Slot Context is valid (i.e. an *Address Device Command* has been issued for the slot), the xHC shall set the *Slot State* field to *Disabled* upon the completion of a *Disable Slot Command*.

When in the *Disabled* state, the slot shall transition to the *Enabled* state due to the successful completion of an *Enable Slot Command*.

Note: Software shall not write to the Doorbell register of slots that are in the *Disabled* state.

Note: A Device Slot shall not generate events when it is in the *Disabled* state.

4.5.3.3 Enabled

In this slot state the *Device Slot* has been allocated to software by the *Enable Slot Command*, however the Doorbell register for the slot is not enabled and the pointer to the slot's Output Device Context in the *Device Context Base Address Array* is invalid. The only commands that software is allowed to issue for a slot in this state are the *Address Device* and *Disable Slot*.

When in the *Enabled* state, the slot shall transition to the *Default* state due to the successful completion of an *Address Device Command* with the *Block Set Address Request* (BSR) flag set to '1'.

When in the *Enabled* state, the slot shall transition to the *Addressed* state due to the successful completion of an *Address Device Command* with the *Block Set Address Request* (BSR) flag cleared to '0'.

When in the *Enabled* state, the slot shall transition to the *Disabled* state due to a *Disable Slot Command*.

Note: The *Enabled* state is a logical slot state that is maintained internally by the xHC. A unique value for the *Enabled* state is not defined for the Slot Context *Slot State* field in Table 6-7, i.e. the *Slot State* field value '0' is overloaded for the *Disabled* and *Enabled* states, refer to *Slot Context Slot State value* column in Table 4-1. Software initializes the Device Context data structure to '0', hence *Slot State* = *Disabled*. The Device Context is then assigned to the xHC with an *Address Device Command*. The *Address Device Command* also transitions the slot to the *Default* or *Addressed* state, so there never is a case where the xHC would actually set the *Slot State* field to *Enabled*.

Note: Software shall not write to the Doorbell register of slots that are in the *Enabled* state.

4.5.3.4 Default

In this slot state the USB device is in the Default state, the pointer to the *Device Slot's* Output *Device Context* in the *Device Context Base Address Array* is valid, the *Slot Context* and *Endpoint Context 0* in the Output *Device Context* have been initialized by the xHC, and the Doorbell register for the slot is enabled only for *DB Target = Control EP 0 Enqueue Pointer Update*. The only commands that software is allowed to issue for the slot in this state are the *Address Device (BSR = 0), Reset Endpoint, Stop Endpoint, Evaluate Context, Set TR Dequeue Pointer,* and *Disable Slot*.

When in the *Default* state, the slot shall transition to the *Addressed* state due to the successful completion of an *Address Device Command* with the *Block Set Address Request* (BSR) flag cleared to '0'.

When in the *Default* state, the slot shall transition to the *Disabled* state due to a *Disable Slot Command*.

Upon the completion of a *Evaluate Context*, *Reset Endpoint*, *Stop Endpoint,* or *Set TR Dequeue Pointer Command* while in the *Default* state, the slot shall remain in *Default* state.

The xHC shall set the Output Slot Context *Slot State* field to *Default* and the *USB Device Address* field to '0' when this state is entered.

Note: Software shall ensure that only one Device Slot is in the Default state at time, otherwise undefined behavior may occur.

4.5.3.5 Addressed

In this slot state the USB device is in the Address state, the pointer to the *Device Slot's* Output *Device Context* in the *Device Context Base Address Array* is valid, the *Slot Context* and *Endpoint Context 0* in the Output *Device Context* have been initialized by the xHC, and the Doorbell register for the slot is enabled only for *DB Target = Control EP 0 Enqueue Pointer Update*. The only commands that software is allowed to issue for the slot in this state are the *Evaluate Context*, *Configure Endpoint*, *Reset Endpoint*, *Stop Endpoint*, *Negotiate Bandwidth, Set TR Dequeue Pointer*, *Reset Device*, and *Disable Slot*.

When in the *Addressed* state, the slot shall transition to the *Configured* state due to the successful completion of a *Configure Endpoint Command* and the *Deconfigure* (DC) flag = '0'.

When in the *Addressed* state, the slot shall remain in the *Addressed* state due to the successful completion of a *Configure Endpoint Command* and the *Deconfigure* (DC) flag = '1', i.e. the *Configure Endpoint Command* is treated like a *No Op Command*.

When in the *Addressed* state, the slot shall transition to the *Default* state due to a *Reset Device Command*.

The xHC shall set the Output Slot Context *Slot State* field to *Addressed* when this state is entered.

Upon the completion of an *Evaluate Context*, *Stop Endpoint,* or *Set TR Dequeue Pointer Command* while in the *Addressed* state, the slot shall remain in *Addressed* state.

While in the *Addressed* state, the *Reset Device Command* may be used to transition the slot to the *Default* state.

When in the *Addressed* state, the slot shall transition to the *Disabled* state due to the successful completion of a *Disable Slot Command*.

4.5.3.6 Configured

In this slot state the USB device is in the Configured state, the pointer to the *Device Slot's* Output *Device Context* in the *Device Context Base Address Array* is valid, the *Slot Context*, *Endpoint Context 0*, and enabled IN and OUT Endpoint Contexts between 1 and 15 in the Output *Device Context* have been initialized by the xHC, and the *Device Context* doorbell for the slot is enabled for *DB Target = Control EP 0 Enqueue Pointer Update* and any enabled endpoint. The only commands that software is allowed to issue for the slot in this state are the *Configure Endpoint (DC = '0' or '1')*, *Reset Endpoint, Stop Endpoint, Set TR Dequeue Pointer*, *Evaluate Context*, *Reset Device*, *Negotiate Bandwidth*, and *Disable Slot*.

The xHC shall set the Output Slot Context *Slot State* field to *Configured* when this state is entered.

Upon the completion of an *Evaluate Context*, *Configure Endpoint*, *Reset Endpoint, Stop Endpoint*, *Negotiate Bandwidth, or Set TR Dequeue Pointer Command* while in the *Configured* state, the slot shall remain in *Configured* state.

Upon the completion of a "deconfigure" *Configure Endpoint Command (DC = '0')* while in the *Configured* state, the slot shall transition to the *Addressed* state.

When in the *Configured* state, the *Reset Device Command* may be used to transition the slot to the *Default* state.

When in the *Configured* state, the completion of a *Disable Slot Command* shall transition the slot to the *Disabled* state.

4.5.4 USB Standard Device Request to xHCI Command Mapping

The Standard Device Requests (as described in section 9.4 of the USB2 spec.) are generated to USB devices using Setup Stage TDs on a device's Default Control Endpoint. This section discusses the relationship of specific Standard Device Requests to xHCI commands. Refer to the USB or Device Class specifications for the order and timing of all other Standard Device Requests.

4.5.4.1 SET_ADDRESS Request

During the execution of the *Address Device Command* with *BSR* = '0', the xHC shall automatically issue a SET_ADDRESS request to a device with the *USB Device Address* assigned in the Output Slot Context and block any SET_ADDRESS Requests issued by software. Therefore a Setup Stage TD with the bmRequestType field set to Host-to-Device, Standard, and Device (0h), and the bRequest field set to SET_ADDRESS (5h) issued by software on the Default Control Endpoint shall not generate a Setup transaction on the USB and shall complete with a *TRB Error* completion code.

4.5.4.2 SET_CONFIGURATION Request

For a USB device to be successfully configured with new endpoint settings, system software shall complete a successful *Configure Endpoint* command to the xHC and a successful SET_CONFIGURATION request to a device. Undefined results may occur otherwise.

If software wishes to "deconfigure" a device by issuing a SET_CONFIGURATION Setup Stage TD with the Configuration Value (wValue) = '0', and issue a *Configure Endpoint Command* with all *Add Context* flags cleared to = '0', and the *Drop Context* flags of all enabled endpoints set to '1'. After both operations are completed successfully, the device is deconfigured.

Note: A *Configure Endpoint Command* is *not* necessary if a SET_CONFIGURATION request does not change any Endpoint Context parameters.

Refer to section 4.6.6 for more details.

4.5.4.3 SET_INTERFACE Request

For an alternate interface of a USB device to be successfully set, system software shall complete a successful *Configure Endpoint Command* and a successful SET_INTERFACE Setup request to a USB device. Undefined results may occur otherwise.

Note: A *Configure Endpoint Command* is *not* necessary if a SET_INTERFACE request does not change any Endpoint Context parameters.

Refer to section 4.6.6 for more details.

4.6 Command Interface

The command interface of the xHC is managed through the *Command Ring Control Register* (CRCR). The CRCR *Command Ring Pointer* field provides a pointer to the Command Ring. Software places commands on the Command Ring, then rings the Host Controller Doorbell Register to notify the xHC. The xHC processes the commands and generates Command Completion Events on the Primary Event Ring to notify software of their completion status. This section describes the operation of the Command Ring and each of the commands.

Refer to Table 3-1 for a summary of the xHCI command set.

Note: Undefined xHC behavior may result if commands and all data structures that they reference are not correctly formed by software. The algorithms below define checks that xHC should perform and the error conditions that may result when executing a command. The extent of command and data structure validity checking performed by an xHC implementation will vary. More comprehensive checking will ease the development and debugging process, but it is ultimately software's responsibility to ensure that the xHC does not receive invalid commands.

Note: A command shall return an *TRB Error* code if the command (i.e. *TRB Type*) is not recognized by the xHC.

Note: A command may return an Undefined Error or Vendor Defined Error codes. A vendor should identify the possible sources of these error codes to ease debugging and error handling.

Note: Software shall not ring the doorbell of an endpoint that has a state modifying command pending. The *Configure Endpoint*, *Evaluate Context*, *Reset Endpoint*, *Stop Endpoint*, and *Set TR Dequeue Pointer Commands* affect specific endpoints of a device. The *Address Device*, *Disable Slot*, and *Reset Device Commands* affect all endpoints of a device.

Note: Software shall be responsible for all command timeouts. If a command times out, software may abort the command using the mechanism described in section 4.6.1.2.

4.6.1 Command Ring Operation

The Command Ring is a dedicated TRB Ring (refer to section 4.9 for a description of TRB Ring operation), which only allows those TRB types defined in Table 6-86. Only one Command Ring exists per xHC instance.

System software is the producer of all Command TRBs and the xHC is the consumer.

The **Command Ring Dequeue Pointer** is an internal register maintained by the xHC, which is not directly exposed to software. Its value is reported in the *Command TRB Pointer* field of *Command Completion Events*.

The initial value of the Command Ring Dequeue Pointer is defined by the *Command Ring Pointer* field in the *Command Ring Control Register* (CRCR), described in section 5.4.5. The *Command Ring Pointer* field shall be set by system software to point to the Command Ring prior to running the xHC (i.e. setting the *Run/Stop* (R/S) flag to '1' and ringing the *Host Controller Command Doorbell* for the first time). The *Command Ring Pointer* field may only be modified by software while the Command Ring is stopped, as indicated by the *Command Ring Running* (CRR) flag equal to '0'.

A Work Item on a Command Ring is called a *Command Descriptor* (CD). CDs enable the management of Device Slots, virtualization, and the controller as a whole. A CD shall be comprised of one Command TRB data structure. Refer to section 4.11.4 for information on the commands supported by the xHCI and section 6.4.3 for details of the Command TRB data structures.

Commands are issued by software to the xHC by:

1. Placing one or more Command Descriptors on the Command Ring and

2. Ringing the *Host Controller Doorbell*.

To ring the *Host Controller Doorbell* software shall write the *Host Controller Doorbell* register (offset 0 in the *Doorbell Register Array*), asserting the *Host Controller Command* value in the *DB Target* field and '0' in the *DB Stream ID* field.

The xHC, upon detecting a *Host Controller Command* Doorbell ring, shall execute commands until the Command Ring is stopped or empty.

Note: If multiple commands are posted to the Command Ring, they are executed in order, so a delay may be incurred before a particular command is executed.

The xHC shall generate a *Command Completion Event* for every command. The *Command TRB Pointer* field of the *Command Completion Event* shall point to the Command TRB that initiated the event. The *Completion Code* field of the *Command Completion Event* shall indicate the completion status of the command. The *Slot ID and VF ID* fields shall reflect the values of the respective fields of the Command TRB that initiated the event.

The Primary Event Ring receives all *Command Completion Events*.

The *Command Completion Events* that result from processing the commands shall be ordered with respect to their location in the Command Ring.

Command execution times are xHC implementation defined.

The standard and optional commands supported by the xHCI are listed in Table 1.

xHC vendors may define proprietary commands using the *Vendor Defined* TRB Type codes identified in Table 6-86. All vendor defined commands shall utilize the *Command Completion Event TRB* to report completions.

4.6.1.1 Stopping the Command Ring

System software may stop the execution of commands on the Command Ring by writing a '1' to the *Command Stop* (CS) bit of the *Command Ring Control* register. Writing a '1' to the *CS* bit shall stop the xHC from fetching additional CDs after the currently executing command completes, "stopping" the Command Ring. After the Command Ring has been successfully stopped, a *Command Completion Event* shall be generated with the *Completion Code* set to *Command Ring Stopped* and the *Command TRB Pointer* set to the current value of the Command Ring Dequeue Pointer.

While the Command Ring is stopped, ownership of all Command Descriptors on the ring is passed to software, which may remove, add, or rearrange Command Descriptors. Software restarts command execution by writing the *Host Controller Doorbell* register with the *DB Reason* field set to *Host Controller Command*. If the *Command Ring Pointer* field of the *Command Ring Control Register* (CRCR) was written while the ring is stopped the xHC shall restart Command Ring execution at the new value defined by the CRCR write, otherwise Command Ring execution shall restart at the current Dequeue Pointer value, i.e. the TRB following the last command executed (or aborted). Software may modify the value of the Command Ring Dequeue Pointer prior to restarting it by writing a new value to the *Command Ring Pointer* field of the *Command Ring Control* register.

4.6.1.2 Aborting a Command

System software may abort the execution of the current command by writing a '1' to the *Command Abort* (CA) bit of the *Command Ring Control* register. Aborting a command on the Command Ring shall perform the following operations:

- If a command is currently executing:
  - A *Command Completion Event* shall be generated for the aborted command with its *Completion Code* set to *Command Aborted*.
  - Advance the Command Ring Dequeue Pointer to point to the next Command TRB.
- Generate a *Command Completion Event* with the *Completion Code* set to *Command Ring Stopped* and the *Command TRB Pointer* set to the current value of the Command Ring Dequeue Pointer.

Software may follow the method described in section 4.6.1.1 to restart the "stopped" Command Ring.

Note: If the xHC detects the assertion of an abort request between the execution of two commands or after the last command, a *Command Completion Event* with the *Completion Code* set to *Command Aborted* may not be found on the Event Ring after an abort operation.

IMPLEMENTATION NOTE

Aborting Commands

Typically when software asserts the *Command Abort* (CA) flag, the Command Ring will normally stop after the completion of a command, i.e. *Completion Code* is not equal to *Command Aborted* in the last Event Ring *Command Completion Event TRB*. Only if a command is "blocked" will it be aborted.

An example of a command that may hang is the *Address Device Command*, because waiting for a SET_ADDRESS request to be acknowledged by a USB device is outside of the xHC's ability to control.

An xHC implementation should "checkpoint" the state associated with a command before a command is initiated. If the *CA* flag is set before the command is complete (e.g. its *Command Completion Event TRB* is posted to the Event Ring), then the command's previous state should be restored by the xHC using the checkpoint information and its *Completion Code* shall be set to *Command Aborted*.

Software should time the completion of all xHCI commands, including the Command Abort operation, i.e. the delay between the negation of *CRR* ('0') and the assertion of CA ('1'). If software doesn't see *CRR* negated in a timely manner (e.g. longer than 5 seconds), then it should assume that the there are larger problems with the xHC and assert *HCRST*.

4.6.2 No Op

The *No Op* command can be issued by software to exercise the TRB Ring mechanism of the xHC without affecting any xHC or USB Device state, or to report the current value of the Command Ring Dequeue Pointer.

Note: A *No Op Command* may be inserted on the Command Ring by software to modify the alignment memory boundaries of Command TDs.

The format of the *No Op Command TRB* is defined in section 6.4.3.1.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

To issue a *No Op Command*, system software shall perform the following operations:

- Insert a *No Op Command TRB* on the Command Ring and initialize the following fields:
  - *TRB Type* = *No Op Command* (refer to Table 6-86).
  - Clear all other fields of the command TRB to '0'.

    - *Cycle bit* = Command Ring's PCS flag. Refer to section 4.9.2 for a discussion of the *Cycle bit* and PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

When a *No Op Command* is executed by the xHC it shall perform the following operations:

- Insert a *Command Completion Event* on the Event Ring and initialize the following fields:
    - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
    - *Command TRB Pointer* = The address of the No Op Command TRB.
    - *Completion Code* = *Success* (refer to Table 6-85).
    - Clear all other fields of the event TRB to '0'.
    - *Cycle bit* = Event Ring's PCS flag.

4.6.3 Enable Slot

The *Enable Slot Command* is issued by software to obtain an available Device Slot and to transition a Device Slot from the *Disabled* to the *Enabled* state. Refer to section 3.3.2 for a high level description of the *Enable Slot Command* and it's usage.

When an *Enable Slot Command* is processed by the xHC, it will look for an available Device Slot. If a slot is available, the *ID* of a selected slot will be returned in the *Slot ID* field of a successful *Command Completion Event* on the Event Ring. If a Device Slot is not available, the *Slot ID* field shall be cleared to '0' and a *No Slots Available Error* shall be returned in the *Command Completion Event*.

Upon the successful completion of an *Enable Slot Command*, system software shall use the *Slot ID* to link a *Device Context* data structure to the slot by writing a pointer to the *Device Context* in the *Device Context Base Address Array[Slot ID]* location. Undefined operation may occur if the *Context Base Address Array* entry is not updated prior to issuing a Command for the slot, or ringing the Default Control Endpoint (0) doorbell.

To ensure proper operation of the xHC, system software shall provide "valid" *Input Control Context*, *Slot Context* and *Endpoint Context* data structures in the *Input Context* data structure.

The requirements of a valid *Slot Context* data structure are defined in section 6.2.2.

The requirements of a valid *Endpoint Context* data structure are defined in section 6.2.3.1.

The format of the *Enable Slot Command TRB* is defined in section 6.4.3.2.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

Sections 6.2.2.1 and 6.2.3.1 also define the *Completion Code* values that will be found in the *Command Completion Event* if an invalid context is detected.

To issue an *Enable Slot Command*, system software shall perform the following operations:

- Insert an *Enable Slot Command TRB* on the Command Ring and initialize the following fields:
  - *TRB Type* = *Enable Slot* command (refer to Table 6-86).
  - *Slot Type* = value specified by the *Protocol Slot Type* field of the associated *xHCI Supported Protocol* Capability structure (refer to Table 7-9).
  - Clear all other fields of the command TRB to '0'.
  - *Cycle bit* = Command Ring's PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

When an *Enable Slot Command* is executed by the xHC it shall perform the following operations:

- Insert a *Command Completion Event* on the Event Ring and initialize the following fields:
  - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
  - *Command TRB Pointer* = The address of the *Enable Slot Command TRB*.
  - Determine if a Device Slot is available.
  - If a Device Slot is available:
    - *Slot ID* = ID of the selected Device Slot.
    - *Completion Code* = *Success* (refer to Table 6-85).
  - else // Device Slot is not available
    - *Slot ID* = '0'.
    - *Completion Code* = *No Slots Available*.
  - Clear all other fields of the event TRB to '0'.
  - *Cycle bit* = Event Ring's PCS flag.

The algorithm for Device Slot ID selection is xHC implementation dependent.

Note: If this command is aborted (i.e. *Completion Code* = *Command Aborted*) the *Slot ID* field should be considered by software to be invalid (e.g. no slot was allocated).

4.6.4 Disable Slot

The *Disable Slot Command* is issued by software to force a Device Slot to the *Disabled* state. A typical use would be to free a Device Slot when a USB device is disconnected.

When a *Disable Slot Command* is processed by the xHC it shall:

- Disable the Doorbell register for the slot
- Free any bandwidth allocated to the periodic endpoints of the device
- Terminate any slot related USB activity (e.g. packet transfers)
- Free any internal resources associated with the slot
- Internally flag the slot as "available" for subsequent reassignment by an *Enable Slot Command*. i.e. the *Device Context Base Address Array* entry for the slot is no longer considered valid by the xHC and software can free the *Device Context*, *Transfer Ring*, *Stream Context Array*, etc. data structures associated with the slot.

A Command Completion Event is always returned for a Disable Slot Command.

The format of the Disable Slot Command TRB is defined in section 6.4.3.3.

The format of the Command Completion Event TRB is defined in section 6.4.2.2.

Note: Before software issues a *Disable Slot Command* the following conditions shall be true, otherwise undefined behavior may occur:

- Any active endpoints associated with the slot shall be in the *Stopped* state or Idle in the *Running* state, and any outstanding Transfer Events shall have been received.
- Any commands targeted at the slot that is being disabled shall be complete, i.e. any outstanding Command Completion Events for the slot have been received.

To issue a *Disable Slot Command*, system software shall perform the following operations:

- Insert a *Disable Slot Command* on the Command Ring and initialize the following fields:
  - *TRB Type* = *Disable Slot Command* (refer to Table 6-86).
  - *Slot ID* = ID of the Device Slot to be disabled.
  - Clear all other fields of the command TRB to '0'.
  - *Cycle bit* = Command Ring's PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

When a *Disable Slot Command* is executed by the xHC it shall perform the following operations:

- Insert a *Command Completion Event* on the Event Ring and initialize the following fields:
  - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
  - *Command TRB Pointer* = The address of the *Disable Slot Command TRB*.
  - *Slot ID* = The value of the command's Slot ID.

- If the Device Slot identified by the *Slot ID* has been previously enabled by an *Enable Slot Command*:
  - Any xHC resources assigned to the Device Slot are freed and the Device Slot is made available for reassignment.
  - The *Slot State* of the associated Slot Context is set to *Disabled*.
  - *Completion Code* = *Success* (refer to Table 6-85).
- else // The slot has not been enabled by an *Enable Slot Command*
  - *Completion Code* = *Slot Not Enabled Error*.
- Clear all other fields of the event TRB to '0'.
- *Cycle bit* = Event Ring's PCS flag.

Note: After software receives the Command Completion TRB for a *Disable Slot Command* it shall clear the respective DCBAA entry to '0'. This action allows the xHC to identify valid vs. invalid Device Slots after a Restore State operation.

Note: Any pending events not already posted to an Event Ring may be aborted when this command is executed.

4.6.5 Address Device

The *Address Device Command* is issued by software to transition a Device Slot from the *Enabled* to the *Default* or *Addressed* state or from the *Default* to the *Addressed* state, depending on the state of the *Block Set Address Request* (BSR) flag.

When an *Address Device Command* is processed by the xHC it shall enable the device's Default Control Endpoint, select an address for the USB device, and issue a USB SET_ADDRESS request to the USB Device. The SET_ADDRESS request for a USB2 device shall be issued to Address '0'. The SET_ADDRESS request for a USB3 device shall be issued using the *Route String*.

Upon successful completion of an *Address Device Command*, the Default Control Endpoint will be added to the xHCs' endpoint scheduling list, the Default Control Endpoint 0 Context Doorbell shall be enabled, and TRBs can be posted to its endpoint Transfer Ring.

A *USB Transaction Error* shall be generated if an error is detected on the USB SET_ADDRESS request and the Device Slot shall not transition to the *Addressed* state.

Once a successful *Address Device Command* has completed, system software can issue USB GET_DESCRIPTOR requests through the Default Control Endpoint to retrieve the USB Device, Configuration, etc. descriptors from the USB device. Using the information in these descriptors system software may determine which Class Driver(s) to load for the USB device, and hand off the device.

Note: A USB SET_ADDRESS request does not include a data stage, so the default Max Packet Size is sufficient to issue the request. However subsequent USB device requests require that the xHC use the Max Packet Size defined by the device. The first request that system software should issue to a USB Device is a GET_DESCRIPTOR request with the wLength set to 8, to retrieve is the USB *Device Descriptor*. The last byte of the returned partial Device Descriptor (bMaxPacketSize0) identifies the maximum packet size of the Default Control Endpoint. This value shall be used by system software to update the Max Packet Size field in the Control Endpoint 0 Context.

Note: If the *Block Set Address Request* (BSR) flag is '0' in the *Address Device Command TRB*, then the xHC shall select a *USB Device Address* and issue a SET_ADDRESS request to a USB device as part of an *Address Device Command*. If the *Block Set Address Request* (BSR) flag is '1' then the xHC shall *not* issue a SET_ADDRESS request to a USB device as part of an *Address Device Command*. In either case, all other operations described in this section for the Address *Device Command* are performed. The BSR flag may be used by software to provide compatibility with legacy USB devices which require their Device Descriptor to be read before receiving a SET_ADDRESS request.

Note: If the xHC detects a SET_ADDRESS request on the Default Control Endpoint Transfer Ring, it shall generate a *TRB Error* Completion Status for the TD. The xHC shall never forward a SET_ADDRESS request on a Default Control Endpoint Transfer Ring to a USB device.

The format of the *Address Device Command TRB* is defined in section 6.4.3.4.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

The *Address Device Command* utilizes the *Address Device Command TRB* data structure defined in section 6.4.3.4, which points to an *Input Context* data structure defined in section 6.2.5.

The *Add Context* flags *A0* and *A1* of the *Input Control* Context data structure (in the *Input Context*) shall be set to '1', and all remaining *Add Context* and *Drop Context* flags shall all be cleared to '0'.

System software shall initialize *Slot Context* and *Endpoint Context 0* entries of the *Input Context*. All other *Endpoint Contexts* in the *Input Context* shall be ignored by the xHC during the execution of this command.

To issue an *Address Device Command*, system software shall perform the following operations:

- Ensure that the *Device Context Base Address Array* entry points to a properly sized and initialized *Device Context* data structure for the device.
- Allocate and initialize an *Input Context* data structure for the command.
    - The *Add Context* flags for the *Slot Context* and the *Endpoint 0 Context* shall be set to '1'. All fields of the Input Context *Slot Context* data structure shall define

valid values, refer to section 4.5.2. The *Endpoint 0 Context* data structure in the *Input Context* shall define valid values for the *TR Dequeue Pointer*, *EP Type, Error Count* (CErr), and *Max Packet Size fields*. The *MaxPStreams, Max Burst Size*, and *EP State* values shall be cleared to '0'.

- Insert an *Address Device Command* on the Command Ring and initialize the following fields:
  - *TRB Type* = *Address Device* command (refer to Table 6-86).
  - *Slot ID* = ID of the target Device Slot.
  - *Input Context Pointer* = The base address of the *Input Context* data structure.
  - Clear all other fields of the command TRB to ‘0’.
  - *Cycle bit* = Command Ring’s PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

Note: A Slot or Endpoint Context contained in the *Input Context* is referred to as an ***Input*** Slot or Endpoint Context. And a Slot or Endpoint Context contained in the *Device Context* data structure pointed to by the *Device Context Base Address Array* is referred to as an ***Output*** Slot or Endpoint Context and the Device Context itself is referred to as the *Output Device Context*.

When an *Address Device Command* is executed by the xHC it shall perform the following operations:

- Insert a *Command Completion Event* on the Event Ring and initialize the following fields:
  - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
  - *Command TRB Pointer* = The address of the *Address Device Command TRB*.
  - *Slot ID* = The value of the command’s Slot ID.
  - If the Device Slot identified by the command’s *Slot ID* field has been previously enabled by an *Enable Slot Command*:
    - Retrieve the pointer to the Output *Device Context* of the selected Device Slot.
    - If the *Block Set Address Request* (BSR) flag = ‘1’
      - If the slot is in the *Enabled* state:
        - Copy all fields of the Input Slot Context to the Output Slot Context.
        - Copy all fields of the Input Endpoint 0 Context to the Output Endpoint 0 Context.
        - Set the *Endpoint State* (EP State) field in the Output Endpoint 0 Context to *Running*.
        - Set the *Slot State* in the Output Slot Context to *Default*.
        - Set the *USB Device Address* field in the Output Slot Context to ‘0’.
        - *Completion Code* = *Success* (refer to Table 6-85).
      - else // The slot is not in the *Enabled* state:
        - *Completion Code* = *Context State Error*.

- else // BSR = '0'
  - If the slot is in the *Enabled* or *Default* state:
    - Select a Device Address for the target USB device.
    - Construct a SET_ADDRESS request to be sent the device
      - bmRequestType = 0.
      - wValue = Selected Device Address.
      - wIndex = 0.
      - wLength = 0.
    - Retrieve the *Route String* from the Input Slot Context.
    - Issue a SET_ADDRESS request to the target USB device.
    - If the SET_ADDRESS request is successful:
      - Copy all fields of the Input Slot Context to the Output Slot Context.
      - Copy all fields of the Input Endpoint 0 Context to the Output Endpoint 0 Context.
      - Set the *Endpoint State* (EP State) field in the Output Endpoint 0 Context to *Running*.
      - Set the *Slot State* in the Output Slot Context to *Addressed*.
      - Set the *USB Device Address* field in the Output Slot Context to the address selected for the USB device by the xHC.
      - *Completion Code* = *Success* (refer to Table 6-85).
    - else // SET_ADDRESS request is not successful
      - *Completion Code* = *USB Transaction Error*.
  - else // The slot is not in the *Enabled* or *Default* state:
    - *Completion Code* = *Context State Error*.
- else // The slot has not been enabled by an *Enable Slot Command*
  - *Completion Code* = *Slot Not Enabled Error*.
- Clear all other fields of the event TRB to '0'.
- *Cycle bit* = Event Ring's PCS flag.

Note: The xHC should check that all referenced contexts are valid before executing the command. If an invalid context is detected, the state of the Output *Device Context* shall not change and the a *Command Completion Event* shall be generated with the *Completion Code* set to *Parameter Error*.

Note: The Slot Context (*Add Context* flag 0 (A0)) and the Default Endpoint Context (*Add Context* flag 1 (A1)) shall be valid in the Input Context referenced by the *Address Device Command*. All other Endpoint Contexts (*A2* to *A31*) in the Input Context shall be ignored by the xHC.

Note: If the SET_ADDRESS request was unsuccessful, system software may issue a *Disable Slot Command* for the slot or reset the device and attempt the *Address*

Device Command again. An unsuccessful *Address Device Command* shall leave the Device Slot in the *Default* state.

Note: If an *Address Device Command* is received and all available USB Device Addresses have been assigned for the BI that the device is associated with, then a *Command Completion Event* shall be generated with the *Completion Code* set to *Resource Error*.

Note: Software shall be responsible for timing the SetAddress() "recovery interval" required by USB and aborting the command on a timeout. Refer to section 9.2.6.3 in the USB2 spec.

Note: If *BSR* = '0' and this command is aborted (i.e. *Completion Code* = *Command Aborted*), software should assume that the USB device is in an unknown state (e.g. the USB device may or may not be in the Address state) and take the appropriate action to recover it to a known state, otherwise undefined behavior may occur.

Note: A *USB Transaction Error* Completion Code for an *Address Device Command* may be due to a Stall response from a device. Software should issue a *Disable Slot Command* for the Device Slot then an *Enable Slot Command* to recover from this error. Refer to section 4.11.2.2 Implementation note.

Note: All endpoints shall be in the **Stopped** state or if in the **Running** state, shall be "idle" (e.g. no USB Transactions are in progress, the Transfer Ring is empty, and software has processed all outstanding events for the Transfer Ring) when this command is executed. If this condition is not met undefined behavior may occur.

Note: If an *Address Device Command* fails with USB Transaction Error and the target device is behind a TT, software shall issue a ClearFeature(CLEAR_TT_BUFFER) request to TT in the HS hub.

Refer to section 6.2.1 for the definition of a *Device Context* data structure and its access constraints.

The requirements of a "valid" *Slot Context* data structure are defined in section 6.2.2.1.

The requirements of a "valid" *Endpoint Context* data structure are defined in section 6.2.3.1.

4.6.6 Configure Endpoint

The *Configure Endpoint Command* is issued by software to enable, disable, or reconfigure endpoints associated with a target configuration.

The format of the *Configure Endpoint Command TRB* is defined in section 6.4.3.5.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

This command is issued by software under the following circumstances:

- **Configuring a device.** To set a configuration in a device, software shall issue a *Configure Endpoint Command* to the xHC in conjunction with issuing USB SET_CONFIGURATION request to the device. This command shall be used to enable the set of Device Slot endpoints selected by the target configuration, and transition a Device Slot from the *Addressed* to the *Configured* state. Undefined behavior may occur if TDs are posted for endpoints enabled by this command and the SET_CONFIGURATION request associated with this command is not successfully completed by the USB device.
- **Deconfiguring a device.** To deconfigure a device, software shall issue a *Configure Endpoint Command* to the xHC in conjunction with "deconfiguring" the device. A USB device is "deconfigured" by issuing a SET_CONFIGURATION request to a USB device with configuration '0' selected. Software shall issue a *Configure Endpoint Command* with *Deconfigure* (DC) = '1' to inform the xHC that a SET_CONFIGURATION request with a configuration value of zero shall be sent to the device. This command shall be used to disable all enabled endpoints (except for the Default Control Endpoint) of a Device Slot, and transition the Device Slot from the *Configured* to the *Addressed* state. Undefined USB device behavior may occur if the SET_CONFIGURATION request associated with this command is not successfully completed.

Note: Setting the *Deconfigure* (DC) flag to '1' in the *Configure Endpoint Command TRB* is equivalent to setting Input Context *Drop Context* flags 2-31 to '1' and *Add Context* 2-31 flags to '0'. If the *DC* flag = '1', the *Input Context Pointer* field shall be ignored by the xHC and the Output Slot Context *Context Entries* field shall be set to '1'.

Note: If the device only has a Default Control Endpoint, then a *Configure Endpoint Command* is *not* necessary prior to issuing a SET_CONFIGURATION "deconfigure" request to a device.

- **Setting an Alternate Interface on a device.** To set an Alternate Interface on a device, software shall issue a *Configure Endpoint Command* to the xHC in conjunction with issuing USB SET_INTERFACE request to the device. This command shall be used to disable, enable, or reconfigure a selected set of endpoints determined by the target Alternate Interface. Undefined behavior may occur if the SET_INTERFACE request associated with this command is not successfully completed.

Note: A USB device presents one or more Configuration options to system software. System software selects a specific configuration with a USB SET_CONFIGURATION request. Also, each Interface defined by a Configuration may optionally present multiple Alternate Interface settings. System software selects a specific Alternate Interface setting with a USB SET_INTERFACE request. The result of the USB SET_CONFIGURATION and SET_INTERFACE requests allow system software to enable a selected set endpoints on a USB device. The specific endpoints enabled by a Configuration or Alternate Interface setting depend on the respective descriptors reported by the device. The xHC does ***not*** maintain information about the relationships between the Configuration and

Alternate Interface options presented by a USB device and the endpoints enabled by a specific configuration option. System software shall use the *Add Context* and *Drop Context* flags of the *Configure Endpoint Command* to explicitly identify to the xHC the endpoints of a Device Slot that shall be enabled due to the selected USB device Configuration and Alternate Interface settings.

Note: Slot or Endpoint Contexts are found in Device and Input Contexts. A Slot or Endpoint Context contained in the *Input Context* is referred to as an ***Input*** Slot or Endpoint Context, and a Slot or Endpoint Context contained in the *Device Context* data structure is referred to as an ***Output*** Slot or Endpoint Context.

The *Add Context* flag *A1* and *Drop Context* flags *D0* and *D1* of the Input *Control Context* (in the *Input Context*) shall be cleared to '0'. *Endpoint 0 Context* does not apply to the *Configure Endpoint Command* and shall be ignored by the xHC. A0 shall be set to '1' and refer to section 6.2.2.2 for the Slot Context fields used by the *Configure Endpoint Command*. The state of the remaining *Add Context* and *Drop Context* flags depend on the specific endpoints affected by the command. System software shall initialize the *Endpoint Contexts* of the *Input Context* referenced by *Add Context* flags. All *Endpoint Context* data structures not referenced by an *Add Context* flag shall be ignored by the xHC. Note that *Endpoint Context* flags referenced only by a *Drop Context* flag does not need to be initialized. Refer to section 6.2.3.2 for the Endpoint Context fields used by the *Configure Endpoint Command.*

Note: An endpoint shall be in the *Stopped* state or if in the *Running* state shall be "idle" (e.g. no USB Transactions are in progress, the Transfer Ring is empty, and software has processed all outstanding events for the Transfer Ring) if its *Drop Context* flag is set. If this condition is not met undefined behavior may occur.

The following rules apply to processing a *Configure Endpoint Command*:

- The xHC resources assigned to a Device Slot are not modified until after all *Drop Context* and *Add Context* flags are evaluated.
- The *Slot State* field of a Device *Slot Context* is not modified until after all *Drop Context* and *Add Context* flags are evaluated.
- The xHC maintains a global *Resources Available* variable, which is initialized to indicate all xHC resources are available. A **Resource** is an xHC implementation defined metric, which refers to the internal xHC data structures, buffer space, or other implementation specific resources required to support an endpoint type.
- For each USB bus instance, a *Bandwidth Available* variable is maintained, which initialized to the respective maximum available value. **Bandwidth** is a commodity allocated by the host controller. Refer to section 4.14.2 (Reserved Bandwidth) for more information on how bandwidth requirements are calculated for an endpoint.
- Two temporary variables are maintained by the xHC when evaluating the *Configure Endpoint Command*: **Resource Required** and **Bandwidth Required**. Both variables are initialized to '0'.

- The *Resource Required* variable identifies the "sum" of xHC resources required to support all endpoints affected by a *Configure Endpoint Command*. Note that the "units" of xHC resource measurement is an implementation specific value.
- The *Bandwidth Required* variable identifies the "sum" of USB bandwidth necessary to support all endpoints affected by a *Configure Endpoint Command*.

- The *Drop Context* flags are evaluated before the *Add Context* flags.
- For each endpoint indicated by a *Drop Context* flag = '1':
  - If the Output Endpoint Context is not in the *Disabled* state:
    - The endpoint related resources are subtracted from the *Resource Required* variable.
    - If the endpoint is periodic, then the bandwidth assigned to the endpoint is subtracted from the *Bandwidth Required* variable.
  - else // Output Endpoint Context is in the *Disabled* state
    - Do nothing
- For each Input Endpoint Context indicated by an *Add Context* flag = '1':
  - The resources required to support the endpoint described by the Input Endpoint Context shall be added to the *Resource Required* variable.
  - If the endpoint described by the Input Endpoint Context is periodic, then the bandwidth required to support the endpoint shall be added to the *Bandwidth Required* variable.
- If the *Drop Context* flag is set for an endpoint and the Output Endpoint Context is in the *Disabled* state, the *Drop Context* flag shall be ignored and no resource or bandwidth evaluation shall be performed for the endpoint.
- After all *Drop Context* and *Add Context* flags are evaluated the xHC determines whether the command was successful:
  - The *Resources Required* variable is compared to the *Resources Available* variable, if the result indicates an oversubscription of resources by the command (i.e. *Resources Available - Resources Required is less than* 0), then the command shall be unsuccessful and a *Resource Error* Completion Code shall be returned in the *Command Completion Event*. Refer to section 4.14.1.1 for more information on xHC resources.
  - The *Bandwidth Required* variable is compared to the *Bandwidth Available* variable, if the result indicates an oversubscription of bandwidth by the command (i.e. *Bandwidth Available - Bandwidth Required is less than* 0), then the command shall be unsuccessful and a *Bandwidth Error* Completion Code shall be returned in the *Command Completion Event*.
  - If the Resource and Bandwidth requirements of the command can be met, then the command is successful and a *Success* Completion Code shall be returned in the *Command Completion Event*.
- If the command is unsuccessful:

    - Current xHC resource allocations shall be unchanged for the endpoint.
    - Current xHC bandwidth allocations shall be unchanged for the endpoint.
    - The Output Slot Context *Slot State* field shall be unchanged for the device.
    - The Output Endpoint Contexts referenced by the command in the Device Context shall be unchanged.
    - The *Command Completion Event* shall indicate the appropriate error Completion Code.

xHC behavior is undefined if the *Drop Context* (D) flag is '0', the *Add Context* (A) flag is '1', and the Output Endpoint Context is not in the *Disabled* state (i.e. software is trying to add an endpoint without dropping its current resources).

- If the command is successful:
    - The *Resources Available* variable shall be updated to reflect the new resource allocation.
    - The *Bandwidth Available* variable shall be updated to reflect the adjusted bandwidth allocation.
    - For each endpoint:
        - If the *Drop Context* flag is '0' and the *Add Context* flag is '0', the xHC shall:
            - Do nothing.
            - The respective Input *Endpoint Context* is ignored by the xHC.
        - If the *Drop Context* flag is '1' and the *Add Context* flag is '0', the xHC shall:
            - Drop the endpoint from its pipe scheduling list if it is scheduled.
            - Set the *Endpoint State* (EP State) field of the Output *Endpoint Context* to *Disabled.*
            - The Input *Endpoint Context* data structure is ignored by the xHC.
        - If the *Drop Context* flag is '0' and the *Add Context* flag is '1', the xHC shall:
            - Add the endpoint to its pipe scheduling list.
            - All fields of the Input Endpoint Context data structure in the *Configure Endpoint Context* are copied to the Output Endpoint Context fields in the *Device Context*.

              Note that when the Input Endpoint Context is copied to the Output Endpoint Context, the ownership of a Stream Context Array pointed to by the Input *TR Dequeue Pointer* is passed from software to the xHC.
            - The *Endpoint State* (EP State) field of the Output Endpoint Context is set to *Running*.
            - Enable the associated *Device Context Doorbell*.
        - If the *Drop Context* flag is '1' and the *Add Context* flag is '1', the xHC shall:
            - Release the current Resources and Bandwidth allocated to the endpoint

        and assign the new Resources and Bandwidth requested for the endpoint.
    - All fields of the Input Endpoint Context data structure in the *Configure Endpoint Context* are copied to the Output Endpoint Context fields in the *Device Context*.

      Note that when the Input Endpoint Context is copied to the Output Endpoint Context, the ownership of a Stream Context Array pointed to by the Input *TR Dequeue Pointer* field is passed from software to the xHC. Software shall not deallocate any Stream Context Array data structures while they are owned by the xHC. It is software's decision whether to set the Input *TR Dequeue Pointer* equal to the Output *TR Dequeue Pointer*, thus reusing the currently allocated Stream Contexts/Transfer Rings, or allocating new data structures and changing the Input *TR Dequeue Pointer* value. If new data structures are allocated, software shall be responsible for recovering the old data structures after the command completes.
    - Set the *Endpoint State* (EP State) field of the Output Endpoint Context to *Running.*
  - If the device is "deconfigured" by this command (i.e. all Output Endpoint Contexts (DCI 2-31) are in the *Disabled* state), the Output Slot Context *Slot State* field shall be set to the *Addressed* state by the xHC.
  - If any Output Endpoint Context (2 through 31) is not in the *Disabled* state, the Output Slot Context *Slot State* field shall be set to the *Configured* state by the xHC.
  - The *Command Completion Event* Completion Code shall indicate *Success*.

When this command is used to "Set an Alternate Interface on a device", software shall set the *Drop Context* and *Add Context* flags as follows:

- If an endpoint is not modified by the Alternate Interface setting, then software shall set the *Drop Context* and *Add Context* flags to '0'.
- If an endpoint previously disabled, is enabled by the Alternate Interface setting, then software shall set the *Drop Context* flag to '0' and *Add Context* flag to '1', and initialize the Input Endpoint Context.
- If an endpoint previously enabled, is disabled by the Alternate Interface setting, then software shall set the *Drop Context* flag to '1' and *Add Context* flag to '0'.
- If a parameter of an enabled endpoint is modified by an Alternate Interface setting, the *Drop Context* and *Add Context* flags shall be set to '1'.

When configuring or deconfiguring a device, only after completing a successful *Configure Endpoint Command* and a successful USB SET_CONFIGURATION request may software schedule data transfers through a newly enabled endpoint or Stream Transfer Ring of the Device Slot.

When setting an Alternate Interface on a device, only after completing a successful *Configure Endpoint Command* and a successful USB SET_INTERFACE request may software schedule data transfers through a newly enabled endpoint or Stream Transfer Ring of the Device Slot.

When the command is complete, a *Command Completion Event* is posted to the *Event Ring* indicating the success or failure of the command.

If the *Slot State* is *Disabled* when a *Configure Endpoint Command* is received, the xHC shall generate a *Slot Not Enabled Error* on the Event Ring.

The xHC shall reject a *Configure Endpoint Command* with *Bandwidth Error* if it determines that the bandwidth required by the configuration is not available.

The xHC shall reject a *Configure Endpoint Command* with *Resource Error* if it determines that it does not have enough internal resources (buffer space, etc.) available to service all the endpoints defined in the configuration.

If the configuration defines periodic endpoints, system software may optionally issue a *Negotiate Bandwidth Command* to cause the xHC to renegotiate bandwidth with other devices. Refer to section 4.16.1 for more information on bandwidth negotiation.

Upon successful completion of a *Configure Endpoint Command*, the enabled endpoints will be added to the xHCs' pipe scheduling list, the respective *Device Context Doorbells* shall be enabled, and TRBs can be posted to any enabled endpoint or Stream Transfer Ring.

Refer to section 4.11.4.5 for more information on the *Configure Endpoint Command*.

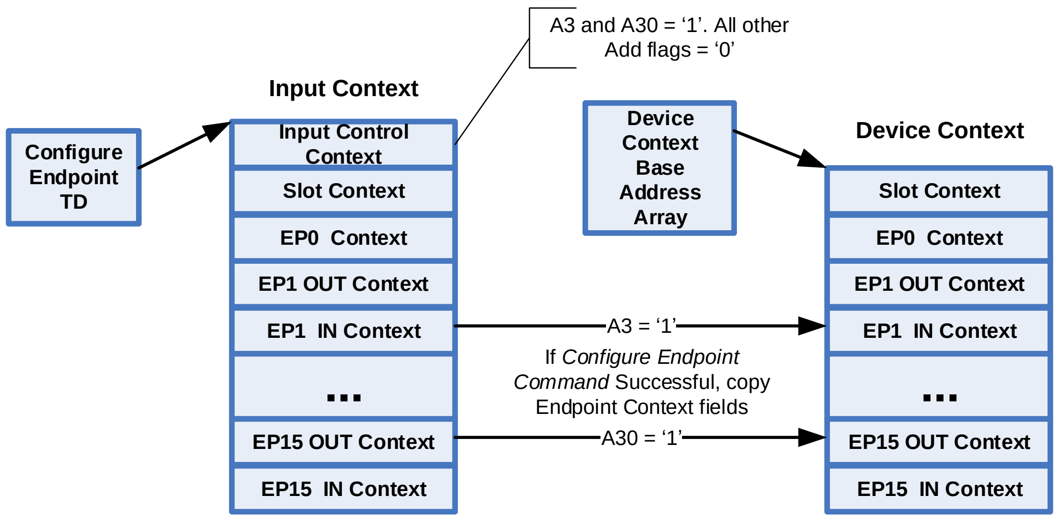
The requirements of a "valid" *Slot Context* data structure are defined in section 6.2.2.2.

The requirements of a "valid" *Endpoint Context* data structure are defined in section 6.2.3.2.

The requirements of a "valid" *Stream Context* data structure are defined in section 6.2.2.1.

If the successful completion of the *Configure Endpoint Command* results in endpoints being enabled, then information in the *Input Context* is copied to the *Device Context*. As illustrated in the figure below.

Figure 4-3: Example Configure Endpoint Command



To issue a *Configure Endpoint Command* system software shall perform the following operations:

- Allocate and initialize an *Input Context* data structure for the command.
- Insert a *Configure Endpoint Command* on the Command Ring and initialize the following fields:
  - *TRB Type* = *Configure Endpoint Command* (refer to Table 6-86).
  - *Slot ID* = ID of the target Device Slot.
  - Input Context Pointer = The base address of the *Input Context* data structure.Clear all other fields of the command TRB to '0'.
  - *Cycle bit* = Command Ring's PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

Note: This example assumes the existence of two global variables: *Bandwidth Available*, and *Resource Available*, which identify the amount of the respective parameter available for allocation. And two temporary variables: *Bandwidth Required*, and *Resource Required*, which define the amount of the respective parameter required to successfully complete the *Configure Endpoint Command*. *Bandwidth* is a commodity allocated by the host controller. Refer to section 4.14.2 for the maximum bus bandwidth may be allocated to periodic endpoints. *Resource* is an xHC implementation specific parameter which may refer to internal xHC data structure or buffer space.

Note: A *Slot* or *Endpoint Context* contained in the *Input Context* is referred to as an ***Input*** Slot or Endpoint Context. And a Slot or Endpoint Context contained in the *Device Context* data structure pointed to by the *Device Context Base Address Array* is referred to as an ***Output*** Slot or Endpoint Context.

When a *Configure Endpoint Command* is executed by the xHC it shall perform the following operations:

- Insert a *Command Completion Event* on the Event Ring and initialize the following fields:
    - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
    - *Command TRB Pointer* = The address of the *Configure Endpoint Command TRB*.
    - *Slot ID* = The value of the command's *Slot ID*.
    - Initialize the *Bandwidth Required* variable to 0.
    - Initialize the *Resource Required* variable to 0.
    - If the Device Slot identified by the *Slot ID* field has been previously enabled by an *Enable Slot Command*:
        - Retrieve the Output Device Context of the selected Device Slot.

        // Release the resources and bandwidth for the endpoints to be disabled.
        - If the Output Device Context *Slot State* is equal to *Configured*:
            - If the *Deconfigure* (DC) flag = '1':
                - For each *Endpoint Context* not in the *Disabled* state:
                    - Subtract the resources allocated to the endpoint from the *Resource Required* variable.
                    - If the endpoint is periodic:
                        - Subtract bandwidth allocated to the endpoint from the *Bandwidth Required* variable.
                    - Set the Output *EP State* field to *Disabled*.
                - Set the *Slot State* in the Output Slot Context to *Addressed*.
                - *Completion Code* = *Success* (refer to Table 6-85). Note: This value may be overwritten by a later operation.
            - else // *DC* = '0'
                - For each *Endpoint Context* designated by a *Drop Context* flag = '1':
                    - Subtract the resources allocated to the endpoint from the *Resource Required* variable.
                    - If the endpoint is periodic:
                        - Subtract bandwidth allocated to the endpoint from the *Bandwidth Required* variable.
                - *Completion Code* = *Success* (refer to Table 6-85). Note: This value may be overwritten by a later operation.

        // Calculate the resource and bandwidth requirements for the endpoints to be enabled.

- If the Output Device Context *Slot State* is equal to *Addressed* or *Configured* and *DC* = ‘0’:
  - If all Input Endpoint Contexts identified by *Add Context* flag fields = ‘1’ are valid:
    - For each *Endpoint Context* designated by an *Add Context* flag = '1':
      - If the xHC resources required by the enabled endpoints are available:
        - Add the resources allocated to the endpoint to the *Resource Required* variable.
      - If the endpoint is periodic:
        - Evaluate the bandwidth requirements define by the Endpoint Context.
        - Add bandwidth allocated to the endpoint from the *Bandwidth Required* variable.
    - If the Resource Required is less than or equal to the Resource Available:
      - If the Bandwidth Required is less than or equal to the Bandwidth Available:

        // The resource and bandwidth allocations will allow a successful completion, so update Endpoint Context(s).
        - Subtract the Bandwidth Required from the Bandwidth Available.
        - For each *Endpoint Context* designated by a *Drop Context* flag = '1':
          - Set the *EP State* field to *Disabled*[14].
        - For each *Endpoint Context* designated by a *Add Context* flag = '1':
          - Copy all fields of the Input Endpoint Context to the Output Endpoint Context.

            Note that this action passes ownership of the Transfer Ring or Stream Context Array/Transfer Rings from software to the xHC. If the Output Endpoint Context had previously pointed to a Transfer Ring or a Stream Context Array, software is responsible for performing any garbage collection necessary for recovering them.
          - Set the Output *EP State* field to *Running*.

[14]Note, if both the *Add* and *Drop* flags are set for an Endpoint Context, the xHC is not expected to write out the intermediate Disabled state to the Output Device Context. The only requirement is that the Endpoint Context is correct when the Command Completion Event is generated.

            - Load the xHC Enqueue and Dequeue Pointers with the value of the *TR Dequeue Pointer* field from the *Endpoint Context*.
          - If all Endpoints are *Disabled*:
            - Set the *Slot State* in the Output Slot Context to *Addressed*.
            - Set the *Context Entries* field in the Output Slot Context to ‘1’.
          - else // An Endpoint is *Enabled*
            - Set the *Slot State* in the Output Slot Context to *Configured*.
            - Set the *Context Entries* field in the Output Slot Context to the index of the last valid Endpoint Context in its Output Device Context structure.
          - *Completion Code* = *Success* (refer to Table 6-85).
        - else[15] // The Bandwidth Required is greater than the Bandwidth Available
          - If the Bandwidth Error is encountered in the primary Bandwidth Domain:
            - *Completion Code* = *Bandwidth Error*.
          - else // The Bandwidth Error is encountered in a Secondary Bandwidth Domain, refer to section 4.16.2 for more information on Bandwidth Domains.
            - *Completion Code* = *Secondary Bandwidth Error*.
      - else // The Resource Required is greater than the Resource Available
        - *Completion Code* = *Resource Error*.
    - else // Not all Input Endpoint Contexts identified by *Add Context* flag fields = ‘1’ are valid
      - *Completion Code* =*Parameter Error*.
  - else // The Output Device Context *Slot State* is not equal to *Addressed* or *Configured.*
    - Completion Code = *Context State Error.*
- else // The slot has not been enabled by an *Enable Slot Command*
  - *Completion Code* = *Slot Not Enabled Error*.
- Clear all other fields of the event TRB to ‘0’.
- *Cycle bit* = Event Ring’s PCS flag.

[15]It is not required that the following checks for Primary and Secondary Bandwidth availability occur in this order. An xHCI implementation may check for Secondary Bandwidth availability first.

Note: Disabled endpoints have no resources or bandwidth allocated to them, so if the *Drop Context* flag is '1' for a Disabled endpoint it is ignored.

Note: The xHC shall consider an Input *Endpoint Context* invalid if the DCI of an *Add Context* flag = '1' is greater than the value of *Context Entries* field of the Input Slot Context.

Note: A *Configure Endpoint Command* may generate a *Max Exit Latency Too Large Error*, because the current *Max Exit Latency* value caused the xHC to reject the configuration, e.g. the *Max Exit Latency* value prevented a PING required by an Isoch endpoing to be scheduled. Refer to section 4.23.5.2.2 for more information on the *Max Exit Latency Too Large Error*.

Refer to sections 6.2.2

The requirements of a "valid" *Slot Context* data structure are defined in section 6.2.2.2.

The requirements of a "valid" *Endpoint Context* data structure are defined in section 6.2.3.2.

The requirements of a "valid" *Stream Context* data structure are defined in section 6.2.4.1.

4.6.6.1 Exit Latency Delta (ELD)

The **Exit Latency Delta** (ELD) provides a hint to software for optimizing power management.

The *ELD* shall be reported by a *Configure Endpoint Command* as a non-zero value in the *Command Completion Parameter* field in the *Command Completion Event*. If the *Command Completion Parameter* = '0', then ELD hinting is not available.

If the *Completion Code* of a *Configure Endpoint Command* = *Success*, then the *ELD* shall define the amount time in microseconds by which the current *Max Exit Latency* value for the slot may be successfully increased and still allow the configuration to succeed. If the *Completion Code* = *Max Exit Latency Too Large Error*, then the *ELD* shall define the amount of time in microseconds that *Max Exit Latency* must be reduced by to enable success. The *Command Completion Parameter* field shall be cleared to '0' for all other *Configure Endpoint Command* completion *Condition Code* values.

Internally an xHC adjusts its timing with an implementation specific granularity. An xHC shall report ELD = '1' if the computed *ELD* value is too small to allow a successful command completion.

4.6.7 Evaluate Context

The *Evaluate Context Command* is issued by software to inform the xHC that specific fields in the Device Context data structures have been modified. There are several cases where parameters associated with a Slot Context or the Default Control Endpoint Context are initially unknown, which shall be updated after the slot has entered the Addressed state. e.g. the *Max Packet Size* of the control endpoint may be determined only after software reads the Device Descriptor from the device through the control endpoint. The Device Descriptor shall be read to determine whether a device is a hub or not, etc.

When an *Evaluate Context Command* is processed by the xHC it shall only affect the parameters identified by the respective context. Refer to the *Evaluate Context Command Usage* sub-sections in section 4.5.2 and 6.2.2.3 for more information on the specific context fields that are affected.

The format of the *Evaluate Context Command TRB* is defined in section 6.4.3.6.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

When the command is complete, a *Command Completion Event* is posted to the *Event Ring* indicating the success or failure of the command.

If the Slot State is *Disabled* when an *Evaluate Context Command* is received, the xHC shall generate a *Slot Not Enabled Error* Event on the Event Ring.

Upon successful completion of an *Evaluate Context Command*, the xHC shall begin executing with the updated context parameters.

The *Evaluate Context Command* utilizes the *Input Context* data structure defined in section 6.2.5 to define which Contexts are to be evaluated. The state of the *Add Context* flags depends on the specific endpoints affected by the command. All *Drop Context* flags of the *Input Control* Context shall be cleared to '0' (these flags do not apply to the *Evaluate Context Command*). System software shall initialize *Contexts* of the *Input Context* affected by the command. All Contexts not referenced by an *Add Context* flag in the *Input Context* are ignored by the xHC.

To issue an *Evaluate Context Command*, system software shall perform the following operations:

- Allocate and initialize an *Input Context* data structure for the command.
- Insert an *Evaluate Context Command* on the Command Ring
    - *TRB Type = Evaluate Context Command* (refer to Table 6-86).
    - The *Add Context* flags shall be initialized to indicate the IDs of the Contexts affected by the command. Refer to sections 6.2.2.3 and 6.2.3.3 for the specific Context fields that shall be evaluated.
    - Set all *Drop Context* flags to '0'.

- *Slot ID* = ID of the target Device Slot.
- *Input Context Pointer* = The base address of the *Input Context* data structure.
- Clear all other fields of the command TRB to '0'.
- *Cycle bit* = Command Ring's PCS flag.

- Write the Host Controller Doorbell with *DB Target = Host Controller Command*.

When an *Evaluate Context Command* is executed by the xHC it shall perform the following operations:

- Insert a *Command Completion Event* on the Event Ring.
  - *TRB Type = Command Completion Event* (refer to Table 6-86).
  - *Command TRB Pointer* = The address of the *Evaluate Context Command TRB*.
  - *Slot ID* = The value of the command's *Slot ID*.
  - *Completion Code = Success* (refer to Table 6-85).
  - If the Device Slot identified by the *Slot ID* fields has been previously enabled by an *Enable Slot Command*:
    - Retrieve the Output *Device Context* of the selected Device Slot.
    - If the Output *Slot State* is equal to *Default*, *Addressed* or *Configured*:
      - For each Context designated by an *Add Context* flag = '1':
        - Evaluate the parameter settings defined by the selected Contexts.
        - If the Context parameters are not valid:
          - *Completion Code = Parameter Error*.
      - If the Max Exit Latency is non-zero:
        - Calculate the *Isoch Scheduling Delay*.
        - If the *Max Exit Latency + Isoch Scheduling Delay* does not allow an Isoch endpoint to be scheduled:
          - *Completion Code = Max Exit Latency Too Large Error.*
      - If *Completion Code = Success*:
        - For each Endpoint Context designated by a *Add Context* flag = '1':
          - Update Output Device Context parameters.
    - else // The Output *Slot State* is not equal to *Default*, *Addressed* or *Configured*
      - *Completion Code = Context State Error*.
  - else // The slot has not been enabled by an *Enable Slot Command*
    - *Completion Code = Slot Not Enabled Error*.
  - Clear all other fields of the event TRB to '0'.
  - *Cycle bit* = Event Ring's PCS flag.

Note: The xHC shall consider an *Endpoint Context* invalid if the DCI of an *Add Context* flag = '1' is greater than the value of *Context Entries*.

Note The Output *Slot/Endpoint Context* parameters shall not be changed if any error is detected by this command.

Note: When an *Evaluate Context Command* modifies the value of *Max Exit Latency*, the xHC shall not drop the data of any Isoch TDs of any endpoints associated with the Device Slot targeted by the command.

Note: Prior to issuing an *Evaluate Context Command* that modifies the value of the Slot Context *Interrupter Target* software shall ensure that all Endpoints (including the Default Control Endpoint), are in the Stopped state.

The requirements of a "valid" *Slot Context* data structure are defined in section 6.2.2.3.

The requirements of a "valid" *Endpoint Context* data structure are defined in section 6.2.3.3.

4.6.8 Reset Endpoint

The *Reset Endpoint Command* is issued by software to recover from a halted condition on an endpoint.

The format of the *Reset Endpoint Command TRB* is defined in section 6.4.3.7.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

When a Transfer Ring or Stream is halted; the associated endpoint is removed from the xHC's Pipe Schedule, the Doorbell Register for that pipe is disabled, the state of the associated Endpoint Context is set to *Halted*, and any subsequent packets received for the endpoint will be silently dropped.

The *Reset Endpoint Command* defines *Slot ID and Endpoint ID* fields. The *Slot ID and Endpoint ID* fields identify the USB device, and the endpoint of that device that is the target of the command.

The xHC shall perform the following operations when Resetting an endpoint:

- If the endpoint is not in the Halted state when an Reset *Endpoint* Command is executed:
    - The xHC shall reject the command and generate a *Command Completion Event* with the *Completion Code* set to *Context State Error*.
- else
    - If the *Transfer State Preserve* (TSP) flag is '0':
        - Reset the Data Toggle for USB2 devices or the Sequence Number for USB3 devices.
        - Reset any USB2 split transaction state associated with the endpoint.
        - Invalidate any xHC TDs that may be cached, forcing xHC to fetch Transfer

TRBs from memory when the pipe transitions from the *Stopped* to *Running* state.

- else // TSP = ‘1’
  - The USB2 Data Toggle or the USB3 Sequence Number for the pipe shall be preserved.
  - Maintain any USB2 split transaction state associated with the endpoint.
  - The endpoint shall continue execution by retrying the last transaction the next time the doorbell is rung, if no other commands have been issued to the endpoint.
- Set the Endpoint Context *EP State* field to *Stopped*.
- Enable the Doorbell Register for the pipe.
- Generate a *Command Completion Event* with the *Completion Code* set to *Success*.

After the command completes, the Transfer Ring will be reinstated on the xHC’s Pipe Schedule the next time its doorbell is rung.

Note: The *Reset Endpoint Command* maintains the state of an endpoint so that the previously executed packet may be retried, irrespective of the value of the *TSP* flag. e.g. if the endpoint halted retrying the 3rd 1K packet of a 4KB TRB, a doorbell ring immediately after a *Reset Endpoint Command* would cause the endpoint to retry the same packet and move the data to/from a 2KB offset within the buffer referenced by the TRB. Clearing the *TSP* flag to ‘0’ resets the Data Toggle/Sequence Number of the endpoint, however it has no other effect on other state associated with the endpoint,

Note: Prior to restarting the Transfer Ring, software may use the *Set TR Dequeue Pointer Command* to modify the value of the *TR Dequeue Pointer* field of the Endpoint Context and clear the endpoint state associated with the previously executed packet. If the *Reset Endpoint Command* is followed with a *Set TR Dequeue Pointer Command*, the endpoint shall start execution at the beginning of the TRB referenced by the *TR Dequeue Pointer* the next time the doorbell is rung.

Note: Software shall execute the following sequence to “reset a pipe”, i.e. clear the xHC endpoint halt condition, reset the host-side Data Toggle/Sequence Number, clear a stall on the device, and reset the device-side Data Toggle/Sequence Number. Also, if the device was behind a TT, the TT buffer would also need to be cleared.

- *Reset Endpoint Command* (TSP = ‘0’).
- If the device was behind a TT and it is a Control or Bulk endpoint:
  - Issue a ClearFeature(CLEAR_TT_BUFFER) request to the hub.
- If not a Control endpoint:
  - Issue a ClearFeature(ENDPOINT_HALT) request to device.

        - Issue a *Set TR Dequeue Pointer Command*, clear the endpoint state and reference the TRB to start.
        - Ring Doorbell to restart the pipe.

    The *Set TR Dequeue Pointer Command* resets the state of the endpoint so that the xHC starts transferring data at the beginning of the TRB referenced by the *TR Dequeue Pointer* (rather than at the location associated with the previous packet that caused the halt) when the doorbell is rung.

Note: Undefined behavior may occur if this command is executed with TSP = ‘0’ and the associated device endpoint is not successfully reset by system software. E.g. the Data Toggle may not be synchronized between the xHC and a USB2 device (refer to section 8.6 in the USB2 spec).

To issue a *Reset Endpoint Command* system software shall perform the following operations:

- Insert a *Reset Endpoint Command TRB* on the Command Ring and initialize the following fields:
    - *TRB Type* = *Reset Endpoint Command* (r refer to Table 6-86).
    - *Transfer State Preserve* (TSP) = Desired Transfer State result.
    - *Endpoint ID* = ID of the target endpoint.
    - *Slot ID* = ID of the target Device Slot.
    - Clear all other fields of the command TRB to ‘0’.
    - *Cycle bit* = Command Ring’s PCS flag.
- Write the *Host Controller Doorbell* with *DB Target* = *Host Controller Command* (*DB Stream ID* = ‘0’).

When a *Reset Endpoint Command* is executed by the xHC it shall perform the following operations:

- Insert a *Command Completion Event* on the Event Ring and initialize the following fields:
    - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
    - *Command TRB Pointer* = The address of the *Reset Endpoint Command TRB*.
    - *Slot ID* = The value of the command’s *Slot ID*.
    - If the Device Slot identified by the *Slot ID* has been enabled by an *Enable Slot Command*:
        - Retrieve the Device Context of the selected Device Slot.
        - If the *Slot State* is set to *Default*, *Addressed*, or *Configured*:
            - If the *Endpoint State* (EP State) field is set to *Halted*:
                - Set the *Endpoint State* (EP State) field to *Stopped*.
                - If the *Transfer State Preserve* (TSP) flag is cleared to ‘0’:
                    - Set the USB2 Data Toggle or the USB3 Sequence Number for the pipe to ‘0’.

            - Enable the Doorbell register for the endpoint.
            - *Completion Code* = *Success* (refer to Table 6-85).
        - else // The *Endpoint State* (EP State) field is not set to *Halted*
            - *Completion Code* = *Context State Error*.
    - else // The *Slot State* is not set to *Default*, *Addressed*, or *Configured*
        - *Completion Code* = *Context State Error.*
- else // The slot has not been enabled by an *Enable Slot Command*
    - *Completion Code* = *Slot Not Enabled Error*

- Clear all other fields of the event TRB to '0'.
- *Cycle bit* = Event Ring's PCS flag.

Note: The xHC resources and bandwidth associated with a reset endpoint are not released by the *Reset Endpoint Command*.

Note: After the successful completion of a *Reset Endpoint Command* with TSP = '0', system software may issue a CLEAR_FEATURE(ENDPOINT_HALT) request to the USB device to reset the halt condition on the endpoint of the device.

Note: Software shall be responsible for timing the Reset "recovery interval" required by USB.

Note: After a *Reset Endpoint Command* is executed for a control endpoint, software shall execute a *Set TR Dequeue Pointer Command* to ensure that the endpoint's Dequeue Pointer references a *Setup TD*.

Note: Software is responsible for cleaning up any partially completed transfers after issuing a *Reset Endpoint Command*, e.g. after this command completes, software shall update the associated Transfer Ring to ensure that any endpoint specific requirements are met (e.g. as identified in the previous note), before ringing the endpoint's doorbell.

Note: The *Reset Endpoint Command* may only be issued to endpoints in the *Halted* state. If software wishes to reset the Data Toggle or Sequence Number of an endpoint that isn't in the *Halted* state, then software may issue a *Configure Endpoint Command* with the *Drop* and *Add* bits set for the target endpoint that is in the *Stopped* state or *Running* but *Idle* state.

4.6.8.1 Soft Retry

A *Soft Retry* may effectively be used to recover from a *USB Transaction Error* that was due to a temporary error condition (e.g. electrical interference caused by a cell phone transmitting too close to a USB cable). Often the delay introduced between software detecting the error and attempting a Soft Retry is enough to let the temporary condition clear and allow a successful transfer.

Section 4.10.2.3 describes how the xHC shall halt an endpoint with a *USB Transaction Error* after *CErr* retries have been performed. The USB device is not aware that the xHC has halted the endpoint, and will be waiting for another retry, so a *Soft Retry* may be used to perform additional retries and recover from an error which has caused the xHC to halt an endpoint.

Software performs a *Soft Retry* with the following operations:

1. Issue a *Reset Endpoint Command* with the *TSP* flag set to ‘1’. This causes the endpoint to advance from the *Halted* to the *Stopped* state, but does not change the state of the Data Toggle or Sequence Number, and allows the xHC to continue the retry process another *CErr* times.

2. Ring the doorbell for the endpoint to initiate up to another *CErr* retries.

To support *Soft Retry*, the state of a partially completed TRB transfer (e.g. if 1K of a 4K TRB has been moved) shall be maintained by a *Reset Endpoint Command* if *TSP* = ‘1’.

Note: *Soft Retry* attempts shall not be performed on Isoch endpoints. Any attempt to do so may result in undefined behavior.

Note: *Soft Retry* attempts shall not be performed if the device is behind a TT in a HS Hub (i.e. *TT Hub Slot ID* > ‘0’). Any attempt to do so may result in undefined behavior.

Note: Recovery of lost data on an Interrupt endpoint may be handled by class specific mechanism.

Note: Software shall limit the number of unsuccessful Soft Retry attempts to prevent an infinite loop.

4.6.9 Stop Endpoint

The *Stop Endpoint Command* is issued by software to stop the xHC execution of the TDs on an endpoint. An endpoint may be stopped by software so that it can temporarily take ownership of Transfer Ring TDs that had previously been passed to the xHC, or to stop USB activity prior to powering down the xHC. While the endpoint is stopped, software may add, delete, or otherwise rearrange TDs on an associated Transfer Ring. e.g. this command allows software to insert “high-priority” TDs at the Dequeue Pointer so they will be executed immediately when the ring is restarted, or to “abort” one or more TDs by removing them from the ring.

The *Stop Endpoint Command* is expected to stop endpoint activity as soon as possible, which may mean that it stops in the middle of a TRB. When the endpoint stops, it saves the value of the *TR Dequeue Pointer* and DCS fields (and possibly other “Opaque” state) in the Endpoint/Stream Context so that it can pick up where it left off the next time its doorbell is rung, e.g. if the endpoint stopped after moving the first 1KB of data in a 4KB TRB, then transfer related state maintained by the xHC will allow it to transfer the remaining 3KB of data when the doorbell is rung. If a *Set TR Dequeue Pointer Command* is issued while an endpoint is in the Stopped state, the transfer related state of the endpoint will be dumped when the Output Endpoint/Stream Context *TR Dequeue Pointer* and *DCS* fields are overwritten. The next time the doorbell is rung, the endpoint shall start execution at the beginning of the TRB referenced by the *TR Dequeue Pointer*.

Note: If the *TR Dequeue Pointer* references an *Event Data TRB* when a TD is stopped, the xHC shall execute it before generating the *Command Completion Event*, by generating an *Event Data Transfer Event* if the *IOC* flag was set and advancing to the next TRB.

Before generating a *Command Completion Event* for this command, the xHC shall write the final value of the endpoint's Dequeue Pointer to the *TR Dequeue Pointer* field and CCS flag to the *DCS* field of the Output Endpoint Context or Stream Context associated with the stopped Transfer Ring. And if *Stopped EDTLA Capability* (SEC) = '1', then the xHC shall write the value of the EDTLA to the *Stopped EDTLA* field of the Stream Context associated with the stopped Transfer Ring. The xHC shall also ensure that the Stream Context *TR Dequeue Pointer*, *DCS*, and *Stopped EDTLA* fields reflect the forward progress of any Stream that entered the Move Data state while the endpoint was in the Running state. Refer to section 4.12 for more information on Stream endpoint Stopped state transitions.

Note: *Stopped EDTLA Capability* support (i.e. *SEC* = '1') shall be mandatory for all xHCI 1.1 compliant xHCs.

The format of the *Stop Endpoint Command TRB* is defined in section 6.4.3.8.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

The *Stopped - Short Packet Capability* (SPC) flag in the HCCPARAMS1 register (5.3.6) indicates whether the xHC is capable of generating a *Stopped - Short Packet* Completion Code. Any discussion of the *Stopped - Short Packet* Completion code assumes that the *Stopped - Short Packet Capability* is supported (*SPC* ='1').

Note: The *Stopped - Short Packet Capability* (i.e. *SPC* = '1') shall be mandatory for all xHCI 1.1 compliant xHCs.

Depending on the timing of the execution of the *Stop Endpoint* command relative to the execution of the TDs on the ring, one of three of scenarios may result:

- If the command is executed between TDs, then the xHC shall perform a *Force Stopped Event* (FSE) operation by generating a Transfer Event for the endpoint with *Condition Code* = *Stopped - Invalid Length*, *TRB Pointer* = current Dequeue Pointer value, and *TRB Transfer Length* = 0, then generate a *Success Command Completion Event* for the command.

- If *SPC* = '1' and the command is executed, after a Short Packet condition has been detected, but before the end of the TD has been reached, (i.e. the TD is in progress for the pipe), then a *Transfer Event TRB* with its *Completion Code* set to *Stopped - Short Packet* and its *TRB Transfer Length* set to the value of the *EDTLA* shall be forced for the interrupted TRB, irrespective of whether its *IOC* or *ISP* flags are set. This *Transfer Event TRB* will precede the *Command Completion Event TRB* for the command, and is referred to as a **Stopped Transfer Event**.
- If a TD is in progress for the pipe when the command is executed, and a Short Packet condition has not been detected or *SPC* = '0' and a Short Packet condition has been detected, then a *Transfer Event TRB* with its *Completion Code* set to *Stopped*, *TRB Pointer* = current Dequeue Pointer value, and *TRB Transfer Length* set to the residual bytes to transfer, shall be forced for the interrupted TRB, irrespective of whether its *IOC* or *ISP* flags are set. This *Transfer Event TRB* will precede the *Command Completion Event TRB* for the command, and is also referred to as a **Stopped Transfer Event**.

While an endpoint is stopped, any USB packets received for it shall be silently dropped by the xHC.

Note that when an endpoint is stopped, the xHC maintains the state necessary to restart the last active Transfer Ring where it left off, however software may not want to do this. The options are discussed below:

1. Temporarily Stop Transfer Ring Activity - If the intent of software in issuing the *Stop Endpoint Command* was just to temporarily stop activity on the Transfer Ring, then software may restart the stopped ring where it left off by simply ringing its doorbell.

2. Aborting a Transfer - If, because of a timeout or other reason, software issued the *Stop Endpoint Command* to abort the current TD. Then the *Set TR Dequeue Pointer Command* may be used to force the xHC to dump any internal state that it has for the ring and restart activity at the new *Transfer Ring* location specified by the *Set TR Dequeue Pointer Command*.

3. Modifying the order of execution of TDs on a Transfer Ring - It may be necessary for software to place a "high priority" TD on a ring, by inserting a TD ahead of any pending TDs. To safely modify the order of execution of TDs on a ring, software shall first stop the endpoint. When an endpoint is stopped, software may examine the Event Ring to determine the current state of TDs on the associated Transfer Ring(s). If the xHC stopped in the middle of a TD, then that TD may not be modified by software, however any other TDs on the ring may be. If the xHC stopped between TDs, then it may modify any TD on the transfer ring. After the TDs are inserted, removed, or rearranged to the satisfaction of software, it may ring the doorbell to restart operation on the ring.

Note: The xHC cannot distinguish whether software temporarily stopped Transfer Ring activity or stopping the Transfer Ring to modifying the order of execution of its TDs. In either case, if the xHC has read-ahead and cached TRBs for the Transfer Ring, it shall invalidate all TRBs not associated with the current TD before continuing execution of the Transfer Ring. This ensures that any TDs modified by software shall be correctly executed by the xHC.

Note: If software is issuing the *Stop Endpoint Command* due to suspending a device or a function on a device, it shall set the *Suspend* (SP) flag to '1' in the *Stop Endpoint Command TRB* (refer to *SP* definition in Table 6-66).

The xHC shall perform the following operations when Stopping an endpoint:

- The xHC shall stop the USB activity for the pipe.
  - If a USB IN or OUT transaction is in-flight, it shall be completed.
  - ERDYs shall be ignored on the pipe for that endpoint.
  - If LS, FS, or HS, then polling of the pipe shall cease.
  - If an SS pipe is waiting for an ERDY, the xHC shall clear the flow control condition and cease waiting for the ERDY.

Note: A *Set TR Dequeue Pointer Command* clears any transfer related state associated with an endpoint. If an SS pipe was waiting for an ERDY when the endpoint was stopped, then if the endpoint transfer state was not cleared by a *Set TR Dequeue Pointer Command,* the xHC shall reissue an IN or OUT for the pipe when the ring is restarted.

  - The current endpoint Service Opportunity (SO) shall be terminated.
- Stop the Transfer Ring activity for the pipe. Refer to Table 4-2 for Stop conditions and Actions.
- Remove the endpoint from the xHC's Pipe Schedule.
- Generate a *Command Completion Event*.

After the command completes, the endpoint shall be reinstated on the xHC's Pipe Schedule the next time its doorbell is rung.

Note: Prior to restarting the ring, software may use the *Set TR Dequeue Pointer Command* to modify the value of the *TR Dequeue Pointer* field of the Endpoint or Stream Context. The *Set TR Dequeue Pointer Command* shall invalidate any xHC TDs that may be cached, forcing xHC to fetch Transfer TRBs from memory when the pipe is restarted.

Note: If software wants to know the exact number of bytes transferred when a TD is stopped:

If the *ED* flag is '0' and the Completion Code equals *Stopped*, software may subtract the value of the *TRB Transfer Length* field reported by the Transfer Event from the sum of the *TRB Transfer Length* fields of all Transfer TRBs in the TD executed prior to and including the TRB referenced by the Transfer Event.

If the *ED* flag is '0' and the Completion Code equals *Stopped - Short Packet*, software shall use the *TRB Transfer Length* field of the Transfer Event.

If the *ED* flag is '0' and the Completion Code equals *Stopped - Length Invalid*, software shall ignore the *TRB Transfer Length* field of the Transfer Event, and simply sum of the *TRB Transfer Length* fields of all Transfer TRBs in the TD executed prior to the TRB referenced by the Transfer Event.

If the *ED* flag is '1' then the *TRB Transfer Length* field reflects the number of bytes transferred prior to stopping.

Note: If the *ED* flag is '0' in the *Stopped Transfer Event* software may emulate an *Event Data Transfer Event* for the stopped Transfer Ring. It does this by starting at the TRB referenced by the *Stopped Transfer Event* and advancing through the TD, searching for the next *Event Data TRB*. If one is found, the *Parameter Component* of the *Event Data TRB* and the "number of bytes transferred" as described in the previous Note may be used to emulate an *Event Data Transfer Event*.

Note: After the command is complete, the *TR Dequeue Pointer* field of all Endpoint/Stream Contexts associated with an endpoint shall contain the current value of the Dequeue Pointer for the respective ring.

The xHC shall generate a *Stopped Transfer Event* every time a Transfer Ring is stopped with a *Stop Endpoint Command*. This operation is referred to as *Force Stopped Event* (FSE). The forced *Stopped Transfer Event* explicitly indicates to software that the selected Transfer Ring has stopped. If a Transfer Ring is empty when a *Stop Endpoint Command* is issued, a *Stopped Transfer Event* shall be generated on the Event Ring indicated by the Slot Context *Interrupter Target* field.

The Table 4-2 identifies the Action that shall be taken by the xHC on the TRB referenced by the Dequeue Pointer when the transfer ring stops. When restarting a Stopped endpoint, Table 4-2 also identifies whether the xHC shall advance the Dequeue Pointer prior to executing a TRB, or if it shall continue the execution at the Stopped TRB.

Note: The cases in Table 4-2 that reference a "FSE" Action shall force an additional *Stopped Transfer Event*.

Note: A Busy endpoint may asynchronously transition from the *Running* to the *Halted* or *Error* state due to error conditions detected while processing TRBs. A possible race condition may occur if software, thinking an endpoint is in the *Running* state, issues a *Stop Endpoint Command* however at the same time the xHC asynchronously transitions the endpoint to the *Halted* or *Error* state. In this case, a *Context State Error* may be generated for the command completion. Software may verify that this case occurred by inspecting the *EP State* for *Halted* or *Error* when a *Stop Endpoint Command* results in a *Context State Error*.

Table 4-2: Stop Endpoint Command TRB Handling

TRB Type referenced by TR Dequeue Pointer	*Chain* bit (CH)	Condition	Action	Advance TR Dequeue Pointer on Doorbell Ring
Transfer TRB[16] (Completed) Residual Length = 0	1	Stopped on TRB boundary within a TD[17].	Generate Transfer Event. An ISSE[18] else Length = 0, CC = Stopped.	Yes
	0	Stopped on TD boundary.[19]	Generate event if IOC flag set. FSE[20].	Yes
Transfer TRB (Incomplete) Residual Length > 0	X	Stopped within a TRB	Generate Transfer Event. An ISSE[18] else Length = Residual bytes to transfer, CC = Stopped.	No
Event Data	1	Stopped on intermediate Event Data TRB	Generate Transfer Event. An ISSE[18] else ED = 1, Length = EDTLA, CC = Stopped.	Yes

[16]A "Transfer" TRB is a Normal, Setup Stage, Data Stage, Status Stage, or Isoch TRB. Note, this row identifies the case where the endpoint has stopped on a TRB (that is not the last TRB of a TD), where all the data associated with the TRB has already been transferred.

[17]This condition is interpreted identically to a "Transfer TRB (Incomplete)", where 0 bytes have been transferred.

[18] ISSE - If *SPC* = '1', and a Short Packet condition has been detected, and the end of the TD has not been reached, then the xHC shall perform a *Intermediate Short Stopped Event* (ISSE) operation, generating a Transfer Event for the endpoint with *Condition Code* = *Stopped - Short Packet*, *TRB Pointer* = current Dequeue Pointer value, and *TRB Transfer Length* = EDTLA.

[19]In this case the xHC is expected to complete the TD normally (e.g. generate a Transfer Event with CC = Success if the IOC flag is set and the transfer was successful) and then perform a *Force Stopped Event* (FSE) operation.

[20]FSE - The xHC shall perform a *Force Stopped Event* (FSE) operation by generating a Transfer Event for the endpoint with *Condition Code* = *Stopped - Invalid Length*, *TRB Pointer* = current Dequeue Pointer value, and *TRB Transfer Length* = 0.

	0	Stopped on terminating Event Data TRB[21]	Generate Transfer Event. ED = 1. Length = EDTLA. CC = previous Transfer TRB CC. FSE[20]	Yes
Link	1	Stopped on Link TRB within a TD	Generate Transfer Event. An ISSE[18] else Length = 0. CC = Stopped, Length Invalid.	Yes[22]
	0	Stopped on Link TD	Generate event if IOC flag set. FSE[20].	Yes[22]
No Op	X	Stopped on Terminating No Op TRB	Generate Transfer Event if IOC flag set. FSE[20].	Yes
Vendor Defined	X	Stopped on Vendor Defined TRB	Vendor defined. FSE[20].	Vendor Defined
Invalid TRB (C != DCS)	Prev TRB[23] *CH* = 1	Stopped while waiting for more TRBs to be posted for TD	Generate Transfer Event.[24] Length = 0. CC = Stopped - Length Invalid.	No
	Prev[23] TRB *CH* = 0	Stopped on TD boundary	FSE[20].	No

Note: If a Transfer Ring has been Halted due to error condition when a Stop Endpoint Command is received, no *Stopped Transfer Event* shall be generated.

[21]Force normal completion of Event Data TRB before generating Command Completion Event.

[22]When the Dequeue Pointer is advanced, the xHC shall begin parsing TRBs at the address identified by the Link TRB Ring Segment Pointer field.

[23]In this case the TRB referenced by the TR Dequeue Pointer is invalid, so use the state of the Chain (CH) bit from the last executed TRB. If no TRBs had been executed previously, assume C = '0' case.

[24]The event generated by the "Stopped while waiting for more TRBs to be posted for TD." condition uses the Slot Context Interrupter Target field to identify the target Event Ring.

To issue a *Stop Endpoint Command* system software shall perform the following operations:

- Insert a *Stop Endpoint Command* on the Command Ring and initialize the following fields:
  - *TRB Type* = *Stop Endpoint Command* (refer to Table 6-86).
  - *Endpoint ID* = ID of the target endpoint.
  - *Slot ID* = ID of the target Device Slot.
  - Clear all other fields of the command TRB to '0'.
  - *Cycle bit* = Command Ring's PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

When a *Stop Endpoint Command* is executed by the xHC it shall perform the following operations:

- If the *Stop Endpoint Command* interrupted the execution of a TD, then insert a *Transfer Event* on the Event Ring and initialize the following fields:
  - *TRB Type* = *Transfer Event*.
  - *Slot ID* = The value of the command's *Slot ID*.
  - *Endpoint ID* = The value of the command's *Endpoint ID*.
  - If the TRB referenced by the *TR Dequeue Pointer* is an *Event Data TRB*:
    - *ED* = '1'.
    - *Parameter Component (TRB Pointer)* = 64 bits of *Event Data TRB* Parameter component.
    - *Length* = The value of the *Event Data Transfer Length Accumulator* (EDTLA). Refer to section 4.11.5.2 for a description of EDTLA.
  - *else* // The the TRB referenced by the *TR Dequeue Pointer* is not an *Event Data TRB*
    - *ED* = '0'.
    - *TRB Pointer* = The address of the TRB interrupted by the command.
    - *Length* = The number of bytes remaining to be moved for the interrupted TRB.
  - *Completion Code* = *Stopped* (refer to Table 6-85).
  - Clear all other fields of the event TRB to '0'.
  - *Cycle bit* = Event Ring's PCS flag.
- Insert a *Command Completion Event* on the Event Ring and initialize the following fields:
  - *TRB Type* = *Command Completion Event*.
  - *Command TRB Pointer* = The address of the *Stop Endpoint Command TRB*.
  - *Slot ID* = The value of the command's *Slot ID*.

- If the Device Slot identified by the *Slot ID* has been previously enabled by an *Enable Slot Command*:
  - Retrieve the Device Context of the selected Device Slot.
  - If the *Slot State* is set to *Default*, *Configured*, or *Addressed*:
    - If the *Endpoint State* (EP State) field equals *Running*:
      - Stop the USB activity for the pipe as described above.
      - Stop the Transfer Ring activity for the pipe as described above.
      - Write Dequeue Pointer value to the Output Endpoint or Stream Context *TR Dequeue Pointer* field.
      - Write CCS value to the Output Endpoint or Stream Context *Dequeue Cycle State* (DCS) field.
      - Removed the endpoint from the xHC's Pipe Schedule.
      - Set the *Endpoint State* (EP State) field to *Stopped*.
      - *Completion Code* = *Success*.
    - else // The *Endpoint State* (EP State) field is not *Running*
      - *Completion Code* = *Context State Error*.
  - else // The *Slot State* is not set to *Default*, *Configured*, or *Addressed*
    - *Completion Code* = *Context State Error.*
- else // The slot has not been enabled by an *Enable Slot Command*
  - *Completion Code* = *Slot Not Enabled Error*
- Clear all other fields of the event TRB to '0'.
- *Cycle bit* = Event Ring's PCS flag.

Note: The xHC resources and bandwidth associated with an endpoint are not released by the *Stop Endpoint Command*.

Note: The xHC shall wait for any partially completed USB2 split transactions to finish before completing the *Stop Endpoint Command*.

4.6.10 Set TR Dequeue Pointer

The *Set TR Dequeue Pointer Command* is issued by software to modify the *TR Dequeue Pointer* field of an Endpoint or Stream Context.

The *Slot ID and Endpoint ID* fields of the *Set TR Dequeue Pointer Command TRB* identify the USB device, and the endpoint of that device, that is the target of the command. If Streams are enabled for the endpoint, the *Set TR Dequeue Pointer Command* TRB *Stream ID* field identifies the Stream Context that shall be modified.

This command may be executed only if the target endpoint is in the *Error* or *Stopped* state.

The format of the *Set TR Dequeue Pointer Command TRB* is defined in section 6.4.3.9.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

The xHC shall perform the following operations when setting a ring address:

- If the endpoint is not in the *Error* or *Stopped* state when the *Set TR Dequeue Pointer Command* is executed:
    - The xHC shall reject the command and generate a *Command Completion Event* with the *Completion Code* set to *Context State Error*.
- else // The endpoint is in the *Error* or *Stopped* state
    - Set the Dequeue Pointer to the value of the *New TR Dequeue Pointer* field in the *Set TR Dequeue Pointer TRB*.
    - Invalidate any xHC TDs that may be cached, forcing xHC to fetch Transfer TRBs from memory when the pipe transitions from the *Stopped* to the *Running* state.
    - Copy the value of the *New TR Dequeue Pointer* field in the *Set TR Dequeue Pointer TRB* to the *TR Dequeue Pointer* field of the target Endpoint or Stream Context.
    - Clear any prior transfer state, e.g. setting the EDTLA to 0, clearing any partially completed USB2 split transactions, etc.
    - Generate a *Command Completion Event* with the *Completion Code* set to *Success*.

Note: If, when the Transfer Ring was stopped a TD was only partially executed, then any remaining TRBs in that TD shall not be executed when the endpoints' *TR Dequeue Pointer* is updated by the *Set TR Dequeue Pointer Command*.

Note: A *Set TR Dequeue Pointer Command* may be issued to modify the *TR Dequeue Pointer* field of a non-active Stream Context while a Stream endpoint is in the *Running* state. Refer to section 4.12 for active vs. non-active Stream Context information.

To issue a *Set TR Dequeue Pointer Command* system software shall perform the following operations:

- Insert a *Set TR Dequeue Pointer Command* on the Command Ring and initialize the following fields:
    - *TRB Type* = *Set TR Dequeue Pointer Command* (refer to Table 6-86).
    - *Endpoint ID* = ID of the target endpoint.
    - *Stream ID* = ID of the target Stream Context or '0' if *MaxPStreams* = '0'.
    - *Slot ID* = ID of the target Device Slot.
    - *New TR Dequeue Pointer* = The new *TR Dequeue Pointer* field value for the target endpoint.
    - *Dequeue Cycle State* (DCS) = The state of the xHCI CCS flag for the TRB pointed to by the *TR Dequeue Pointer* field.

  - Clear all other fields of the command TRB to ‘0’.
  - *Cycle bit* = Command Ring’s PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

When a *Set TR Dequeue Pointer Command* is executed by the xHC it shall perform the following operations:

- Insert a *Command Completion Event* on the Event Ring and initialize the following fields:
  - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
  - *Command TRB Pointer* = The address of the *Set TR Dequeue Pointer Command TRB*.
  - *Slot ID* = The value of the command’s *Slot ID*.
  - If the Device Slot identified by the *Slot ID* has been enabled by an *Enable Slot Command*:
    - Retrieve the Device Context of the selected Device Slot.
    - If the *Slot State* is set to *Default*, *Configured*, or *Addressed*:
      - If the *Endpoint State* (EP State) field equals *Stopped or Error*:
        - If the *Stream ID* field is non-zero a Stream Context is referenced so perform a Stream ID boundary check as described in section 4.12.2.1:
          - If the Stream ID is valid:
            - Copy the value of the *New TR Dequeue Pointer* field to the *TR Dequeue Pointer* field of the target Stream Context.
            - Copy the value of the *Dequeue Cycle State* (DCS) field to the *Dequeue Cycle State* (DCS) field of the target Stream Context.
            - *Completion Code* = *Success* (refer to Table 6-85).
          - else // The Stream ID is invalid
            - *Completion Code* = *TRB Error.*
        - else (Stream ID = ‘0’)
          - If MaxPStreams = ‘0’:
            - Copy the value of the *New TR Dequeue Pointer* field to the *TR Dequeue Pointer* field of the target Endpoint Context.
            - Copy the value of the *Dequeue Cycle State* (DCS) field to the *Dequeue Cycle State* (DCS) field of the target Endpoint Context.
            - *Completion Code* = *Success*.
          - else // MaxPStreams > ‘0’
            - *Completion Code* = *TRB Error.*
      - else // The *Endpoint State* (EP State) field is not *Stopped or Error*
        - *Completion Code* = *Context State Error*.

        - else // The *Slot State* is not set to *Default*, *Configured*, or *Addressed*
            - *Completion Code = Context State Error.*
    - else // The slot has not been enabled by an *Enable Slot Command*
        - *Completion Code = Slot Not Enabled Error*
    - Clear all other fields of the event TRB to '0'.
    - *Cycle bit* = Event Ring's PCS flag.

Note: Consider the case where there are multiple TDs posted for pipe for a single data transfer and an error condition on one TD means that the data transfer is terminated, and that the subsequent TDs associated with the data transfer are now invalid. The xHC may have read ahead on the Transfer Ring and cached the subsequent TDs. To ensure that xHC frees any cached information associated with a pipe in a timely manner (so that it can reuse the cache space for other pipes), software shall issue a *Set TR Dequeue Pointer Command* for the pipe when the data transfer is terminated, vs. waiting for the next data transfer to be ready before issuing the command.

Note: If software issues a *Set TR Dequeue Pointer Command* that points to a TRB that had previously been partially completed TD, the xHC shall treat that TRB as the first TRB of the TD. i.e. any prior state associated with a partially completed TRB is lost.

Note: The xHC does *not* maintain knowledge of which Streams are active or non-active. If software issues a *Set TR Dequeue Pointer Command* that targets an active Stream of an endpoint, undefined behavior may occur. Refer to section 4.12 for active vs. non-active Stream Context information)

4.6.11 Reset Device

The *Reset Device Command* is used by software to inform the xHC that the USB Device associated with a Device Slot has been Reset (by either; setting the Root Hub port *PR* flag if the device is attached to a Root Hub port, or issuing a SetPortFeature(PORT_RESET) request the external hub port upstream of the device). In the Slot Context of the selected device slot, the reset operation sets the *Slot State* field to the *Default* state and the *USB Device Address* field to '0'. The reset operation also disables all endpoints of the slot except for the Default Control Endpoint by setting the Endpoint Context *EP State* field to *Disabled* in all enabled Endpoint Contexts. Software should stop all endpoint activity before issuing a Reset Device Command.

For all endpoints except the Default Control Endpoint the xHC shall:

- Terminate any USB activity (e.g. packet transfers).
- Disable the endpoints' Doorbell.
- Drop any pending events not already posted to an Event Ring.
- Free any bandwidth allocated to the periodic endpoints.

- Free any internal resources associated with the endpoint.

For the Default Control Endpoint the xHC shall terminate any USB activity, abort any pending events not already posted to an Event Ring, and transition the endpoint to the *Running* state. Undefined behavior may occur if this command is executed and the device associated with it is not successfully reset. E.g. if the USB device is not in the Default state, then a subsequent *Address Device Command* shall fail.

The format of the *Reset Device Command TRB* is defined in section 6.4.3.10.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

To issue a *Reset Device Command* system software shall perform the following operations:

- Insert an *Reset Device Command* on the Command Ring and initialize the following fields:
  - *TRB Type* = *Reset Device Command* (refer to Table 6-86).
  - *Slot ID* = The ID of the Device Slot to reset.
  - Clear all other fields of the command TRB to '0'.
  - *Cycle bit* = Command Ring's PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

When a *Reset Device Command* is executed by the xHC it shall perform the following operations:

- If the Device Slot is in the *Addressed* or *Configured* state:
  - Abort any USB transactions to the Device.
  - Set the *Slot State* field of *Slot Context* to the *Default* state.
  - Set the *Context Entries* field of *Slot Context* to '1'.
  - Set the *USB Device Address* field of *Slot Context* to '0'.
  - For each *Endpoint Context* of the *Device Context* (except the Default Control Endpoint):
    - Set the Endpoint Context *EP State* field to *Disabled.*
- Insert a *Command Completion Event* on the Event Ring of Interrupter 0 and initialize the following fields:
  - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
  - *Command TRB Pointer* = The address of the *Reset Device Command TRB*.
  - If the Device Slot was in the *Addressed* or *Configured* state:
    - *Completion Code* = *Success* (refer to Table 6-85).
  - else // The Device Slot was not in the *Addressed* or *Configured* state
    - *Completion Code* = *Context State Error*.
  - Clear all other fields of the event TRB to '0'.

- *Cycle bit* = Event Ring's PCS flag.

Note: Software is responsible for recovering any memory data structures (Stream Context Arrays, Transfer Rings, etc.) owned by disabled Endpoint Contexts the slot when the *Reset Device Command* is issued.

The *Reset Device Command* forces a Device Slot to the *Default* state, however the *Reset Device Command TRB* (section 6.4.3.10) does not reference an Input Context, so there is no Input Context available to use to set the values of the Output Device Context. After the completion of a *Reset Device Command* the Slot and Endpoint 0 Contexts shall contain values that allow the xHC to issue requests to the Default Control Endpoint of the USB device that has just been reset. Refer to sections 6.2.2.4 and 6.2.3.7 for the respective Slot Context and Endpoint Context field value settings.

4.6.12 Force Event (Optional Normative)

The *Force Event Command* is used by a VMM to insert an Event TRB in an Event Ring of a target VM when the VMM is emulating an xHC device to a VM.

When a *Force Event Command* is processed by the xHC it shall insert an Event TRB on the target VFs' Event Ring and copy the data pointed to by the *Force Event Command*, with the exception of the *Cycle* bit, to the target Event TRB. The xHC shall set the *Cycle* bit to be consistent with the target VFs' Event Ring.

A *Command Completion Event* with a *TRB Error* will be generated if the *VF ID* of the *Force Event Command* is not valid. A *VF Event Ring Full Error* shall be generated if the Target VF's Event Ring is full.

Refer to section 8 for detailed information on the use of the *Force Event Command* in a virtualized environment. And refer to section 3.3.11 for a high level description of the *Force Event Command* and it's usage.

The format of the *Force Event Command TRB* is defined in section 6.4.3.11.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

To issue a *Force Event Command* system software shall perform the following operations:

- Allocate and initialize the *VF Event TRB* that will be sent to the target VF's Event Ring. The details of the *VF Event TRB* initialization will depend on the type of Event that is being forced.
- Insert a *Force Event Command* on the Command Ring of the PF0 and initialize the following fields:
  - *TRB Type* = *Force Event Command* (refer to Table 6-86).
  - *VF ID* = ID of the target VF.

- *VF Interrupter Target* = The ID of the target Interrupter assigned to the VF. Refer to Table 6-72 for more information on this value.
  - *Event TRB Pointer* = The address of the *VF Event TRB*.
  - Clear all other fields of the command TRB to '0'.
  - *Cycle bit* = Command Ring's PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

When a *Force Event Command* is executed by the xHC it shall perform the following operations:

- Insert a *Command Completion Event* on the Event Ring and initialize the following fields:
  - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
  - *Command TRB Pointer* = The address of the *Force Event Command TRB*.
  - If the *VF ID* is valid:
    - If the *VF Interrupter Target* is in range for the VF:
      - If the target VF's Event Ring is not full:
        - Insert the VF's Event TRB referenced by the *Force Event Command Event TRB Pointer* into target VF's Event Ring specified by the *VF ID* and the *VF Interrupter Target* fields:
          - Copy all fields of the *VF Event TRB* except the *Cycle bit* field to the target VF's Event Ring.
          - *Cycle bit* = Target VF's Event Ring's PCS flag.
        - *Completion Code* = *Success* (refer to Table 6-85).
      - else // The target VF's Event Ring is full
        - *Completion Code* = *VF Event Ring Full Error*.
    - else // The *VF Interrupter Target* is not in range for the VF
      - *Completion Code* = *TRB Error*.
  - else // The *VF ID* is not valid
    - *Completion Code* = *TRB Error*.
  - Clear all other fields of the event TRB to '0'.
  - *Cycle bit* = Event Ring's PCS flag.

Note: When the command completes, the VMM may release the buffer containing the Event TRB pointed to by the *Force Event Command*.

Note: The "forced" event shall be dropped if the target Event Ring is full. Software should reschedule a *Force Event Command* if an *VF Event Ring Full Error* is returned.

4.6.13 Negotiate Bandwidth (Optional Normative)

The *Negotiate Bandwidth Command* is used by system software to initiate *Bandwidth Request Events* for periodic endpoints. This command should be used recover unused USB bandwidth from the system.

If the *BW Negotiation Capability* (BNC) bit in the HCCPARAMS1 register is ‘1’, the xHC shall support this command.

This command shall complete with a *Success Completion Code* if the command is supported, or a *TRB Error* Completion Code if the command is not supported.

The xHC shall generate *Bandwidth Request Events* upon the reception of the command to all target periodic endpoints. The command will complete when all *Bandwidth Request Events* have been generated.

The format of the *Negotiate Bandwidth Command TRB* is defined in section 6.4.3.12.

The format of the *Bandwidth Request Event TRB* is defined in section 6.4.2.4.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

To issue a Negotiate Bandwidth Command system software shall perform the following operations:

- Insert an *Negotiate Bandwidth Command* on the Command Ring and initialize the following fields:
    - *TRB Type* = *Negotiate Bandwidth Command* (refer to Table 6-86).
    - *Slot ID* = The ID of the slot that requires the bandwidth negotiation.
    - Clear all other fields of the command TRB to ‘0’.
    - *Cycle bit* = Command Ring’s PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

When a *Negotiate Bandwidth Command* is executed by the xHC it shall perform the following operations:

- If the command is supported:
    - If the Device Slot identified by the *Slot ID* has been enabled by an *Enable Slot Command*:
        - If the *Slot ID* identifies a slot in the *Addressed* or *Configured* state then:
            - If there are devices that define candidate periodic endpoints for receiving *Bandwidth Request Events*:
                - For each device, identify the target Event Ring (specified by the *Interrupt Target* field of the device’s *Slot Context*).
                    - If there is space on the device’s target Event Ring:

```
                    •  Insert a Bandwidth Request Event and initialize the following
                       fields:
                    •  TRB Type = Bandwidth Request Event (refer to Table 6-86).
                    •  Slot ID = ID of the device slot.
                    •  Completion Code = Success (refer to Table 6-85).
                    •  Clear all other fields of the event TRB to ‘0’.
                    •  Cycle bit = Device’s target Event Ring’s PCS flag.
                •  else // No space on the device’s target Event Ring
                    •  Skip the device.
            •  Completion Code = Success (refer to Table 6-85).
        •  else // The Slot ID identifies slot not in the Addressed or Configured state
            •  Completion Code =Context State Error.
    •  else // The slot has not been enabled by an Enable Slot Command
        •  Completion Code = Slot Not Enabled Error
•  else // The Negotiate Bandwidth Command is not supported
    •  Completion Code =TRB Error.
•  Insert a Command Completion Event on the Event Ring of Interrupter 0 and initialize
   the following fields:
    •  TRB Type = Command Completion Event (refer to Table 6-86).
    •  Command TRB Pointer = The address of the Negotiate Bandwidth Command
       TRB.
    •  Clear all other fields of the event TRB to ‘0’.
    •  Cycle bit = Event Ring’s PCS flag.
```

Note: System software may never issue a *Negotiate Bandwidth Command*, however if the *BNC* flag is ‘1’ an unsolicited *Bandwidth Request Event* may be generated by hardware, e.g. if the system software is running in a Virtual Machine and communicating with an xHCI Virtual Function. This condition occurs when system software running in another Virtual Machine issues a *Negotiate Bandwidth Command* through its xHCI Virtual Function. System software should immediately honor an unsolicited *Bandwidth Request Event* and free unused USB bandwidth by selecting lower bandwidth alternate configurations or interfaces on the devices that it owns.

Note: If the target Event Ring for a device is full, the *Bandwidth Request Event* shall be dropped by the xHC.

Note: The xHCI may generate a *Bandwidth Request Event* for the same slot that a *Negotiate Bandwidth Command* was issued to.

4.6.14 Set Latency Tolerance Value (LTV) (Optional Normative)

Refer to section 4.13.1 for an overview of the xHCI's USB3 *Latency Tolerance Messaging* (LTM) support. This section describes the *Set LTV Command* which is one component of the *Latency Tolerance Reporting* (LTR) mechanism.

The *Set LTV Command* provides a simple means for host software to provide a *Best Effort Latency Tolerance* (BELT) value to the xHC. This command is optional normative, however it shall be supported if the xHC also supports a corresponding host interconnect LTR mechanism.

Note: The host's interconnect LTR definition is owned by the respective bus specification and is outside the scope of this document. (e.g. PCI Express, AHBA, etc.)

The value of the *BELT* field in the *Set LTV Command TRB* shall be treated in exactly the same way as BELT values received from USB3 devices by the xHC. Refer to section 4.13.1.

Note: The manner in which these values are stored is implementation specific and as such falls outside the scope of this specification.

If the *Latency Tolerance Messaging Capability* (LTC) bit in the HCCPARAMS1 register is '0', the xHC shall not support this command.

Note: If *LTC* = 0, then this xHC implementation does not translate LTM messages from a device into system LTM messages. However, if enabled in the DNCTRL register (*N2* = '1'), then LTM Device Notification TPs are received by the xHC shall generate *Device Notification Events*. Refer to section 4.13.1.

This command will complete with a *Success Completion Code* if the command is supported, or a *TRB Error* Completion Code if the command is not supported.

The format of the *Set Latency Tolerance Value Command TRB* is defined in section 6.4.3.13.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

To issue a *Set Latency Tolerance Value Command* system software shall perform the following operations:

- Insert an *Set Latency Tolerance Value Command* on the Command Ring and initialize the following fields:
  - *TRB Type* = *Set Latency Tolerance Value Command* (refer to Table 6-86).
  - *BELT* = The *Best Effort Latency Tolerance* value provided by software.
  - Clear all other fields of the command TRB to '0'.
  - *Cycle bit* = Command Ring's PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

When a *Set Latency Tolerance Value Command* is executed by the xHC it shall perform the following operations:

- Record the value of the *BELT* field as the host defined LTV.
- If the value of the *BELT* field is less than the “current” LTV maintained by the xHC:
    - Set the value of the *BELT* field as the “current” xHC LTV.
    - Send the host-specific LTM to the host, reporting the new LTV to the system.
- Insert a *Command Completion Event* on the Event Ring of Interrupter 0 and initialize the following fields:
    - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
    - *Command TRB Pointer* = The address of the *Set Latency Tolerance Value Command TRB*.
    - *Completion Code* = *Success* (refer to Table 6-85).
    - Clear all other fields of the event TRB to ‘0’.
    - *Cycle bit* = Event Ring’s PCS flag.

4.6.15 Get Port Bandwidth

The *Get Port Bandwidth Command* is issued by software to retrieve the percentage of *Total Available Bandwidth* on each Root Hub Port of the xHC or on the downstream facing ports of a external USB hub. This information can be used by system software to recommend topology changes to the user if they were unable to enumerate a device due to a *Bandwidth Error* (Root Hub) or *Secondary Bandwidth Error* (external hub).

An xHC may support multiple *USB Bus Instances* (BI), where each BI represents a “unit” bandwidth at the speed that the BI supports. Also note that multiple Root Hub ports may be assigned to a single BI.

For instance, an xHCI implementation that supports 8 ports may provide 1 SS BI, 2 HS BIs, and 4 LS/FS BIs. So in this example there are 7 USB BIs, 1 SS (5Gb/s), 2 HS (480 Mb/s) and 4 LS/FS (12Mb/s). Any SS device attached to a root hub port shares the SS BI bandwidth. If the 2 HS BIs are mapped to ports 0 to 3 and 4 to 7, and the 4 LS/FS BIs are mapped to ports 0 and 1, 2 and 3, 4 and 5, and 6 and 7, respectively, then an LS/FS device attached to port 5 shares the BW available on port 4 provided by one the LS/FS BIs, but not with any other ports. A more sophisticated xHC implementation may have the ability to dynamically map ports to BIs as function a device’s bandwidth requirements.

A USB2 hub may support a single or multiple *Transaction Translators* (TT), where a single TT is capable of providing the equivalent of a LS/FS BI’s bandwidth. If a USB2 Hub supports a single TT, then all of its downstream facing ports attached to LS or FS devices shall share the bandwidth of the single TT (i.e. a LS/FS BI). If a USB2 Hub supports a multi-TT capability, then a separate TT exists for each of its downstream facing ports and each port is capable of providing the bandwidth of a LS/FS BI.

When software issues a *Get Port Bandwidth Command* it is trying to accommodate the bandwidth requirements of a particular device. By providing a *Device Speed* parameter in the *Get Port Bandwidth Command*, the xHC can supply software with *Total Available Bandwidth* on each port of the Root Hub or USB2 hub, at a particular speed, without exposing its BI or TT to Port mapping scheme.

Software, knowing the percentage of *Total Available Bandwidth* on a hub port, the speed that the device in question is operating at, and the device's bandwidth requirement, may determine if a particular port will meet the device's bandwidth needs.

The xHC uses the *Device Speed* parameter to identify which Bus Instance(s) to use when it calculates the *Total Available Bandwidth* on that port.

The *Get Port Bandwidth Command* passes a pointer to a *Port Bandwidth Context* data structure to the xHC. The xHC updates this context with the percentage of *Total Available Bandwidth* on each port. If a hub is attached to a Root Hub port then the reported bandwidth is available on any unused port of the hub or any port of the hub that is operating at the *Device Speed*.

For the Root Hub the *Port Bandwidth Context* shall be at least *NumPorts*+1 bytes in size or for an external hub the *Port Bandwidth Context* shall be at least *bNbrPorts*[25]+1 bytes in size, rounded up to the nearest Dword boundary.

The xHC overwrites the *Port Bandwidth Context* when it executes the *Get Port Bandwidth Command*, so software does not need to initialize the context data structure before passing it to the xHC.

- A Root Hub port assigned to the *Debug Capability* shall report '0' bandwidth available.
- If the *Device Speed* parameter is LS, FS, or HS, then USB3 (SS) Root Hub ports shall report '0' bandwidth available.
- If the *Device Speed* parameter is SS, then USB2 Root Hub ports shall report '0' bandwidth available.

Note: Software shall consider any port that reports '0' bandwidth available as being unusable. A port that, as far as software is concerned, does not have a device attached may report '0' bandwidth available. e.g. a VMM shall report '0' bandwidth for a port if the device attached to it is assigned to another VF.

[25]Refer to section 11.23.2.1 in the USB2 spec for the definition Hub Descriptor *bNbrPorts* field.

Consider a physical connector that is “USB3 compatible” and has a SS device attached it. The connector will be wired to a USB2 and a USB3 Root Hub Port. When the USB2 Root Hub Port is queried for its HS bandwidth availability, it will not know that a SS device is attached to physical connector and report a non-zero HS bandwidth availability, when in reality the USB2 Root Hub port is not available because it is associated with a physical connector that is attached to SS device. The same problem will occur with a USB3 Root Hub port if a USB2 device or hub is attached to the physical connector. Note that the problem does not occur if a USB3 hub is attached because both Root Hub Ports see a hub attached. Software, knowing the Root Hub Port to physical USB connector mapping (refer to section 4.19.7) and whether the attached device is a USB2 or USB3 hub, shall be responsible for correcting the reported Port Bandwidth Values.

The format of the *Get Port Bandwidth Command TRB* is defined in section 6.4.3.14.

The *Get Port Bandwidth Command* utilizes the *Port Bandwidth Context* data structure defined in section 6.2.6.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

To issue a *Get Port Bandwidth Command*, system software shall perform the following operations:

- Allocate and initialize an *Port Bandwidth Context* data structure.
- Insert a *Get Port Bandwidth Command TRB* on the Command Ring
  - *TRB Type* = *Get Port Bandwidth Command* (refer to Table 6-86).
  - *Dev Speed* = The bus speed of the target device. Refer to the *Dev Speed* field in Table 6-76 for the encoding.
  - *Hub Slot ID* = ‘0’ if referencing Root Hub ports (i.e. the Primary Bandwidth Domain) or the value of the respective hub’s *Slot ID* if referencing the ports of a USB2 hub (i.e. a Secondary Bandwidth Domain). Refer to section 4.16.2 for more information on Bandwidth Domains.
  - *Port Bandwidth Context Pointer* = The base address of the *Port Bandwidth Context* data structure.
  - Clear all other fields of the command TRB to ‘0’.
  - *Cycle bit* = Command Ring’s PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

When a *Get Port Bandwidth Command* is executed by the xHC it shall perform the following operations:

- Insert a *Command Completion Event TRB* on the Event Ring.
  - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
  - *Command TRB Pointer* = The address of the *Get Port Bandwidth Command TRB*.

- *Slot ID* = ‘0’.
- If the *Dev Speed* field is valid (i.e. not equal to Undefined or Reserved):
  - If the *Hub Slot ID* field = ‘0’:
    - Compute Percentage of *Total Available Bandwidth* for each Root Hub port based on its *Speed*. Use the value of the *Dev Speed* field for ports that do not have devices attached.
  - else
    - Compute the percentage of *Total Available Bandwidth* for the ports of the hub specified by the *Hub Slot ID* based on their *Speed*. Use the value of the *Dev Speed* field for ports that do not have devices attached.
  - Copy the results to the *Port Bandwidth Context*.
  - *Completion Code* = *Success* (refer to Table 6-85).
- else // The *Dev Speed* field is not valid
  - *Completion Code* = *TRB Error*.
- Clear all other fields of the event TRB to ‘0’.
- *Cycle bit* = Event Ring’s PCS flag.

Note: If a non-zero *Hub Slot ID* references a Device Slot whose Slot Context *Hub* field = ‘0’ or *Speed* field is not equal to *High-speed*, may result in undefined behavior by the xHC.

4.6.16 Force Header

The *Force Header Command* is issued by software to send a Link Management (LMP) or Transaction Packet (TP) to a USB device, through a selected Root Hub Port. For instance, it may be used to send a PING TP or a Vendor Device Test LMP.

Note: Inappropriate or incorrect use of this command may cause the xHC link state machines to get out of sync with those on an attached device. Software shall comprehend the possible side effects of the specific headers that are forced on the USB. If a forced header results in undefined behavior by the device or the xHC (e.g. a DPH with no DP), software may have to reset the device, a Root Hub port, the xHC, or all of them to restore normal operating conditions.

The xHC is not required to comprehend the content of the header being forced. Depending upon the type of header forced, it is possible for various parameters in the header (such a Data Packet sequence numbers) to be out of sync with the host controller and/or device. In addition, some TPs may result in Device responses which will not be comprehended by the xHC. It may be necessary to reset the xHC to recover from these conditions.

The format of the *Force Header Command TRB* is defined in section 6.4.3.15.

The format of the *Command Completion Event TRB* is defined in section 6.4.2.2.

To issue a *Force Header Command*, system software shall perform the following operations:

- Insert a *Force Header Command TRB* on the Command Ring
  - *TRB Type* = *Force Header Command* (refer to Table 6-86).
  - *Root Hub Port Number* = The number of the Root Hub Port that defines the target of the Header packet.
  - *Packet Type* = The field identifies the SS packet type. Refer to section 8.3.1.2 in the USB3 specification for valid values.
  - *Header Info* = The header Type specific data to send to the target device. Refer to section 8 in the USB3 specification for the encoding information.
  - Clear all other fields of the command TRB to '0'.
  - *Cycle bit* = Command Ring's PCS flag.
- Write the Host Controller Doorbell with *DB Target* = *Host Controller Command*.

When a *Force Header Command* is executed by the xHC it shall perform the following operations:

- Insert a *Command Completion Event* on the Event Ring.
  - *TRB Type* = *Command Completion Event* (refer to Table 6-86).
  - *Command TRB Pointer* = The address of the *Force Header Command TRB*.
  - *Slot ID* = 0.
  - If the value *Root Hub Port Number* field is in range:
    - If the Force Header packet was transmitted successfully:
      - *Completion Code* = *Success* (refer to Table 6-85).
    - else // The Force Header packet was not transmitted successfully
      - *Completion Code* = *Undefined Error*.
  - else // The value *Root Hub Port Number* field is not in range
    - *Completion Code* = *TRB Error*.
  - Clear all other fields of the event TRB to '0'.
  - *Cycle bit* = Event Ring's PCS flag.

4.7 Doorbells

The xHCI presents an array of up to 256 32-bit Doorbell Registers (refer to section 5.6), which reside in MMIO space and are indexed by Device Slot ID. The base of the Doorbell Register Array is pointed to by the Doorbell Offset (DBOFF) register in the xHCI Capability Registers (refer to section 5.3.7).

Each Doorbell Register contains a *DB Target* field, which is used to indicate the reason for a software reference to the register. System software "rings" a doorbell by writing a Doorbell Register with the appropriate value in the *DB Target* field.

Doorbell Register 0 is dedicated to the Host Controller. For this register, there is only one valid value for the DB Target field, 0 (Host Controller Command). The remaining values (1-255) are reserved.

Doorbell Registers 1-255 are referred to as the *Device Context Doorbell* registers. There is a 1:1 mapping of *Device Context Doorbell* registers to *Device Slots*. System software rings a *Device Context Doorbell* after it has inserted work on a Transfer Ring (endpoint/Stream) associated with the respective Device Slot. The *DB Target* and *DB Stream ID* fields of a *Device Context Doorbell* register is used to identify which Transfer Ring of a device has been modified. Refer to Table 5-40 for the encoding of the DB Target field.

The xHC internally records all Doorbell Register write references and uses the information to determine if the Command Ring or a Transfer Ring has newly posted work items (TDs). There is no need to “clear” a Doorbell Register. To inform the xHC that work has been posted to two separate Transfer Rings of a device, system software shall post two writes to the associated Doorbell Registers, where the value of the DB Target field identifies the respective Transfer Ring.

Doorbell registers return no information when read.

Software shall not write to a Doorbell register:

- If the associated Device Slot is in the *Disabled* state.
- If the associated Device Slot is not in the *Disabled* state and the *DB Target* field is set to an endpoint that is in the *Disabled* state.

If a doorbell register is written by software with the *DB Target* value that references an endpoint that is in the *Disabled* state, the xHC should generate a *Transfer Event TRB* with the *TRB Pointer*, TRB Transfer *Length*, *Event Data* (ED) fields set to ‘0’, a *Completion Code* of *Endpoint Not Enabled Error*, and the *Slot ID* and *Endpoint ID* fields contain the IDs of device slot/endpoint that the doorbell that was rung for. This transfer Event TRB shall be posted to the Primary Event Ring.

An *Endpoint Not Enabled Error* should be generated for doorbell register writes to Device Slots that are in the *Disabled* state regardless of the *DB Target* value provided.

The xHC may ignore doorbell references to Device Slots in the *Disabled* state or endpoints in the *Disabled* state.

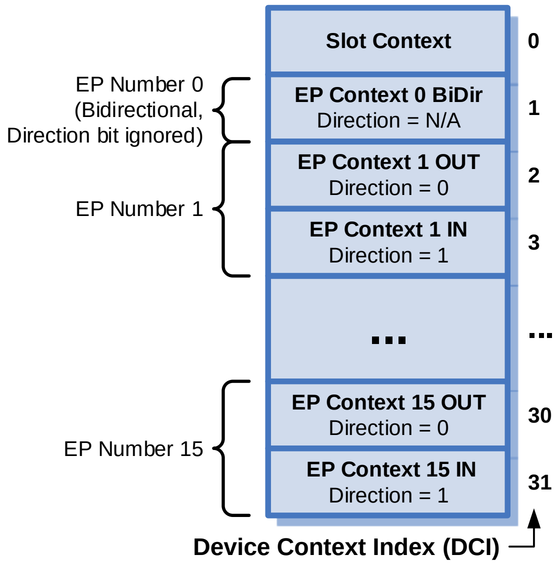
The xHC shall ignore doorbell references to endpoints in the *Halted* or *Error* state.

4.8 Endpoint

A USB device supports up to 31 endpoints (EPs): e.g.15 IN, 15 OUT, and 1 Control. The Default Control EP (0) is a bidirectional EP defined for all USB devices.

4.8.1 Endpoint Addressing

Figure 4-4: Endpoint Context Addressing



An *Endpoint Address* defined by a USB Endpoint Descriptor allows up to 31 possible values, where a 4-bit *Endpoint Number* is combined with a *Direction* bit (refer to section 9.6.6 in the USB2 spec). The xHCI parallels this organization by using the *Endpoint Number* to select one of 16 *Endpoint Context* data structure pairs, and the *Direction* bit to select the IN or OUT *Endpoint Context* of a pair. Refer to Figure 4-4 to the right.

A Control endpoint (e.g. *EP Number 0* in Figure 4-4) is a bidirectional endpoint and, per the USB specification, the *Direction* bit is "ignored" when calculating its *Endpoint Address*, i.e. only the *Endpoint Number* is used to calculate the location of a Control Endpoint Context data structure. To accommodate the addressing anomaly of USB bidirectional endpoint addressing the xHC shall use the IN (odd) Endpoint Context of the pair to manage bidirectional endpoints.

The USB specification allows a device to define additional Control (bidirectional) endpoints, beyond the Default Control Endpoint (EP 0) required by the USB Framework. Using the rules defined above, the xHCI is capable of supporting additional Control endpoints.

For all Endpoint Numbers greater than 0, the xHC shall ignore the OUT (even) Endpoint Context of the pair of any endpoint that declares itself as a bidirectional. Software shall use the IN (odd) Endpoint Context of a pair for managing a Control Endpoint.

4.8.2 Endpoint Context Initialization

All fields of an Input Endpoint Context data structure (including the Reserved fields) shall be initialized to ‘0’ with the following exceptions:

4.8.2.1 Default Control Endpoint 0

- *EP Type* = Control. Refer to Table 6-9 for the encoding.
- *Max Packet Size* = For USB2 devices: Device Descriptor:bMaxPacketSize0 or for USB3 devices” Device Descriptor:$2^{bMaxPacketSize0}$. May be set to Default Endpoint Max Packet Size until USB Device Descriptor is retrieved. An *Evaluate Endpoint Command* shall be used to modify the value of *Max Packet Size* when the device slot is in the Addressed state.
- *CErr* = 3. Enables 3 retries.
- *TR Dequeue Pointer* = Start address of the first segment of the previously allocated Transfer Ring.
- *Dequeue Cycle State* (DCS) = 1. Assuming that all TRBs in the segment referenced by the TR Dequeue Pointer have been initialized to ‘0’, this field reflects Cycle bit state for valid TRBs written by software.

4.8.2.2 Control Endpoints

Identical to the Default Control Endpoint except that the *Max Packet Size* shall be set to the value of the associated Endpoint Descriptor:wMaxPacketSize.

4.8.2.3 Bulk Endpoints

- *EP Type* = Bulk IN or Bulk OUT. Refer to Table 6-9 for the encoding.
- *Max Packet Size* = Endpoint Descriptor:wMaxPacketSize.
- *Max Burst Size* = For USB3 devices: SuperSpeed Endpoint Companion Descriptor:bMaxBurst, for USB2 devices: ‘0’.
- *CErr* = 3. Enables 3 retries.
- If Streams are enabled (i.e. SuperSpeed Endpoint Companion Descriptor:bmAttributes MaxStreams field > 0):
    - Allocate and clear *Primary Stream Array*.
    - *MaxPStreams* = Size of *Primary Stream Array*.
    - *TR Dequeue Pointer* = Start address of *Primary Stream Array*.

- *HID* = Initialize as required to enable or disable *Host Initiated Move Data* operations.
- *LSA* = Initialize as required to enable or disable *Linear Stream Array* operations.

- else
  - *MaxPStreams* = '0'.
  - *TR Dequeue Pointer* = Start address of the first segment of the previously allocated Transfer Ring.
  - *Dequeue Cycle State* (DCS) = 1. Assuming that all TRBs in the segment referenced by the TR Dequeue Pointer have been initialized to '0', this field reflects Cycle bit state for valid TRBs written by software.

Note: The Endpoint Context *Dequeue Cycle State* (DCS) field is not applicable if the Streams are enabled.

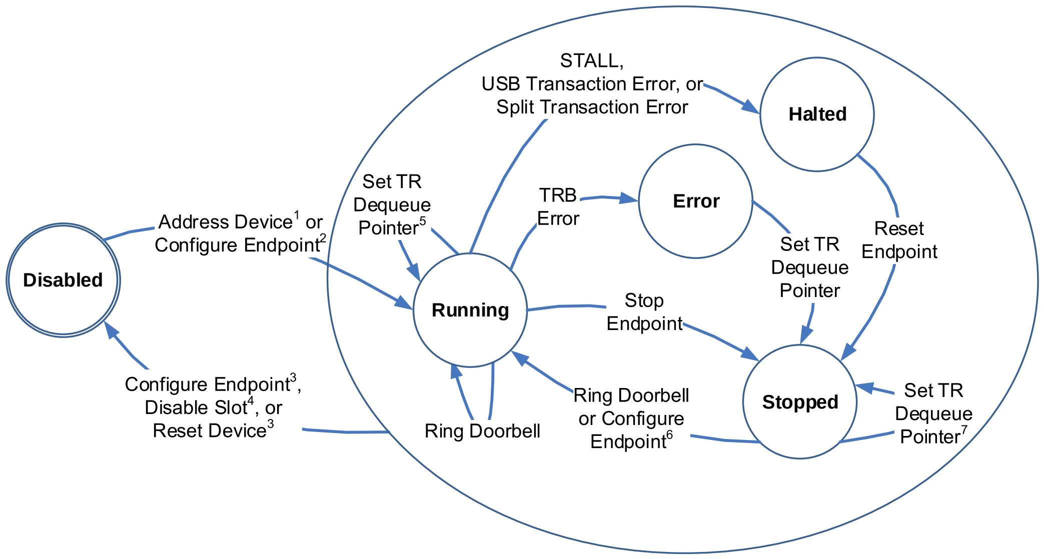
4.8.2.4 Isoch or Interrupt Endpoints

- *EP Type* = Isoch IN, Isoch OUT, Interrupt IN or Interrupt OUT. Refer to Table 6-9 for the encoding.
- *Max Packet Size* = Endpoint Descriptor:wMaxPacketSize & 07FFh.
- *Max Burst Size* = SuperSpeed Endpoint Companion Descriptor:bMaxBurst or (Endpoint Descriptor: wMaxPacketSize & 1800h) >> 11.
- *Mult* = SuperSpeed Endpoint Companion Descriptor:bmAttributes:Mult field. Always '0' for Interrupt endpoints.
- *CErr* = 3 for Interrupt endpoints. Enables 3 retries.
  CErr = 0 for Isoch endpoints. Retries are not performed for Isoch endpoints.
- *TR Dequeue Pointer* = Start address of the first segment of the previously allocated Transfer Ring.
- *Dequeue Cycle State* (DCS) = 1. Assuming that all TRBs in the segment referenced by the TR Dequeue Pointer have been initialized to '0', this field reflects Cycle bit state for valid TRBs written by software.
- *Max ESIT Payload* = Refer to section 4.14.2 for value.

4.8.3 Endpoint Context State

The current state of an *Endpoint Context* is identified by its respective *Endpoint State* (EP State) field. Figure 4-5 defines the Endpoint States.

Figure 4-5: Endpoint State Diagram



1. The *Address Device Command* transitions the Default Control Endpoint from the *Disabled* to the *Running* state.

2. The *Configure Endpoint Command* (*Add* (A)= '1'and *Drop* (D) = '0') shall transition an endpoint, except the Default Control Endpoint, from the *Disabled* to the *Running* state.

3. The *Configure Endpoint Command* (*Add* (A)= '0' and *Drop* (D) = '1') or *Reset Device Command* shall transition an endpoint, from any state to the *Disabled* state, except the Default Control Endpoint which shall transition from the *Stopped* to the *Running* state.

4. The *Disable Slot Command* shall transition all endpoints of a Device Slot, including the Default Control Endpoint, from any state to the *Disabled* state.

5. In the *Running* state, a *Set TR Dequeue Pointer Command* should only be issued to the non-active Transfer Rings of a Stream endpoint. Refer to section 4.12 for active vs. non-active Stream Context information.

6. The *Configure Endpoint Command* (*Add* (A) = '1' and *Drop* (D) = '1') shall transition an endpoint, except the Default Control Endpoint, from the *Stopped* to the *Running* state.

7. In the *Stopped* state, a *Set TR Dequeue Pointer Command* may be used to modify the starting TRB of an endpoint or non-active Stream prior to ringing the Doorbell. Refer to section 4.12 for active vs. non-active Stream Context information.

Figure 4-5 illustrates the state transitions presented by an endpoint. The *Disabled* to *Running* transition for the Default Control Endpoint shall occur due to an *Address Device Command*, and for all other endpoints the transition shall be invoked by a *Configure Endpoint Command*. Refer to Appendix E for state machine notation.

A Halt condition, e.g. a *Stall Error, Invalid Stream Type Error*, *Invalid Stream ID Error*, *Babble Detected Error*, *Event Lost Error*, *USB Transaction Error*, or a *Split Transaction Error* detected on a USB pipe shall cause a *Running* Endpoint to transition to the *Halted* state. A *Reset Endpoint Command* shall be used to clear the Halt condition on the endpoint and transition the endpoint to the *Stopped* state. A *Stop Endpoint Command* received while an endpoint is in the *Halted* state shall have no effect and shall generate a *Command Completion Event* with the Completion Code set to *Context State Error*.

Note: A STALL detected on any stage (Setup, Data, or Status) of a Default Control Endpoint request shall transition the *Endpoint Context* to the *Halted* state. A Default Control Endpoint STALL condition is cleared by a *Reset Endpoint Command* which transitions the endpoint from the Halted to the Stopped state. The Default Control Endpoint shall return to the *Running* state when the Doorbell is rung for the next Setup Stage TD sent to the endpoint.

Section 8.5.3.4 of the USB2 spec and section 8.12.2.3 of the USB3 spec state of Control pipes, "Unlike the case of a functional stall, protocol stall does not indicate an error with the device." The xHC treats a functional stall and protocol stall identically, by Halting the endpoint and requiring software to clear the condition by issuing a *Reset Endpoint Command*.

Note: If the STALL condition is detected on the Setup or Data Stage TD of a request, software shall be responsible for removing the Data Stage or Status Stage TDs, respectively, associated with the request from the Transfer Ring.

A *TRB Error* condition should cause a *Running* Endpoint to transition to the *Error* state. A *Set TR Dequeue Pointer Command* shall be used to transition the endpoint to the *Stopped* state. A *Stop Endpoint Command* received while an endpoint is in the *Error* state shall have no effect and shall generate a *Command Completion Event* with the Completion Code set to *Context State Error*.

Note: An endpoint in the *Running* state may be *Busy* (actively processing TRBs on its Transfer Ring) or *Idle* (the endpoint is not processing TRBs and waiting for a doorbell ring) sub-state, i.e. an endpoint does not exit the *Running* state if it exhausts its Transfer Ring.

Note: Some xHC implementations may not handle a *TRB Error* gracefully, resulting in undefined behavior and possibly the assertion of *HCE*. It is the responsibility of software to *always* present correctly formed TRBs to the xHC.

A *Stop Endpoint Command* shall also transition the endpoint to the *Stopped* state. While in the *Stopped* state, the ownership of the Transfer Ring is relinquished up by the xHC, allowing software to add, delete, or modify any TD on the ring.

If an endpoint is in the *Stopped* state when the doorbell is rung, it will transition to the *Running* state. A *Configure Endpoint Command* shall also transition a *Stopped* endpoint to the *Running* state. Note that a *Configure Endpoint Command* does not affect the Default Control Endpoint, therefore shall not transition the Default Control Endpoint from the *Stopped* to the *Running* state.

A *Configure Endpoint* "deconfigure" (*DC* = '1') or *Reset Device Command* shall transition all endpoints, except for the Default Control Endpoint, from the *Running*, *Halted*, *Error*, or *Stopped* states to the *Disabled* state.

A *Disable Slot Command* shall transition all endpoints, including the Default Control Endpoint, from the *Running*, *Halted*, *Error*, or *Stopped* states to the *Disabled* state, as noted by the large bubble. System software is responsible for issuing a *Disable Slot Command* when a device detach event is detected.

A *Set TR Dequeue Pointer Command* may be issued to a non-active Stream Context of an endpoint to set its Dequeue Pointer while the endpoint is in the *Running* state. Refer to sections 4.6.10 and 4.12.

An endpoint in the *Stopped* state shall not generate Transfer Events.

When an endpoint transitions from the *Stopped* to the *Running* state due to a doorbell ring, the *EP State* field of the Output *Endpoint Context* shall be updated by the xHC to running before any Transfer Events are generated.

Note: If the xHC is reset while an endpoint is not in the *Disabled* state, the value of the *Endpoint State* (EP State) field shall be invalid.

Note: An Endpoint is considered "enabled" if it is not in the *Disabled* state.

Note: Software shall not write to the Doorbell register with the *DB Target* field value set to an endpoint that is in the *Disabled* state.

Note: A control, bulk, or Interrupt endpoint shall transition to the *Halted* state if a tHostTransactionTimeout occurs (refer to Table 8-36 in the USB3 spec). For Isoch transactions the host shall not perform any more transactions to the endpoint in the current Service Interval. And the host shall not halt the endpoint and shall restart transactions to the endpoint in the next Service Interval. And retries are not performed for any endpoint type if a tHostTransactionTimeout occurs. Note that the tHostTransactionTimeout is an xHC implementation specific delay within the range specified in the USB3 spec.

Note: There are several cases where the *EP State* field in the Output *Endpoint Context* may not reflect the current state of an endpoint. The xHC should attempt to keep *EP State* as current as possible, however it may defer these updates to perform higher priority references to memory, e.g. Isoch data transfers, etc. Software should maintain an internal variable that tracks the state of an endpoint and not depend on *EP State* to represent the instantaneous state of an endpoint.

For example, when a Command that affects *EP State* is issued, the value of *EP State* may be updated anytime between when software rings the Command Ring doorbell for a command and when the associated *Command Completion Event* is placed on the Event Ring by the xHC. The update of *EP State* may also be delayed relative to a Doorbell ring or error condition (e.g. TRB Error, STALL, or USB Transaction Error) that causes an EP State change not generated by a command.

Software should maintain an accurate value for *EP State*, by tracking it with an internal variable that is driven by Events and Doorbell accesses associated with an endpoint using the following method:

- When a command is issued to an endpoint that affects its state, software should use the *Command Completion Event* to update its image of *EP State* to the appropriate state.
- When a Transfer Event reports a *TRB Error*, software should update its image of *EP State* to *Error*.
- When a Transfer Event reports a *Stall Error* or *USB Transaction Error*, software should update its image of *EP State* to *Halted*.
- When software rings the Doorbell of an endpoint to transition it from the *Stopped* to *Running* state, it should update its image of *EP State* to *Running*.

Refer to section 6.2.3 for more information on the *Endpoint Context* data structure.

4.9 TRB Ring

A TRB (Transfer Request Block) Ring defines a queue, which is used to transfer Work Items between producer and consumer entities[26].

A TRB Ring is defined as a circular queue of TRB data structures. TRB rings are used to pass **Work Items** from the producer to the consumer. Two pointers (Enqueue and Dequeue) associated with each ring identify where the producer will Enqueue the next Work Item on the ring and where the consumer will Dequeue the next Work Item from the ring.

A Work Item is comprised of one or more TRB data structures. A Work Item may define an operation to perform, or the result of an operation that has been performed.

[26] Note: The xHCI Producer/Consumer model is not related to the PCI Producer/Consumer model.

There are 3 basic types or TRB Rings; **Transfer, Event**, and **Command**. Each type of ring defines an exclusive set of TRB data structures; however they all employ the underlying TRB Ring mechanism to organize their work items and the basic TRB template.

Transfer Rings provide data transport to and from USB devices. There is a 1:1 mapping between Transfer Rings and USB Pipes. They are defined by an Endpoint Context data structure contained in a Device Context, or the Stream Context Array pointed to by the Endpoint Context.

The **Event Ring** provides the xHC with a means of reporting to system software: data transfer and command completion status, Root Hub port status changes, and other xHC related events. An Event Ring is defined by the Event Ring Segment Table Base Address, Segment Table Size, and Dequeue Pointer registers which reside in the Runtime Registers.

The **Command Ring** provides system software the ability to issue commands to enumerate USB Devices, configure the xHC to support those devices, and to coordinate virtualization features. The Command Ring is managed by the Command Ring Control Register that resides in the Operational Registers.

The **Enqueue Pointer** and **Dequeue Pointer** are terms used to refer to the logical beginning and end of the valid entries in a TRB Ring. The size of a TRB ring is determined by the number and size of the segments that comprise the ring.

Note: The Dequeue and Enqueue Pointers for Transfer and Command Rings are NOT defined as physical xHC registers. However a facsimile of these pointers are maintained internally by the xHC and system software to manage a respective ring.

Note: Only the Dequeue Pointer for an Event Ring is defined as a physical xHC register. A facsimile of the Enqueue Pointer is maintained internally by the xHC and system software to manage an Event Ring.

This section describes how these “facsimiles” are maintained. The Enqueue and Dequeue Pointers are always advanced starting from the TRB entry pointed to by their initial values.

The Enqueue Pointer is the address of the next TRB in a ring available to the producer. The producer constructs new Work Items starting with the TRB at this location, and advances the Enqueue Pointer when the construction is complete.

The Dequeue Pointer is the address of the next TRB to be serviced by the consumer.

If the Dequeue Pointer equals the Enqueue Pointer, then the TRB Ring is empty. If the “Enqueue Pointer + 1” = Dequeue Pointer, then the ring is full. Note that the calculation of the “Enqueue Pointer + 1” value requires comprehending Link TRBs. Refer to section 4.11.5.1 for more information on Link TRBs.

TRBs between the Enqueue Pointer -1 and Dequeue Pointer are owned by the consumer of the Work Items. All other TRBs in a ring are owned by the producer of the Work Items. TRB ownership is passed to the consumer when the Enqueue Pointer is advanced by the producer. TRB ownership is passed to producer when the Dequeue Pointer is advanced by the consumer.

A consumer or producer may modify any TRB that it owns, at any time, and in any order. The producer shall never modify a TRB that is owed by the consumer. And the consumer shall never modify a TRB that is owed by the producer.

TRBs shall be executed by the consumer in order, starting at the TRB referenced by the Dequeue Pointer.

All TRB data structures shall be 16 bytes in size.

TRB Rings may be larger than a Page, however they shall not cross a 64K byte boundary. Refer to section 4.11.5.1 for more information on TRB Rings and page boundaries.

Initially when the TRB Ring is created in memory, or if it is ever re-initialized, all TRBs in the ring shall be cleared to ‘0’. This state represents an empty queue.

Note: Refer to Table 6-86 for a definition of the valid TRB types allowed on a specific TRB ring type. Table 6-87 defines the allowable Transfer Ring TRB Types as function of endpoint type.

Note: Ownership of TRBs on a Transfer Ring is strictly determined by the location of its Enqueue and Dequeue pointers. A Short Packet, error, or other condition reported for a TRB that is not the last TRB of a TD shall not be interpreted by the producer (software) as indicating that the ownership of the remaining TRBs in the TD have also transitioned to the producer.

4.9.1 Transfer Descriptors

Transfer Rings support *Transfer Descriptors* (TDs) that consists of 1 or more TRBs. The TRB *Chain* (C) bit is set in all but the last TRB of a TD.

The xHC shall schedule *Max Packet Size* USB transactions for all packets associated with a TD, except possibly for the last packet if the TD does not define an integer multiple of *Max Packet Size* data bytes.

To generate a “zero-length” USB transaction, software shall explicitly define a TD with a single Transfer TRB, and its *TRB Transfer Length* field shall equal ‘0’. Note that this TD may include non-Transfer TRBs, e.g. an Event Data or Link TRB.

Refer to section 4.14.1 for an Implementation Note that discusses TRBs and system bandwidth management.

There are many conditions described in this specification where the xHC shall “advance to the next TD”. However, if the xHC is processing a partially formed TD when one of these conditions occurs, then advancing to the next TD is not possible and the xHC shall stop advancing when it reaches the Enqueue Pointer (i.e. the Cycle bit transition). In this case, the xHC sees the Transfer Ring as empty (i.e. the Dequeue Pointer is equal to the Enqueue Pointer), and the next time the doorbell is rung for the endpoint, the xHC shall attempt to advance to the next TD boundary. Note that the xHC shall always interpret the *New TR Dequeue Pointer* field of a *Set TR Dequeue Pointer Command* as a pointer to the “next TD”, terminate any effort to “advance to the next TD”.

A “partially completed TD” is identified by the case where the *Chain* bit (CH) set to ‘1’ in the TRB referenced by the Dequeue Pointer and advancing the Dequeue Pointer sets it equal to the Enqueue Pointer.

Note: Command and Event TRBs do not support a Chain bit (CH), so all Command Descriptors (CDs) and Event Descriptors (EDs) only consist of a single TRB.

Note: If the xHC receives a Short Packet from a device, then it shall retire the current TD. If another TD is defined on the Transfer Ring, the xHC shall advance to it and begin IN transactions. If the *EOB* flag was set in a short DP received on a SS IN pipe, then the host shall retire the current TD, and wait for an ERDY from the device before beginning IN transactions for the next TD (if one exists). Refer to section 4.10.1.1 for detailed information on Short Packet handling.

Note: If an error is detected while processing a multi-TRB TD, the xHC shall generate a Transfer Event for the TRB that the error was detected on with the appropriate error *Condition Code*, then may advance to the next TD. If in the process of advancing to the next TD, a Transfer TRB is encountered with its *IOC* flag set, then the *Condition Code* of the Transfer Event generated for that Transfer TRB should be *Success*, because there was no error actually associated with the TRB that generated the Event. However, an xHC implementation may redundantly assert the original error *Condition Code*. As a general rule, the *Completion Code* of a Transfer Event represents the status of the buffer referenced by the Transfer TRB that generated it, however there may be exceptions.

4.9.2 Transfer Ring Management

This section describes the operation of Enqueue and Dequeue Pointers in Transfer Rings. The operation of Enqueue and Dequeue Pointers in Command Rings is described in section 4.9.3 and Event Rings in section 4.9.4.

Figure 4-6 shows a graphical representation of a Transfer Ring. The producer (host) places items in a Transfer Ring at the **Enqueue Pointer**, and the consumer (xHC) removes items from the Transfer Ring at the **Dequeue Pointer**.

The ***Cycle bit*** field in a TRB identifies the location of the Enqueue Pointer in a Transfer Ring, eliminating the need to define a physical Enqueue Pointer register.

Software uses and maintains private copies of the Enqueue and Dequeue Pointers for each Transfer Ring. The Enqueue and Dequeue Pointers are set to the address of the first TRB location in the Transfer Ring and written to the Endpoint/Stream Context *TR Dequeue Pointer* field, when a Transfer Ring is initially set up. Software uses the Enqueue Pointer to determine where to place the next Work Item on a Transfer Ring. Software advances its copy of the Enqueue Pointer, by either incrementing it by the TRB size, or reloading it with the value of the *Ring Segment Pointer* field when it encounters a Link TRB, every time it writes a TRB to the Transfer Ring. The position of the Enqueue pointer is also marked in the Transfer Ring itself, by a transition of the Cycle bit.

The xHC also maintains private copies of the Enqueue and Dequeue Pointers for each Transfer Ring. When a Transfer Ring is enabled or reset, the xHC initializes its copies of the Enqueue and Dequeue Pointers with the value of the Endpoint/Stream Context TR Dequeue Pointer field.

The xHC uses the Dequeue Pointer to determine where to fetch the next Work Item from a Transfer Ring. The xHC advances its copy of the Dequeue Pointer, by either incrementing it by the TRB size, or reloading it with the value of the *Ring Segment Pointer* field when it encounters a Link TRB, every time it fetches a TRB from the Transfer Ring.

The xHC employs the Event Ring to report the current value of the Dequeue Pointer to system software. Each Transfer Event placed on the Event Ring points to the Transfer TRB that generated it. Software may interpret the pointer value from the latest Transfer Event as the "current value" of the xHC Dequeue Pointer.

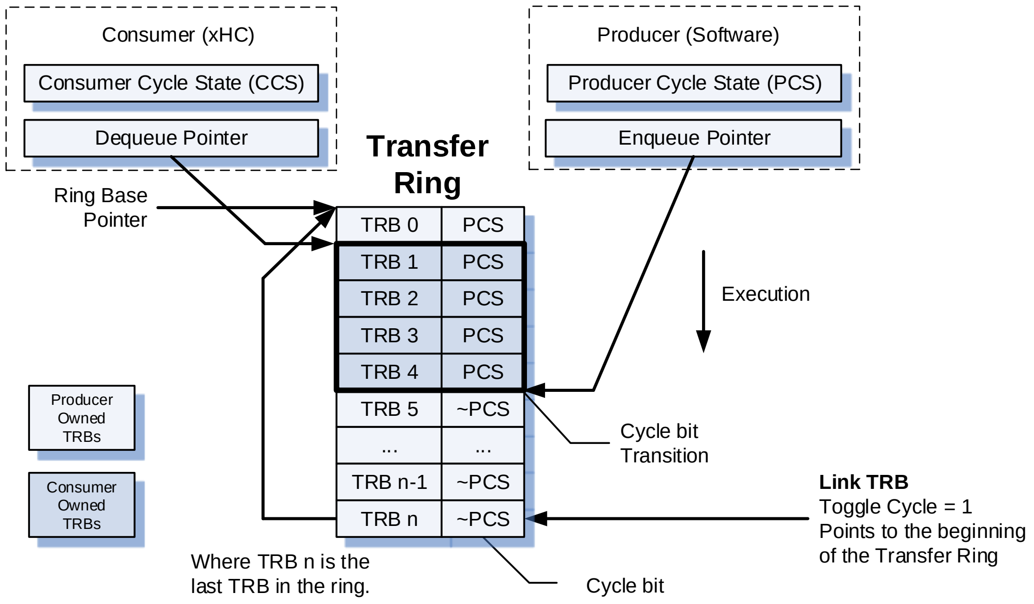
The xHC uses the Enqueue Pointer to determine when a Transfer Ring is empty. As it fetches TRBs from a Transfer Ring it checks for a Cycle bit transition. If a transition detected, the ring is empty.

Software uses the Dequeue Pointer to determine when a Transfer Ring is full. As it processes Transfer Events, it updates its copy of the Dequeue Pointer with the value of the Transfer Event *TRB Pointer* field. If advancing the Enqueue Pointer would make it equal to the Dequeue Pointer then the Transfer Ring is full and software shall wait for Transfer Events that will advance the Dequeue Pointer.

The *Enqueue Pointer* is managed by the producer and the *Dequeue Pointer* is managed by the consumer. The producer maintains a **Producer Cycle State** (PCS) flag which identifies the value that it shall write to the TRB Cycle bit. The consumer maintains a **Consumer Cycle State** (CCS) flag, which it compares to the Cycle bit in TRBs that it fetches. If the CCS flag is equal to the value of the TRB Cycle bit, then the consumer owns the TRB pointed to by the Dequeue

Pointer and may process it. If they are not equal, then the consumer shall stop processing TRBs and wait for a notification of more work.

Figure 4-6: Index Management



In Figure 4-6, TRBs are written by the producer setting the Cycle bit to the value of PCS. Note that in Figure 4-6, "~PCS" is the inverted version of PCS.

To form a ring (or circular queue) a *Link TRB* may be inserted at the end of a ring to point to the first TRB in the ring. A ring may contain multiple Link TRBs which are used to chain together Transfer Ring Segments.

In the example of Figure 4-6 the *Toggle Cycle* flag is set in the Link TRB. If the Producer encounters a *Toggle Cycle* flag set in a Link TRB it shall toggle the state of its PCS flag. If the Consumer encounters a *Toggle Cycle* flag set in a Link TRB it shall toggle the state of its CCS flag. The producer sets the TRB Cycle bit to the value of the PCS flag when it writes a TRB to set the position of the Enqueue Pointer. In Figure 4-6, the next TRB written by the producer after encountering the Link TRB will be TRB 0. The assertion of the Toggle Cycle bit in the Link TRB will cause the Producer to toggle the state of the PCS flag. The Cycle bit in TRB0 will be set to the value of PCS.

Link TRBs allow Transfer Rings to span Page boundaries and to be dynamically sized.

Note: All TRBs between the Dequeue Pointer and the Enqueue Pointer-1 are owned by the Consumer and may not be modified by the Producer. If the Ring is empty (Dequeue Pointer = Enqueue Pointer) then no TRBs are owned by the Consumer. Any TRBs in a ring not owned by the Consumer are owned by the Producer.

Note: If Streams are not enabled for an endpoint, the Transfer Ring CCS flag shall be set to the value of the Endpoint Context *DCS* flag by a *Configure Endpoint Command* if the associated *Add Context* flag is '1', or by a *Set TR Dequeue Pointer Command*.

If Streams are enabled for an endpoint, then when a Stream is selected, the CCS flag shall be set to the value of the DCS flag in the associated Stream Context, and when the Stream state is saved, the DCS flag in the associated Stream Context shall be set to the value of the CCS flag.

4.9.2.1 Segmented Rings

The Link TRB provides support for non-contiguous TRB Rings. For instance, if contiguous Pages of memory cannot be allocated by system software to form a large TRB Ring, then Link TRBs can be used to tie together multiple memory Pages to form a single large Transfer Ring.

A non-contiguous TRB Ring is composed of Ring Segments. A Ring Segment is a contiguous block of physical memory. The Link TRB provides a 64-bit pointer which points to the next segment of a ring. If the ring is comprised of only a single segment then the only Link TRB points to the beginning of the ring, as illustrated in Figure 4-6 above. A multi-segment ring will use a Link TRB to delimit the end of one Segment and the start of the next. The last TRB in a Ring Segment is always a Link TRB.

Figure 4-7: Segmented Ring Example

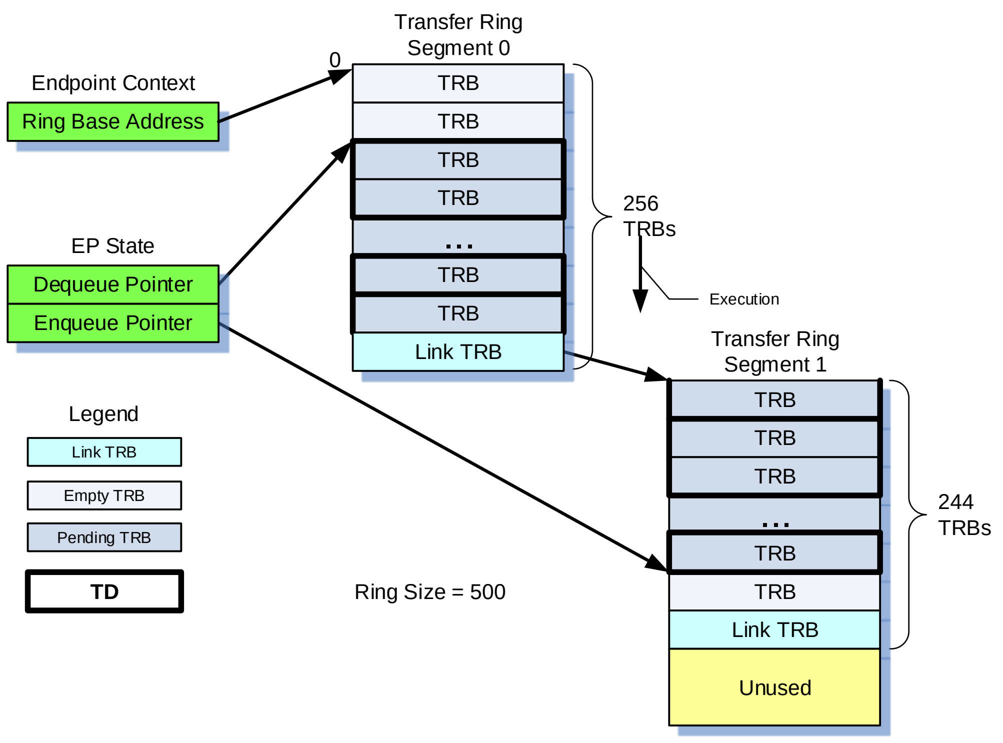


Figure 4-7 illustrates a Segmented Ring that contains two segments. In this example both segments are allocated as 4KB contiguous blocks of memory. Segment 0 defines 256 TRBs, where the last TRB is a Link TRB that points to the beginning of the next segment. Segment 1, which defines 244 TRBs, does not fully utilize the 4K buffer that was allocated for it. The two segments together define ring size of 500 total TRBs, where 498 of them are available for TDs. Note that the *Toggle Cycle* flag is set only in Segment 1's Link TRB.

4.9.2.2 Pointer Advancement

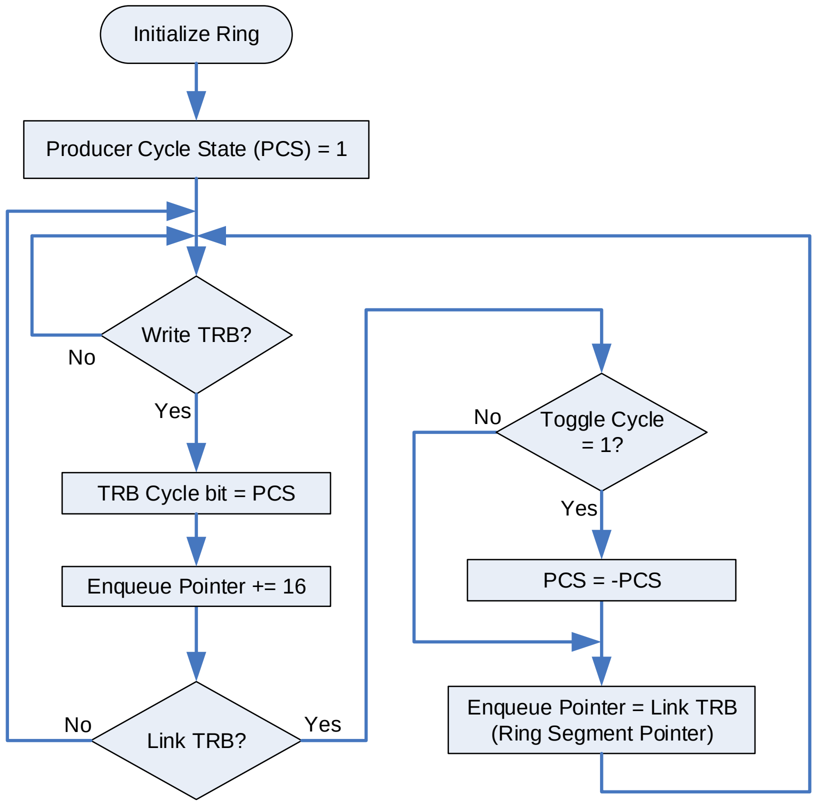
When a Dequeue Pointer is "advanced", its value is adjusted to point to the next transfer related (Isoch, Setup Stage, Normal, etc.) TRB to be executed. The xHC increments the pointer value by 16 bytes to point to the next TRB, however if the next TRB is a Link TRB and its Cycle bit indicates that it is a valid TRB, then the xHC will automatically set the Dequeue Pointer to the address provided by the Link TRB. This operation will point the Dequeue Pointer to the first TRB of the next segment.

Software is responsible for advancing the Enqueue pointer. It does this by toggling the Cycle bit each pass through the ring as it writes TRBs.

Once started (by a doorbell), the xHC processes TRBs until the ring is empty. A ring is defined as "empty" if the Dequeue Pointer is equal to the Enqueue pointer. The value of the Enqueue Pointer is defined by the Cycle bit transition.

To prevent overruns, software shall determine when the Ring is full. The ring is defined as “full” if advancing the Enqueue Pointer will make it equal to the Dequeue Pointer. Software shall take Link TRBs into account when evaluating the full condition. If the Enqueue Pointer *is not* pointing at a Link TRB, software can determine if the Ring is full by adding the size of a TRB (16) to the Enqueue Pointer and checking if the result is equal to the value of the Dequeue Pointer. If the Enqueue Pointer *is* pointing at a Link TRB, then software shall compare the Ring Segment Pointer value in the Link TRB with the Dequeue Pointer.

Figure 4-8: Enqueue Pointer Advancement



Note: The **Producer Cycle State** (PCS) and the **Consumer Cycle State** (CCS) flags are maintained internally by the xHC and software to aid in identifying the value of the Enqueue pointer. These flags are *NOT* defined in xHC registers or data structures.

The Pointer Advancement rules:

- The Cycle bit shall be initialized by software to ‘0’ in all TRBs of all segments when initializing a ring.
- The Producer Cycle State (PCS) and the Consumer Cycle State (CCS) bits shall be set to ‘1’ when a ring is initialized.

Note: The initial state of a Transfer Ring's CCS flag is determined by the Endpoint Context *DCS* flag. The initial state of the Command Ring's CCS flag is determined by the *Command Ring Control* Register *Ring Cycle State* (RCS) flag. The initial state of the Event Ring's CCS flag is always '1'. The previous two bullets assume that the *DCS* and *RCS* flags are initialized to '1' by software. If software chooses to initialize a CCS flag (*DCS* or *RCS*) to '0', the Cycle bits in the respective ring shall be set to '1'.

- The Cycle bit shall be written by the producer with the current value of the PCS bit.
- The Cycle bit shall be treated as Read-Only by the consumer.
- The Consumer may execute a TRB referenced by the Dequeue Pointer whose Cycle bit equals CCS.
- If the Enqueue Pointer references a Link TRB, then the Enqueue Pointer shall be set to Link TRB Ring Segment Pointer and if the *Toggle Cycle* bit is set to '1' in the Link TRB, the PCS bit shall be toggled by the Producer.
- If the Dequeue Pointer references a Link TRB then the Dequeue Pointer shall be set to Link TRB Ring Segment Pointer and if the *Toggle Cycle* bit is set to '1' in the Link TRB, the CCS bit shall be toggled by the Consumer.

Note: A Cycle bit transition takes place between a Link TRB and the first TRB of the segment that the Link TRB Ring Segment Pointer references.

Note: The *TR Dequeue Pointer* and Link TRB are not required to point to the beginning of a memory page.

4.9.2.3 Enlarging a Transfer Ring

To increase the size of a Transfer Ring, software shall allocate and initialize a new segment.

Software then identifies a segment boundary (Link TRB) where it will add the new segment.

Note: Only Link TRBs that are owned by the producer may be modified to point to the new segment.

Figure 4-9: Initial State of Transfer Ring

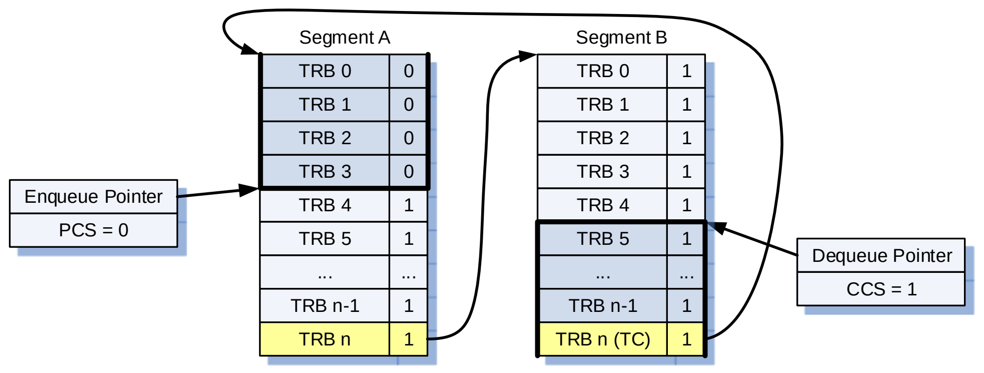
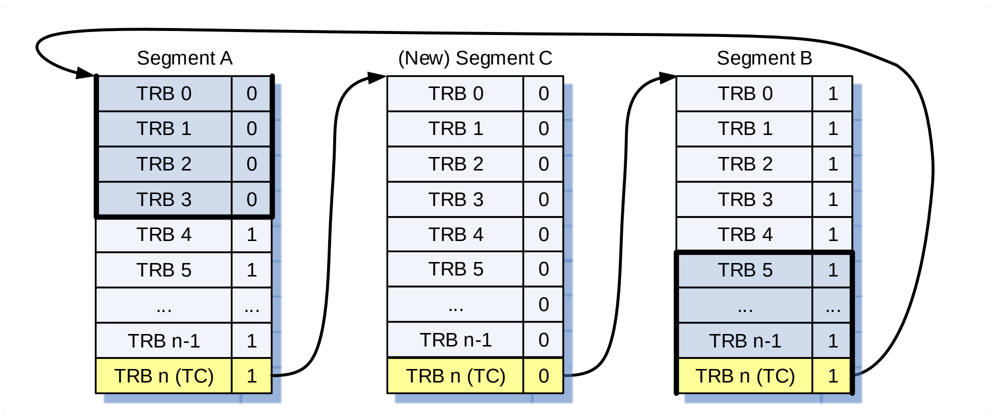


Figure 4-9 illustrates a two segment Transfer Ring (A and B) where TRBs 5 to n of Segment B and TRBs 0 to 3 of Segment A are owned by the consumer (xHC), and the remaining TRBs are available to the producer (software) for creating new TDs. Note that the *Toggle Cycle* (TC) bit is set in the Link TRB of segment B and not set in the Link TRB of segment A, hence the state of the Cycle bit is toggled once each pass through the Transfer Ring.

Now, consider the case where software needs to grow the ring size of Figure 4-9. Software may pause its insertion of TDs on the Transfer Ring, which temporarily stops the Enqueue Pointer from advancing, to insert a new segment. Software may only modify Link TRBs that it owns, so the new segment C may only be inserted between existing segments A and B as illustrated in Figure 4-10.

Note: If a Link TRB is *not* owned by software and not an "intermediate" TRB of the TD currently being executed by the xHCI, software may stop the Transfer Ring to modify the Link TRB, then restart it. If the Link TRB is an "intermediate" TRB of the TD currently being executed by the xHCI, then software shall use a *Set TR Dequeue Pointer Command* after stopping the Transfer Ring to ensure that the xHCI flushes any cached TRBs before restarting it. Refer to section 4.6.9 for more information on the requirements of stopping a Transfer Ring.

Figure 4-10: Final State of Transfer Ring



In this example software initializes the new segment with the following operations:

- All TRBs in the new segment C to ‘0’, including the Cycle bit.
- The *TRB Type* of the last TRB (n) in segment C shall be set to *Link TRB*.
- And the *Ring Segment Pointer* field of the segment C Link TRB (n) shall be initialized to point to the first TRB (0) of segment B.
- The *Toggle Cycle* (TC) flag of the segment C Link TRB (n) shall be set, to indicate the Cycle bit transition between the last TRB in segment C and first TRB in segment B.

Software then modifies segment A’s Link pointer to point to link the new Segment C into the ring.

- The *Ring Segment Pointer* field of the segment A Link TRB (n) shall be initialized to point to the first TRB (0) of segment C.
- The *Toggle Cycle* (TC) flag of the segment A Link TRB (n) shall be set to ‘1’, to indicate the Cycle bit transition between the consumer owned TRBs in segments A and C.

Software is required to ensure that the state of the Cycle bits in the new segment(s) and the *Toggle Cycle* flags in the Link TRBs that are used to connect the new segment to existing segments, do not cause an inconsistency in the definition of the Enqueue Pointer position.

Given the initial conditions illustrated in Figure 4-9, to ensure Cycle bit consistency when inserting segments software may either: 1) clear all the Cycle bits in all TRBs in the new segment(s) to ‘0’ and modify the Link TRB *Toggle Cycle* flags in the segment that points to the new segment and the new segment, or 2) set all the Cycle bits in all TRBs in the new segment to ‘1’. Figure 4-10 illustrates the case 1.

4.9.2.4 Shrinking a Transfer Ring

To decrease the size of a Transfer Ring, software shall identify a segment boundary (Link TRB) where it will perform the shrink operation.

Note: The producer shall not modify Link TRBs that it does not currently own.

Software may modify the Link TRB *Ring Segment Pointer* to map out one or more intermediate segments and/or set the Link TRB *Ring Segment Pointer* to a TRB location in the segment terminated by the Link TRB.

Software shall ensure that the state of the Cycle bits in all remaining segments do not cause an inconsistency in the definition of the Enqueue Pointer position by managing the Link TRB *Toggle Cycle* bits.

4.9.3 Command Ring Management

This section describes the operation of Enqueue and Dequeue Pointers in the Command Ring.

The operation of a Command Ring is identical to Transfer Rings with the following exceptions:

- If the *Command Ring Control Register* (CRCR) is written while the Command Ring is stopped (CRR = ‘0’) the xHC shall initialize the Command Ring Dequeue Pointer with the value of the *Command Ring Pointer* field (refer to section 5.4.5).
- When the Host Controller Doorbell Register (0) is written by system software, the xHC will evaluate the Command TRB pointed to by the Command Ring Dequeue Pointer. Once started (by a doorbell write), the xHC processes Command TRBs and advances the Command Ring Dequeue Pointer until the ring is empty.
- The location of the Command Ring Dequeue Pointer is reported on the Event Ring in Command Completion Events.
- No multi-TRB TDs are allowed on the Command Ring.

All other aspects of Command Ring management are identical to those described for the Transfer Rings. i.e.:

- Software is responsible for advancing the Enqueue pointer. It does this by toggling the Cycle bit each pass through the Command Ring as it writes Command TRBs.
- A Command Ring is defined as “empty” if the Dequeue Pointer is equal to the Enqueue pointer. The Enqueue Pointer is defined by a Cycle bit transition.

Note: Refer to the description of the CRCR *RCS* bit in Table 5-23 for information on Command Ring CCS flag initialization.

Note: While the Command Ring is in the *Running* state (*CRR* = ‘1’), it may be *Busy* (actively processing Command TRBs) or *Idle* (not processing Command TRBs and waiting for a doorbell ring), i.e. *CRR* is not negated when the Command Ring has completed all queued commands.

4.9.4 Event Ring Management

This section describes the operation of Enqueue and Dequeue Pointers in the Event Ring. The operation of Enqueue and Dequeue Pointers in Transfer Rings is described in section 4.9.2 and Command Rings in section 4.9.3. Note an xHC may implement multiple Interrupters, each with its own Event Ring. This section describes the operation of a single Event Ring.

A fundamental difference between an Event Ring and a Transfer or Command Ring is that the xHC is the producer and system software is the consumer of Event TRBs. The xHC writes Event TRBs to the Event Ring and updates the Cycle bit in the TRBs to indicate to software the current position of the Enqueue Pointer.

The xHC maintains an Event Ring *Producer Cycle State* (PCS) bit, initializing it to '1' and toggling it every time the *Event Ring Enqueue Pointer* wraps back to the beginning of the Event Ring. The value of the *PCS* bit is written to the *Cycle* bit when the xHC generates an Event TRB on the Event Ring.

Software maintains an Event Ring *Consumer Cycle State* (CCS) bit, initializing it to '1' and toggling it every time the *Event Ring Dequeue Pointer* wraps back to the beginning of the Event Ring. If the *Cycle* bit of the Event TRB pointed to by the *Event Ring Dequeue Pointer* equals *CCS*, then the Event TRB is a valid event, software processes it and advances the *Event Ring Dequeue Pointer*. If the Event TRB *Cycle* bit is not equal to *CCS*, then software stops processing Event TRBs and waits for an interrupt from the xHC for the Event Ring. When the interrupt occurs, software picks up where it left off, checking the *Cycle* bit of the Event TRB pointed to by the *Event Ring Dequeue Pointer* against its *CCS* bit.

System software shall write the *Event Ring Dequeue Pointer* (ERDP) register to inform the xHC that it has completed the processing of Event TRBs up to and including the Event TRB referenced by the ERDP.

Note: The detection of a *Cycle* bit mismatch in an Event TRB processed by software indicates the location of the xHC Event Ring Enqueue Pointer and that the Event Ring is empty. Software shall write the ERDP with the address of this TRB to indicate that it has processed all Events in the ring.

Event Ring segments are defined by an **Event Ring Segment Table** (ERST). The ERST consists of an array of Base Address/Size pairs (ERST.BaseAddress and ERST.Size), each defining a single Event Ring segment. The first element in the ERST (0) is pointed to by the **ERST Base Address Register** (ERSTBA section 5.5.2.3.2). The number of elements in the ERST is defined by the **ERST Size Register** (ERSTSZ section 5.5.2.3.1). When the xHC is initialized, it begins writing Event TRBs starting at the address referenced by the 0th ERST entry. The xHC maintains a count of the Event TRBs that it has written to a segment. When the count exceeds the value of the associated ERST.Size entry, the xHC shall fetch the next ERST entry. The ERST entries are treated as a circular queue, wrapping back to the ERST(0) after the ERST(*ERSTSZ* – 1) is fetched. Refer to section 6.5 for the definition of an ERST entry.

Figure 4-11: Segmented Event Ring Example

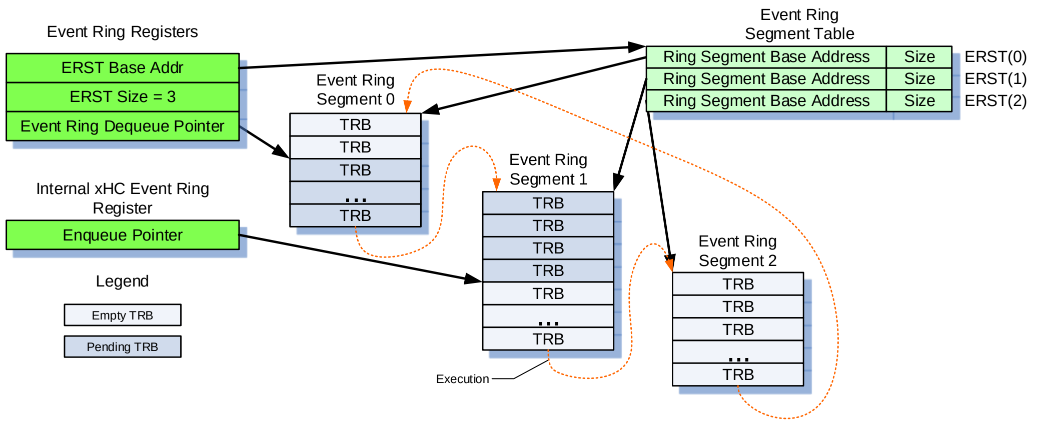


Figure 4-11 illustrates a segmented Event Ring that consists of 3 segments.

Rules for operation of an Event Ring:

- Prior to writing the *ERST Base Address* (ERSTBA) register system software shall:
    - Initialize the Event Ring Segments that will be referenced by the Event Ring Segment Table (ERST) to ‘0’.
    - Initialize the ERST by initializing the *ERST.BaseAddress* and *ERST.Size* fields of each element in the table. The *ERST.BaseAddress* field shall point to the associated Event Ring Segment, and the *ERST.Size* field shall indicate the number of TRBs supported by the segment.
    - Write the *ERST Size* (ERSTSZ) Register with the number of valid entries in the ERST and *Event Ring Dequeue Pointer* (ERDP) Register with the value of ERST(0).BaseAddress.
- Write the *ERST Base Address* (ERSTBA) register with the value of ERST(0).BaseAddress. When the ERSTBA register is written, the Event Ring State Machine (Figure 4-12) is set to the Start state.
- System software shall advance the Event Ring Dequeue Pointer by writing the address of the last processed Event TRB to the *Event Ring Dequeue Pointer* (ERDP) register. Note, the “last processed Event TRB” includes the case where software detects a Cycle bit mismatch when evaluating an Event TRB and the ring is empty.
- System software is responsible for ensuring valid values for ERST entries in paged environments.
- System software is responsible for ensuring the Size of every ERST entry (Event Ring segment) is at least 16.

Figure 4-12: Event Ring State Machine

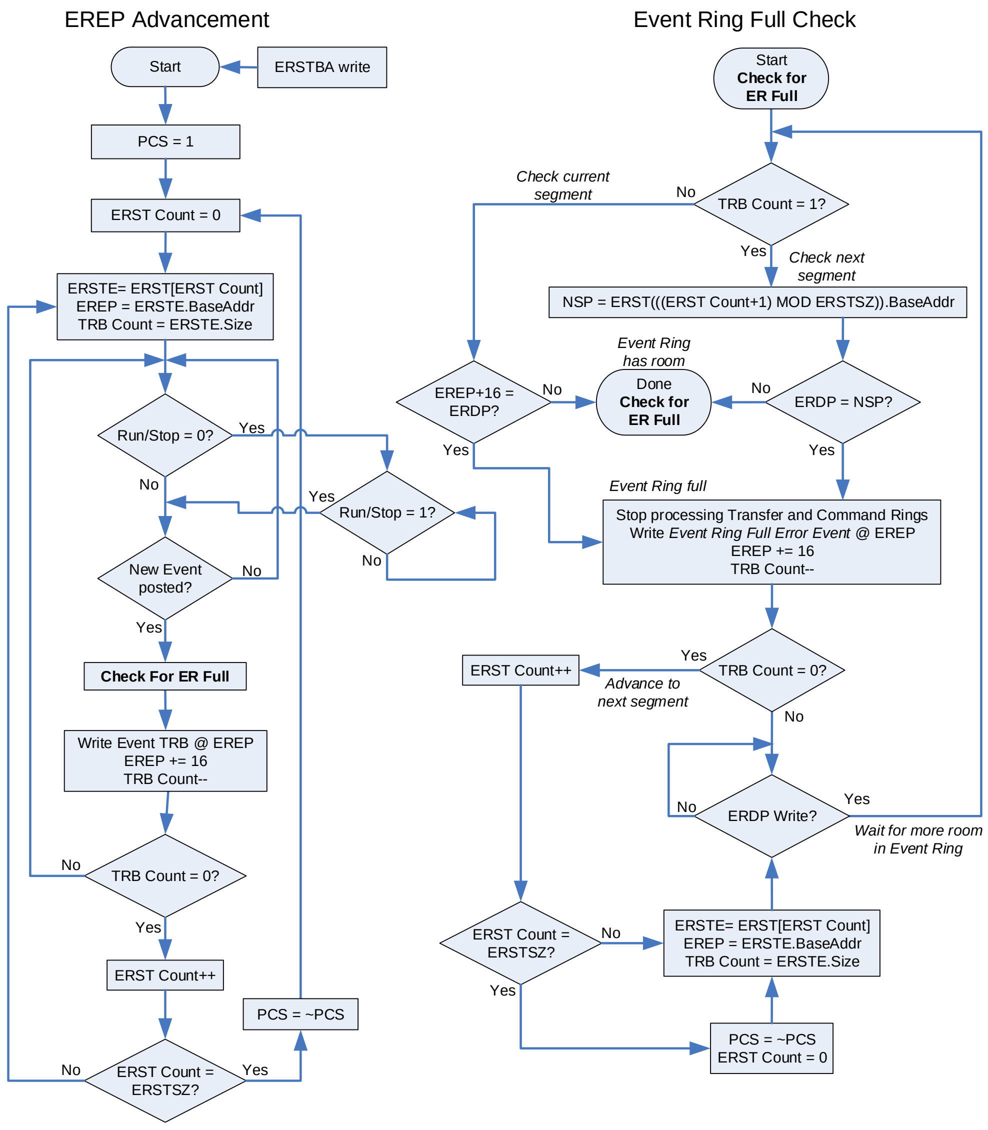


Figure 4-12 describes the algorithm the xHC employs for advancing its internal *Event Ring Enqueue Pointer* (EREP). The left side of the figure describes the EREP Advancement algorithm. The right side of the figure describes the algorithm for checking if the Event Ring is full.

Note: The *Producer Cycle State* (PCS) flag for the Event Ring is toggled **only** when the Event Ring wraps back to the beginning.

Note: The Event Ring State machine is Stopped if the USBCMD *Run/Stop* (R/S) flag is ‘0’.

Note: A blocked Event Ring may impact forward progress on endpoints whose TDs target other Event Rings.

Note: It is recommended that software process as many Events as possible before writing the ERDP. This approach not only minimizes the number of MMIO writes, but is particularly important if the Event Ring is full. If an Event Ring Full condition exists, writing the ERDP after processing individual Events may cause no work to progress because the Event Ring becomes filled with Event Ring Full Events.

Ideally, software writes the ERDP after processing all Events on an Event Ring. Practically, software should maximize the number of Events processed before writing the ERDP, e.g. processing a minimum of 4 Events before each ERDP write.

Note: Section 4.23.2 describes the xHC Restore process. Step 2 in the restore process requires software to load all registers (including the ERSTBA) with previously saved values. Writing the ERSTBA initializes the Event Ring State Machine internal variables and advances it to wait for *Run/Stop* (R/S) to be asserted or an event to be posted. A Restore operation, which always follows the register load by software, shall overwrite the Event Ring State Machine internal variables (ERSTE, ERST Count, EREP, and TRB Count) with previously saved values, allowing the Event Ring State Machine to "pick up where it left off" after a power event.

Note: Software writes to the ERDP register shall always advance the Event Ring Dequeue Pointer value, i.e. software shall not write the same value to the ERDP register on two consecutive write operations.

Table 4-3: Event Ring State Machine Definitions

Name	Label	Description
Event Ring Segment Table	ERST	Resides in host memory. Contains the addresses and lengths of the Event Ring segments. Refer to section 6.5.
Event Ring Dequeue Pointer	ERDP	Resides in Runtime register space. Advanced by software. Refer to section 5.5.2.3.3.
Event Ring Enqueue Pointer	EREP	Internal xHC variable. Advanced by Figure 4-12 algorithm
Event Ring Segment Table Count	ERST Count	Internal xHC variable. Identifies the offset into the ERST of the segment that is currently being filled with Event TRBs by the xHC.
Event Ring Segment Table Entry	ERSTE	Internal xHC variable. A pointer to an ERST entry.
Event Ring Segment Table Base Address	ERSTE.BaseAddr	*Ring Segment Base Address* field of current ERST entry.

Event Ring Segment Size	ERSTE.Size	*Segment Size* field of current ERST entry.
Event Ring Segment Table Size	ERSTSZ	Number of entries in the in the ERST.
Next Segment Pointer	NSP	Base address for next Segment of ERST, based on the current EREP.
TRB Count	TRB Count	Internal xHC variable. Identifies the number of remaining TRBs in the current segment.

The following steps describe the xHC Event Ring Enqueue Pointer (EREP) Advancement algorithm (left side of Figure 4-12):

1. When the *ERST Base Address* (ERSTBA) register is initially written the Event Ring State Machine enters the Start state.

2. The xHC initializes its internal PCS flag to '1'.

3. The xHC sets its internal ERST Count to '0'.

4. The xHC then fetches the entry in the Event Ring Segment Table referenced by the ERST Count (ERSTE = ERST[ERST Count]) and initializes its Enqueue Pointer (EREP) with the value of the *Ring Segment Base Address* field (ERSTE.BaseAddr), and the TRB Count with the value of the *Segment Size* field (ERSTE.Size).

5. If the USBCMD *Run/Stop* (R/S) flag = '0' the Event Ring State Machine shall wait for *Run/Stop* (R/S) to return to '1'[27]. When *Run/Stop* (R/S) flag = '1' the xHC shall proceeds to check if an event is posted (step 6., otherwise it proceeds immediately to step 6.

6. When an event is posted for the ring, the xHC shall first check if the ring is full. If not, the xHC writes the Event TRB to the location identified by the EREP, increments the EREP by 16, and decrements the TRB Count. The *Cycle* bit of the Event TRB is set to the value of the *PCS* flag. If no event is posted, the xHC will return to step 5.

7. As long as the TRB Count is non-zero, the xHC shall return to step 5, continuing to check *Run/Stop* (R/S) or for new events.

[27] A *Controller Restore State* (CRS) operation overwrites the Event Ring State Machine internal variables. This may occur while waiting for *Run/Stop* (RS) to be set to '1' when restoring state from a power event. Refer to section 4.23.2.

8. When the TRB Count reaches ‘0’, the xHC shall increment the ERST Count and evaluate it, otherwise it returns to step 5.

    a. If the ERST Count is not equal to the value of the ERSTSZ register, then the xHC returns to step 4 to process events starting in the next segment of the ERST.

    b. If the ERST Count equals the value of the ERSTSZ register, then the xHC sets the ERST Count to ‘0’, toggles the *Producer Cycle State* (PCS) flag, and return to step 3 to process events starting in the first segment of the ERST.

If the Event Ring is full, the xHC shall flag the condition by reporting an *Event Ring Full Error*, which requires placing an Event on the Event Ring. To ensure that there is space on the Event Ring for this error, the xHC shall consider the Event Ring full when there is still room for one more entry.

The following steps describe the xHC algorithm for checking if the Event Ring is full (right side of Figure 4-12):

1. If the TRB Count is greater than ‘1’, then the xHC can simply add 16 to the EREP and compare it to the ERDP to determine whether the Event Ring is full.

2. If the TRB Count is equal to ‘1’, then the xHC shall check if the ERDP points to the first entry in the next segment. To obtain the base address for the next segment the xHC retrieves the ERST.BaseAddress entry for the ERST Count + 1 modulus the ERSTSZ. Then calculates the address of the next Event Ring segment (NSP).

    a. If the NSP does not equal the ERDP, then the Event Ring has room and the Event Ring Full Check exits.

    b. If the NSP equals the ERDP, then the Event Ring is full. The xHC stops processing the Transfer and Command Rings, writes a *Event Ring Full Error* Event to the EREP, advances the EREP and decrements the TRB Count. Refer to Step 2b note below.

3. If the TRB Count is not equal ‘0’, then there is room in the current segment for more events so go to step 6 and wait for the ERDP to advance.

4. If the TRB Count is equal ‘0’, then increment the ERST Count to advance the EREP to the next segment.

    a. If the ERST Count is not equal to the value of the ERSTSZ register, then the xHC goes to step 5 to initialize the state machine parameters for the next segment of the ERST.

b. If the ERST Count equals the value of the ERSTSZ register, then advance the EREP to the first segment of the ERST by setting the ERST Count to '0' and toggling the *Producer Cycle State* (PCS) flag, then go to step 5 to initialize the state machine parameters for the first segment of the ERST.

5. To initialize the state machine parameters, the xHC fetches the entry in the Event Ring Segment Table referenced by the ERST Count (ERSTE = ERST[ERST Count]) and initializes its Enqueue Pointer (EREP) with the value of the *Ring Segment Base Address* field (ERSTE.BaseAddr) and the TRB Count with the value of the *Segment Size* field (ERSTE.Size). Once the EREP has been advanced to the next segment go to step 6 and wait for the ERDP to advance.

6. The Event Ring will remain full until the next time that software writes the ERDP. When the ERDP is written, the xHC will determine if the new ERDP value has freed space on the Event Ring by returning to step 1).

Note: The expectation is that the xHC shall *gracefully* stop execution on the Command and Transfer Rings when the Event Ring is full. An "Event Ring Stop" will propagate all the way to the USB when all the buffered operations in the xHC are exhausted. The xHC is expected to not lose Control, Interrupt, or Bulk data under these conditions, however if the condition persists, the xHC will begin to miss periodic endpoint Service Opportunities (SOs), resulting in the loss of Isoch data and the possible loss of Interrupt data. The *Missed Service Error* may be used to report this condition in an Isoch Transfer Event once the Event Ring Stop condition is cleared. The *Event Ring Full Error* shall be reported whether data is lost or not, to inform system software that the Event Ring is under provisioned.

Note: Step 22.b above states that "the xHC stops processing the Transfer and Command Rings" if an Event Ring is full. This action is further qualified with the type of Event Ring that has gone full. If the Primary Event Ring is full, then all command and transfer rings shall stop processing TRBs. If a Secondary Event Ring becomes full, then the xHC may stop all command and transfer ring processing, or only stop processing on those transfer rings that target the full Event Ring. If virtualization is enabled, an xHC implementation shall ensure that a full condition on a Secondary Event Ring does *not* stop the processing of TRBs on the Command Ring, the Primary Event Ring, or other Secondary Event Rings.

4.9.4.1 Changing the size of an Event Ring

To increase the size of an Event Ring, software shall allocate and initialize a new segment.

Software then initializes ERST entries, starting at the offset defined by ERSTSZ, with the Address and Size of the new Event Ring segment(s) and writes new size of the ERST to the *ERSTSZ* Register.

Software may determine when the xHC has started using the new segment by evaluating the Completion Code of the first TRB in the new segment for a non-zero (valid) condition.

Consider the case were there the 2 segments ‘0’ and ‘1’ (ERSTSZ = 2, ERST(0) and ERST(1)) are active, and a *new* segment ‘2’ is being added. Software initializes all TRBs in the new segment to ‘0’. Then sets the ERST(2).BaseAddr equal to the base address of the new segment, the ERST(2).Size equal to the number of Event TRBs supported by the new segment, and the *ERSTSZ* to 3.

If the EREP just passed the end of segment 1 when the *ERSTSZ* was written, the xHC will not start using the new segment until the next pass through the Event Ring. If the EREP is positioned at the last TRB of segment ‘1’ when the *ERSTSZ* was written, the xHC will start using the new segment.

Note that the xHC will write the Cycle bit in the segment 2 TRBs with the same value as it had been using for segment 1. Software may determine when the xHC started using the new segment as it is evaluating Event TRBs pointed to by the Dequeue Pointer. When software evaluates the Event TRB after the last TRB of segment 1, it shall check for a *Valid* (non-zero) Completion Code in the first TRB of segment 2 as an indicator that the xHC has started using the new segment. If the Completion Code is Valid, then software shall advance the Dequeue Pointer to the first TRB of segment 2. If the Completion Code is Invalid (‘0’) value, software shall check the state of the Cycle bit in the first TRB of segment 0 to see whether it matches the expected state for the next pass through the Event Ring. If it does not match, it means that the EREP is pointing at the last TRB of segment 1 and the Event Ring is empty. If it does match, then software shall advance the Dequeue Pointer to the first TRB of segment ‘0’. If the Event Ring is empty, software shall reevaluate direction of the EREP at the segment 1 to segment 2 boundary the next time it receives an interrupt.

The *Valid* (non-zero) to *Invalid* (‘0’) transition of the Event TRB Completion Code field shall be used by software to determine the position of the Enqueue Pointer during the first pass of the Dequeue Pointer through the new segment(s). The TRB Cycle bit field shall be treated as invalid during the first pass through the new segment(s) and shall *not* be used by software to determine the position of the Enqueue Pointer.

After the first pass of the Enqueue Pointer through the new segment(s), the xHC has initialized the Cycle bit in all newly added Event TRBs.

After the first pass of the Dequeue Pointer through the new segment(s), software shall evaluate the Cycle bit state in segment 2 to determine the Enqueue Pointer position.

Note: ERST entries (Segment Base Address and Size fields) between 0 and ERSTSZ-1 are not allowed to be modified by software when *HCHalted* (HCH) = ‘0’.

4.9.4.2 Shrinking an Event Ring

To decrease the size of an Event Ring, software shall decrement value of the *ERSTSZ* Register.

Software may determine when the xHC has stopped using the segment that is to be removed by evaluating the state of the Cycle bit of the first TRB in the deleted segment(s).

Consider the case where there are 3 segments 0, 1, and 2 (ERST Count = 3) and segment 2 is being deleted. Software writes the *ERSTSZ* register, setting it to 2. If the EREP is pointing into segment 2 when the *ERSTSZ* was written, the xHC will not stop using the "deleted" segment until the next pass through the Event Ring. If the EREP is positioned at the last TRB of segment 1 when the *ERSTSZ* was written, the xHC will stop using the new segment immediately.

Software may determine when the xHC stopped using the "deleted" segment as it is evaluating Event TRBs pointed to by the Dequeue Pointer. When software evaluates the Event TRB after the last TRB of segment 1, it may check the Cycle bit of the first TRB in segment 2. If the Cycle bit state matches the expected state then it shall continue processing the Event TRBs in the deleted segment. If the Cycle bit state of the first TRB in segment 2 does not match the expected state, then software shall check the state of the first TRB in segment 0. If the Cycle bit in the first TRB in segment 0 matches the state of the last TRB in segment 1, then the EREP is pointing at the last TRB of segment '1' and the Event Ring is empty. If it does not match, then the EREP has advanced to segment 0 and the next Event TRB to process is the first TRB of segment 0, and the xHC has stopped using the deleted segment. If the Event Ring is empty, software shall reevaluate direction of the EREP at the segment '1' to segment '2' boundary the next time it receives an interrupt.

4.9.4.3 Primary and Secondary Event Rings

The number of Interrupters available to software is defined by the *MaxIntrs* field in the HCSPARAMS1 register. If more than one Interrupter is available then the 0th Interrupter is referred to as the **Primary Interrupter** and all other Interrupters are referred to as the **Secondary Interrupters**. Each Interrupter defines an associated Event Ring. The Event Ring associated with the 0th Interrupter is referred to as the **Primary Event Ring**. The Event Rings associated with the other Interrupters are referred to as the **Secondary Event Rings**. The only Event TRB types that may be found on a Secondary Event Ring are:

- Transfer Event
- Bandwidth Request Event
- Device Notification Event
- Host Controller Event

- Vendor defined event (optional)

Transfer Events generated by a Device Slot may be directed to a Secondary Event Ring by a non-'0' value in the Transfer TRB *Interrupter Target* field. All Transfer Events with the TRB *Interrupter Target* field cleared to '0', shall be directed to the Primary Event Ring by the xHC.

Bandwidth Request and Device Notification Events are targeted at a Device Slot. The xHC shall use the Device Slot's Slot Context *Interrupter Target* field to determine the Event Ring that shall receive the event.

4.10 Host Controller TRB Handling

4.10.1 Transfer TRBs

A fully configured host controller can support 255 USB Devices, where each device can declare up to 31 endpoints. 30 of the endpoints may declare up to 64K Streams each. This means that approximately 500M Transfer Rings may exist for a single xHC. Of course this is a worst case value; however the xHC architecture shall cope efficiently with reporting the completion status of hundreds, or possibly thousands, of Transfer Rings. Transfer Ring completions are queued on Event Rings as Transfer Event TRBs for the host. Refer to section 4.11.3.1 for more information on Transfer Event TRBs.

When the data transfer associated with a Transfer TRB is completed, the xHC will evaluate the completion status of the transfer and the Transfer TRB flags to determine whether to generate a *Transfer Event TRB* for the *Transfer TRB*.

If upon transfer completion of a TRB the *Interrupt On Completion* (IOC) flag is set, the xHC shall generate a *Transfer Event TRB*. Note the generation of an Event TRB always generates an interrupt to the host. The Completion Code and Length fields of the Transfer Event TRB will reflect the completion status of the Transfer TRB that generated the event.

The detection of a USB Short Packet (i.e. the actual number of bytes received was less than the expected number of bytes defined by the Transfer TRB) during a transfer does not necessarily generate an Event. A Short Packet will trigger the generation of a Transfer Event TRB on the Event Ring if the *Interrupt-on-Short* (ISP) or *Interrupt On Completion* (IOC) flags are set in the TRB that the Short Packet was detected on. The Completion Code field of the Transfer Event shall be set to *Short Packet*. The *Length* field of the Transfer Event shall be set to the residual number of bytes not written to the Transfer TRBs' data buffer. A Short Packet may occur on an intermediate TRB of a TD. In this case the xHC shall advance to the first TRB of the next TD after completing the transfer.

Note: The xHC shall execute the first *Event Data TRB* encountered while advancing to the end of the Short Packet TD.

The detection of an error during a transfer shall always generate a Transfer Event, irrespective of whether the *Interrupt-on-Short* or *Interrupt On Completion* (IOC) flags are set in the Transfer TRB. The Completion Code of the Transfer Event shall identify the detected error condition. If a *Missed Service Error* occurs on an intermediate TRB of a TD of an Isoch endpoint the xHC shall advance to the first TRB of the next TD or the Enqueue Pointer (i.e.Cycle bit transition), whichever is encountered first, when continuing execution on the Transfer Ring. When an error condition is encountered which requires an endpoint to halt; the xHC shall stop on the TRB in error, the endpoint shall be halted, and software shall use a *Set TR Dequeue Pointer Command* to advance the Transfer Ring to the next TD.

Note: If the xHC encounters a Cycle bit transition and is unable to advance to a TD boundary when it encounters an error, it shall advance to the next TD boundary the next time the doorbell is rung. The only exception is if a *Set TR Dequeue Pointer Command* is issued before the doorbell is rung, modifying the Dequeue Pointer. In this case the xHC shall assume that the modified Dequeue Pointer references the first TRB of a TD.

A Transfer Event TRB identifies the location of the TRB that "generated the event" (the Device ID, Endpoint ID, and address of the source TRB). The *Completion Code* field of the Transfer Event TRB shall contain the originating TRBs' completion status. The location information in the Transfer Event TRB allows system software to identify the device, endpoint, and TRB that generated the event. The location information also allows the host to update its copy of the Dequeue Pointer for the Transfer Ring that generated the event.

If interrupts to the host are enabled, Interrupt Moderation (refer to section 4.17) is used to gracefully manage bursts of Transfer Events.

A host controller implementation may delay the generation of Events associated with Transfer TRBs. The following conditions should force Transfer Event generation to take place immediately:

- The completion of a TRB that has its IOC flag set.
- The completion of a Short Packet on a TRB that has its ISP flag set.
- An error occurs on any Transfer TRB.
- An xHC implementation dependent threshold, designed to prevent the TRB Ring state from getting too far behind, is reached.

Note: The TRB Pointer field in a Transfer Event TRBs not only references the TRB that generated the event, but it also provides system software with the latest value of the xHC Dequeue Pointer for the Transfer Ring. Software may choose to use Event Data TRBs exclusively to report TD completions (e.g. never setting an IOC flag in the Transfer TRBs of TDs). However, to keep the software copy of the Transfer Ring Dequeue Pointer current, software will occasionally have to set the *IOC* flag in a Transfer TRB, except if an Event Data TRB is declared. The frequency with which the IOC flag is set in Transfer TRBs will depend on many system and software factors, that are outside the scope of this specification.

Note: System software should not generate unnecessary Events. Typically there is no need to set the IOC flag in more than one Transfer TRB per TD. The only exceptions would be for 1) very large TDs (e.g. > 16MB transfers) where *Intermediate Event Data TRBs* are declared, or 2) if the IOC flag is set to refresh the software Dequeue Pointer value.

Note: An *Event Lost Error* shall be generated for the endpoint if the xHC is unable to generate all the Events defined by a TD. An *Event Lost Error* shall halt the endpoint. By following the recommendations in the notes above, this condition may be avoided. The conditions that generate this error are xHC implementation specific.

4.10.1.1 Short Transfers

The **TD Transfer Size** is defined by the sum of the Length fields in all TRBs that comprise the TD. On an IN endpoint the xHC shall schedule ((*TD Transfer Size* - 1) / *Max Packet Size*) + 1 USB packets for each TD.

If the TD Transfer Size is larger than *Max Packet Size*, all USB packets shall be *Max Packet Size* except for the last packet, which shall be sized to contain the remaining TD data.

A **Short Packet** condition shall occur if the number of bytes received for a USB packet associated with a TD is less than the number of bytes expected.

4.10.1.1.1 Short Transfers when using Event Data TRBs

When a Short Packet condition occurs and Event Data TRBs are being used, the xHC shall perform the following operations:

- If the *Interrupt-on Short Packet* (ISP) or if the *Interrupt On Completion* (IOC) flag is set to '1' in the TRB that the Short Packet condition occurred on, a Transfer Event shall be generated for that TRB with the Completion Code set to *Short Packet*.
- Automatically advance the Dequeue Pointer for the Transfer Ring to the beginning of the next TD.
    - When an *Event Data TRB* is encountered in the process of advancing the Dequeue Pointer from the Short Packet TRB to the beginning of the next TD, the xHC shall parse the Event Data TRB, i.e. if the *IOC* flag is set in the Event Data TRB, an Event Data Transfer Event shall be generated with the *Completion Code* set to *Short Packet* and the *Length* field set to the actual number of bytes received by the TD.
        - If subsequent *Event Data TRBs* are encountered in the process of advancing the Dequeue Pointer from the first Event Data TRB encountered to the beginning of the next TD, the xHC shall parse them if the *Parse All Event Data* (PAE) flag is set ('1'), and shall not parse them if the *PAE* flag is cleared ('0').

Refer to section 5.3.6 for more information on *PAE*.

- If a *Link TRB* is encountered, the xHC shall parse the *Link TRB* and if its *IOC* flag is set ('1'), then a *Transfer Event* shall be generated with its *Completion Code* set to *Success*. All Link TRBs encountered in TD shall be parsed.

If a Short Packet condition does not occur while receiving the data for a TD, the xHC shall parse all TRBs of the TD. i.e. any TRB with its *IOC* flag shall generate a Transfer Event.

Note: A USB packet may be comprised of the data from many TRBs, or many USB packets may be required to transfer a single TRB.

Note: No relationship is assumed between USB packet boundaries and TRB data buffer boundaries.

When a Short Packet condition occurs and Event Data TRBs are being used, the xHC shall perform the following operations:

Software shall perform the following operations when using Event Data TRBs to flag the completion of a TD that may receive a Short Packet, then:

- The *ISP* and *IOC* flags shall be cleared ('0') in all Transfer TRBs.
- The *IOC* shall be set ('1') in all Event Data TRB(s).

Event Data Transfer TRBs encountered prior to the occurrence of a Short Packet shall generate an Event Data Transfer Event with its *Completion Code* = *Success* (assuming no errors) and *TRB Transfer Length* field equal to the number of bytes transferred since the beginning of the TD or the previous Event Data Transfer TRB of the TD.

If a Short Packet occurs and the *PAE* flag is set ('1'), then all subsequent Event Data Transfer TRBs encountered while advancing to the end of the TD shall generate an Event Data Transfer Event with its *Completion Code* = *Short Packet* and should set the *TRB Transfer Length* field equal to the number of bytes transferred since the beginning of the TD or the previous Event Data Transfer TRB of the TD. If a Short Packet occurs and the *PAE* flag is cleared ('0'), then subsequent Event Data Transfer TRBs encountered while advancing to the end of the TD shall not generate Event Data Transfer Events.

If a Short Packet does not occur, then the last Event Data Transfer TRB shall generate an Event Data Transfer Event with its *Completion Code* = *Success* (assuming no errors) and *TRB Transfer Length* field equal to the number of bytes transferred since the beginning of the TD (i.e. EDTLA) or the previous Event Data Transfer TRB of the TD. Refer to section 4.11.5.2 for more information on Event Data TRB usage.

- If a TD on an IN endpoint is terminated with an *Event Data TRB*, there is no need to set the *ISP* flag in every TRB of the TD because the length of the transfer (including the terminating Short Packet) shall be reported by the *TRB Transfer Length* field of the *Event Data TRB*.

- Software shall not interpret an Short Packet Event Data Transfer Event as indicating that the TD that it is associated with is “complete”, unless the Event Data Transfer Event is the last TRB of the TD.

4.10.1.1.2 Short Transfers when not using Event Data TRBs

If software is *not* using Event Data TRBs, but it wants to flag the completion of a TD that may receive a Short Packet, then:

- The *ISP* flag shall be set (‘1’) in all Transfer TRBs of the TD, and
- The *IOC* flag shall be set (‘1’) in the last Transfer TRB of the TD.

If a Short Packet occurs, then a Transfer Event shall be generated with the *Completion Code* = *Short Packet*, its *TRB Pointer* field pointing to the Transfer TRB that the Short Packet occurred on, and its *TRB Transfer Length* field shall indicate the residue bytes in the buffer.

If a Short Packet does not occur, then the last TRB of the TD shall generate a Transfer Event with its *Completion Code* = *Success* (assuming there was no error), its *TRB Pointer* field pointing to the last Transfer TRB, and the *TRB Transfer Length* field shall equal 0.

If the Short Packet occurred while processing a Transfer TRB with only an *ISP* flag set, then two events shall be generated for the transfer; one for the Transfer TRB that the Short Packet occurred on, and a second for the last TRB with the *IOC* flag set. In the second event, the *Completion Code* shall be set to *Short Packet*, and the *TRB Transfer Length* should be set to the same value that was reported by the initial Short Packet Event.

Software shall not interpret a Short Packet Event as indicating that the TD that it is associated with is “complete”, unless the *TRB Pointer* field of the Transfer Event references the last TRB of the TD.

If Event Data TRBs are not used, then the total number of received bytes for a Short Packet TD is the sum of the *TRB Transfer Length* fields in all Transfer TRBs up to and including the one that generated the Short Packet Event, minus the residue value of the *TRB Transfer Length* field in the Short Packet Event.

Note: Typically an *IOC* flag is only set in the last TRB of a TD, and the event that is generated by the TRB is referred to as the "TD Completion Event", i.e. the Event that completes the TD. Also note that due to errors or Short Packet conditions, the TD Completion Event may not occur on the last TRB of a TD. And for Transfer Ring management or other reasons, software may set the *IOC* flag in any TRB of a TD, including a TD that is configured to handle Short Packets (i.e. with the *ISP* flag set in one or more TRBs). Because of this the xHC must handle the generation of multiple Events for a single TD, and those events may occur before and after the "TD Completion Event".

TD Completion Events are generated by few basic conditions:

- If the *IOC* flag = ‘1’ and the TD completes successfully, then a Transfer Event shall be generated with its *Completion Code* = *Success* and *TRB Transfer Length* identifying the number bytes transferred.
- If a Short Packet occurs and the *ISP* or *IOC* flags are set, then a Transfer Event shall be generated with set its *Completion Code* = *Short Packet* and *TRB Transfer Length* identifying the number bytes transferred.
- If an error condition is detected while processing any TRB within a TD, an Event shall be forced for that TRB (irrespective of whether the *IOC* or *ISP* flags are set on the TRB) with the *Condition Code* indicating the error that occured and the *TRB Transfer Length* indicating the number of bytes that were successfully transferred.

Events generated for a TD by TRBs encountered before the TD Completion Event shall set their Completion Code to Success. Where Data Transfer Events (*ED* = '0') shall set their *TRB Transfer Length* to 0 (since the TRB Transfer Length field represents the "residue" of a transfer and all the bytes the buffer referenced by the TRB were successfully moved), and Event Data Transfer Events (*ED* = '1') shall set their *TRB Transfer Length* to the current value of the EDTLA and then reset the EDTLA to zero.

After the TD Completion Event, if any subsequent Transfer TRBs are encountered with their *IOC* flag set while advancing to the end of the TD then those TRBs shall also generate an Event, where the *Completion Code* field shall return the same value as TD Completion Event and *TRB Transfer Length* field should return the same value as TD Completion Event. There are a couple of exceptions to this rule:

- If the *IOC* flag is set in an Event Data TRB then an Event Data Transfer Event shall be generated only if *PAE* = '1'.
- If the *IOC* flag is set in a Link TRB then the Transfer Event shall be generated with *Completion Code* = Success, and the *TRB Transfer Count* = 0.

Note: Setting the *IOC* flag in a TRB always forces an Event for that TRB (whether a Short Packet condition occurs or not), therefore also setting the *ISP* flag in the same TRB is redundant (but allowed).

4.10.2 Errors

The detection of an error during a USB transfer shall always generate a Transfer Event, irrespective of whether the *Interrupt-on-Short Packet* (ISP) or *Interrupt On Completion* (IOC) flags are set in the Transfer TRB. The Completion Code of the Transfer Event shall identify the detected error condition. An error may occur on any TRB of a TD.

All Transfer Ring error conditions force the state of the associated endpoint to *Halted* and require system software intervention to recover.

Refer to section 4.11.2.2 for more information on Control Endpoint error handling.

An isoch endpoint *never* halts because there is no handshake to report a halt condition. Errors are reported as a completion code associated with a TRB for an isochronous transfer, but an isoch pipe is not halted in an error case. If an error is detected, the xHC shall continue to process the data associated with the next ESIT of the transfer. Only limited error detection is possible because the protocol for isochronous transactions do not provide per-transaction handshakes. Refer to section 5.6.5 of the USB2 spec. There is no equivalent text in the USB3 spec, however SuperSpeed isoch endpoints are treated the same way.

4.10.2.1 Stall Error

A STALL PID (USB2) or STALL LMP (USB3) may be returned by a USB function in response to an IN token or after the data phase of an OUT or in response to a PING transaction. The STALL PID indicates that a function is unable to transmit or receive data, or that a control pipe request is not supported. The state of a USB device after returning a STALL for any endpoint (except the Default Control Endpoint) is undefined. The host controller shall not return a STALL under any condition.

When a STALL PID is received from a USB device by the xHC, it shall stop further activity on the associated Transfer Ring by removing it from its Pipe Schedule, set the associated *Endpoint State* (EP State) field to *Halted*, and generate a *Transfer Event TRB* with a *Stall Error*.

Note: If a device responds to a SETUP packet with a STALL[28] the endpoint shall generate a *Stall Error* for the Setup TRB and shall be halted.

A two step process is required to recover a halted endpoint:

1. System software shall use a *Reset Endpoint Command* (section 4.11.4.7) to remove the *Halted* condition in the xHC. After the successful completion of the *Reset Endpoint Command*, the Endpoint Context is transitioned from the *Halted* to the *Stopped* state and the Transfer Ring of the endpoint is reenabled. The next write to the Doorbell of the

[28]Typically control endpoints only return STALL TPs due to a Protocol Stall condition (as described in the USB3 spec section 8.12.2.3), however section 8.1 of the USB3 spec states "For non-isochronous transfers, an endpoint may respond to valid transactions by:... Returning a STALL Transaction Packet if there is an internal endpoint error". This condition describes a "Functional Stall" case, which applies to a SuperSpeed Control Endpoint if an internal endpoint error is detected by the device, hence any TP or DP issued to a Control Endpoint may return a STALL TP, including a Setup DP.

Endpoint will transition the Endpoint Context from the *Stopped* to the *Running* state.

Note: The *Reset Endpoint Command* for the endpoint shall complete successfully *and* the halt condition on the USB device shall be successfully cleared before attempting to restart the Transfer Ring by ringing its doorbell.

2. Software intervention is required to recover the pipe within the USB device.

4.10.2.1.1 Non-Control Endpoints

Removal of the halt condition on an interrupt or bulk pipe in a USB device is achieved via software intervention through a separate control pipe.

Note: The software intervention required to remove the halt condition on the USB device shall be invoked after the pipe has been transitioned to the *Stopped* state by a successful *Reset Endpoint Command*, but before writing to the *Doorbell* register of the Endpoint to restart activity on the pipe.

Note: Since an Isoch endpoint does not generate a transaction handshake, they cannot generate a Stall Error.

4.10.2.1.2 Control Endpoints

Removal of the halt condition on the Control endpoint of a USB device is achieved by the device accepting the next SETUP PID.

For Control endpoints, a reset of the USB device shall be required to clear the halt or error condition if the device does not accept the next Setup PID.

Refer to section 4.11.2.2 for additional Control Endpoint error handling.

4.10.2.2 TRB Error

A *TRB Error* indicates the TRB field values are out of range or that the xHC has determined that a TRB is incorrectly formed.

This error condition may be reported in a Transfer Event or a Command Completion Event due to an error detected on a Transfer or Command TRB, respectively. This error will not be reported in any other Event TRB types.

Note: A Transfer Ring *TRB Error* should transition an endpoint to the *Error* state (refer to section 4.8.3), however an xHC implementation may assert *HCE*[29] due to the

[29]A *TRB Error* is generated due to a malformed TRB or a SET_ADDRESS Setup Stage TRB, hence their generation is solely due to a xHCI driver error. So as not to burden xHCI implementations with complex error handing logic that only applies to the driver debug process, an xHC is allowed to assert *HCE* when *TRB Error* conditions are detected.

detection of *TRB Error* related error conditions. It is the responsibility of software to always present correctly formed TRBs to the xHC.

4.10.2.3 USB Transaction Error

A transaction error is any error that causes the host controller to not complete a transfer successfully. Table 4-4 lists the events/responses that the xHC can observe as a result of a transaction. The effects of the Bus Error Counter and interrupt status are summarized in the following paragraphs. Most of these errors set the *USB Transaction Error* Completion Code in the appropriate Transfer Event TRB.

There is a small set of protocol errors that relate only when executing a *Setup Stage TRB* and fit under the umbrella of a *Bad PID* error that are significant to explicitly identify. When these errors occur, the *Bus Error Counter* (4.10.2.7) is decremented. When the USB PID Code[30] indicates a SETUP, the following responses are protocol errors and shall result in a *USB Transaction Error* if not resolved after *CErr* retries.

- A high-speed device and returns a NAK handshake to a SETUP.
- A high-speed device and returns a NYET handshake to a SETUP.
- A low- or full-speed device complete-split receives a NAK handshake.
- A SuperSpeed device responds to a SETUP DP with an NRDY TP.

Table 4-4: Summary of USB Transaction Errors

Event / Result	Error Tries	TRB Error Status
USB2 CRC or USB3 DPP Error[31]	CErr	USB Transaction[32]
Timeout	CErr (USB2), N/A (USB3)[33]	USB Transaction[32]

[30]Refer to Table 8-1 in the USB2 spec for a list of the PID Codes (Types).

[31]Refer to section 0 for the definition of a DPP Error. Note that the xHCI definition is slightly different than the definition of DPP Error in the USB3 spec because it includes the case where an ACK TP is received for a DPP with the *Retry Data Packet* (rty) bit set,

[32]If error occurs on a USB transaction, then a *USB Transaction Error* (XactErr) is asserted immediately on an Isoch pipe or after *CErr* unsuccessful attempts on all other pipe types. In addition non-Isoch Transfer Ring shall be halted, refer to section 4.10.2.1.

[33]Section 8.13 of the USB3 spec states that if a *tHostTransactionTimeout* occurs, for control, bulk, and Interrupt transactions the host shall assume that the transaction has failed and halt the endpoint. For Isoch transactions the host shall not perform any more transactions to the endpoint in the current Service Interval. And the host shall not halt the endpoint and shall restart transactions to the endpoint in the next Service Interval. No retries are performed for any transaction type if a *tHostTransactionTimeout* occurs.

USB2 Bad PID[34]	CErr	USB Transaction[32]
Babble	N/A	Babble Detected Error
Buffer Error	N/A	Data Buffer Error

This error condition shall only be reported in a Transfer Event due to an error detected on a Transfer TRB. This error shall not be reported in any other Event TRB types.

Note: No retries shall be performed if the xHC does not see a response to a Data Transaction (either IN or OUT) within tHostTransactionTimeout on a SuperSpeed or SuperSpeedPlus pipe. The endpoint shall transition to Halted state when this condition is detected.

Note: The USB3 spec defines a range of possible tHostTransactionTimeout values. The specific value applied by an xHC implementation may be hardcoded by an xHC vendor or programmable through a vendor defined mechanism, e.g. a Vendor Defined xHCI Extended Capability.

4.10.2.4 Babble Detected Error

When a device transmits more data on the USB than the host controller is expecting for a transaction, it is defined to be babbling. In general, this is called a **Babble Error**[35]. When a device sends more data than the *TD Transfer Size* bytes (**TD Babble**), unexpected activity that persists beyond a specified point in a (micro)frame (**Frame Babble**), or a packet greater than Max Packet Size (**Packet Babble**), the host controller shall set the *Babble Detected Error* in the *Completion Code* field of the TRB, generate an Error Event, and halt the endpoint (refer to Section 4.10.2.1). The *Bus Error Counter* is not decremented for a *Babble Error* condition.

This error condition shall only be reported in a Transfer Event due to an error detected on a Transfer TRB. This error shall not be reported in any other Event TRB types.

[34]The xHC received a response from the device, but it could not recognize the PID as a valid PID. Not applicable to USB3.

[35]The USB3 spec describes a (Packet) babble condition as receiving "sDataSymbolsBabble symbols without receiving a valid DPPEND ordered set or DPPABORT".The USB2 spec describes a (Frame) babble condition as "unexpected bus activity that persists beyond a specified point in a (micro)frame. Refer to section 8.7.4 in the USB2 spec for more details.The EHCI spec describes two (TD and Packet) babble conditions as "the device sends more than Tr*ansaction X Length* or Ma*ximum Packet Size* bytes (whichever is less)". Where Tr*ansaction X Length is* equivalent to TD *Transfer Size,* i.e. a TD *Babble* condition. The EHCI spec also states that a babble error "is considered a fatal error for the transfer".

Note: When *Babble Detected Error* is generated, software shall assume that any excess received data has been lost and not attempt a Soft Retry.

Note: If a Babble Error is detected and the received data passes all integrity checks, the host controller may write the received data (up to the expected data length) to the data buffer, and the value of the *TRB Transfer Length* field in the *Babble Detected Error* Transfer Event shall be consistent with the number of data bytes written to the buffer.

4.10.2.4.1 USB2 Protocol

A babble condition also exists if IN transaction is in progress at High-speed EOF2 point. This is called a **Frame Babble**. If a Frame Babble condition is detected while a TRB is being processed the xHC shall set the *Babble Detected Error* in the *Completion Code* field of the TRB, generate an Error Event, and halt the endpoint. In addition, the xHC shall disable the Root Hub port to which the Frame Babble is detected. The xHC shall never start an OUT transaction that will babble across a microframe EOF.

Note: *Frame Babble* is also a *Port_Error* condition which shall transition a port in the *Enabled* state to the Disabled state, assert the *PEC* flag ('1'), and generate a *Port Status Change Event*. Refer to section 4.19.1.1.6.

IMPLEMENTATION NOTE

PID Mismatch and Babble Checking

When a host controller detects a data PID mismatch, it shall either: disable the Packet Babble checking for the duration of the bus transaction, or do Packet Babble checking based solely on Maximum Packet Size. The USB core specification defines the requirements on a data receiver when it receives a data PID mismatch (e.g. expects a DATA0 and gets a DATA1 or visa-versa). In summary, the xHC shall ignore the received data and respond with an ACK handshake, in order to advance the transmitter's data sequence.

The xHCI allows System software to provide buffers for a Control, Bulk or Interrupt IN endpoint that are not an even multiple of the maximum packet size specified by the device. Whenever a device misses an ACK for an IN endpoint, the host and device are out of synchronization with respect to the progress of the data transfer. The xHC may have advanced the transfer to a buffer that is less than maximum packet size. The device will re-send its maximum packet size data packet, with the original data PID, in response to the next IN token. In order to properly manage the bus protocol, the host controller shall disable the Packet Babble check when it observes the data PID mismatch.

4.10.2.4.2 USB3 Protocol

A babble condition also exists if on an IN transaction the DPP exceeds the Max Packet Size. If a babble condition is detected the xHC shall set the *Babble*

Detected Error in the *Completion Code* field of the TRB, generate an Error Event, and halt the endpoint.

4.10.2.5 Data Buffer Error

This event indicates that an overrun of incoming data or an underrun of outgoing data has occurred for this Transfer TRB. This would generally be caused by the host controller not being able to access required data buffers in memory within necessary latency requirements. These conditions are not considered transaction errors, and do not effect the Bus Error Count. When these errors do occur, a Transfer Event TRB will be generated (pointing to the TRB that the error was detected on) with the Completion Status set to *Data Buffer Error*.

If the *Data Buffer Error* occurs on a non-isochronous IN, the host controller shall not issue a handshake to the endpoint. This will force the endpoint to resend the same data (and data toggle) in response to the next IN to the endpoint.

If the *Data Buffer Error* occurs on an OUT, the host controller shall corrupt the end of the packet so that it cannot be interpreted by the device as a good data packet. Simply truncating the packet is not considered acceptable. An acceptable implementation option is to 1's complement the CRC bytes and send them. There are other options suggested in the Transaction Translator section of the USB2 spec.

This error condition shall only be reported in a Transfer Event due to an error detected on a Transfer TRB. This error will not be reported in any other Event TRB types.

Note: A Data Buffer Error may be generated for a USB2 or USB3 transfer.

4.10.2.6 Host System Errors

Interrupts are used by xHCI to report Events generated by the controller. The reporting requires that the xHC hardware that manages the Event Ring, and the host system hardware that the xHCI is communicating over, is operating properly.

If a catastrophic system error occurs, it may prevent the xHC from properly completing a TRB in the Event Ring. This means that software could receive an interrupt with an inconsistent Event Ring. If in the process of normal Event TRB processing software suspects a problem, it may examine the *Host System Error* (HSE) bit in the USBSTS register to determine whether the problem was due to a host controller related catastrophic fault condition.

If a catastrophic error occurs during a host system access involving the Host Controller module the *Host System Error* (HSE) bit in the USBSTS register shall be set to ‘1’. (In a PCI system, conditions that set this bit to ‘1’ include PCI Parity error, PCI Master Abort, and PCI Target Abort.) When this error occurs, the Host

Controller shall clear the *Run/Stop* (R/S) bit in the USBCMD register to prevent further execution of the scheduled TDs.

The following conditions shall indicate an Event TRB problem:

- System software receives an xHC interrupt and a Valid Transfer Event TRB does not point to a Valid source TRB.
- System software receives an xHC interrupt and a Valid Transfer Event TRB does not identify an enabled Device Slot.
- System software receives an xHC interrupt and a Valid Transfer Event TRB does not identify to an enabled endpoint.
- Out of range, incomplete, or inconsistent Event TRB field values.

It is recommended that system software check for these conditions.

Note: A *Host System Error* (HSE = '1') may be generated due to transfer integrity errors on the system bus. Some modern system bus interrupt mechanisms (e.g MSI, MSI-X) utilize specialized writes to the host address space to generate interrupts. These writes require that the address and data paths of the system bus to be functioning properly. A catastrophic error condition may prevent these writes from completing successfully. It is recommended that an xHC implementation uses and "Out-of-Band" mechanism for reporting Host System Errors. This may be a hardwired interrupt, bus or system error signal provided by the system bus.

Host System Error (HSE) may optionally be used to report other internal xHC errors that might jeopardize system level operation or data integrity. It should be assumed, however, that the assertion of *HSE* should generate a critical system interrupt (e.g., NMI or Machine Check) and is, therefore, fatal. Consequently, care should be taken in using *HSE* to report non-parity or system errors. Both the xHC and software shall assume that system integrity has been compromised when *HSE* is asserted.

Note: *Host Controller Error* (HCE) should be used to report internal xHC error conditions which may be recovered from by software resetting and reinitialization of the xHC. Refer to section 4.24.1.

IMPLEMENTATION NOTE

Out-of-Band Error Reporting

The PCI *PERR#* (Parity ERRor) and *SERR#* (System ERRor) error reporting pins are required for all PCI implementations. xHC implementations shall assert the PERR# pin if a parity error is detected during a PCI transaction (other than Special Cycle). The xHC shall assert the SERR# pin if an address parity error, data parity error on the Special Cycle command, the *Host System Error* (HSE) bit in the USBSTS register is set to '1', or any other system error is detected by the xHC where the result will be fatal. Assertion of the *PERR#* or *SERR#* pins shall set the *HSE* bit in the USBSTS register to '1'.

If an MSI or MSI-X write transaction is terminated with a Master-Abort or a Target-Abort, the xHC shall report the error by asserting SERR# (if bit 8 in the PCI Configuration Space *Command* register is set) and to set the appropriate bits in the PCI Configuration Space *Status* register (refer to Section 3.7.4.2 of the PCI specification). An MSI or MSI-X memory write transaction is ignored by the target if it is terminated with a Master-Abort or Target-Abort. Refer to section 5.2.1 for more information on the PCI Configuration Space registers.

If SERR# is not enabled, software should implement an algorithm for checking the *HSE* flag if the xHC is not responding.

Non-PCI xHC implementations shall provide an equivalent out-of-band notification mechanism for xHC notification of catastrophic errors.

4.10.2.7 Bus Error Counter

The ***Bus Error Counter*** is an internal 2-bit down counter that the xHC maintains. This counter determines the number of consecutive Errors allowed while executing a USB Transaction.

Section 4.10.2.3 describes how when *CErr* bus errors are encountered on any packet of a TD, the TD is aborted, the endpoint is Halted and an Error Event will be generated. The xHC is expected to maintain an internal *Bus Error Counter* for each endpoint, which allows retries and differentiating “soft-errors” from “hard-errors”.

The xHC initializes this internal *Bus Error Counter* to the value defined by the Endpoint Context *Error Count* (CErr) field on the first transmission of a packet and decrements it when an error is detected, if the *Bus Error Counter* reaches 0, then a hard-error is generated. If a packet transmission successfully completes prior to the Bus Error Counter reaching 0, it is considered successful and no error will be generated.

Table 4-5: CErr Management

Error	Decrement Counter	Comment
Transaction Error	Yes	Refer to section 4.10.2.3.
Stalled	No	Detection of Babble or Stall automatically halts the ring. Thus, count is not decremented.

No Error	No	If a bus transaction completes and the host controller does not detect a transaction error, then the host controller should reset the Bus Error Counter to extend the total number of errors for this TD. For example, Bus Error Counter should be reset with value of CErr on each successful completion of a USB transaction. The xHC shall not reset the Bus Error Counter if the value at the start of the transaction is 00b.
Data Buffer Error	No	Data buffer errors are host problems. They don't count against the device's retries.
Babble Detected	No	Detection of Babble or Stall automatically halts the ring. Thus, count is not decremented.

Note: Software shall not program *CErr* to a value of ‘0’ when the Slot Context *Speed* field indicates a Full- or Low-speed device. This combination could result in undefined behavior.

4.10.2.8 Isoch Endpoint Error Handling

CErr does not apply to Isoch Data Transactions because retries are not performed on Isoch endpoints. Also an Isoch endpoint shall not halt due to a Data Transaction error, but instead shall advance to the next Isoch TD and attempt to execute it during the next ESIT. An Isoch Data Transaction error shall force the generation of a *Transfer Event*, irrespective of whether the *Interrupt-on-Short Packet* (ISP) or *Interrupt On Completion* (IOC) flags are set in the *Transfer TRB*, where the *Transfer Event's*:

- *TRB Pointer* field shall point to the *Transfer TRB* that the error was detected on, and
- *TRB Transfer Length* field shall indicate the residue of the number of bytes not successfully transferred.

If a Timeout, USB2 CRC Error, USB3 DPP Error, or a USB2 Bad PID was detected on an Isoch IN Data Transaction, the *Completion Code* of the *Transfer Event* shall be set to *USB Transaction Error*.

If a Babble condition was detected on an Isoch IN Data Transaction, the *Completion Code* of the *Transfer Event* shall be set to *Babble Error*.

While advancing to the next Isoch TD:

- If an *Event Data TRB* is encountered, the xHC shall parse it, i.e. if the its *IOC* flag is set, an *Event Data Transfer Event* shall be generated with its *Completion Code* set to the same error value reported by the Transfer Event and *TRB Transfer Length* field set to the number of bytes successfully transferred. The first *Event Data TRB* encountered shall be parsed.

    - If subsequent *Event Data TRBs* are encountered in the process of advancing to the next Isoch TD, the xHC shall parse them if the *Parse All Event Data* (PAE) flag is set ('1'), and shall not parse them if the *PAE* flag is cleared ('0'). Refer to section 5.3.6 for more information on PAE.

- If a Link *Data TRB* is encountered, the xHC shall parse the *Link TRB*, i.e. if its *IOC* flag is set, a *Transfer Event* shall be generated with its *Completion Code* set to *Success*. All Link TRBs encountered shall be parsed.

Note: Isoch TD shall follow the TD Fragment rules which define when an IOC flag may be set within a TD.

Note: If a tHostTransactionTimeout occurs a SuperSpeed or SuperSpeed Plus Isoch IN endpoint shall not perform any more transactions to the endpoint in the current Service Interval. And the host shall not halt the endpoint and shall restart transactions to the endpoint in the next Service Interval (refer to Table 8-33 in the USB3 spec). Note that the tHostTransactionTimeout is an xHC implementation specific delay.

4.10.3 Events

Refer to section 4.17.4 for information on Event to Interrupter mapping.

4.10.3.1 Ring Overrun and Underrun

If an Isoch endpoint is *Running*, the xHC periodically schedules the endpoint as a function of the ESIT. Each ESIT the xHC shall execute one Isoch TD on the endpoint's Transfer Ring. If the Isoch ring is empty when the xHC is ready to perform the transfer, it shall generate a Transfer Event on the Event Ring indicated by the Slot Context *Interrupter Target* field. An IN Isoch endpoint shall set the Completion Code to *Ring Overrun* and an OUT Isoch endpoint shall set the Completion Code to *Ring Underrun*.

When a Ring Overrun or Ring Underrun condition occurs, the TRB referenced by the Dequeue Pointer is not valid. Ring Underrun and Ring Overrun Transfer Events shall clear the *TRB Transfer Length* field to '0', and set the *TRB Pointer* field to the address of the invalid TRB (i.e. the value of the Dequeue Pointer where the Overrun or Ring Underrun condition was detected). Refer to section 4.11.3.1 for a detailed description of the *Transfer Event TRB*. The functionality described in this paragraph shall be mandatory for all xHCI 1.1 compliant xHCs.

Note: Pre-1.1 xHC implementations clear the *TRB Pointer* field of a Ring Underrun or Ring Overrun Transfer Event TRB to '0'.

After a Ring Overrun or Ring Underrun condition is reported the endpoint shall remain in the *Running* state, and be removed from the Pipe Schedule. The endpoint shall be placed back on the Pipe Schedule the next time system software rings the doorbell for the endpoint.

A Ring Overrun or Ring Underrun condition may occur unintentionally if software posts Isoch TDs late, i.e. software does not meet the *Isochronous Scheduling Threshold* (IST) requirement. In this case the xHC detects an empty Transfer Ring for the ESIT, generates a Ring Overrun or Ring Underrun Event, and removes the endpoint from the Pipe Schedule. However software, not knowing that it is late, rings the endpoint's doorbell, posting an Isoch TD for the ESIT that just incurred the Over/Underrun condition. The doorbell ring causes the xHC to put the endpoint back on the Pipe Schedule, and in preparation for the next ESIT, the xHC may fetch a TD that software had intended for a previous ESIT. If the *SIA* flag is set, then the TD (and all subsequent TDs) will be transferred one ESIT late. If the *SIA* flag is cleared, then the xHC will inspect the TD's *Frame ID,* recognize that the TD is not within the current *Valid Frame Window*[36], and generate a *Missed Service Error*, because the xHC is unable to service the TD within the specified ESIT. After the *Missed Service Error* the xHC will attempt to "resynchronize" the Isoch pipe. If resynchronization is successful, then subsequent Isoch TDs will be transferred in their correct ESITs. Refer to section 4.10.3.2 for more information on *Missed Service Error* handling. Refer to section 4.11.2.5.2 for more information on Resynchronization.

Software typically posts multiple Isoch TDs with each doorbell ring. If software is very late (e.g. multiple ESITs) when it rings the doorbell after an Overrun/Underrun condition, then multiple Isoch TDs may not be within the current *Valid Frame Window*. In this case, a *Missed Service Error* shall be generated for each TD skipped in the process of resynchronizing. Refer to section 4.11.2.5 for the definition of *Valid Frame Window*.

Note: For Isoch TDs with *SIA* = '0' that are not scheduled in advance of the *Isochronous Scheduling Threshold* (IST):

- If an Isoch endpoint is *Running* and *Busy*, then TDs that are not scheduled in advance of the *IST* shall result in an *Ring Overrun* or *Ring Underrun* condition, because the Transfer Ring appears empty when the xHC goes to fetch the next TD (refer to section 4.14.2.1.4 for more information on *IST*).
- If an Isoch endpoint is *Running* and *Idle*, then TDs that are not scheduled in advance of the *IST* shall result in a *Missed Service Error*, because the doorbell is rung too late for the xHC to schedule the TD for the ESIT targeted by the *Frame ID* (refer to section 4.10.3.2 for more information).

Refer to section 4.11.2.5.1 for scheduling ESITs less than 1 ms., i.e. *Microframe Alignment*.

Note: A late doorbell ring may result in the generation of two Events; a *Ring Overrun* or *Ring Underrun* condition, being followed immediately by a *Missed Service Error*.

[36]If the ESIT is less 1 ms., then subsequent TDs within the same frame report the same *Frame ID* value and pass the "current Valid Frame Window" test, but they may still be late. Refer to section 4.11.2.5.1 for how software may ensure that an Isoch TD is transferred within the correct ESIT of a Frame.

The xHC generates a *Ring Overrun* or *Ring Underrun* condition because an Isoch Transfer Ring is empty when it tries to move the data for a scheduled Interval. The *Ring Overrun* or *Ring Underrun* condition also causes the endpoint to transition to the Runnung *Idle* state, i.e. requiring a doorbell ring to restart it. When the (late) doorbell ring does occur, assuming it posted data buffers for the Interval that generated the *Ring Overrun* or *Ring Underrun* condition, the xHC will fetch buffer targeted at an Interval that has already passed and generate a *Missed Service Error* because it cannot deliver the data associated with an Isoch TD.

4.10.3.2 Missed Service Error

This error only applies to Isochronous endpoints. A *Missed Service Error* Completion Code indicates that the xHC was unable to complete the data transfer associated with an Isoch TRB within the ESIT. The cause of the error may be due to an Event Ring full condition, excessive DMA latency when accessing periodic data causing an internal xHC buffer overrun or underrun, etc. The data associated with the TD in error shall be lost, however for the next ESIT the xHC shall advance to the next Isoch TD and attempt to execute it.

A *Missed Service Error* shall utilize the *Transfer Event TRB* format. The *TRB Pointer* field of *Missed Service Error* Transfer Event shall reference the TRB that was missed and its *TRB Transfer Length* the residue data bytes in the buffer. Since a *Missed Service Error* forces a Transfer Event, the Event's *TRB Pointer* field may not reference a TRB that has its IOC flag set ('1') within the skipped Isoch TD.

If the conditions that cause a *Missed Service Error* persist, multiple consecutive Isoch transfers may not be completed. In this case, a *Missed Service Error* Transfer Event shall be generated for every ESIT missed. The only exception to this rule is if an Event Ring full condition prevents the posting of *Missed Service Error* Transfer Events. When the Event Ring full condition clears, the xHC shall post a *Missed Service Error* Transfer Event for the last Isoch TD (of each Transfer Ring) not completed.

Note: xHC implementations that do not support the *Contiguous Frame ID Capability* (CFC) may not generate a *Missed Service Error* Transfer Event for every ESIT missed.

A *Missed Service Error* shall not be reported if an Isoch transfer was not completed due to another error condition, e.g. USB Transaction Error, etc.

Refer to section 4.10.3.1 for more information on the relationship of *Missed Service Errors* to *Ring Overrun* and *Ring Underrun* conditions.

4.10.3.3 Split Transaction Error

This error only applies to USB2 protocol endpoints for reporting an error on a split transaction, e.g. that the xHC was unable to schedule a required complete-split transaction of a HS Split Interrupt IN transaction. If a *Split Transaction Error* is detected, there is the possibility of data loss and the endpoint shall be halted.

Note: Software shall not attempt a Soft Retry to recover from a Split Transaction Error.

4.10.3.4 Short Packet

A *Short Packet* Completion Code shall be reported if number of bytes received was less than the TD Transfer Size and the *Interrupt-on Short Packet* (ISP) or *Interrupt on Completion* (IOC) flag was set to '1' in the associated *Transfer TRB*. Refer to section 6.4.5 Table 6-85 for the definition of the *Short Packet* completion code. Refer to section 4.10.1.1 for more information in Short Packet handling.

Note: If a Short Packet ends between two TRBs, either TRB may report a *Short Packet* Completion Code.

4.10.4 IOC Flag

The general rule for how the xHC should handle the *IOC* flag is simple: if the *IOC* flag is set, then generate an event. There are some exceptions to this rule described in the spec, e.g. if the Event Ring is full, but normally this rule should *always* be applied.

If software wants to know when the xHC has completed processing all the TRBs associated with a TD, it must set the *IOC* flag in the last TRB of TD. The event that the *IOC* in the last TRB generates informs software that last TRB of the TD is complete, which means that the TD is complete, and that the space on the Transfer Ring that the TD consumed may be reclaimed.

The *ISP* flag generates an event *only* if less data was received, than was specified by a TRB. The TRB *Transfer Length* field of the Transfer Event that a Short Packet condition generates informs software of the exact number of bytes transferred when the condition was detected. Software may also set the *BEI* flag if it is not interested in generating an interrupt due to a Short Packet Event.

And if the *ISP* flag is set and *IOC* flag is not set in the last TRB of TD that may received a Short Packet, an event shall not be generated if a Short Packet condition does not occur on that TRB, i.e. if the buffer defined by the TRB is completely filled.

In some cases, the xHC response to an error condition may look very similar to a Short Packet condition, because after the xHC generates an event for either condition, the xHC may automatically advance to the next TD. An example of this behavior is when a *USB Transaction Error* is detected during an Isoch IN transfer, where the Isoch pipe does not stall, but advances to the next Isoch TD in preparation for the next Interval. The error will generate an event, however if the event does not point to last TRB of the Isoch TD and the *IOC* flag is not set in last TRB of the TD with the error, software will have to wait until the next *IOC* flag is encountered by the endpoint before it can reclaim the Isoch TD that had the error. This may take many milliseconds depending on the size of the Interval and where software set the *IOC* flags.

In summary, software can not only use the *IOC* flag to report specific TD completions, but it can also be used to provide timely updates of the Dequeue Pointer position so that TRBs can be reclaimed, to reduce error recovery times, or to allow Transfer Rings to grow or shrink as function of system loading or resource changes.

Note: An exception is if the *PAE* flag is cleared ('0'). In this case when a Short Packet occurs, the *IOC* flag in the first Event Data TRB encountered generates an Event Data Transfer Event and the *IOC* flag is ignored in subsequent Event Data TRBs that are encountered in the process of advancing the Dequeue Pointer to the beginning of the next TD. Refer to section 5.3.6 for more information on PAE.

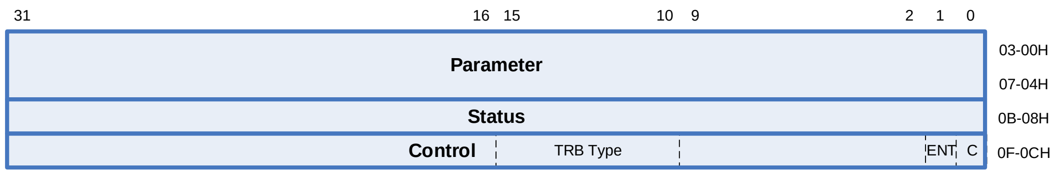
4.11 TRBs

This section discusses the properties and uses of TRBs that are outside of the scope of the general data structure descriptions that are provided in section 6.4.

4.11.1 TRB Template

TRBs adhere to the generalized template illustrated in Figure 4-13.

Figure 4-13: TRB Template



A TRB consist of 3 basic components: Parameter, Status, and Control. The following sub-sections identify the properties of each component.

4.11.1.1 Command and Transfer TRB Components

Command and Transfer TRB Components adhere to the following general rules, where the producer is system software and the consumer is the xHC.

All components of all Command and Transfer TRBs shall be initialized to '0' by the system software when the Command Ring or a Transfer Ring is created.

All components of all Command and Transfer TRBs shall be treated as read-only by the xHC.

The format/contents of all Command or Transfer TRB components shall be defined by the Control component *TRB Type* field. *TRB Type* field shall always reside in bits 10-15 of the Control component.

The Enqueue Pointer of a ring is defined by the transition of the Control component *Cycle* (C) bit in the TRB Ring. Refer to section 4.9 for a detailed explanation of Cycle bit operation. *Cycle* bit shall always reside in bit 0 of the Control component.

If the xHC does not pre-fetch TRBs the *Evaluate Next TRB* (*ENT*) flag forces the xHC to evaluate the next TRB of a TD before advancing to the next endpoint in the Pipe Schedule. The *ENT* flag does not span TDs, therefore the *ENT* flag is valid only if the *Chain* bit (CH) is '1'. Refer to section 4.12.3 for more information on the *ENT* flag.

Note: if all 4 Dwords of a TRB are not written as an atomic memory operation, then it is required that the Parameter and Status components of a TRB shall be initialized prior to writing the Control Component. Violating this rule shall cause undefined xHC behavior.

How a Transfer Ring is managed is described in section 4.11.2. How a Command Ring is managed is described in section 4.11.4.

IMPLEMENTATION NOTE

xHC Bus Mastering

The xHCI specification is designed around the assumption that hardware will issue a single, atomic system bus transaction when reading and writing TRBs and other data structures. For example, at least a 16 byte read transaction would be issued as an atomic operation to fetch a TRB from memory. Larger read or write transactions may be used to minimize the system bus overhead associated with moving data structures to or from memory, e.g. an xHC implementation could fetch 4 TRBs with a single 64B atomic operation, or use the system bus's maximum transaction size. Failure to read or write TRBs as atomic operations may result in undefined behavior.

4.11.1.2 Event TRB Components

Event TRB Components adhere to the following general rules, where the consumer is system software and the producer is the xHC.

All components of all Event TRBs shall be initialized to '0' by the system software when the Event Ring is created.

After Event Ring initialization, all components of Event TRBs shall be treated as read-only by system software.

The format/contents of all Event TRB components shall be defined by the Control component *TRB Type* field. *TRB Type* field shall always reside in bits 10-15 of the Control component.

The Enqueue Pointer of a ring is defined by the transition of the Control component Cycle (C) bit in the Event TRB Ring. Refer to section 4.9 for a detailed explanation of Cycle bit operation.

How an Event Ring is managed is described in section 4.11.3.

4.11.2 Transfer TRBs

Transfer TRBs shall be found on a **Transfer Ring**. A Work Item on a Transfer Ring is called a **Transfer Descriptor** (TD) and is comprised of one or more Transfer TRB data structures. This section describes the transfer related TRBs.

System software is the producer of all Transfer TRBs and the xHC is the consumer.

Upon completion of a Transfer TRB one of 4 conditions shall cause an associated Transfer Event to be generated on the Event Ring:

1. The *Interrupt On Completion* (IOC) flag is set.
2. A Short Packet has been received and the *Interrupt-on Short Packet* (ISP) flag is set.
3. An error occurred while executing a Transfer TRB.

In each case, the Completion code will indicate either Success or the cause of the Transfer Event generation.

The *IOC* flag will typically only be set (‘1’) in the last TRB of Transaction Descriptor (TD) to minimize Event TRB generation and system interrupts.

Each Endpoint Context defines one Transfer Ring if the *MaxPStreams* field = '0' or multiple Transfer Rings if the *MaxPStreams* field > '0'.

Table 6-86 defines the *TRB Types* found on a Transfer Ring. Table 6-87 defines the allowable Transfer Ring TRB Types as function of endpoint type.

Note: Software shall only utilize Transfer Events to determine TRB completions. Software shall not infer TRB completions based on Frame ID, MFINDEX, or other information.

Refer to section 4.11.7 for more information on TRB requirements.

4.11.2.1 Normal TRB

A *Normal TRB* is used in several ways; exclusively on Bulk and Interrupt Transfer Rings for normal and Scatter/Gather operations, to define additional data buffers for Scatter/Gather operations in Isoch and for Data stage TDs.

The direction of a data transfer associated with a Normal TRB depends on the direction defined by the Endpoint Context that it is associated with, or the preceding *Data Stage TRB* in the TRB Ring associated with a Control endpoint.

The *Chain* bit (*CH* field in Figure 6-8) may be set to '1' in Normal, Data Stage, Status Stage, and Isoch TRBs to form multi-TRB Transfer Descriptors. Chaining allows scatter/gather operations. Chaining can be used by system software to concatenate Pages of virtual memory, or to concatenate byte aligned data.

Refer to section 6.4.1.1 for the definition of a *Normal TRB*.

4.11.2.2 Setup Stage, Data Stage, and Status Stage TRBs

All USB devices respond to requests from the host on the device's Default Control Pipe. These requests are made using Control Transfers. At the USB packet level, a Control Transfer consists of multiple transactions partitioned into stages: a *setup stage*, an optional *data stage*, and a terminating *status stage*. The xHCI defines the *Setup Stage TRB*, *Data Stage TRB*, and *Status Stage TRB* to provide a 1:1 mapping to the respective USB Control transfer stages. Refer to section 3.2.9 for an overview of xHCI Control transfer support.

Refer to sections 6.4.1.2.1, 6.4.1.2.2, and 6.4.1.2.3 for detailed definitions of a *Setup Stage TRB*, *Data Stage TRB*, and *Status Stage TRB*, respectively. Also see section 8.5.3 in the USB2 spec. or section 8.12.2 in the USB3 spec. for a description of "Control Transfers".

Table 9-2 of the USB2 or USB3 specification defines the format of the **USB SETUP Data**. The host is responsible for establishing the values passed in the *USB SETUP Data* fields. Every USB Setup packet is comprised of an eight byte *USB SETUP Data* structure.

Figure 4-14: *SETUP Data*, the Parameter Component of Setup Stage TRB

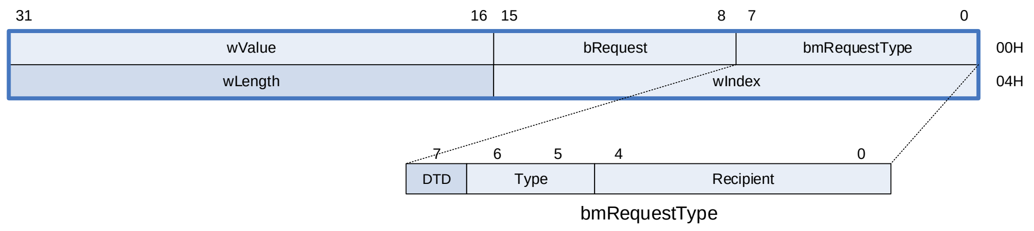


Figure 4-14 illustrates the mapping of the *USB SETUP Data* defined in section 9.3 (Table 9-2) of the USB2 or USB3 spec. to the *Setup Stage TRB* Parameter component.

The Transfer Ring associated with a Control Endpoint adheres to the following rules:

- The Control Transfer Ring may contain *Setup Stage* and *Status Stage TDs*, and optionally *Data Stage TDs*.
- Each *Setup Stage TD* shall contain a single *Setup Stage TRB*.
- A *Data Stage TD* shall consist of a *Data Stage TRB* chained to zero or more *Normal TRBs*, or *Event Data TRBs*.
- A *Status Stage TD* shall contain of a single *Status Stage TRB*, optionally chained to an *Event Data TRB*.
- All Control transfers require a *Setup Stage TD* followed by a *Status Stage TD*. If a data stage is required for the transfer, then system software is responsible for ensuring that a *Data Stage TD* is inserted between the *Setup Stage TD* and the *Status Stage TD*. "No-data" Control transfers do not require a *Data Stage TD*.
- A No-data Control transfer is generated by software if a *Data Stage TD* does not exist between the Setup Stage and Status Stage TDs.
- A *Setup Stage TRB* shall contain immediate data (*IDT* flag = '1'), its *Parameter* fields shall contain the 8-byte *USB SETUP Data*, which defines the request and the request's parameters that will be sent to the device in the USB Setup stage transaction, and its *Length* field shall be set to '8'.
- System software is responsible for setting the values passed in the *USB SETUP Data* fields as function of the desired USB Control Endpoint request. Refer to section 9.3 in the USB2 or section 9.3 in the USB3 spec. for the format of the *USB Setup Data*.
- System software is responsible for ensuring that the *Direction* (DIR) flag of the *Data Stage* and *Status Stage TRBs* are consistent with the *USB SETUP Data* defined *bmRequestType:Data Transfer Direction* (DTD) flag and *wLength* field. Refer to Table 4-6 for mapping.
- No more than one *Data Stage TD* may be defined between a pair of *Setup* and *Status Stage TDs*.

Table 4-6: USB SETUP Data to Data Stage TRB and Status Stage TRB mapping

USB SETUP Data		Transfer Type flag (TRT) Status Stage TRB	Direction flag (DIR)	
Data transfer direction (DTD)	wLength		Data Stage TRB	Status Stage TRB
Host-to-device	0	No Data Stage	No Data Stage TD defined	IN

	>0	OUT Data Stage	OUT	IN
Device-to-host	0	No Data Stage	No Data Stage TD defined	IN
	>0	IN Data Stage	IN	OUT

Note: The *Direction* (DIR) flag in the *Status Stage TRB* indicates the direction of the control transfer acknowledgement. For USB2 devices, *DIR* directly determines the PID that shall be used for the associated USB2 transaction. For USB3 devices, a *Status TP* is defined which is used for the status stage of all SuperSpeed (SS) control transfers. Refer to section 8.5 of the USB3 spec for the definition of the SS Status TP *Direction* flag.

Note: The *Direction* (DIR) flag in the *Data Stage TRB* defines the transfer direction for all TRBs in the *Data Stage TD*. For USB2 devices, *DIR* directly determines the PID that shall be used for the Data Stage transaction. For USB3 devices, if *DIR* = OUT a DP is generated with write data, if *DIR* = IN an ACK TP is generated to request read data from the device.

- If the data associated with a *Data Stage TD* is not contiguous, then additional *Normal TRBs* shall be chained in a *Data Stage TD*.
- System software is responsible for ensuring that the total data length defined by a *Data Stage TD* (i.e. the sum of the *Length* fields of the *Data Stage TRB* and all *Normal TRBs*) is equal to *wLength*. Note that communicating with some non-compliant devices may require violating this rule. The transfer lengths managed by the xHC depend strictly on the TRB Length fields.
- The Transfer Event generated by a *Status Stage TRB* shall report a *Success*, *Stall Error*, or other error Completion Code.
- ***Success*** indicates that the USB device has completed the command and is ready to accept a new command. Refer to "Function completes" row in Table 8-7 of the USB2 spec. Refer to "Request completes" row in Table 8-27 of the USB3 spec.
- ***Stall Error*** indicates that the USB device has an error that prevents it from completing the command. Refer to "Function has an error" row in Table 8-7 of the USB2 spec. Refer to "Request has an error" row in Table 8-27 of the USB3 spec. Software shall provide a timeout for all control operations and abort them using a *Stop Endpoint Command* if the operation times out.

Note: If a USB device is still processing the command when the Status Stage TD is executed, the device will return a Busy[37] response. The xHC shall wait indefinitely for a *Success*, *Stall Error* or other error response from device for the Status stage.

- The xHC shall **NOT** check for the following Control transfer error conditions.

Note: Some (non-compliant) USB devices use the SETUP Data *wLength* field as a custom parameter for non-data control transfers. xHCI implementations should not tie a non-zero wLength value to the existence of a Data Stage TD in a control transfer to ensure compatibility with those devices.

  - If a *Data Stage TD* follows a *Setup Stage TD*, where *wLength* = '0'.
  - If a *Status Stage TD* does not follow a *Setup Stage TD*, where *wLength* = '0'.
  - If a *Data Stage TD* does not follow a *Setup Stage TD*, where *wLength* > '0'[38].
  - If the total size of the *Data Stage TD* is not equal to *wLength*.
  - If the *Data Stage TRB Direction* (DIR) flag does not correspond to the definition in Table 4-6.
  - If the *Status Stage TRB Direction* (DIR) flag does not correspond to the definition in Table 4-6.
- The xHC is **NOT** required to check for the following Control transfer error conditions. If system software is properly designed these error conditions will never occur. However if the xHC does check for these conditions it shall generate a Transfer Event for the TRB that the error was detected on with the *Completion Code* set to *TRB Error*.
  - If a *Status Stage TD* does not follow a *Data Stage TD*.
  - If the *Setup Stage TRB* defines a Length not = 8.
- The xHC shall inspect the *bRequest* field in Setup Stage TRBs for a SET_ADDRESS request code and the *bmRequestType* field for Data Transfer Direction (DTD) = Host-to-device, Type = Standard, and Recipient = Device. If these values are detected for *bRequest* and *bmRequestType*, no Control transfer shall be issued to the USB, and the *Transfer Event* associated with the *Setup Stage TRB* shall return a *TRB Error* completion code. The SET_ADDRESS request is the ONLY Standard Device Request trapped by the xHC. This error shall not generate a stall condition on the Default Control Endpoint.

[37]Refer to "Function is busy" row in Table 8-7 of the USB 2 spec. Refer to "Device is busy" row in Table 8-27 of the USB 3 spec.

[38]This condition violates the definition of a USB Control Transfer, however this condition should be ignored by the xHC to ensure legacy device compatibility. The Setup Stage *Transfer Type* (TRT) field strictly indicates the presence and the Direction of the Data Stage TD, and determines the direction of the Status Stage TD so the *wLength* field should be ignored by the xHC.

- On a SS endpoint, if a STALL TP is received for a Setup, Data, or Status Stage TD, the xHC shall generate a Transfer Event pointing to the TRB that the error occurred on, with the Completion Code set to *Stall Error*.
- On a USB2 endpoint, if an error is detected on a Setup, Data or Status Stage TD, the xHC shall generate a Transfer Event pointing to the TRB that the error occurred on, with the Completion Code set to *USB Transaction Error*.
- All Control transfers begin with a Setup Stage TD and end with a Status Stage TD. A Control transfer may be aborted prior to executing its Data Stage or Status Stage TDs using a Stop Endpoint Command. Software is responsible for cleaning up the Transfer Ring after issuing a Stop Endpoint Command. And this is the only case where the xHC may expect to see a Setup Stage TD not follow a Status Stage TD.

Note: Undefined behavior may occur if software does not schedule a Status Stage TD to terminate a control transfer.

IMPLEMENTATION NOTE

Control Endpoint Recommendations

The USB2 specification section 8.5.3 is silent about what to do if a STALL is returned for a Setup Transaction handshake. The EHCI spec (e.g. section 4.12.1) treats a STALL generically, retrying the transaction indefinitely. Receiving a STALL for any Transaction handshake (including a Setup) halts the endpoint. The EHCI treats a NAK to a Setup Transaction as a USB Transaction Error (i.e. decrements *CErr*). It is recommended that xHCI provides the same response.

The USB3 specification section 8.12.2 is silent about what to do if an NRDY or STALL is returned for a Setup TP. xHCI implementations should treat these NRDYs like a *USB Transaction Error*, retrying the transaction CErr times (refer to section 4.10.2.3), and if a STALL is received for a Setup TP the xHC should halt the endpoint (refer to section 4.10.2.3).

4.11.2.3 Isoch TRB

An *Isoch Transfer Descriptor* (TD) shall consist of an *Isoch TRB* chained to zero or more *Normal TRBs*.

The direction of a data transfer associated with an Isoch Transfer Ring (and the Isoch TD that it defines) depends on the direction defined by the Endpoint Context that it is associated with. Refer to the *EP Type* field definition in Table 6-9 for the direction encoding.

The USB Endpoint Descriptor **bInterval** and **wMaxPacketSize**, and USB SuperSpeed Endpoint Companion Descriptor **bMaxBurst** and **bmAttributes:Mult** parameters define the bandwidth requirements of an isochronous pipe. These parameters specify a Quality of Service contract between the device and the host. This contract ensures that during an Interval, up to Max ESIT Payload bytes may be transferred between the host and the device. Another way of looking at it is; the USB Descriptor fields; bInterval, wMaxPacketSize, bMaxBurst, Mult, define a bandwidth that is guaranteed to be available on the USB for moving the data associated with this endpoint. The xHCI defines more generic versions of these parameters in the Endpoint Context; **Interval**, **Max Packet Size**, **Max Burst Size**, and **Mult** fields. System software is responsible for converting the endpoint type and speed dependent values defined in the USB Endpoint and SuperSpeed Endpoint Companion Descriptors to the generic values utilized by the xHCI. Refer to section 6.2.3 for more information on the Endpoint Context fields and their relationship to the USB Descriptor fields.

An *Isoch TD* defines an isochronous data transfer that will occur during a single Interval. An *Isoch TD* consists of one or more TRBs, where the first TRB of TD is always an *Isoch TRB*. If the data associated with an *Isoch TD* is not contiguous or larger than 64K bytes, then additional Normal TRBs may be chained to the initial Isoch TRB, forming a multi-TRB Isoch TD.

The xHC shall consume one *Isoch TD* each Interval on an Isoch Transfer Ring. To ensure streaming data, system software is required to place at least one Isoch TD on the Transfer Ring each Interval, prior to the *Isochronous Scheduling Threshold* (refer to IST, section 4.14.2.1).

For Isoch OUT endpoints, if the associated Transfer Ring is empty, then no Isoch transfers shall be scheduled over the USB during the intervening Intervals, the endpoint shall be removed from the xHC's Pipe Schedule, and a *Ring Underrun Event* shall be generated for the EPs' Transfer Ring to flag the condition.

For Isoch IN endpoints, if the Transfer Ring is empty, then any Isoch data that may have been transferred during the intervening Interval(s) shall be lost, the endpoint shall be removed from the xHC's Pipe Schedule, and a *Ring Overrun Event* shall be generated for the EPs' Transfer Ring. In either case, the endpoint shall remain in the *Running* state. The xHC shall remove the endpoint from the Isoch Pipe Schedule and restart the Isochronous transfers the next time the endpoint's doorbell is rung.

Note: A *Ring Underrun* or *Ring Overrun Event* is only generated the first Interval that an empty Transfer Ring is detected.

Note: Refer to section 4.10.3.1 for a description of *Ring Underrun* or *Ring Overrun Transfer Events*.

An Isoch Transfer Ring will be reinstated on the xHC's Pipe Schedule the next time its doorbell is rung.

If the xHC is unable meet an Isochronous deadline, a *Missed Service Error Event* shall be generated for the endpoint.

Note: The xHC may not generate a *Missed Service Error* for each Isochronous deadline missed, e.g. if the Event Ring is full.

The Ring Underrun, Ring Overrun, and Missed Service Error Events shall utilize a Transfer Event TRB format.

The Isoch TRB Frame ID field may be used to specify the Service Interval Boundary that an Isoch transfer may start on. If the Start Isoch ASAP (SIA) flag is cleared to '0' in the Isoch TRB, the xHC shall schedule the Isoch TD within one Service Interval of the next match of the Frame ID field with the Frame Index portion (bits 13:3) of the Microframe Index (MFINDEX) register. Refer to Figure 4-21. The range of possible values for the Frame ID field are 0 to 2047, with the constraints defined in section 4.11.2.5. If the Start Isoch ASAP (SIA) flag is set to '1' in the Isoch TRB, the Frame ID field is ignored and the Isoch TD is scheduled as soon as possible.

Service Interval Boundaries are aligned. I.e. if Interval = '1', then the Service Interval is 2 microframes long and begins when the low order bit of the MFINDEX register = 0. If Interval = '2', then the Service Interval is 4 microframes long and begins when the low order two bits of the MFINDEX register = 0, and so on.

The Isoch TRB Transfer Burst Count (TBC) and Transfer Last Burst Packet Count (TLBPC) fields may be used by the xHC to identify the exact number of packets that will comprise an Isoch TD without having to read in the complete TD. The xHC may use this information to better manage its periodic schedules. If Extended TBC Capability (ETC) and Extended TBC Enable (ETE) = '1' then TBC field supports the definition of Burst Counts up to 32 (and the TD Size field is deprecated in an Isoch TRB), otherwise the TBC field supports the definition of Burst Counts up to 4 (and the TD Size field is valid). Refer to section 6.4.1.3 for more information.

The *TBC* field (Table 6-34) shall be initialized by software. The following method shall be used to compute *TBC*, where *TDPC* is the *Transfer Descriptor Packet Count* described in section 4.14.1.

TBC = ROUNDUP (*TDPC* / (*Max Burst Size* + 1)) - 1

The *TLBPC* field (Table 6-34) shall be initialized by software. The following method shall be used to compute *TLBPC*, where *TDPC* is the *Transfer Descriptor Packet Count* described in section 4.14.1.

IsochBurstResiduePackets = *TDPC* MODULUS (*Max Burst Size* + 1)

TLBPC = IF (*IsochBurstResiduePackets* == 0)
THEN *Max Burst Size*
ELSE *IsochBurstResiduePackets* - 1

Refer to section 6.4.1.3 for the detailed definition of an *Isoch TRB*.

Note: The *ETC* shall not be enabled by an xHC implementation if the *Large ESIT Payload Capability* (*LEC* = '1') is not supported.

Note: If *LEC* = ‘1’ and *ETC* = ‘0’, then the largest Isoch Transfer that the *TBC* and *TLBPC* fields can describe is 64 KB. If the *Max ESIT Payload* indicates a value greater than 64 KB, then the *TBC* and *TLBPC* fields shall be used as a hint, rather than to compute an explicit Isoch TD packet count.

4.11.2.4 TD Size

The *TD Size* field of a TRB defines a number of packets that remain to be transferred for a TD after processing all Max Packet Sized packets in the current TRB and all previous TRBs. This field may be used by the xHC to estimate the size of a TD without requiring it to read ahead TRBs to the end of the TD. The *TD Size* field shall be initialized by software in Transfer TRBs, with a value calculated for a TRB using the following method:

TD Packet Count defines the number of packets that must be transferred to complete a TD.

TD Packet count = ROUNDUP(TD Transfer Size / Max Packet Size)

where, ROUNDUP (x) rounds fractional x up, away from 0 (zero), to the nearest integer value.

x is the number of Transfer TRBs in a TD.

n is the index of a Transfer TRB in a TD, where n = 1 for the first Transfer TRB of a TD.

TRB Transfer Length Sum (n) is the sum of the TRB Transfer Length fields in TRBs 1 through n.

Packets Transferred (n) defines the number of Max Packet Sized packets that have been transferred for the TD, up to and including the data described by TRB (n).

Packets Transferred (n) = ROUNDDOWN(TRB Transfer Length Sum (n) / Max Packet Size)

TRB Residue (n) defines the number of bytes remaining in TRB (n)'s buffer after processing all Max Packet Sized packets in the current TRB and all previous TRBs of a TD.

TRB Residue (n) = TRB Transfer Length Sum (n) - (Max Packet Size * Packets Transferred (n))

TD Size (n), For all Transfer TRBs except the last in a TD, TD Size identifies the number of packets that still need to be scheduled to complete this TD after sending TRB Residue (n) + the data for TRBs n+1 through x. The value of the *TD Size* in the last Transfer TRB of a TD (TD Size (x)) shall be cleared to '0' to explicitly indicate that it is the last Transfer TRB of the TD. Since the *TD Size* field is only 5 bits, its value shall be forced to 31 if the number of packets to be scheduled is greater than 31.

For all Transfer TRBs of a TD except the last (n = 1 through x-1):

TD Size (n) = IF (TD Packet Count - Packets Transferred (n) > 31, then 31,
else TD Packet Count - Packets Transferred (n))

For the last Transfer TRB of a TD:

TD Size (x) = 0.

Note: If the TRB Residue for the last Transfer TRB (TRB Residue (x)) is greater than 0, then a terminating Short Packet shall be generated for the TD. Also note that the TRB Residue value is always less than Max Packet Size.

Note: If *ETE* = ‘1’, then the *TD Size* is not available in Isoch TRBs. Refer to section 6.4.1.3.

Refer to section 6.4.1 for more information on the *TD Size* field.

4.11.2.5 Frame ID

The *Frame ID* field of an Isoch TD identifies the target frame that the Interval associated with this Isochronous Transfer Descriptor will start on. The *Frame ID* is valid only if the *Start Isoch ASAP* (SIA) field of an Isoch TRB equals ‘0’.

Software shall not schedule an Isoch TD with a *Frame ID* value that is greater than the *End Frame ID*, where:

End Frame ID = (Current MFINDEX register value + 895 ms.) MOD 2048

This limitation allows the xHC to properly manage Isoch TDs when a *Missed Service Error* occurs.

Note: When a *Missed Service Error* occurs, the Isoch TD that was supposed to be transferred during the missed service interval is dropped, and the xHC is expected resynchronize the Isoch pipe by advancing to the next Isoch TD for the next Interval. If the *Frame ID* of an Isoch TD is used to identify the specific Frame associated with a TRB of an Isoch TD, then the scheduling limit on the *Frame ID* (i.e. the *Valid Frame Window*) allows the xHC to unambiguously determine if an Isoch TD should be skipped or executed.

Software should not schedule an Isoch TD with a *Frame ID* value that is less than the *Start Frame ID*, where:

Start Frame ID = (Current MFINDEX register value + IST + 1) MOD 2048

This limitation allows the xHC sufficient time to fetch and schedule Isoch TDs. For more information on the *Isochronous Scheduling Threshold* (IST), refer to section 4.14.2.1.4.

Note: The *Frame ID* value is calculated as the modulus of 2048, i.e. the size of the *Frame Index* portion of the MFINDEX register (refer to Figure 4-21).

If the *Contiguous Frame ID Capability* is supported (*CFC* = '1'), then the xHC shall match the *Frame ID* in every Isoch TD with *SIA* = '1' against the *Frame Index* of the MFINDEX register. This rule ensures resynchronization of Isoch TDs even if some are dropped due to *Missed Service Errors* or *Stopping* the endpoint. Note that the xHC may advance through Isoch TDs faster than the Service Interval rate to resynchronize the Isoch data flow. Refer to section 4.11.2.5.2 for more information.

Note: If the *Contiguous Frame ID Capability* is supported (*CFC* = '1') by the xHC, then software should set the *Frame IDs* (i.e. *SIA* = '0') in all Isoch TDs. To induce a gap in the data stream of a Running Isoch endpoint, software simply specifies a gap in the *Frame IDs* assigned to the TDs of the data stream, and the xHC will pause the data stream until the *Frame ID* matches the *Frame Index* of the MFINDEX register.

Contiguous Frame ID Capability support (*CFC* = '1') is mandatory for all xHCI 1.1 compliant xHCI implementations.

A **Valid Frame Window** is defined by a *Start Frame ID* and an *End Frame ID.*

If the *Contiguous Frame ID Capability* is not supported (*CFC* = '0'), then the xHC may start the Isoch data flow when the MFINDEX *Frame Index* matches the *Frame ID* value specified in the first Isoch TD and ignore the *Frame ID* fields in subsequent Isoch TDs until the data flow is terminated, e.g. due to an Overrun or Underrun condition. A *Missed Service Error* does not terminate an Isoch data flow, therefore if a *Missed Service Error* occurs (i.e. one or more Isoch TDs are dropped), the xHCI will not be able to determine whether the subsequent Isoch TDs are within a *Valid Frame Window* and properly resynchronize the Isoch data flow.

Note: If the *Contiguous Frame ID Capability* is not supported (*CFC* = '0') by the xHC, then software may set the *Frame ID* (i.e. *SIA* = '0') only in the first Isoch TD of an Isoch data flow, and shall set *SIA* = '1' in all subsequent Isoch TDs of the data flow. To induce a gap in the data flow of a Running Isoch endpoint, software must force a *Ring Overrun* or *Ring Underrun* condition (by letting the Transfer Ring go empty), then specify the starting *Frame ID* in the first Isoch TD of the next data flow, and ring the doorbell.

4.11.2.5.1 Frame ID ESIT Rules

The ESIT of an endpoint may be smaller, equal to, or larger than the 1 ms. Frame period that may be specified by the *Frame ID* field of an Isoch TD. This section defines how to defined *Frame ID* values as a function of the ESIT value.

For endpoints with an ESIT greater than or equal to 1 ms.

- Software shall specify a *Frame ID* value that begins on an *ESIT Boundary*. E.g. if the *Interval* of an endpoint is 4 ms. (32 microframes) the valid *Frame ID* values for the endpoint are 0, 4, 8, 12, and so on.

- The xHC shall transfer an Isoch TD during the ESIT that starts on Frame boundary specified by the Frame ID. E.g. the TD in the example above with the Frame ID value of 4, may be transferred over the USB any time during Frames 4, 5, 6, or 7, where the xHC ensures that the data transfer will not take place before Frame 4 begins or after Frame 7 ends.

For endpoints with an ESIT less than 1 ms.:

- All Isoch TDs transferred within the same Frame (1 ms.) period shall have the same *Frame ID* value. So depending upon the value of the *Interval* field, up to 8 consecutive Isoch TDs may have the same *Frame ID* value. E.g. if the ESIT of an endpoint is 250 µs. (2 microframes) then groups of 4 consecutive Isoch TD shall have the same *Frame ID* value; 0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, and so on.
- To ensure proper *Microframe Alignment* of Isoch TDs for ESITs less than 1 ms., the xHC shall assume that the *Frame ID* of the first TD posted to a Transfer Ring shall begin transferring during the first ESIT of a Frame (1 ms.) period. The xHC shall also assume that when a transition in *Frame ID* values is detected (e.g. between the 4th and 5th TDs in the example above), the TD where the transition occurred (i.e. the 5th TD) should be transferred during the first ESIT of the next Frame period. E.g. in the above example, 4 Isoch TDs are transmitted each Frame period, where the first TD is transmitted during Frame 0/microframe 0 or 1, the second TD is transmitted during Frame 0/microframe 2 or 3, the third TD is transmitted during Frame 0/microframe 4 or 5, etc.

Note: Starting an IN or OUT Isoch data transfer at an ESIT that is not the first ESIT of a Frame period is currently not supported by this specification.

4.11.2.5.2 Resynchronization

An Isoch Transfer Ring is “synchronized” when the xHC is able to successfully transfer the data associated with an Isoch TD during the correct ESIT. If the *Frame ID* value of an Isoch TD is not valid for a target ESIT, then synchronization is lost. The rules defining the valid *Frame ID* values for a specific ESIT are identified in section 4.11.2.5.1.

If a *Missed Service Error* occurs then the xHC is required to advance through a Transfer Ring until it is “resynchronized” or the ring is exhausted. The data associated with Isoch TDs that are skipped over while attempting to resynchronize a pipe is not moved, however a *Missed Service Error* should be generated for every skipped Isoch TD.

The xHC shall not drop Events associated with TRBs as it attempts to resynchronize an Isoch pipe, e.g. if *IOC* = ‘1’ in a Link TRB then it returns *Success*, if *IOC* = ‘1’ in an Event Data or Normal TRB then it returns *Missed Service Error*, etc.

4.11.3 Event TRBs

Event TRBs shall be found on an **Event Ring**. A Work Item on an Event Ring is called an **Event Descriptor** (ED). An ED shall be comprised of only one Event TRB data structure. This section describes the operational characteristics of the event related TRBs.

The xHC is the producer of all Event TRBs and system software is the consumer.

Event TRBs are used to report events associated with the Command Ring and Transfer Rings, as well as a variety of other host controller related events (Port Status Change, Bandwidth Requests, etc.).

The field definitions of the Parameter, Length, and the high word of the Control components of Event TRBs are all *Event Type Dependent*. Refer to the specific Event definitions below for more information on these definitions. The Event Type field shall define the contents of the *Event Type Dependent* fields.

The *Event Type* field shall indicate Event Ring TRB Types as defined in Table 6-86. Any *Event Type* may be found on the Primary Event Ring. Only Transfer, Bandwidth Request, and Device Notification Events may be found on a secondary Event Ring. Refer to section 4.9.4.3 for a discussion of Primary and Secondary Event Rings.

IMPLEMENTATION NOTE

Event TRB Updating

The xHC shall ensure that all Dwords in an Event TRB are updated before it toggles the *Cycle* (C) bit in Dword 3. An xHC implementation may update all 4 Dwords of the *Event TRB* as an atomic (single DMA) operation, or if it updates the *Event TRB* Dwords as discrete operations, then it shall update Dword 3 (toggling the *Cycle* bit) last.

4.11.3.1 Transfer Event TRB

Transfer Event TRB generation shall *only* occur under the following conditions:

- If the *Interrupt On Completion* (IOC) flag is set.
- When a short transfer occurs during the execution of a Transfer TRB and the Interrupt-on-Short Packet (ISP) flag is set.
- If an error occurs during the execution of a Transfer TRB.

Several transfer related errors may be detected that cannot be attributed to a specific TRB, e.g. *Ring Overrun*, *Ring Underrun*, etc. In these cases, the xHC shall set the *TRB Pointer* to '0' and software shall treat it as invalid.

When the data transfer associated with a Transfer TRB completes, a Transfer Event shall be generated by the xHC if the TRB *IOC* or *ISP* flags are set to '1', or if an error occurs on the transfer associated with the TRB. And while advancing to the end the current TD after generating this event, each Transfer TRB encountered with its *IOC* flag set to '1' shall generate a Transfer Event. The *Condition Code* of the "current" Transfer Event shall be set to the value of the *Condition Code* in the original Transfer Event, and the *TRB Transfer Length* of the current Transfer Event should be set to value of the *TRB Transfer Length* field in the original Transfer Event.

4.11.4 Command TRBs

The *Parameter*, *Status* and *Length* TRB components shall be cleared to '0' by system software unless otherwise noted by a specific command.

The *TRB Type* field of the Control component shall indicate the Command Type. Table 6-86 defines the available Command TRBs, i.e. *TRB Types* allowed on a Command Ring.

For every command, the xHC notifies system software of its completion by placing a Command Completion Event TRB on the Event ring.

When a Command TRB is initialized on the Command ring, the *Cycle* bit will be set to the value of the Command Ring's Producer Cycle State (PCS) flag.

If an endpoint defines Streams, then commands that affect Endpoint Contexts may also affect the associated Stream Contexts. In cases where both contexts may be affected, the combined contexts are referred to as the "Endpoint/Stream" Context.

The remaining fields shall be managed by system software as a function of the command type, and are described below.

Note: The Address Device, Configure Endpoint, and Evaluate Context Commands utilize an Input Context data structure.

4.11.4.1 No Op Command TRB

The *No Op Command TRB* provides a simple means for verifying the operation of the basic TRB Ring mechanisms offered by the xHC, or to report the current value of the Command Ring Dequeue Pointer.

The format of the *No Op Command TRB* is defined in section 6.4.3.1.

Refer to section 4.6.2 for more information on the *No Op Command*.

4.11.4.2 Enable Slot Command TRB

The *Enable Slot Command TRB* causes the xHC to select an available Device Slot and return the ID of the selected slot to the host in a *Command Completion Event*.

The *Enable Slot Command* utilizes the same format as the *No Op Command TRB*, described in section 6.4.3.1.

Refer to section 4.6.3 for more information on the *Enable Slot Command*.

4.11.4.3 Disable Slot Command TRB

The *Disable Slot Command TRB* releases any bandwidth assigned to the disabled slot, frees any internal xHC resources assigned to the slot, and sets the *Slot State* field of the associated Slot Context to *Disabled*.

The format of the *Disable Slot Command TRB* is defined in section 6.4.3.3.

Refer to section 4.6.4 for more information on the *Disable Slot* command.

4.11.4.4 Address Device Command TRB

The *Address Device Command TRB* transitions the selected Slot Context from the *Default* to the *Addressed* state. It also causes the xHC to select an address for the USB device and issue a SET_ADDRESS request to the USB device.

The format of the *Address Device Command TRB* is defined in section 6.4.3.4.

Refer to section 3.3.4 for more information on the *Address Device Command*.

4.11.4.5 Configure Endpoint Command TRB

The *Configure Endpoint Command TRB* is used to enabled and/or disable selected endpoints of a Device Slot. When enabling endpoints the xHC evaluates the host controller resource and USB bandwidth requirements identified by the selected Endpoint Contexts in the command. If the requirements can be met, then the endpoints are enabled.

The format of the *Configure Endpoint Command TRB* is defined in section 6.4.3.5.

Refer to section 3.3.5 for more information on the *Configure Endpoint Command*.

4.11.4.6 Evaluate Context Command TRB

The *Evaluate Context Command TRB* is used by system software to notify the xHC that parameters associated with selected contexts have been modified. The current state of a context is not changed by the execution of an *Evaluate Context Command*. Refer to section 4.3 for more information on the use of this command.

Note: Refer to the Slot and Endpoint Context data structure descriptions (sections 6.2.2 and 6.2.3, respectively) for information on the specific Context fields that are evaluated by this command. A typical use of this command is immediately after an *Address Device Command* to inform that xHC that software has updated the *Max Packet Size* field of the Control endpoint. Refer to section 4.3 for more information on this usage.

The format of the *Evaluate Context Command TRB* is defined in section 6.4.3.6.

Refer to section 4.6.6 for more information on the *Evaluate Context Command*.

4.11.4.7 Reset Endpoint Command TRB

The *Reset Endpoint Command TRB* command is used by system software to reset an individual endpoint. This command may be used to restart a Halted endpoint.

The format of the *Reset Endpoint Command TRB* is defined in section 6.4.3.7.

Refer to section 4.6.8 for more information on the *Reset Endpoint Command*.

4.11.4.8 Stop Endpoint Command TRB

The *Stop Endpoint Command TRB* command is used by system software to stop the packet stream of an individual endpoint and transfer ownership of all the TDs on the associated Transfer Ring to software.

The format of the *Stop Endpoint Command TRB* is defined in section 6.4.3.8.

Refer to section 4.6.9 for more information on the *Stop Endpoint Command*.

4.11.4.9 Set TR Dequeue Pointer Command TRB

The *Set TR Dequeue Pointer Command TRB* command is used by system software to set the *TR Dequeue Pointer* field of an individual endpoint to a new value.

The format of the *Set TR Dequeue Pointer Command TRB* is defined in section 6.4.3.9.

Refer to section 4.6.10 for more information on the *Set TR Dequeue Pointer Command*.

4.11.4.10 Reset Device Command TRB

The *Reset Device Command TRB* command is used by system software to inform the xHC that it has reset a USB Device.

The format of the *Reset Device Command TRB* is defined in section 6.4.3.10.

Refer to section 4.6.11 for more information on the *Reset Device Command*.

4.11.4.11 Force Event Command TRB (Optional Normative)

The *Force Event Command TRB* allows a VMM to inject an Event TRB on the Event Ring of a selected Virtual Function. VMMs utilize this command when emulating a USB device to a VM. Refer to section 8 for more information on virtualization.

The format of the *Force Event Command TRB* is defined in section 6.4.3.11.

Refer to section 4.6.12 for more information on the *Force Event Command*.

4.11.4.12 Negotiate Bandwidth Command TRB (Optional Normative)

The *Negotiate Bandwidth Command TRB* is used by system software to initiate *Bandwidth Request Events* for periodic endpoints. This command may be used to recover unused USB bandwidth from the system.

If the *BW Negotiation Capability* (BNC) bit in the HCCPARAMS1 register is '1', then the xHC shall support this command.

The format of the *Negotiate Bandwidth Command TRB* is defined in section 6.4.3.12.

Refer to section 4.16 for more information on Bandwidth Negotiation.

Refer to section 4.6.13 for more information on the *Negotiate Bandwidth Command*.

4.11.4.13 Set Latency Tolerance Value Command TRB (Optional Normative)

The *Set Latency Tolerance Value Command TRB* is used by system software to provide a Best Effort Latency Tolerance (BELT) value to the xHC. This command is optional normative, however it shall be supported if the xHC also supports a corresponding host interconnect LTM mechanism.

If the *Latency Tolerance Messaging Capability* (LTC) bit in the HCCPARAMS1 register is '1', then the xHC shall support this command.

The format of the *Set Latency Tolerance Value Command TRB* is defined in section 6.4.3.13.

Refer to section 4.6.14 for more information on the *Set Latency Tolerance Value Command*.

4.11.4.14 Get Port Bandwidth Command TRB

The *Get Port Bandwidth Command TRB* is issued by software to retrieve the percentage of periodic bandwidth available on each Root Hub Port of the xHC. This information can be used by system software to recommend topology changes to the user if they were unable to enumerate a device due to a *Bandwidth Error* or a *Secondary Bandwidth Error*.

The format of the *Get Port Bandwidth Command TRB* is defined in section 6.4.3.14.

Refer to section 4.6.15 for more information on the *Get Port Bandwidth Command*.

4.11.4.15 Force Header Command TRB

The *Force Header Command TRB* is issued by software to send a Link Management or Transaction Packet to a USB device. For instance, it may be used to send a Vendor Device Test LMP.

The format of the *Force Header Command TRB* is defined in section 6.4.3.15.

Refer to section 4.6.16 for more information on the *Force Header Command*.

4.11.5 Other TRBs

4.11.5.1 Link TRB

The Link TRB provides support for sizing and non-contiguous Transfer and Command Rings. A Link TRB indicates the end of a ring by providing a pointer to the beginning of the ring.

If contiguous Pages cannot be allocated by system software to form a large Transfer Ring, then Link TRBs may also be used to link together multiple memory Pages to form a single Transfer Ring.

A non-contiguous TRB Ring is composed of Ring ***Segments***.

Software shall invoke the following rules when constructing a TRB Ring:

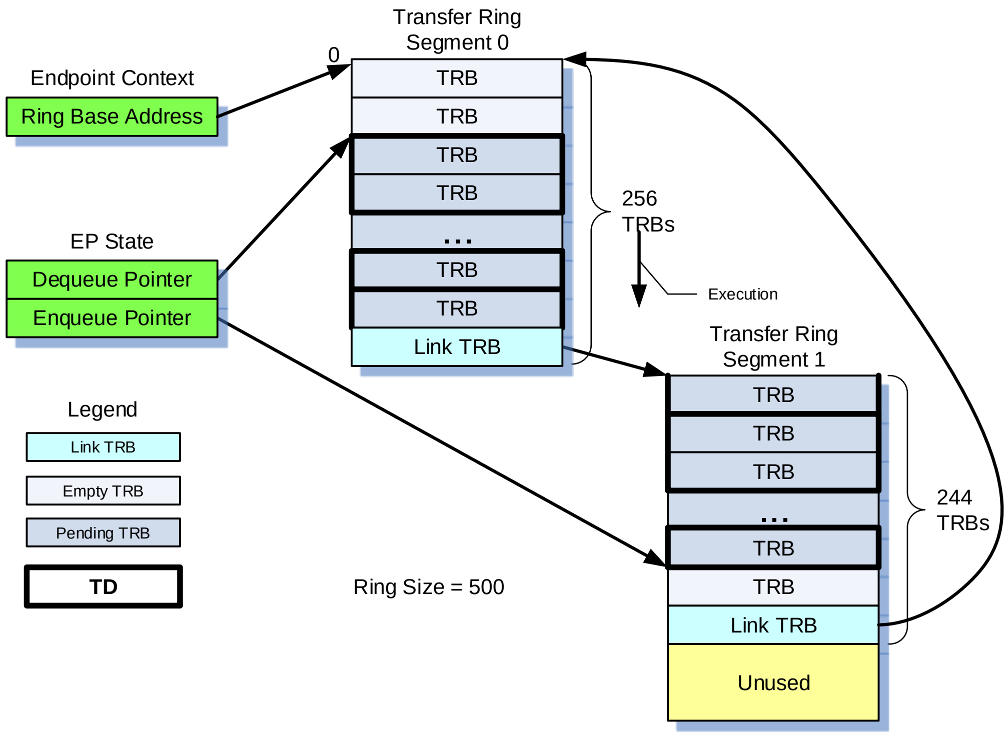
- All Transfer Ring Segments shall be aligned to 16-byte (TRB) boundaries.
- All Command Ring Segments shall be aligned to 64-byte boundaries.
- All Transfer and Command Ring Segments are multiples of 16 bytes in size.
- A *Link TRB* shall be the last TRB of each Transfer or Command Ring Segment
- The *Ring Segment Pointer* field of a *Link TRB* shall point to the next Segment of a multi-segment TRB Ring, or to first segment in a single Segment ring.
- The Link TRB of the last Ring Segment in a ring shall point to the beginning of the first segment of the ring.
- The *Toggle Cycle* flag should be set in at least one Link TRB of a ring.

Note: The *Ring Segment Pointer* field in a Link TRB is not required to point to the beginning of a physical memory page.

Note: A Link TRB may be found on Transfer or Command Rings.

Refer to Figure 4-15 for an illustration of TRB Ring Segments and Link TRBs.

Figure 4-15: Link TRB Example



Transfer Descriptors (Chained TRBs) may cross Segment boundaries.

Refer to section 4.11.7 for how the *Chain* (CH) flag shall be set in a Link TRB. In a Transfer Ring a Link TRB is always assumed to be linked to the first TRB of the next segment. If the *Chain* bit (CH) of the previous TRB is '1', then the multi-TRB TD that it defines spans segments and shall continue with the first TRB of the next segment. In a Command Ring the Link TRB *Chain* bit (CH) is ignored by the xHC.

As software advances its Enqueue Pointer and advances over a Link TRB, the *Cycle* (C) bit shall be updated with the value of the PCS flag.

The *Interrupt On Completion* (IOC) flag of a Link TRB may be used by system software to generate an event indicating the Dequeue Pointer has reached the Link TRB. This feature provides software with the ability to track the Dequeue Pointer as a function of segment boundary crossings.

Note: A TD Fragment shall not span segments. Refer to section 4.11.7.1.

When the Link TRB resides on a Transfer Ring the *Interrupt On Completion* (IOC) flag of a Link TRB may be used by system software to generate a Transfer Event, where the *Transfer Event Slot ID* and *Endpoint ID* shall reflect the slot and endpoint that the Transfer Ring is associated with, the *Length* = '0', the *TRB Pointer* field shall point to Link TRB, and the *Completion Code* = *Success*.

When the Link TRB resides on a Command Ring the *Interrupt On Completion* (IOC) flag of a Link TRB may be used by system software to generate a *Command Completion Event*, where the Command Completion Event *Slot ID* = '0', *VF ID* = '0', the *Command TRB Pointer* field shall point to Link TRB, and the *Completion Code* = *Success*.

Note: The Primary Interrupter ('0') is the target of all *Command Completion Events*. The *Interrupter Target* field shall be ignored by the xHC in Link TRBs found on the Command Ring.

IMPLEMENTATION NOTE

xHC TRB Fetching

All TRBs between the Enqueue and Dequeue Pointers of a TRB Ring are owned by the xHC. No constraints are placed on how many TRBs an xHC implementation may fetch in a single DMA operation or the order that the xHC may fetch them in. System software shall not modify a TRB owed by the xHC.

4.11.5.2 Event Data TRB

The *Event Data TRB* allows system software to generate a software defined event, and fully specify the Parameter Component of a generated event.

The Event Data TRB has the unique properties of inheriting the Completion Code of the previous (non-Event Data) TRB executed on a ring, and accumulating the transfer Lengths of preceding TRBs.

A typical use of the Event Data TRB would be to provide a 64-bit software defined identifier (or address) upon the completion of a TD. To accomplish this the Event Data TRB would be chained as the last TRB of the TD, and the IOC flag would be set only in the Event Data TRB. When the TD completes, an event is generated where the Completion Code is supplied by the previous TRB executed, and the Parameter Component of the event is loaded with the value supplied by the Event Data TRB.

The *Event Data* (ED) field of a Transfer Event indicates whether the event was generated by a Transfer TRB or an Event Data TRB. A Transfer Event with its *ED* flag equal '1' is referred to as a ***Event Data Transfer Event***.

A key feature of a *Event Data Transfer Event* is its ability to report the number of bytes transferred by a TD, rather than that of an individual TRB. To accomplish this the xHC maintains an internal 24-bit *Event Data Transfer Length Accumulator* (EDTLA) for each endpoint. The rules for EDTLA management are:

- The EDTLA shall be cleared to '0' immediately prior to executing the first Transfer TRB of a TD or when a *Set TR Dequeue Pointer Command* is executed.
- When a Transfer TRB is completed, the number of bytes transferred by the TRB shall be added to the EDTLA. The EDTLA shall wrap, if the total number of bytes transferred is greater than 16,777,215 (16MB-1).
- When an Event Data TRB is encountered an *Event Data Transfer Event* shall be generated, where the *TRB Transfer Length* field shall contain the value of the EDTLA. The EDTLA shall then be cleared to '0' and begin accumulating again.
- If a *Stopped Transfer Event* is generated and the *Condition Code = Stopped - Short Transfer*, then the *TRB Transfer Length* field of the Transfer Event shall contain the value of the EDTLA.

Note that for TDs greater than or equal to 16MBytes the EDTLA will roll-over. It is system software's responsibility to insert "**Intermediate**" Event Data TRBs periodically within a TD to report transfer lengths before the rollover condition occurs. Software is also responsible for accumulating the *Length* fields of Event Data Transfer Events to determine the total number of bytes transferred by a TD that declares multiple Event Data TRBs.

Note: Software shall set the *IOC* flag in all Event Data TRBs. Because the *IOC* flag must be set in an Event Data TRB, the possible locations of an Event Data TRBs within a TD are constrained by the *TD Fragment* rules described in section 4.11.7.1.

If a Short Packet is detected during the execution of a multi-TRB TD, the xHC shall advance to the first TRB of the next TD or the Enqueue Pointer (i.e.Cycle bit transition), whichever is encountered first. If the TD that incurred the Short Packet is terminated by an Event Data TRB (with its IOC flag is set), then the xHC shall generate an *Event Data Transfer Event*, where the *Length* field shall reflect the actual number of bytes transferred.

The following rules apply to Event Data TRBs on a Transfer Ring unless otherwise stated:

- An event shall be generated by an Event Data TRB if its IOC flag is set to '1'.
- An event generated by an Event Data TRB (*Event Data Transfer Event*) shall utilize the format of the Transfer Event TRB. The *Slot ID* and *Endpoint ID* fields shall be set appropriately for the Transfer Ring that contained the Event Data TRB, and the *Event Data* (ED) flag shall be set to '1'.
- The event generated when the *IOC* flag of an *Event Data TRB* is set to '1' shall report the Completion Code of the previously executed Transfer TRB of a TD, or *Success* if inserted as an Event Data TD (i.e. a TD that consists of just one Event Data TRB) on a

ring. The “previously executed Transfer TRB” is either the last Transfer TRB of the TD or the Transfer TRB that generated an error which forced a premature completion of the TD. Intermediate Event Data TRBs shall report “Success”.

- The Parameter Component of the Transfer Event generated by an Event Data TRB shall contain the value of the Event Data TRB Parameter Component.
- The *Length* field of a *Event Data Transfer Event* shall reflect the number of bytes transferred from the beginning of a TD or since the last Event TRB encountered in a TD.

Note: The above rules also apply to *Intermediate* Event Data Transfer Event TRBs.

Note: The *Event Data* (ED) flag in the Transfer Event TRB indicates to system software whether the Parameter Component of the respective event should be interpreted as pointer to system memory or software defined data.

Note: The *IOC* flag is treated generically by the xHC. If it is set in a TRB, then the xHC shall generate an Event for that TRB. If the *IOC* flag is not set in an *Event Data TRB*, the xHC will advance past it, clearing the EDTLA in the process.

Note: An Event Data TRB may only be found on a Transfer Ring.

Note: An Event Data TRB shall not immediately follow another Event Data TRB.

Note: Refer to section 4.12.3 for information on how the *Evaluate Next TRB* (ENT) flag should be used to manage Event Data TRBs.

Note: Refer to section 4.10.1.1 for more information on the handling of Event Data TRBs if a Short Packet condition occurs while executing a TD.

Note: Software shall not define a “stand-alone” Event Data TD (i.e. a TD that only contains a single Event Data TRB) on an Isoch Transfer Ring, however Event Data TRBs may be included in Isoch TDs.

4.11.6 Vendor Defined TRB Types

xHC vendors may define proprietary TRB Types using the *Vendor Defined TRB Type* codes identified in Table 6-86. The Vendor Defined TRB Types may be used to define Command, Event, or Transfer TRBs.

A vendor shall define proprietary xHCI Extended Capability structures using the xHCI Extended Capability Codes identified in Table 7-3 to enumerate any vendor defined TRB types or xHC capabilities.

If an unrecognized Vendor Defined TRB is encountered by the xHC:

- On a Transfer Ring, if a Vendor Defined TRB is preceded by a Transfer TRB and the *Chain* bit (CH) of the Transfer TRB is set (‘1’), then the Vendor Defined TRB is also required to support a valid *Chain* bit, which the xHC shall evaluate to determine if the end of the TD has been reached. Otherwise, the xHC shall advance past an unrecognized Vendor Defined TRB on a Transfer Ring and shall ignore it.

- The xHC shall treat Vendor Defined TRBs encountered on a Command Ring like a No Op Command TRB.
- Software shall advance past and ignore Vendor Defined TRBs encountered on an Event Ring.

Note: All vendor defined TRBs shall define a *Cycle* (C) bit at the same bit position as defined in all xHCI TRBs and manage it as defined in section 4.9 for the respective ring type.

Note: All vendor defined Event TRBs shall define a *Completion Code* field at the same bit position as defined in all xHCI Event TRBs and manage it as defined in section 4.9.4.

Note: Any vendor defined Transfer TRBs that may be included in a multi-TRB TD, shall define a *Chain* bit (CH) field at the same bit position as defined in a Normal TRB and manage it as defined in section 4.9.1.

xHC vendors may use the *Vendor Defined* TRB Type codes to define proprietary xHCI commands. All vendor defined commands shall utilize the Command Completion Event TRB to report completions.

Multiple vendors may define the same xHCI Extended Capability code or *Vendor Defined* TRB code to perform different operations. All vendor defined xHCI Extended Capability codes and TRB Types shall be qualified by system software with the PCI Configuration Space Header Vendor ID and Subsystem Vendor ID.

Vendors may also define Completion Codes. The Vendor Defined completion codes are separated into two groups: error and information. This partitioning allows software to infer the purpose of a Vendor Defined completion code even if it does not have vendor specific knowledge. Refer to Table 6-85.

If software does not have vendor specific knowledge, completion codes in the range defined by *Vendor Defined Info* codes shall be interpreted identically to a *Success* completion code.

If software does not have vendor specific knowledge, completion codes in the range defined by *Vendor Defined Error* codes shall be interpreted as an *Undefined Error* completion code, e.g if a *Vendor Defined Error* code is reported in a Command Completion Event software shall assume that the associated command did not complete successfully.

4.11.7 TD Usage Rules

A Transfer Descriptor (TD) may be composed of 1 or more TRBs. The TRB *Chain* flag is used identify the TRBs of a TD, where the *Chain* flag is set in all the TRBs of a TD except the last. In the simplest case, a TD consists of a single TRB. Larger transfers may require TDs that are comprised of many TRBs. If a TD crosses a TRB Ring Segment boundary it may include one or more *Link* TRBs.

Setting the TRB *Interrupt On Completion* (IOC) flag allows the completion of a TRB to generate an event. An *IOC* flag may be set in the TRBs of a TD identified in section 4.11.7.1.

Note: A "*Transfer TRB*" is any TRB defined in section 6.4.1. *Link* and *Event Data* TRBs are not "Transfer TRBs".

On an IN endpoint, if the device class allows a device to supply less data than the host has provided buffer space for, software has two options in forming a TD.

1. Set the *Interrupt-on Short Packet* (ISP) flag in all TRBs of a TD, and set the *IOC* flag in the last TRB. This action shall cause the xHC to generate a Transfer Event if a Short Packet condition is detected while executing any TRB in the TD, or generate a Transfer Event if the device completely fills the buffer.

   To determine the number of bytes actually transferred, software shall add the *TRB Transfer Length* fields of all TRBs up to and including the TRB that generated the Transfer Event, and subtract the Transfer Event *TRB Transfer Length* field.

2. Terminate the TD with an *Event Data* TRB that has its *IOC* flag set, and not set the *ISP* or *IOC* flag in any Transfer TRB of the TD. This action shall cause the xHC to generate an Event Data Transfer Event if a Short Packet condition is detected while executing any TRB in the TD or if the device completely fills the buffer.

   The *TRB Transfer Length* field of the Event Data Transfer Event identifies the number of bytes actually transferred, from the beginning of the TD or since the last Event Data Transfer Event. The *TRB Transfer Length* field of the Event Data Transfer Event may define up to a 16,777,215 byte transfer.

More than one *Event Data TRB* may be defined within a TD.

If *Event Data TRBs* are defined within a TD, then the *IOC* or *ISP* flags shall not be set in any Transfer TRB of a TD. i.e. the use of Event Data Transfer Events and normal Transfer Events to report a TD completion are mutually exclusive.

Note: Software may insert an Event Data TD immediately following a TD to provide additional information related to the previous TD. An **Event Data TD** is a TD that consists of just one Event Data TRB.

If the IDT flag is set in one TRB of a TD, then it shall be the only Transfer TRB of the TD. An Event Data TRB may be included in the TD.

Software shall specify the same *Interrupter Target* value in all TRBs of a TD. If an invalid *Interrupter Target* value is defined in a TRB, the behavior of the xHC is undefined if the TRB generates a Transfer Event. If virtualization is supported, an xHC implementation shall ensure that this “undefined behavior” does not affect another function (PF0 or VFx).

The Transfer TRB *TD Size* field shall be valid in all Transfer TRBs that define it. Refer to section 4.11.2.4.

Software shall not define a *No Op Transfer TRB* within a multi-TRB TD, i.e. software shall never set the *Chain* bit of a *No Op TRB* to '1' and a *No Op TRB* shall always be preceded by a TRB whose *Chain* bit is also set to '0'.

Software shall not define a *Link TRB* as the first TRB of a multi-TRB TD.

Software shall not define a *Link TRB* as the last TRB of a multi-TRB TD.

One or more Link TDs may precede or follow a TD. A **Link TD** is a TD that consists of just one Link TRB.

Software shall not define consecutive *Link TRBs* within a TD, i.e. software shall not set the *Chain* bit of consecutive *Link TRBs* to '1'.

Undefined xHC behavior may occur if the requirements defined in this section are not met.

Note: Besides reporting an error or the completion of a TD, Events may also be used by software to periodically update the current value of the Dequeue Pointer, to indicate the crossing of a Transfer Ring Segment boundary so it can add or remove a segment, etc., so the xHC shall generate an Event every time it encounters an *IOC* flag equal to ‘1’, irrespective of any error events that may be forced for earlier TRBs in a TD that did not have their *IOC* flag set.

For example, software may periodically set *IOC* flags in TRBs of a large TD so that it may update its Dequeue Pointer and reuse the TRBs that have been consumed by the xHC (rather than having to expand the Transfer Ring). Unless an error is encountered, all the intermediate events shall report Success. If any event generated by a TD reports an error, then that Completion Code overrides any Successful Completion Codes that other TRBs associated with the TD may have asserted, whether they come before or after the error Event.

Note: Software shall not interpret an error Event as indicating that the TD that it is associated with is “complete” (i.e. ownership of all the TRBs of the TD have been relinquished by the xHC), unless the *TRB Pointer* field of the error Transfer Event references the last TRB of the TD.

4.11.7.1 TD Fragments

The xHCI architecture allows TRBs to reference buffers of any length; however hardware works most efficiently when it is dealing with regularly sized buffers, e.g. Max Packet Size or Max Burst Size. Also the event generation mechanisms defined for Transfer Rings are extremely flexible, however constraints must be imposed to ensure that the hardware gate count and validation requirements are minimized for xHC implementations. *TD Fragments* require software to organize the TRBs of a TD in manner that allows the xHC hardware to optimize its internal buffer management and operation.

TD Fragments are designed to:

- Maximize burst opportunities for the xHC by ensuring that when software adds TRBs to a Transfer Ring, it does so in burst friendly units.
- Simplify Event generation by limiting the frequency and locations in a TD where the *IOC* flag may be set.

The **Max Burst Payload** (MBP) is the number of bytes moved by a maximum sized burst, i.e. *Max Burst Size* * *Max Packet Size* bytes.

A TD is comprised of one or more *TD Fragments*. If the *TD Transfer Size* is an even multiple of the MBP then all *TD Fragments* shall define exact multiples of MBP data bytes. If not, then the only last *TD Fragment* shall define less than MBP data (or the Residue) bytes.

Each *TD Fragment* is comprised of one or more TRBs. The first TRB of a *TD Fragment* is written last, ensuring that all the other TRBs of the *TD Fragment* are complete and reference valid buffers in host memory.

TD Fragments require software to construct TDs as sequential groups of TRBs. If the TD Transfer Size is greater than MBP, then the TD consists of 1 or more TD Fragments.

A *TD Fragment* may reference more than MBP bytes; if it is the last or only *TD Fragment* of a TD, or if it references an integral multiple of MBP bytes.

A *TD Fragment* may reference less than MBP bytes, if it is the last or only *TD Fragment* of a TD.

Software is allowed to construct a single TD Fragment that is an integral multiple of MPB bytes, or that defines a complete TD.

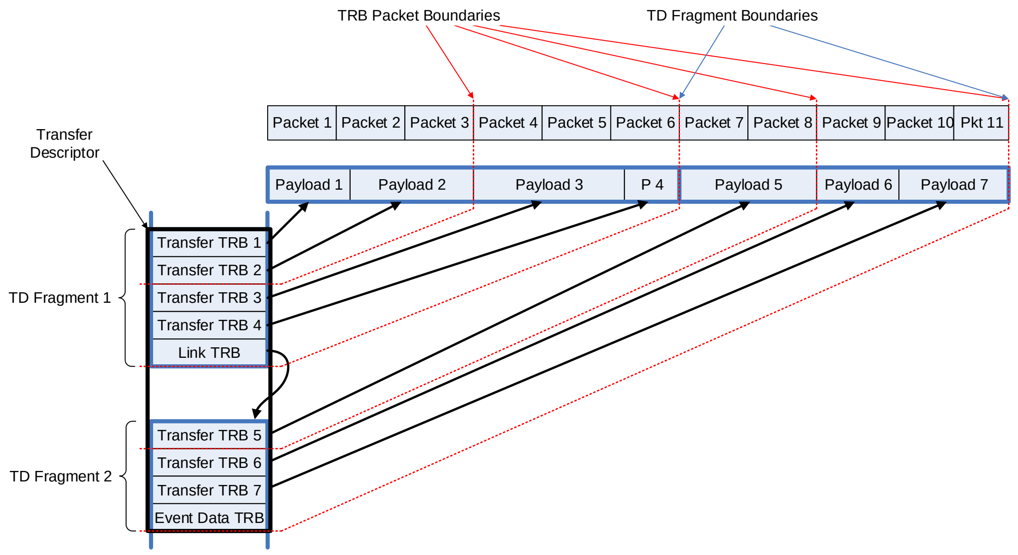
- The first TRB of a TD Fragment shall always be a Transfer TRB.
- A TD Fragment should not span Transfer Ring Segments.
- Link TRB placement in a TD shall follow the rules described in this section and section 4.11.7.
- Event Data TRB placement in a TD Fragment shall follow the rules described in this section and sections 4.11.5.2 and 4.11.7.
- A **TRB Packet Boundary** in a TD immediately precedes a Transfer TRB in which the first byte of the buffer referenced by a Transfer TRB is the also the first byte of a USB packet.

- The first TRB of a TD Fragment shall be the first TRB of a TD or immediately follow a *TRB Packet Boundary*.
- The last TRB of a TD Fragment immediately precedes a *TRB Packet Boundary* or is the last TRB of a TD.

The *IOC* flag may be set in only one TRB of a TD Fragment, with the following conditions:

- The IOC flag may be set in a Transfer TRB that immediately precedes a *TRB Packet Boundary* or the last Transfer TRB of a TD Fragment.
- The IOC flag may be set in a non-Transfer TRB (e.g. a Link TRB, Event Data TRB, etc.) that resides between two Transfer TRBs that form a *TRB Packet Boundary*, or follow the last Transfer TRB of a TD.

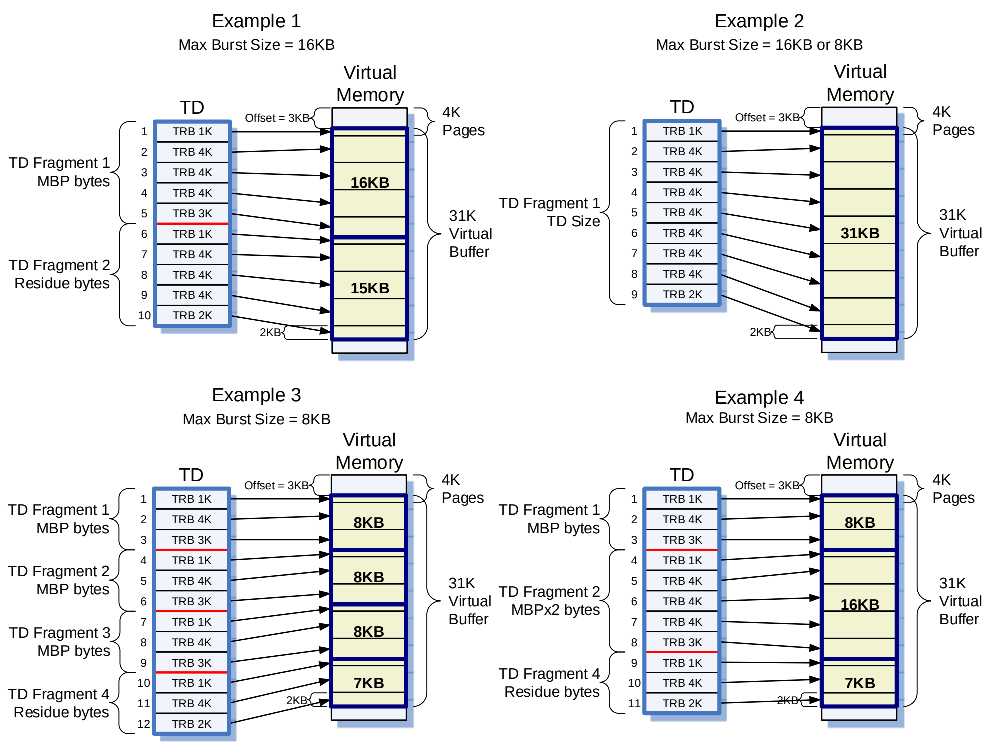
Figure 4-16: TRB Packet Boundary Example



The example of Figure 4-16 illustrates a TD that consists of two TD Fragments. TD Fragment 1 ends on boundary that is also a multiple of Max Packet Size bytes, while TD Fragment 2 ends at the end of the TD. Both TD Fragments end on a *TRB Packet Boundary* (red lines). An additional *TRB Packet Boundary* is defined in each TD Fragment, i.e. between TRBs 2 and 3 in TD Fragment 1 and between TRBs 5 and 6 in TD Fragment 2. Following the rules described above, the *IOC* flag may be set only once in a TD Fragment, i.e. in Transfer TRB 2, Transfer TRB 4, or the Link TRB of TD Fragment 1, and in Transfer TRB 5, Transfer TRB 7, or the Event Data TRB of TD Fragment 2. The IOC flag cannot be set in Transfer TRBs 1, 3 or 6 because they do not immediately precede a TRB Packet Boundary.

The TD Fragment rules above also ensure that the last Transfer TRB of a TD Fragment shall describe a data buffer that ends on a Max Packet Size boundary (Transfer TRB 4) or terminates the TD (Transfer TRB 7).

Figure 4-17: TD Fragment Examples



In Figure 4-17 the TDs in all the examples describe the same Virtual Buffer, which is 31KB in size, begins at a 3KB offset into the first physical Page, and spans 9 Pages.

Example 1 illustrates the *TD Fragments* that would be generated for an endpoint with a *Max Packet Size* = 1KB and a *Max Burst Size* of 16 packets. The first *TD Fragment* describes MBP (16K) bytes of buffer space. The second *TD Fragment* describes the TD Fragment Residue of the TD, or 15K bytes of buffer space. Note that two TRBs (5 and 6) are used to split 5th physical memory Page on a MBP boundary.

Example 2 illustrates a case where the single *TD Fragment* fully describes the TD, or 31K bytes of buffer space. In this case the TD is fully formed when TRB 1 is written, and the xHC will generate the Max Burst Size transactions as appropriate for the endpoint.

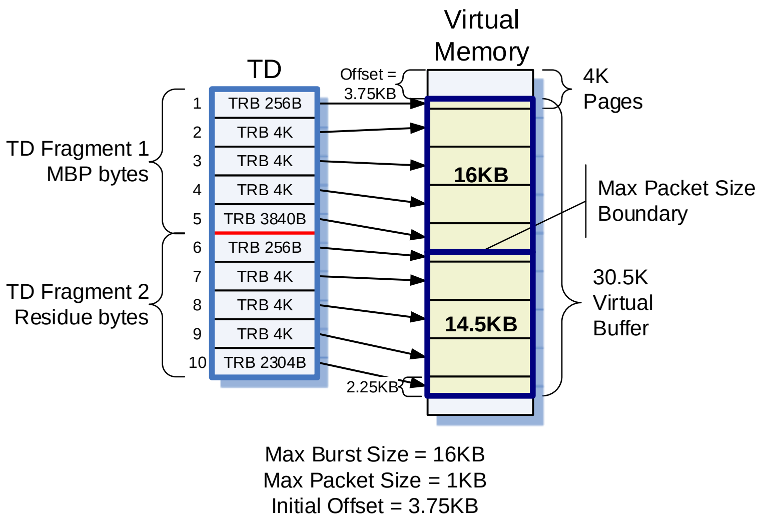
Examples 3 and 4 illustrates *TD Fragments* that may be generated for an endpoint with a *Max Packet Size* = 1KB and a *Max Burst Size* of 8 packets. Each of the first three *TD Fragments* in Example 3 describe MBP (8K) bytes of buffer space, and the last *TD Fragment* describes the Residue of the TD, or 7K bytes of buffer space. In Example 4 software has decided to use *TD Fragment* 2 to describe 2 x MBP bytes of buffer space.

In every case, software shall write the first TRB of a respective TD Fragment last. For instance the write order in Example 4 would be TRBs: 2->3->1, 5->6->7->8->4, and 10->11->9. And so on. Note that it really doesn't matter what order the TRBs of a *TD Fragment* are written in, as long as its first TRB is written last.

Note that in each example of Figure 4-17, the data associated with a single page is split between two TRBs to enforce a TD Fragment boundary, e.g. in example 1, the 4KB page on the boundary between TD Fragment 1 and 2 is defined by TRB 5 (3KB) and TRB 6 (1KB), where TRB 5 defines the last 3KB of the 16KB TD Fragment 1 and TRB 6 defines the first 1KB of TD Fragment 2.

Note: Only fully formed TDs may be scheduled on Isoch endpoints, e.g. write the first TRB of a multi-TRB TD last, irrespective of the number of *TD Fragments* that comprise it, and the *TD Fragment* rules for the assertion of *IOC* in TRBs described above apply.

Figure 4-18: Non-aligned TD Fragment Example



In Figure 4-18 the example defines a TD that transfers 30.5KB of data, where the packet size (Max Packet Size) = 1KB, the Burst Size = 16 KB, and the initial offset of the data in the first 4KB page is 3.75KB (3840B). An important aspect of this example is that due to the initial offset (3.75KB), page boundaries do not land on packet boundaries (as they do in Figure 4-17).

Given the rules defined above for where an *IOC* flag may be set in a TD Fragment:

- In Figure 4-18 the *IOC* flag only may be set in TRBs 5 and 10. In TRB 5 because TRBs 5 and 6 split the data in the page that they reference to force a break on a Burst Size boundary, hence the buffer described by TRB 5 ends on a packet, boundary. The *IOC* flag may be set in TRB 10 because it is the last packet of a TD, which forces a packet boundary. Note that the Link TRB does not land on a packet boundary relative to the start of TD Fragment 1, so its *IOC* flag may not be set.
- In Figure 4-17 all TRBs define buffers that end on Packet boundaries, hence an *IOC* flag may be set in any TRB of a TD Fragment, but only once per TD.

Note: The TD Fragment rules, that define which TRBs of a TD that an *IOC* flag may be set in, apply to Isoch TDs, however a partially formed TD shall not be posted to an Isoch endpoint. Only fully formed TDs may be posted to Isoch endpoints, e.g. software shall write the first TRB of a multi-TRB TD last, irrespective of its size.

4.12 Streams

Streams extend the number of Transfer Rings that may be accessible to a SS Bulk USB endpoint. A standard endpoint defines a single Transfer Ring. Streams allow an individual endpoint to define up to 65533[39] Transfer Rings using Linear Stream Arrays or Primary/Secondary Stream Arrays.

Streams allow the data flow of a bulk pipe to be multiplexed between multiple Transfer Rings associated with the endpoint. The USB device determines which Stream is active at any time, i.e. which Stream Context Transfer Ring is being used to move data.

The *TR Dequeue Pointer* field of an Endpoint Context that supports Streams points to an array of *Stream Context* data structures called the **Stream Context Array** or just **Stream Array**. A Stream (i.e. Stream Context) is selected with a **Stream ID**, where the *Stream ID* is used to index into a *Stream Array*.

A **Stream Context** data structure also contains a TR Dequeue Pointer field, which points to the Transfer Ring associated with the Stream.

A **Stream Protocol** maintained between the xHC and a SS USB device allows the device to establish the **Current Stream** (CStream) of an endpoint and control the movement of data for that Stream. At any time the device may terminate a Stream data transfer and switch to another Stream. Before an endpoint transitions to the Stopped, Halted, or Error state, the xHC shall ensure that the Stream Context *TR Dequeue Pointer*, *DCS*, and if *SEC* = '1', *Stopped EDTLA*[40] fields reflect the forward progress of any Stream that entered the Move Data state while the endpoint was in the Running state, e.g. the Stream Context fields are updated with the CStream state when a Stream exits the Move Data state

[39]Stream IDs 0, 65535 (No Stream) and 65534 (Prime) are reserved.

[40]*Stopped EDTLA Capability* support (i.e. *SEC* = '1') shall be mandatory for all xHCI 1.1 compliant xHCs.

(e.g. after a Stream switch or due to an error), or before the endpoint enters the Stopped, Halted, or Error state. Refer to section 3.3.8 for more information on Stream Context state requirements after a *Stopped Endpoint Command*.

Because all Streams associated with an endpoint share the same bulk pipe, if the Current Stream causes the pipe to stall, then all Streams associated with the pipe are also stalled. There are cases with the Stream Protocol where a Stall may occur and there is no directly attributable TRB that can be referenced by the Transfer Event TRB that reports the error (e.g. due to sending a Prime Pipe transaction). In this case the Slot Context *Interrupter Target* field shall be used to generate the Event and the *TRB Pointer* and *TRB Transfer Length* fields of the Transfer Event shall be set to '0'. Refer to section 4.17.4.

A Stream Context may be "active" or "non-active". A **non-active** Stream Context shall be identified by an empty Transfer Ring or if, through an out-of-band (Device Class) defined mechanism, software knows that the Stream Context will not be selected by a USB device to become the *Current Stream* (CStream). An **active** Stream Context does not meet the criteria described above for a non-active Stream Context. Reliably determining whether a Stream Context is active or not, is a Device Class responsibility. There is no xHCI defined method.

For example, a UASP data Stream Context becomes active (i.e. may be selected at any time by the device and become the *Current Stream*) after software rings the doorbell with the DB Stream ID equal to the Stream ID of the Stream Context. The Stream Context becomes non-active (i.e. shall not be selected by the device to become the *Current Stream*) when the UASP command associated with the Stream Context completes, or after an Abort Task command for the Stream Context is successfully completed by the UASP device.

Note: The value of *CStream* is not exposed for a Stream endpoint by the xHC after an endpoint transitions to the *Stopped* state (e.g. after to a Stop Endpoint Command). So if the Transfer Ring of a Stream Context is not empty, then software shall use an out-of-band mechanism to determine whether a Stream Context is active or not.

For more information on Streams refer to the section 8.12.1.4 of the USB3 specification.

4.12.1 xHCI Stream Protocol

The USB Stream Protocol adheres to the semantics of the standard SS Bulk protocol, so the packet exchanges on a SS bulk pipe that supports Streams are similar to a SS bulk pipe that doesn't. The Stream Protocol is managed strictly through manipulation of the packet header *Stream ID* field.

Stream selection is driven by a USB device. The Stream Protocol allows a device to switch Streams on packet boundaries.

This section references the *General Stream Protocol State Machine* (SPSM) defined in the USB3 specification (Figure 8-19), which applies to both IN and OUT endpoints. Unless otherwise stated, refer to the USB3 specification for the specific details of Stream ID and packet management on IN or OUT endpoints. Refer to USB3 section 8.12.1.4.2 for the IN Stream Protocol, and section 8.12.1.4.3 for the OUT Stream Protocol details.

Figure 4-19: xHC Stream Protocol State Machine (xSPSM)

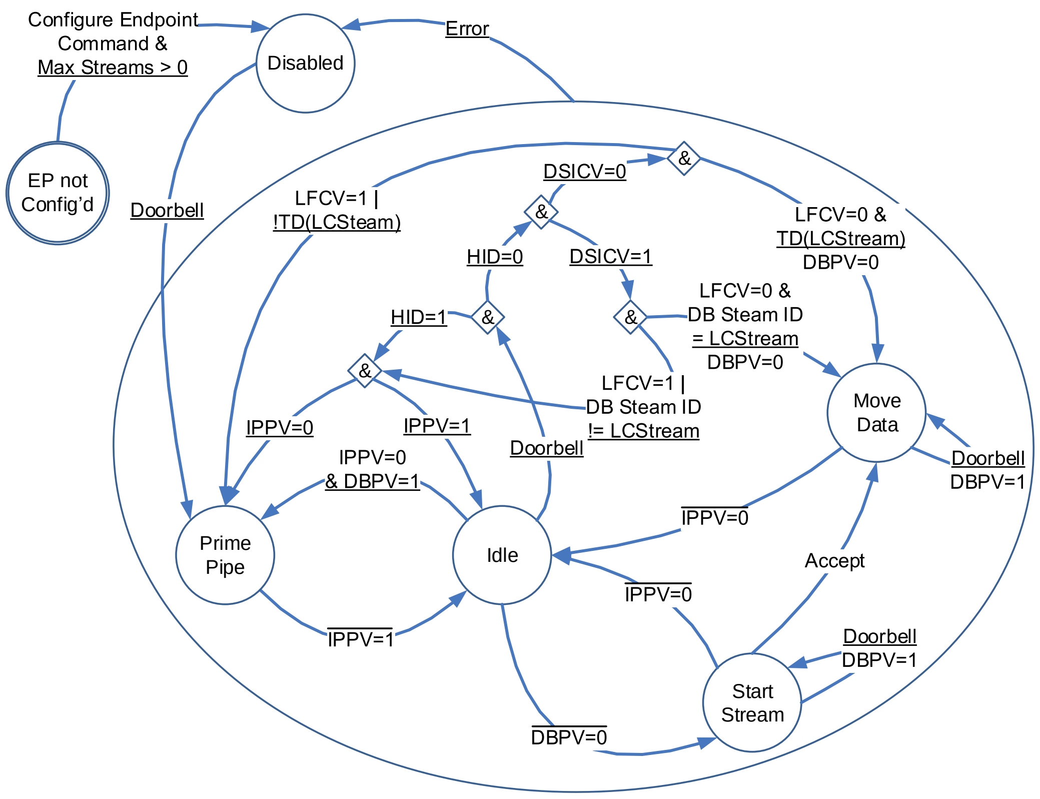


Figure 4-19 illustrates the xHCI Stream Protocol state machine, which overlays the USB Stream Protocol State Machine described in the USB3 spec. This section describes the xHC's role in the execution of the Stream Protocol. There is a 1:1 correspondence of the states described in the xSPSM and those defined in the USB3 SPSM. The xSPSM identifies the xHCI's role in advancing the USB3 SPSM. Refer to Appendix E for state machine notation.

The xSPSM associated with an unconfigured endpoint shall enter the **Disabled** state when a *Configure Endpoint Command* is executed and Streams are enabled (*MaxPStreams* >0).

The first time the endpoint doorbell is rung after entering the **Disabled** state, the xHC shall transition the xSPSM to the **Prime Pipe** state, and the device should automatically transition the xSPSM to the **Idle** state.

Note: The USB packet exchanges that transition the SPSM through its states are described in USB3 specification section 8.12.1.4.

The **Prime Pipe** state is used by the xHC to inform the USB device that host memory buffers have been modified or added to the endpoint by system software. The device may use this information as queue to start or restart stream activity.

To facilitate the xSPSM management of **Prime Pipe** transitions, an *Idle Prime Pipe Value* (IPPV), *LCStream Flow Control Value* (LFCV), and *Doorbell Pending Value* (DBPV) may be implemented by the xHC as a shadow flags. All three flags are initially cleared ('0'). The xSPSM utilizes IPPV. LFCV, and DBPV.

IPPV is cleared ('0') when the **Idle** state is entered from the **Start Stream** or **Move Data** state and set ('1') when the **Prime Pipe** state is entered. IPPV is used to limit **Prime Pipe** transitions to one per **Idle** state entry.

LFCV records if the LCStream was flow controlled by the device. In this case, the xHC should not generate a Host Initiated Data Move if buffers are posted for the LCStream. LFCV is updated when the **Move Data** state is exited. If the **Move Data** state was exited due to an *NRDY(Stream n)* condition then LFCV is set, otherwise LFCV is cleared. Refer to the IMDSM and OMDSM (Figures 8-30 and 8-32, respectively) in the USB3 specification for more information on the *NRDY(Stream n)* condition.

DBPV is cleared when entering the **Start Stream** or **Move Data** states and set if the doorbell is rung while the xSPSM is in the **Start Stream** or **Move Data** states. DBPV records doorbell rings while the xSPSM is not in the **Idle** state, so that a **Prime Pipe** state may be immediately forced when the **Idle** state is reentered.

To further accelerate the Stream protocol an xHC implementation may optionally capture the *DB Stream ID* value when the doorbell is rung. A fourth shadow flag, *DB Stream ID Captured Value* (**DSICV**) is set if the xHC hardware captures the *DB Stream ID* when the doorbell is rung, otherwise it is cleared.

If the doorbell for the endpoint is rung while in the **Idle** state the following algorithm shall be applied:

- If *Host Initiated* transitions are disabled (*HID* = '1'):
  - if IPPV = '0', transition to the **Prime Pipe** state.
  - if IPPV = '1', remain in the **Idle** state.
- If *HID* = '0':
  - If the xHC captures the *DB Stream ID* when the doorbell is rung (DSICV=1):
    - If LFCV = '0' and the *DB Stream ID* value equals LCStream[41], transition to the

[41]Refer to section 8.12.1.4.1 in the USB3 specification for the definition of *LCStream*.

Move Data state.

    - If LFCV = ‘1’ or the *DB Stream ID* value does not equal LCStream:
        - if IPPV = ‘0’, transition to the **Prime Pipe** state.
        - if IPPV = ‘1’, remain in the **Idle** state.
- If the *DB Stream ID* is not captured when the doorbell is rung (DSICV=0), access the Transfer Ring associated with LCStream to determine whether it is empty:
    - If LFCV = ‘0’ and the Transfer Ring is not empty (TD(LCStream)), transition to the **Move Data** state.
    - If LFCV = ‘1’ or the Transfer Ring is empty (!TD(LCStream)), transition to the **Prime Pipe** state.

Note: Due to internal resource or other limitations, an xHC implementation may disable *Host Initiated* transitions for an endpoint, i.e. the xSPSM may operate as if *HID* is always ‘1’, irrespective of the value of the field in the *Endpoint Context*.

Refer to section 4.12.1.1 for more information on *Host Initiated* transitions to the **Data Move** state.

When the xSPSM returns to the **Idle** state from the **Prime Pipe** state the xHC shall set the IPPV flag to ‘1’, flagging the fact that a **Prime Pipe** transition has been executed while in **Idle**.

When the xSPSM transitions from the **Idle** to the **Start Stream** or the **Move Data** state the xHC shall clear the DBPV flag to ‘0’, preparing it to record any Doorbell rings while it is in the **Start Stream** or **Move Data** states.

If the endpoint’s doorbell is rung while in the **Start Stream** or **Move Data** state, the DBPV flag is set to ‘1’.

Note: If an error (USB Transaction, timeout, etc.) is detected in the SuperSpeed ISPSM (Figure 8-29 in the USB3 specification) **Prime Pipe** or **Prime Pipe Ack** state, or the OSPSM (Figure 8-31 in the USB3 specification) **Prime Pipe**, **Start Stream End**, or the **Prime Pike Ack** state, the xHC shall generate a Transfer Event with the *TRB Pointer* and *TRB Transfer Length* fields = ‘0’, to the Event Ring identified by the Slot Context *Interrupter Target* field.

When the **Idle** state is entered from the **Start Stream** or **Move Data** state, the IPPV flag is cleared to ‘0’, enabling one **Prime Pipe** transition while in the **Idle** state.

If in the **Idle** state and the IPPV flag is ‘0’ and DBPV is ‘1’, the xSPSM shall transition to the **Prime Pipe** state, informing the device of the recorded doorbell ring.

A Stream ID is a zero-based value that indexes into the endpoint’s *Stream Context Array* starting at offset ‘0’, as illustrated in Figure 4-20.

The xHC uses the value of the **Stream ID** field, received in a SuperSpeed Transaction Packet (TP) or Data Packet (DP), as an index into the Stream Context

Array(s) to access the Stream Context associated with the packet. Refer to section 8.2 in the USB3 specification for a discussion of SuperSpeed Packet Types.

If Streams are defined for an endpoint, then:

- The Endpoint Context *MaxPStreams* field is > '0'.
- The Endpoint Context *TR Dequeue Pointer* field points to a *Primary Stream Context Array*.
- The Primary Stream Context Array shall contain *MaxPStreams* Stream Context data structures.

Streams may only be defined for Bulk endpoint types.

The *MaxPStreams* field in the *Endpoint Context* identifies the number of Streams supported by the *Primary Stream Array* of the endpoint. If *MaxPStreams* = '0', then the endpoint is a standard endpoint and its *TR Dequeue Pointer* field points to a Transfer Ring. The value of the *MaxPStreams* field shall not exceed the value reported in the *MaxStreams* field of the *SuperSpeed Endpoint Companion Descriptor* for the endpoint.

The *Stream ID* field of USB packets on endpoints that do not define Streams shall be ignored by the xHC.

Refer to section 3.3.8 for more information on how a Stream is affected by a *Stop Endpoint Command*. Refer to section 4.6.10 for more information on how a Stream is affected by a *Set TR Dequeue Pointer Command*.

4.12.1.1 Host Initiated Data Move

A *Host Initiated* transition from the **Idle** to the **Data Move** state is described in the General Stream Protocol State Machine (SPSM) of section 8.12.1.4 in the USB3 specification. The objective of a *Host Initiated* transition to **Data Move** is to initiate a Data Move operation that has a high probability of being accepted by the device.

A doorbell is rung when work is added to a Transfer Ring. The *DB Stream ID* indicates the specific Stream of the endpoint that the doorbell ring references.

An xHC implementation is not required to capture the value of *DB Stream ID* field when the doorbell is rung, however this feature may be used to accelerate SPSM transitions. When the doorbell is rung in the **Idle** state, the *DB Stream ID* value explicitly identifies the Stream that has had work added to it, thus eliminating the need to access the associated Transfer Ring to determine this condition. In Figure 4-19 the *DB Stream ID Capture Value* (DSICV) shadow flag is used to indicate whether an xHC implements this feature.

Some Stream usage models may operate more efficiently if the device maintains full control over Stream selection. *Host Initiated* transitions from the **Idle** to the **Move Data** state may be disabled by setting the *Host Initiated Disable* (HID) flag in the Endpoint Context to '1'.

4.12.2 Stream ID Management

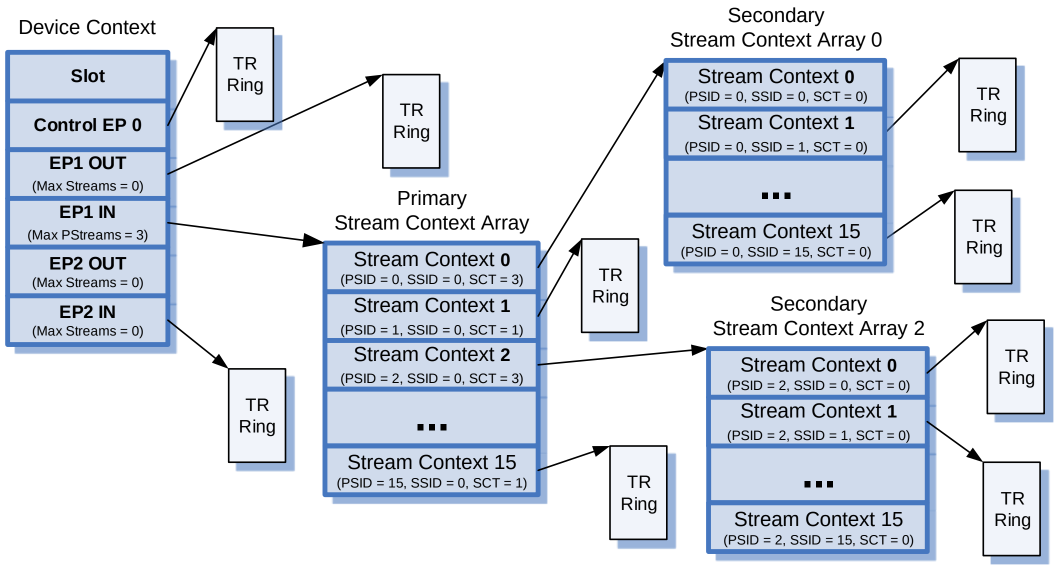
The xHCI architecture provides software with the ability to increase or reduce the number of Streams supported by an xHC Endpoint Context during runtime, and support for the case where a large number of Streams would cause a *Stream Context Array* to exceed a PAGESIZE.

Both of these features are supported through hierarchical *Stream Context Arrays*. With this approach, the Endpoint Context references a *Primary Stream Array*, which in turn may reference a *Secondary Stream Array*. Figure 4-20 illustrates the relationship between the Endpoint Context, the *Primary Stream Context Array*, and the *Secondary Stream Context Array*.

If the *MaxPStreams* field of the *Endpoint Context* is greater than '0', then Streams are supported by the endpoint and the *TR Dequeue Pointer* field points to a *Primary Stream Array* with $2^{MaxPStreams+1}$ entries. Refer to Table 6-8 for the definition of *MaxPStream*.

Note that the *MaxStreams* field of the *SuperSpeed Endpoint Companion Descriptor* identifies the maximum number of Streams that the associated endpoint supports, however software may configure the *Primary Stream Array* of the associated endpoint with less than *MaxStreams* entries and grow the number of hardware supported Streams later.

Figure 4-20: Stream Context Data Structures



In the example of Figure 4-20, to access a specific Stream Context, the xHCI splits the *Stream ID* into two sub-fields; the *Primary Stream ID* (**PSID**) and *Secondary Stream ID* (**SSID**). The *Primary Stream ID* is used as an index into the *Primary Stream Array*. If the *Secondary Stream ID* is equal to ‘0’, then the Stream Context in the *Primary Stream Array* shall contain a pointer to a Transfer Ring (e.g. Primary Stream Context 1 or 15, *SCT* = ‘1’). If the *Secondary Stream ID* is non-zero, then the Stream Context in the *Primary Stream Array* shall contain a pointer to a *Secondary Stream Array* (e.g. Primary Stream Context 0 or 2, *SCT* = ‘3’), and the *Secondary Stream ID* is used as an index into the *Secondary Stream Array*. Also note that the 0th element in *Secondary Stream Context Array 0* (SSID = 0, PSID = 0) does not point to a Transfer Ring because it Stream ID 0 is reserved, however the 0th element in *Secondary Stream Context Array 2* does point to a Transfer Ring because it represents Stream ID 2 (SSID = 0, PSID = 2).

The boundary between the PSID and SSID sub-fields is defined by the *MaxPStreams* field of the *Endpoint Context*, Refer to Table 6-8. The PSID resides in the low order bits of a *Stream ID* and the SSID resides in the high order bits.

All endpoints that declare Streams shall be initialized to point to a *Primary Stream Array*. *Secondary Stream Arrays* may be defined at initialization or run time. Software shall coordinate the allocation of Stream IDs with the *Primary/Secondary Stream Array* layout of an endpoint. Note that in the example of Figure 4-20, Stream Contexts 0 and 2 in the *Primary Stream Context Array* point to *Secondary Stream Context Arrays*. To access a Stream Context in the Secondary Stream Array referenced by Primary Stream Context 0, software shall set the Primary Stream ID to 0, and the Secondary Stream ID to the index of the Secondary Stream Context. Note that the *Stream ID* value ‘0’ (i.e. PSID & SSID = ‘0’) is reserved by the USB3 spec and should never be presented to the xHC by a device that declares a Stream endpoint. Hence in the example of Figure 4-20, *Stream Context 0* in *Secondary Stream Context Array 0* is reserved and shall not be accessed by the xHC.

Note: If *Secondary Stream Arrays* are enabled, then Stream Context 0 of the *Primary Stream Context Array* shall always reference a *Secondary Stream Array* (i.e. *SCT* > ‘1’). An *SCT* value of ‘0’ or ‘1’ may result in undefined behavior.

The value of *MaxPStreams* informs the xHC of the size of the *Primary Stream Array*. If Secondary Streams are enabled, then the maximum size of a *Primary Stream Array* is 256 entries (*MaxPStreams* = ‘7’). The *Stream Context Type* (SCT) field in each Stream Context identifies whether a context in the *Primary Stream Array* points to a Transfer Ring or a *Secondary Stream Array*. The *SCT* field also identifies the number of entries in a *Secondary Stream Array*. This flexible mechanism must be carefully managed by software to ensure that the SIDs that it generates shall not cause the xHC to reference an out-of-range Secondary Stream Context.

The maximum size *Primary Stream Array* supported by an xHC implementation is defined by the *MaxPSASize* field in the HCCPARAMS1 register (refer to Table 5-13).

The *NSS* field in the HCCPARAMS1 register (Table 5-13) identifies whether an xHC implementation supports *Secondary Stream Arrays*.

4.12.2.1 Stream Array Bounds Checking

Stream Array bounds checking shall be supported by the xHCI. This feature ensures that an invalid Stream ID presented by a device or a *Set TR Dequeue Pointer Command* shall not cause the xHC to reference host memory that it doesn't have access to.

The size of the Primary Stream Array shall be determined by *MaxPStreams*.

If Linear Streams are enabled, then the maximum size of a Primary Stream Array shall be 64K entries.

Note: The *Stream ID* values FFFFh (NoStream) and FFFEh (Prime) are reserved by the USB3 spec. Hence, if 64K Stream Contexts are defined, the last two are reserved and shall not be accessed by the xHC.

If Streams are enabled (*MaxPStreams* > '0') then the xHC shall perform the following checks when parsing a Stream ID presented by a USB packet or a *Set TR Dequeue Pointer Command*.

Note: The following tests are defined for a Stream ID presented by a USB packet. If a boundary error is detected on a *Stream ID* presented by a *Set TR Dequeue Pointer Command* a Command Completion Event shall set its Completion Code to *TRB Error*.

- If a *Stream ID* = '0' the xHC shall generate a Transfer Event with the Completion Code set to *Invalid Stream ID Error* and shall halt the endpoint.
- If the *TR Dequeue Pointer* field of a Stream Context data structure equals '0':
  - If the *Stream Context Type* (SCT) equals *Transfer Ring*:
    - The xHC shall interpret the value as an "empty" Transfer Ring and shall not attempt to DMA TRBs from the address and reject the request with a NoStream response.
  - If the *Stream Context Type* (SCT) equals *SSA*:
    - The xHC shall generate a Transfer Event with the Completion Code set to *Invalid Stream Type Error*, shall halt the endpoint, and shall not attempt to DMA a Stream Context data structure from the address.

If Linear Stream Array mode is enabled (*Linear Stream Array*[42] (LSA) flag = '1'):

[42]The *Linear Stream Array* (LSA) field is defined in Table 6-8.

- If a Stream ID is less than the Primary Stream Array size defined by *MaxPStreams* and greater than '0', then the xHC shall check *Stream Context Type* (SCT) of Stream Context data structure in the Primary Context Array as follows:
  - If *Primary:Transfer Ring* (*Stream Context Type*[43] (SCT) field = '1'):
    - The Stream Context is valid.
  - else
    - The Stream Context is not valid and the xHC shall generate a Transfer Event with the Completion Code set to *Invalid Stream Type Error* and shall halt the endpoint.
- If a Stream ID is '0' or greater than or equal to the Primary Stream Array size defined by *MaxPStreams* the xHC shall generate a Transfer Event with the Completion Code set to *Invalid Stream ID Error* and shall halt the endpoint.

If Secondary Stream Arrays are enabled (*LSA* = '0'):

- Use the *MaxPStreams*+1 low order bits of the Stream ID to index into the Primary Stream Array.
  - Check *SCT* field of the Primary Stream Array Stream Context data structure:
    - If *Secondary:Transfer Ring* (*SCT* = '0'):
      - The xHC shall generate a Transfer Event with the Completion Code set to *Invalid Stream Type Error* and shall halt the endpoint.
    - else if *Primary:Transfer Ring* (*SCT* = '1'):
      - If the *SSID* is not '0':
        - The Stream Context is not valid and the xHC shall generate a Transfer Event with the Completion Code set to *Invalid Stream ID Error* and shall halt the endpoint.
      - else
        - The Stream Context is valid.
    - else
      - *Primary:SSA* (*SCT* = '2' to '7').
      - If the *SSID* is '0' or out of range as defined by the *Primary:SCT* Secondary Stream Array Size, then the xHC shall generate a Transfer Event with the Completion Code set to *Invalid Stream ID Error* and halt the endpoint.
      - Check *SCT* of secondary Stream Context data structure:
        - If not *Secondary:Transfer Ring* (*SCT* = '0'):
          - The Stream Context is not valid and the xHC shall generate a Transfer Event with the Completion Code set to *Invalid Stream Type Error* and shall halt the endpoint.
        - else

[43]The *Stream Context Type* (SCT) field is defined in Table 6-13.

- It is a *Secondary:Transfer Ring* type and the Stream Context is valid.

Note: If a non-*CStream* SID is received in the **Move Data** state then the pipe shall halt with a *Invalid Stream Type Error* completion code.

Note: If an NRDY with a non-*Prime* SID is received in the **Prime Pipe** state then the pipe shall halt with an *Invalid Stream Type Error* completion code. The SID shall be ignored if the *Deferred* bit is set in a packet, or an ERDY is received, in the **Prime Pipe** state, and no *Invalid Stream Type Error* shall be generated.

4.12.3 Evaluate Next TRB (ENT)

The *Evaluate Next TRB* (ENT) flag applies to all Transfer Rings, and it is particularly important for Stream Contexts. It provides a means of forcing the execution of a terminating Event Data TRB (4.11.5.2) when a Stream is terminated.

If the device initiates the xSPSM (4.12.1) transition from the **Move Data** to the **Idle** state, the xHC does not have visibility to the conditions that caused it. If the transition is due to a temporary condition e.g. the device needed to switch to a higher priority Stream or flow control the current Stream, then the Stream will be rescheduled at a later time by the device. However, if the transition was due to the device completing the data transfer associated with the Stream, then the Stream may not be scheduled again by the device.

When the transition to the **Idle** state occurs, the xHC is expected to save the state of the Stream (e.g. the Transfer Ring Dequeue Pointer) so that it may pick up where it left off the next time the Stream is scheduled. Note that the transition to the **Idle** state may occur in the middle of a TD, so the saved Stream state shall support the ability to continue a partially completed TD.

If the transition to the **Idle** state was due to one of the temporary conditions described above, then the xHC should wait for the device to reschedule the Stream. However, if the transition to the **Idle** state was due to a completed transfer, then the xHC should complete the TD before saving the Stream state.

If a TD is comprised of one or more Normal TRBs and terminated with an Event Data TRB, then the transition to the **Idle** state (and associated Stream state save) could occur after all the data for the TD has been moved (e.g. after Transfer Event TRBs have been executed), but before the Event Data TRB is executed. Under these conditions, the execution of the Event Data TRB necessary to complete the TD will not occur until the next time the Stream is scheduled. This could lock up the Stream if software was waiting for the TD to complete before scheduling the Stream again.

Before the transitioning a Stream pipe to the **Idle** state, then the xHC shall evaluate the *ENT* flag in the last TRB completed, and if the *ENT* flag is set (‘1’), then the xHC shall evaluate the next TRB before saving the Stream state.

Setting the ENT flag in the last Normal TRB of the TD described above, allows the xHC to execute the terminating Event Data TRB and complete the TD before saving the Stream state, thus eliminating the lock up condition.

Note: System software shall set the *ENT* flag in the last Transfer TRB before a terminating *Event Data TRB* in a TD. This action ensures the timely execution of an Event Data TRB if the Transfer Ring is flow controlled.

When the xHC detects the *Chain* and *ENT* bits both set to '1' in a TRB, it shall evaluate the next TRB. If the next TRB is an *Event Data TRB*, the xHC shall generate the associated *Event Data Transfer Event* before saving the Stream state. If the next TRB is not an *Event Data TRB*, the xHC shall save the Stream state, i.e. evaluate the next TRB the next time the associated Stream is scheduled.

Note: System software should only set the *ENT* flag in a TRB if the next TRB is an *Event Data TRB* and the *Event Data TRB* is the last TRB in a TD. The *ENT* flag does not span TDs, therefore the *ENT* flag is valid only if the *Chain* bit (CH) is '1'.

Note: The *ENT* flag shall "span" a Link TRB if there is a Link TRB between the TRB with the *ENT* flag set and the next Transfer TRB. i,e, if the *ENT* flag is set in a TRB that it is immediately followed by a Link TRB, the xHC shall execute the Link TRB and evaluate the TRB that the Link TRB points to, before advancing to the next endpoint in the Pipe Schedule.

Note: If an endpoint is Halted due to an error while executing a TRB, a Transfer Event shall be generated for that TRB and the xHC is not required to evaluate the *ENT* flag of the TRB that generated the error.

4.13 Device Notifications

The USB3 specification defines a Device Notification Transaction Packet. The Notification Type field in this packet defines 16 possible notification types. Some notification types are handled directly by the xHC and others may be reported to software. The Device Notification Control (DNCTRL) register allows system software to individually select which notifications are important to it and shall generate a Device Notification Event. Refer to section 6.4.2.7 for more information on the Device Notification Event TRB.

Refer to section 7.5.1.6 in the USB3 spec for a complete definition of the various Device Notification packet format and types.

Note: To support debugging, the DNCTRL register allows Device Notification Events to be generated for notification types that are normally only handled by the xHC.

Note: The xHC shall use the Device Slot's Slot Context Interrupter Target field to determine the Event Ring that shall receive the event.

4.13.1 Latency Tolerance Message Handling

Latency Tolerance Messaging (LTM) represents a new, more robust, system technique for managing power consumption on a platform. Current platform power management policies are forced to guess when and for how long to sleep. These guesses usually force the platform to trade power savings at the expense of platform performance, in particular performance of attached devices. LTM adds the capability for attached devices to provide information that can improve the host platform's ability to select when and how long to sleep. This is accomplished by an attached device informing the host of its acceptable service latency between accesses, the device's latency tolerance.

The xHC's role in supporting this new platform capability is to accept latency tolerance values from USB3 devices, evaluate the values and forward the lowest value to the host platform, using the host platform's *Latency Tolerance Reporting* (LTR) mechanism. LTM is optional normative, however shall be supported by any xHC implementation that also supports a corresponding host interconnect LTR mechanism. The form of the LTR mechanism used by the xHC to forward these latency tolerance values to system will be host-specific and will vary based on the interconnect architecture used by the host platform for device communications (e.g. PCI Express, AMBA, etc.). The actual host-specific LTM mechanism for a given platform is outside the scope of this specification.

USB3 defines a complimentary Latency Tolerance Messaging (LTM) mechanism. USB3 LTM is an optional normative USB power management feature that utilizes reported **Best Effort Latency Tolerance** (BELT) values to enable more power efficient platform operation. These messages are supported by USB3 devices (excluding hubs) using an optional USB3 "Device Notification (DEV_NOTIFICATION)" Transaction Packet (TP) with a Notification_Type = LATENCY_TOLERANCE_MESSAGE (LTM). This USB message is also referred to as a Latency Tolerance Message (LTM) TP. The LTM TP contains the BELT value that indicates the current tolerable service latency for that device. Refer to the USB3 Specification (section 8.5.6) for detail on DEV_NOTIFICATION Transaction Packets and the BELT Messaging mechanism.

The *Latency Tolerance Messaging Capability* (LTC) bit in the HCCPARAMS1 register identifies whether the xHC shall support LTM handling. If *LTC* = '1', then an xHC implementation shall support the LTR mechanism as described by the appropriate system bus spec, USB LTM as described in section 8.5.6.4 of the USB3 spec, and the *Set LTV Command* as described in section 4.6.14. If an xHC implementation is designed for a bus/system that does not support an LTR mechanism or decides not to support LTM, then *LTC* shall be '0', and the xHC will not maintain internal LTM related variables described below, and software shall not enable LTM in USB devices.[44]

When the host bus of the platform implements a host-specific LTM mechanism, the xHC shall:

- Maintain an internal **Current BELT** variable, which represents the last BELT value reported to the host. This variable is initialized to the value of *tBELTdefault* (as defined in section 8.13 of the USB3 spec.).
- For each configured USB device, maintain an internal **Device BELT** variable. These variables are initialized to the value of *tBELTdefault*.
- Recognize receipt of an USB *LTM TP*.

Upon receiving an LTM TP the xHC shall determine the lowest service latency value for the attached USB subsystem by performing the following actions:

1. Extract the BELT value and multiplier from the LTM TP.
2. Record the value received for the device in the *Device BELT* variable associated with the device.
3. Compare the *Current BELT* value to each *Device BELT* value.
   a. If a device's *Device BELT* value represents a smaller latency than *Current BELT*, then set *Current BELT* equal to the smallest *Device BELT*.
4. If the *Current BELT* value has been modified, then:
   a. *Format a host-specific Latency Tolerance Reporting (LTR) message for transmission to the host.*
   b. *Place the Current BELT value in the LTR message defined for the host interconnect.*

      Note: Based on the host interconnect used by the platform and the associated LTR mechanism, it may be necessary to translated the BELT value into multiple forms before forwarding to the host.
   c. *Send the LTR message the host.*

Step Record the value received for the device in the requiring that the xHC keep a record of the value received is necessary to enable the comparison operation in step Compare the . In addition, this value shall also be recorded in the event

[44]For example, If LTC = '1', then a PCIe xHC implementation is required to support PCIe Latency Tolerance Reporting (LTR) as described in section 6.18 of the PCIe spec, USB LTM, and the *Set LTV Command*. If the xHC is implemented on a non-PCIe bus then it would use the equivalent LTR mechanism defined for that bus.

that the device is removed and under these circumstances the xHC shall set the *Current BELT* value to *tBELTdefault* and re-evaluate for the lowest latency of the remaining *Device BELT* values by executing Step Compare the and step If the above.

Note: The manner in which the *Current BELT* and *Device BELT* variables are stored is implementation specific and as such falls outside the scope of this specification.

The *Set LTV Command TRB* provides a means for host software to provide its own "*Device BELT"* value. This command is optional normative, however it shall be supported if the xHC also supports a corresponding host interconnect LTM mechanism.

xHCI Device BELT is an internal variable that maintained by the xHC. The *xHCI Device BELT* value is initialized to an "unconfigured" state. While the *xHCI Device BELT* variables is "unconfigured", it is not compared with the other Device BELT variables in Step 3 above.

When a *Set LTV Command* is executed by the xHC:

- The *BELT* field of the *Set LTV Command TRB* is copied to the *xHCI Device BELT* variable. This action transitions the *xHCI Device BELT* variable from "unconfigured" to "configured". When *xHCI Device BELT* is "configured", it is compared with the other Device BELT variables in Step 3 above.
- Re-evaluate for the lowest latency by executing Step 3 and step 4 above.

Refer to section 4.6.14 for more information on the *Set LTV Command*.

Refer to section 6.4.3.13 for more information on the *Set LTV Command TRB*.

Note: The xHC hardware automatically handles LATENCY_TOLERANCE_MESSAGE Device Notifications (*Notification Type* = 2) so there is no need to enable *Device Notification Event* generation for this notification type.

4.13.2 Function Wake

A USB3 device sends a FUNCTION_WAKE Device Notification Transaction Packet to inform the host of a "Function Remote Wake". Software should set flag *N1* in the *DNCTRL* register to enable the generation of Device Notification Events when FUNCTION_WAKE Device Notifications are received.

Note: The FUNCTION_WAKE Device Notification Transaction Packet is used to indicate "Function Remote Wake". A Function Remote Wake is distinct from a "Remote Wake"[45] that is initiated by a low level USB signaling.

[45] Note that section 9.2.5.4 of the USB3 spec defines "remote wake" as being device level wake signaling, "enabled when any function within a device is enabled for function remote wakeup", however the USB2 LPM ECN defines "remote wake" as link level Device Initiated L1 Exit signaling.

Refer to section 8.5.6 of the USB3 spec for more information on FUNCTION_WAKE Device Notification Transaction Packets.

4.14 Managing Transfer Rings

This section presents an overview of how the host controller interacts with Transfer Rings.

A number of terms that are used throughout this section are described below.

System software shall translate the device Endpoint Descriptor (and SuperSpeed Endpoint Companion Descriptors) fields into the appropriate Endpoint Context *Interval*, *Max Packet Size*, *Max Burst Size*, and *Mult* values. Refer to section 6.2.3 for the definition of Endpoint Context.

The xHC uses the *Max Packet Size* and *Max Burst Size* fields in the Endpoint Context to manage transactions on the USB.

Transfer Descriptors (TDs) allow software to define contiguous blocks of data, constructed from non-contiguous host memory buffers, that shall be passed to or from a USB device.

The **TD Transfer Size** is defined by the sum of the *TRB Transfer Length* fields in all TRBs that comprise the TD.

For IN pipes, a device may truncate the data transfer associated with a TD by issuing a Short Packet before the TD is exhausted. In this case the xHC shall retire the TD that received the Short Packet and advance to the next TD on the Transfer Ring or the Enqueue Pointer (i.e.Cycle bit transition), whichever is encountered first.

If the *Interrupt On Completion* (IOC) or *Interrupt-on Short Packet* (ISP) flags are set in the TRB that received the Short Packet, a Transfer Event shall be generated with the Completion Code set to *Short Packet*.

An endpoint is considered **Active** when it is on the xHC's Pipe Schedule, and **Inactive** if it is not. Ringing the Doorbell of an endpoint in the *Running* state will activate it, and the xHC shall place the endpoint in its Pipe Schedule. While the endpoint is *Active* the xHC shall actively process TDs on its Transfer Ring. If the Transfer Ring for the endpoint is exhausted or the endpoint exits the *Running* state, the endpoint is pulled from the xHC's Pipe Schedule and placed in *Inactive* state. Software may ring the Doorbell of an endpoint in the *Running* state to reactive an inactive endpoint.

A **Bus Instance** (BI) represents a "unit" bus bandwidth at the speed that the BI supports. The bit rate cited for a USB bus (e.g. SS 5Gb/s. HS 480Mb/s, etc.) should not be confused with the "Total Available Bandwidth", which is the maximum bandwidth available for actually moving data through a BI.

The **Total Available Bandwidth** identifies a BI's ability to move real data. As rule of thumb, the Total Available Bandwidth will be at least 20% lower than the cited bit rate of a BI, or more depending on the mix of packet sizes. Also note that multiple Root Hub ports may share the bandwidth of a single BI. The mapping of BI to Root Hub ports is xHC implementation dependent and not exposed to software.

During each IN transaction, the xHC shall use the *Max Packet Size* to detect Packet Babble errors. If a babble error is detected, a Transfer Event shall be generated for the offending TRB, with the *Completion Code* set to *Babble Detected Error*.

When the xHC detects that a Transfer Ring will be exhausted after the execution of a TP or DP (e.g. the last packet of the last TRB of the last TD on a Transfer Ring), it should clear the ACK TP or DP *Packet Pending* (PP) bit to '0'. If *Max Exit Latency* is greater than '0', then the xHC should clear the *Packet Pending* flag in the last packet of each Isoch TD. The *Packet Pending* bit shall be set to '1' in all other ACK TPs or DPs generated by the xHC.

4.14.1 General Scheduling Model

When a doorbell is rung for a *Running* endpoint, the xHC places the endpoint on a **Pipe Schedule**. An xHC will typically maintain two Pipe Schedules per Bus Instance, one for periodic pipes (Isoch and Interrupt endpoints) and another for async pipes (Control and Bulk).

Each pass through a Pipe Schedule an endpoint is given one "Service Opportunity". A **Service Opportunity** (SO) is a block of time that the xHC allocates for moving packets on USB, for a specific endpoint.

Depending on the endpoint type and settings,1 to 3 USB Transactions may be executed during a Service Opportunity (SO). USB Standard Transactions transfer a single Data Packet (DP), however a single USB Burst Transaction may transfer multiple DPs.

The **Max Service Opportunity Packet Count** (MSOPC) is the maximum number of DPs that the xHC shall schedule during one Service Opportunity (SO). The MSOPC value for an endpoint is set by the number of packets defined by the Endpoint Context *Max Burst Size* field times the *Mult* field.

The **Transfer Descriptor Packet Count** (TDPC) is the number of packets required to move all the data defined by a TD. Note that a partial or a zero-length packet increments this count by 1.

The **Transfer Ring Packet Count** (TRPC) is the sum of the TDPCs for all TDs on a Transfer Ring.

The **Service Opportunity Packet Count** (SOPC) is the number of packets actually scheduled by the xHC during a SO. The SOPC value shall be initialized at the beginning of a SO, and decremented as each transaction or retry of the SO is completed. When SOPC reaches zero the SO for the current endpoint is complete, the xHC shall initiate a SO for the next endpoint in the schedule. Retries may terminate the current SO and continue on the next SO.

Normally SOPC is less than or equal to MSOPC, however the xHC is allowed to limit the SOPC to a value less that MSOPC. And if only one endpoint is in the Pipe Schedule SOPC may be greater than MSOPC, e.g. a continuous burst on the bus. Refer to the individual pipe type discussions below for more details on SOPC usage.

The endpoints assigned to a periodic schedule are closely controlled by the xHC through the *Address Device* and *Configure Endpoint Commands* to ensure that the periodic Pipe Schedule consumes no more than a maximum percentage of the *Total Available Bandwidth*. Any USB bandwidth not consumed by periodic pipes, is available to async pipes.

Note: The “maximum percentage” of the *Total Available Bandwidth* depends on the speed of the periodic pipe. Refer to section 4.14.2 for more information.

The endpoints assigned to an async schedule are considered “Best Effort” and may consume any USB bandwidth not consumed by periodic pipes. Each endpoint in an async Pipe Schedule is given one Service Opportunity (SO) per pass through the schedule.

4.14.1.1 System Bus Bandwidth Scheduling

System bus bandwidth is limited, especially in cases where the xHC is connected to a system by a bus that provides less bandwidth than the USB bus instances that it supports. To ensure consistent and reliable operation of USB endpoints the xHC shall manage the system bus activity associated with an endpoint using methods that are similar to the way that it manages the USB bandwidth associated with an endpoint.

For example, given the system bus bandwidth available to the xHC it shall distribute that bandwidth across its active endpoints. Periodic endpoints will have priority over async endpoints, and all async endpoints will be given fair access to the remaining system bus bandwidth.

The xHC uses the value of the *Average TRB Length* field in the Endpoint Context as a metric to help compute the system bus bandwidth requirements of an endpoint. The accuracy of this parameter is particularly important for periodic endpoints. An xHC will use the *Average TRB Length* and other metrics to allocate/distribute system bus bandwidth to endpoints. These “other” metrics are xHC implementation specific and outside the scope of this specification. The *Average TRB Length* field is computed by dividing the average *TD Transfer Size* by the average number of TRBs that are used to describe a TD, including Link, No Op, and Event Data TRBs.

A *Configure Endpoint Command* may be rejected by the xHC with a *Bandwidth Error* or a *Secondary Bandwidth Error* if it determines that there is not enough system bandwidth available for it.

IMPLEMENTATION NOTE

TRB Lengths and System Bus Bandwidth

System buses are most efficient when they are moving large transfers. As transfer sizes become smaller, the throughput of a bus can fall off rapidly.

The xHCI supports byte granularity for the TRB *Data Buffer Pointer* and *Length* fields, which enables “fine-grain” scatter/gather operations. The threshold where it is more efficient to declare many small TRBs and allow the xHC to use DMA to scatter/gather data vs. having software copy that data to/from larger buffers will depend on many factors (e.g. the xHC implementation, system I/O bus performance, system memory performance, etc.). The xHCI does not place lower limits on TRB sizes, which could constrain the ability of a system developer to optimize the performance/throughput of their entire system. However, an xHC will place limits on the system bus bandwidth allocated to an individual endpoint, to ensure that other endpoints are not affected by an endpoint that requires disproportionately large number of system bus transactions to complete its USB transactions.

A programmer should assume that defining large numbers of small TRB Data Buffers will affect USB throughput and design accordingly. The extent to which the system bandwidth demands of a single endpoint will affect that endpoint or other endpoints is xHC implementation dependent.

Note that an *Average TRB* Length of 16 implies that 50% of the system bus bandwidth consumed by an endpoint moving TRBs, i.e. each 16 byte TRB defines 16 bytes of data. And an *Average TRB Length* of 1024 implies that 1.5% of the system bus bandwidth consumed by an endpoint moving TRBs. Ideally the *Average TRB Length* represents the true average size of the data buffers that the TRBs of an endpoint reference, which will generally be a class specific or application specific value. If precise values for the *Average TRB Length* of an endpoint are not available, software may calculate a running average of the size of TRBs scheduled for an endpoint in real-time and periodically updating *Average TRB Length*. Reasonable initial values of *Average TRB Length* for Control endpoints would be 8B, Interrupt endpoints 1KB, and Bulk and Isoch endpoints 3KB.

4.14.2 Periodic Transfer Ring Scheduling

Isoch and Interrupt endpoints define “periodic” transfers. Periodic transfers provide guaranteed bandwidth on the USB.

A **Periodic TD** is an Isoch TD or a TD scheduled on an interrupt endpoint Transfer Ring.

A **Periodic Pipe** is an Isoch or interrupt endpoint.

The *Microframe Index Register* (MFINDEX) is advanced at the **Minimum Interval Time** (MIT). The MIT is equal to 125 µs., corresponding to High-Speed and SuperSpeed microframe timing. The time that the *Microframe Index Register* is advanced, is defined as the **MIT Boundary**.

The MIT multiplied by the Endpoint Context ***Interval*** field as a base 2 exponent, defines ***Endpoint Service Interval Time*** (ESIT).

ESIT = 2^{Interval} * 125 µs.

All *ESITs* are temporally aligned with MIT Boundaries.

The xHC uses the *Max Endpoint Service Time Interval Payload* (Max ESIT Payload) and *Interval* fields in the Endpoint Context to compute the USB bandwidth that it shall reserve for a periodic endpoint. A periodic pipe may, on an ongoing basis, use less bandwidth than that reserved. A USB device reports, if necessary, the actual bandwidth used via its normal, non-USB defined mechanisms.

Software shall define the maximum periodic payload per ESIT as follows for USB2 periodic endpoints:

Max ESIT Payload in Bytes = *Max Packet Size* * (*Max Burst Size* + 1).

Software shall define the maximum periodic payload per ESIT as follows if the *SuperSpeed Endpoint Companion:bmAttributes:SSP ISO Companion* bit is cleared (0):

Max ESIT Payload in Bytes = SuperSpeed Endpoint Companion Descriptor:wBytesPerInterval.

Software shall define the maximum periodic payload per ESIT as follows if the *SuperSpeed Endpoint Companion:bmAttributes:SSP ISO Companion* bit is set (1):

Max ESIT Payload in Bytes =
SuperSpeedPlus Isochronous Endpoint Companion
Descriptor:dwBytesPerInterval.

Refer to section 6.2.3.8 for more information on *Max ESIT Payload*.

Note: Undefined behavior may result if an Isoch TD is encountered which defines more that **Max ESIT Payload** bytes.

The xHC bandwidth calculation for a periodic endpoint is defined as follows:

Reserved Bandwidth in MBytes/s = *Max ESIT Payload* / ($2^{\textit{Interval}}$ * 0.000125)

Per the USB specifications, the **Maximum Allowed ESIT Payload** of a FS Interrupt, FS Isoch, HS Interrupt, HS Isoch, SS Interrupt, or SS Isoch periodic pipe are defined as 64B, 1KB, 3KB, 3KB, 3KB, and 48KB, respectively.

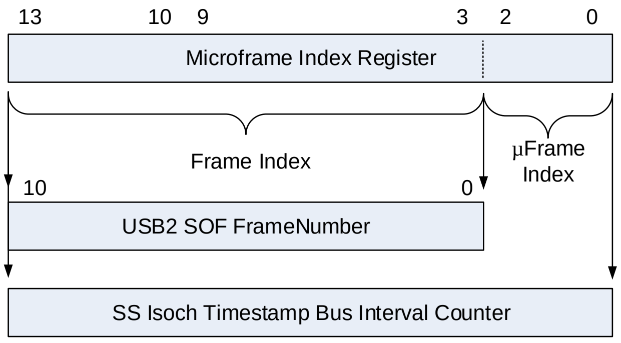
The maximum percentage of Total Available Bandwidth depends on the speed of the BI. The USB requires that no more than 90% of any frame be allocated for periodic (isochronous and interrupt) transfers for SuperSpeed and full-speed endpoints. High-speed endpoints shall allocate at most 80% of a microframe for periodic transfers.

The xHC is free to schedule a isoch transfer at any time within an ESIT as long as the complete TD shall have an opportunity to complete within the ESIT.

For SuperSpeed pipes, if the Endpoint Context *Max Exit Latency* field is greater than '0', the xHC shall transmit a PING packet a minimum of *Max Exit Latency* prior to initiating an Isoch transfer, to transition the links in the path between the xHC and the device to the U0 state. PING generation is optional for Interrupt endpoints. Refer to section 4.23.5.2 for more information on *Max Exit Latency* and its computation.

The Microframe Index (MFINDEX) register is incremented at the beginning of each microframe. Figure 4-21 illustrates the required relationships between the USB2 SOF FrameNumber and the SS Isoch Timestamp (ITS) *Bus Interval Counter* field (refer to section 8.7 of the USB3 spec) 1/8th ms. counter values, and the MFINDEX register. Figure 4-21 also illustrates the partitioning of the *Frame Index* and *µFrame Index* fields of the MFINDEX register.

Figure 4-21: Microframe Index (MFINDEX) Register Mapping


13
10
9
3
2
0
Microframe Index Register
Frame Index
µFrame Index
10
0
USB2 SOF FrameNumber
SS Isoch Timestamp Bus Interval Counter

To enable software computation of larger Microframe Index values, *MFINDEX Wrap Events* may be enabled. If enabled, a *MFINDEX Wrap Event* is inserted on the Event Ring of the Primary Interrupter every time the MFINDEX register wraps from 03FFFh to 0. Refer to section 6.4.2.8 for a description of the *MFINDEX Wrap Event*. Refer to the definition of the USBCMD register (5.4.1) for details on the *Enable Wrap Event* (EWE) flag that may be used to enable *MFINDEX Wrap Events*.

Note: If the target Event Ring is full, *MFINDEX Wrap Events* shall be dropped by the xHC.

If all Root Hub ports are in the **Disconnected**, **Disabled**, **Training**, or **Powered-off** state the MFINDEX counting action may be stopped by the xHC to reduce power consumption. The *EU3S* flag in the USBCMD register may be used to optionally add the **U3** state to list of port states that enable the counting action to be stopped. Exiting any of these states on any port shall automatically restart the MFINDEX counting action.

Refer to section 4.11.2.5 for more information on the use of the MFINDEX register.

4.14.2.1 Isochronous Transfer Ring Scheduling

This section defines the xHCI operational model for isochronous Transfer Rings.

If an Isoch Endpoint Context is *Active*, the xHC shall process one Isoch TD from its Transfer Ring each ESIT.

Software shall not define a *TD Transfer Size* for a TD of an Isoch endpoint that exceeds the *Max ESIT Payload*.

The xHC may schedule multiple Service Opportunities (SOs) per ESIT.

SOPC is set to the smaller of TDPC or MSOPC.

The xHC shall compute the *TD Transfer Size* as it processes a TD. If in the process of executing the TRBs of the TD the *TD Transfer Size* exceeds the *Max ESIT Payload* or the *Maximum Allowed ESIT Payload*, then a *Bandwidth Overrun Error* shall be generated for the offending TRB and the xHC shall advance its Dequeue Pointer to the next Isoch TD boundary or the Enqueue Pointer (i.e.Cycle bit transition), whichever is encountered first. Note that the pipe remains Active after this error, the xHC simply truncates the transfer and advances to the next TD.

If the Transfer Ring is empty and there is no TD defined to receive Isoch IN data, the xHC shall remove the endpoint from the periodic schedule and generate a single Transfer Event with the *Completion Code* set to *Ring Overrun*.

If the Transfer Ring is empty and there is no TD defined to transmit Isoch OUT data, the xHC shall remove the endpoint from the periodic schedule and generate a single Transfer Event with the *Completion Code* set to *Ring Underrun*.

Ringing the doorbell of a periodic endpoint that has encountered a *Ring Overrun* or *Ring Underrun* condition shall place it on back on the periodic schedule.

Interval values are limited to base 2 multiples. An **ESIT Boundary** is defined by when the least significant bits of the MFINDEX register transition to '0'. e.g. if the *Interval* equals 2 microframes, the *ESIT Boundary* is defined by the transition of the least significant bit of the MFINDEX register to '0'. If the *Interval* equals 4 microframes, the *ESIT Boundary* is defined by the transition of the least significant two bits of the MFINDEX register to '0'. And so on.

Note: Section 8.12.6 of the USB3 spec states that "If there is no data to send to an isochronous OUT endpoint during a service interval, the host does not send anything during the interval." The USB2 spec is silent on this subject. When xHC encounters a zero-length Isoch OUT TD on a Transfer Ring, it shall transmit a zero-length DP to the USB bus regardless bus speed, consuming the Isoch TD for the Service Interval. If the Transfer Ring is empty when the xHC attempts to service an Isoch TD, no DPs shall be sent, and an *Underrun Event* shall be generated.

4.14.2.1.1 High-speed endpoints

The USB Endpoint Descriptor (refer to section 9.6.6 in the USB2 spec.) wMaxPacketSize field for a high-speed isochronous endpoint is divided into two fields: the **Maximum Packet Size** (bits 0-10), and the **Multiplier** field (bits 11-12). High-speed USB devices support "high-bandwidth" pipes via the Multiplier field. The USB2 Maximum Packet Size and Multiplier bit fields of the wMaxPacketSize fields are separated and passed to the xHC through the Endpoint Context *Max Packet Size* and *Max Burst* fields respectively.

For high-speed devices, the xHC shall execute the specified number of *Max Packet Sized* bus transactions specified by the *Max Burst Size* field in a single microframe (MIT). The TD is used to service all transactions indicated by the *Max Burst* field.

The maximum sized High-speed isochronous packet size supported is 1024 bytes. The *Max Burst Size* field may define up to up to 3 contiguous packets in a burst.

For OUT transfers, the xHC shall transmit data packets with data fields less than or equal to the endpoint's *Max Packet Size*. If a TD defines more information than will fit into the *Max Packet Size* and the *Max Burst Size* is greater than '0', the xHC shall transmit up to *Max Burst Size*+1 consecutive packets on the USB to move the TD data. If more than one *Max Packet Size* packet is required to move the data defined by a TD, then all packets associated with the TD are transmitted as a contiguous burst in a single microframe of the ESIT. When all bytes have been transmitted for an Isoch TD the xHC advances its Dequeue Pointer to the next TD and waits for the next ESIT delay before scheduling the endpoint again.

For IN transfers, the xHC may issue up to *Max Burst Size*+1 IN transactions of *Max Packet Size* for a single Isoch TD. It is assumed that software has properly initialized the Isoch TD to accommodate all of the possible data that may be received in an ESIT. During each IN transaction, the xHC shall use *Max Packet Size* to detect Packet Babble errors.

For IN transfers, the xHC keeps the sum of bytes received in an internal TD Payload Length register. After all transactions for the endpoint have completed for the ESIT, the local TD Payload Length register contains the total bytes received. If the final value of local TD Payload Length register is less than the value of TD Transfer Size, then less data than was allowed for was received from the associated endpoint. This Short Packet condition shall assert a *Short Packet* completion code only if the *ISP* or *IOC* flag was set on the TRB that the Short Packet condition was detected on. If the device sends more than Max Packet Size bytes, then the xHC shall generate a Transfer Event with the Completion Code set to *Babble Detected Error* for the TRB that the error was detected on. Refer to section 4.10.2.4 for more information on Babble Error handling.

If the *Max Burst Size* field is greater than '0', then the xHC shall automatically attempt to execute *Max Burst Size*+1 transactions on the USB. The xHC shall not execute all *Max Burst Size* transactions if:

- The endpoint is an OUT and the TD is exhausted before all the transactions of the burst have executed (e.g. ran out of data).
- The endpoint is an IN and the endpoint delivers a Short Packet, or an error occurs on a transaction before all the transactions of the burst have been executed.
- The endpoint is an IN and the TD is exhausted before all the transactions of the burst have executed (e.g. ran out of buffer space). This condition shall result in the xHC terminating the Isoch TD with a *Isoch Buffer Overrun* Transfer Event.

Note: The *Isoch Buffer Overrun* condition shall force a Transfer Event for the TRB, irrespective of the state of the *IOC* flag. System software may determine whether to treat this condition as an error or not.

Refer to Appendix B for a table summary of the host controller required behavior for all the High-speed USB2 high-bandwidth transaction cases.

4.14.2.1.2 Full-speed or High-speed endpoints

The end of a microframe may occur before all packets have been executed for a high-speed or full-speed endpoint. When this happens, the xHC shall terminate the Isoch TD with a *Missed Service Error* Transfer Event.

4.14.2.1.3 SuperSpeed endpoints

If the bMaxBurst field of the SuperSpeed Endpoint Companion Descriptor is greater than '0', the SuperSpeed endpoint supports "high-bandwidth" pipes. Software shall pass the bMaxBurst value to the xHC through the Endpoint Context *Max Burst Size* field.

Additionally, the Mult value defined in bits 1:0 of the SuperSpeed Endpoint Companion Descriptor bmAttributes field identifies the number of Bursts within an ESIT that the device supports. This value is passed to the xHC through the Endpoint Context *Mult* field. Note that the range of values for the Mult field is limited by the USB3 spec to ‘0’ to ‘2’, or 1 to 3 bursts.

The maximum sized SuperSpeed isochronous packet size supported is 1024 bytes. The *Max Burst Size* field may define up to up to 16 contiguous packets in a burst, and the *Mult* field may allow up to 3 bursts in an ESIT, allowing for up to 48KB per ESIT.

For OUT transfers, the xHC shall transmit data packets with data fields less than or equal to the endpoint’s *Max Packet Size*. If a TD defines more information than will fit into the *Max Packet Size* and the *Max Burst Size* is greater than ‘0’, the xHC may transmit a burst of up to *Max Burst Size*+1 consecutive packets in a single MIT. If a TD defines more information than will fit into a single burst and *Mult* is greater than ‘0’, the xHC shall transmit up to *Mult*+1 bursts in an ESIT. When all bytes have been transmitted for an Isoch TD the xHC advances its Dequeue Pointer to the next TD and waits for the next ESIT delay before scheduling the endpoint again.

For IN transfers, the xHC may issue up to (*Max Burst Size* + 1) * (*Mult* + 1) IN transactions of *Max Packet Size* for a single Isoch TD. It is assumed that software has properly initialized the Isoch TD to accommodate all of the possible data that may be received in an ESIT. During each IN transaction, the xHC shall use *Max Packet Size* to detect Packet Babble errors.

Refer to section 8.12.6.1 of the USB3 spec for more information on xHC execution of SuperSpeed isochronous transactions.

For IN transfers, the xHC keeps the sum of bytes received in an internal TD Payload Length register. After all transactions for the endpoint have completed for the ESIT, the local TD Payload Length register contains the total bytes received. If the final value of local TD Payload Length register is less than the value of TD Transfer Size, then less data than was allowed for was received from the associated endpoint. This Short Packet condition shall assert a Short Packet completion code only if the *ISP* or *IOC* flag was set on the TRB that the Short Packet condition was detected on. If the device sends more than TD Transfer Size or *Max Packet Size* bytes (whichever is less), then the xHC shall generate a Transfer Event with the Completion Code set to *Babble Detected Error* for the TRB that the error was detected on. Note, that the xHC is not required to update the Transfer Event *TRB Transfer Length* field in this error scenario. Refer to section 4.10.2.4 for more information on Babble Error handling.

The host controller shall not execute all (*Max Burst Size* + 1) * (*Mult* + 1) transactions if:

- The endpoint is an OUT and the TD is exhausted before all the transactions of the burst have executed (ran out of data), or
- The endpoint is an IN and the endpoint delivers a Short Packet, or an error occurs on a transaction before all the transactions of the burst have been executed.
- The endpoint is an IN and the TD is exhausted before all the transactions of the burst have executed (e.g. ran out of buffer space). This condition shall result in the xHC terminating the Isoch TD with a *Isoch Buffer Overrun* Transfer Event.

In addition to the Microframe Index (MFINDEX) register, the xHC shall maintain a 13 bit **Delta Time** down-counter that is cleared to ‘0’ at the *MIT boundary* and incremented every 16.666~ ns. (i.e. 8 HS bit times). The *Delta Time* counter identifies the delay, in 16.666~ ns. increments, between the start of the current packet to the previous *MIT Boundary*. Note: A value of 7500 is reported if the *Delta Time* counter is sampled exactly on a *MIT Boundary*.

The value of the *Microframe Index* (MFINDEX) register shall be written to bits 13:0 and the value of the *Delta Time* register shall be written to bits 26:14 of the *Isochronous Timestamp* (ITS) field of Isochronous Timestamp Packets (ITP) when they are sent. Refer to the USB3 specification section 8.7 for more information on the ITP and the required accuracy of the ITS field.

- If an Isoch IN Transfer Ring is Active and the xHC is unable to send an isochronous IN request (ACK TP) during an ESIT, (due to problems such as internal buffer overrun, excessive DMA access latency, etc.) the xHC shall set the *Completion Code* to *Data Buffer Error* in the Transfer Event generated for the associated Isoch TD. Note that this is an error condition that should never occur.
- If an Isoch OUT Transfer Ring is Active and the xHC is unable to send an isochronous OUT DP data during an ESIT (due to problems such as internal buffer overrun, excessive DMA access latency, etc.), the xHC discards the data and notifies software by setting the *Completion Code* to *Data Buffer Error* in the Transfer Event generated for the associated Isoch TD. Note that this is an error condition that should never occur.
- If the xHC receives a corrupted data packet, it discards the data and informs software by setting the *Completion Code* to *USB Transaction Error* in the Transfer Event generated for the associated Isoch TD.

Note: An IN Isoch endpoint may set the Completion Code to *Isoch Buffer Overrun* if the *Last Packet Flag* (LPF) is not set in the last DP received in a Service Interval. Refer to section 8.6 in the USB3 spec for more information on LPF.

4.14.2.1.4 Isochronous Scheduling Threshold

The *Isochronous Scheduling Threshold* (IST) field in the HCSPARAMS2 capability register is an indicator to system software as to how the host controller pre-fetches and caches TRB structures. It is used by system software when adding isochronous work items. The value of this field indicates to system software the minimum distance (in time) that it is required to stay ahead of the host controller while adding TRBs in order to have the host controller process them at the correct time. In other words, software shall add a TRB to the ring some period of time before that TRB is required to be executed, and the *IST* indicates a *minimum* value for this period of time as required by the specific host controller hardware implementation.

Software shall determine the host controller's current frame/microframe by reading the MFINDEX register, to account for the uncertainty in the actual read latency and position within the microframe, software shall always add a value of one microframe to the value read.

It is recommended that software post sufficient TRB(s) to the ring to allow uninterrupted processing by the host controller. This may be accomplished by always placing multiple TD(s) on the ring that either exceed the time window represented in the IST field or exceeds the round-trip delay in the host software, which ever is greater.

The *Isochronous Scheduling Threshold* (IST) field definition can be found in section 5.3.4.

A value of '2' in the Isochronous Scheduling Threshold (IST) field indicates that software can add a TRB no later than 2 microframes before that TRB is due to be executed.

If bit [3] of IST is cleared to '0', software can add a TRB no later than IST[2:0] Microframes before that TRB is scheduled to be executed.

If bit [3] of IST is set to '1', software can add a TRB no later than IST[2:0] Frames before that TRB is scheduled to be executed.

Note: Undefined behavior may result if a partially formed Isoch TD is encountered, i.e. the enqueue pointer (Cycle bit transition) is encountered before the end of the TD (*Chain* = '0'). This condition may occur if software fails to honor the *IST*.

Note: Ideally the *IST* value declared by an xHC implementation represents a worst case latency, however the xHC may encounter system latencies that cause it to skip a scheduled Isoch TD even if software has met the IST requirements. These conditions are normally indicated as a *Missed Service Error*. If *Missed Service Errors* persist, software may choose to use a larger value for IST than that reported by the xHC.

4.14.3 Interrupt Transfer Ring Scheduling

The value of the Endpoint Context *Interval* field is treated as a throttling parameter or a deadline by the xHC for Interrupt endpoints. The following rules apply to Interrupt Transfer Ring scheduling:

- If an interrupt transfer ring has been idle, the maximum time between the xHC receiving a doorbell ring for the endpoint and scheduling the first associated interrupt transaction on USB for the first TD posted to Transfer Ring shall be equal to IST + ESIT.

- If multiple Interrupt TDs are posted to an Interrupt endpoint Transfer Ring, the xHC should consume no more than one TD per ESIT.
- Software may define a *TD Transfer Size* for a TD of an Interrupt endpoint that exceeds the *Max ESIT Payload*.
- An Interrupt pipe executes a single SO per ESIT.
- SOPC is set to the smaller of TDPC or MSOPC.
- An Interrupt pipe shall transmit or receive no more that one *Max ESIT Payload* per ESIT, e.g. if the Interrupt *TD Transfer Size* is greater than the *Max ESIT Payload*, then the TD may take multiple ESITs to complete.
- A Short Packet shall terminate an IN Interrupt TD and the next TD (if present) shall be scheduled in the next ESIT.
- Unexpected ERDYs shall be silently dropped.

Note: Since Interrupt pipes provide reliable data delivery but the number of packets (including retries) per ESIT is limited by the value of *MSOPC*, packet retries may cause an Interrupt TD to require more ESITs than expected to complete. If a second TD is pending on the Transfer Ring when this condition occurs, it shall be delayed until the first TD is successfully transferred.

To minimize the latency impact of retries on an Interrupt pipe, up to MSOPC packets (including retires) may be transferred in an ESIT even if the initial SOPC value was less than MSOPC.

An xHC implementation may exceed MSOPC packets per ESIT if it can guarantee that additional packets do not affect the bandwidth guarantees that have been established with other periodic endpoints.

4.14.3.1 Low-, Full-, and High-speed Endpoints

- Interrupt IN pipes
  - If an IN transaction is NAKed, then the Interrupt TD will be retried in the next ESIT.
  - If the IN transaction times out, then the xHC shall retry the transaction for the endpoint *CErr* times in the same ESIT if possible, or if the maximum number of transactions per microframe has been reached, the xHC shall retry the transaction in the next ESIT. If *Bus Error Counter* = 0, the endpoint shall halt.
- Interrupt OUT pipes
  - If an OUT transaction is NAKed, then the xHC shall not issue another transaction for the endpoint until 1 ESIT later.

- If the OUT transaction times out, then the xHC shall retry the transaction for the endpoint *CErr* times in the same ESIT if possible, or if the maximum number of transactions per microframe has been reached, the xHC shall retry the transaction in the next ESIT. If *Bus Error Counter* = 0, the endpoint shall halt.

For High Bandwidth endpoints, the Endpoint Context *Max Burst Size* field specifies the maximum number of desired transactions per microframe. If the maximum number of transactions per microframe has not been reached, the xHC may immediately retry a transaction that failed during the current microframe. If possible an xHC implementation should attempt an immediate retry of a failed transaction since this minimizes impact on devices that are bandwidth sensitive. If the maximum number of transactions per microframe has been reached, the xHC shall retry the failed transaction at the next ESIT for the endpoint.

Note that for a high-bandwidth interrupt OUT endpoint, the host controller may optionally immediately retry the transaction if it fails.

The xHC is allowed to issue less than the maximum number of transactions to an endpoint per microframe only if the TD Transfer Size is less than the Max ESIT Payload.

Normal DATA0/DATA1 data toggle sequencing is used for each interrupt transaction during a microframe.

Refer to Table 4-7 for HS/FS Interrupt pipe actions based on Endpoint Response and Residual Transfer State.

Refer to Appendix B for a table summary of the host controller required behavior for all the high-bandwidth transaction cases.

4.14.3.2 SuperSpeed Endpoints

- ESIT*2 defines the maximum latency between an ERDY and an OUT DP or IN TP being scheduled to a SS Interrupt endpoint.
- Interrupt IN pipes
  - If Interrupt IN TDs are available, the xHC shall issue ACK TPs to the interrupt endpoint at one ESIT or less intervals.
  - If an IN request is responded to with an NRDY, then the xHC shall wait indefinitely for a ERDY from the endpoint. System software is responsible for any timeouts. The only exception to this rule is when an endpoint that has been flow controlled by an NRDY is stopped with a *Stop Endpoint Command* then restarted by ringing its doorbell. When the endpoint transitions to the *Running* state, it checks its Transfer Ring and if a TD exists, it shall issue an IN.
  - Once the xHC receives the ERDY TP, it shall send an IN request (via an ACK TP) to the device no later than 2 x ESIT.

- If the xHC is unable to accept a valid Data Packet from a device due to internal issues (e.g. internal buffer overrun, etc.), it shall set the ACK TP *Host Error* (HE) bit to '1'.

- Interrupt OUT pipes
    - If an OUT DP is responded to with an NRDY, then the xHC shall wait indefinitely for a ERDY from the endpoint. System software is responsible for any timeouts. The only exception to this rule is when an endpoint that has been flow controlled by an NRDY is stopped with a *Stop Endpoint Command* then restarted by ringing its doorbell. When the endpoint transitions to the *Running* state, it checks its Transfer Ring and if a TD exists, it shall issue an OUT.
    - If a DP was received by the device with an error, the Retry bit shall be set in the returned ACK TP and the xHC should retry the same DP by the next ESIT at the latest.
    - If an OUT DP is responded to with a STALL TP, the xHC shall set the Halted flag for the EP to '1' and pull the endpoint from the Pipe Schedule. USB System Software intervention is required to recover from the error.

Refer to Table 4-8 for SS Interrupt pipe actions based on Endpoint Response and Residual Transfer State.

4.14.4 Asynchronous Transfer Ring Scheduling

Control and Bulk endpoints define "Asynchronous" transfers. Async endpoints provide "best effort" delivery of their data. As such, their delivery delays are not bounded.

An **Async TD** is a TD scheduled on a control or bulk endpoint Transfer Ring.

An **Async Pipe** is a control or bulk endpoint.

To ensure fairness across the pipes in the async schedule, the xHC shall schedule Service Opportunities for each Async Pipe using a round-robin algorithm. The maximum amount of async data moved for an Async Pipe during a Service Opportunity is called the *Max Service Transfer Size*, and is defined by an Endpoint Context's *Max Packet Size* and *Max Burst Size* fields.

Max Service Transfer Size = *Max Packet Size* * *Max Burst Size*

If the *Max Service Transfer Size* is greater than or equal to the *TD Transfer Size* then one Service Opportunity is used to move the TD data. If the *Max Service Transfer Size* is smaller than the *TD Transfer Size* then multiple service opportunities will be necessary to move the TD data.

The xHC is allowed to schedule less packets during an Async Pipe Service Opportunity than allowed for by the *Max Burst Size*.

If async schedule execution is interrupted by periodic transfers, the xHC shall retain an identifier for the next Async Pipe to be executed. When the asynchronous schedule is restarted, this shall be the first Async Pipe that will be serviced.

The order of Async Pipe execution on the async schedule is xHC determined.

Each Async Pipe is only given one Service Opportunity per pass through the async schedule.

Each Stage of a control transfer is a different Async TD, and may be scheduled during different Service Opportunities.

If there is more than one endpoint in the async schedule the xHC shall limit the number of packets transferred during a Service Opportunity (SO) to MSOPC. However, if only one endpoint is in the async schedule, the xHC may exceed the default MSOPC and continuously stream packets to an endpoint. The xHC shall interrupt a continuous stream when a second endpoint is scheduled and revert to the MSOPC packet limit per endpoint SO.

Note: Retries are counted against the EPs SOPC. e.g. If an error is detected on the last packet of the SO, then the xHC shall advance to the next EP and the packet shall be retried at the beginning of the next SO for the endpoint.

Table 4-7: USB2 Pipe Actions based on Endpoint Response and Residual Transfer State

Direction	Endpoint Response	Transfer State after Transaction (Bytes to transfer)	Pipe Action
IN	Data Packet *Max Packet Size*	Not Zero	Decrement *SOPC*. If *SOPC* = 0: Advance to next endpoint. else Continue moving endpoint packets.
		Zero	Retire TD. Advance to next endpoint.
	Data Packet Short	Don't care	Retire TD. Advance to next endpoint.
	NAK	Don't care	Advance to next endpoint.
	Stall or Babble	Don't care	Note 8-1.

	CRC or Bad PID error	Don't care	Discard packet. Note 8-2.
	Timeout	Don't care	Note 8-2.
OUT	ACK	Not Zero	Decrement *SOPC*. If *SOPC* = 0: Advance to next endpoint in schedule. else Continue moving endpoint packets.
		Zero	Retire TD. Advance to next endpoint.
	NYET, NAK	Don't care	Advance to next endpoint.
	Stall or Babble	Don't care	Note 8-1.
	CRC, Timeout, or Bad PID error	Don't care	Note 8-2.
PING	ACK	Not Zero	Allowed to transfer up to *SOPC* packets.
	NAK	Don't care	Advance to next endpoint.
	Stall	Don't care	Note 8-1.
	CRC, Timeout, or Bad PID error	Don't care	Note 8-2.

Note 8-1:

```
If Stall
    Generate *Stall Error* Transfer Event.
else
    Generate *Babble Detected Error* Transfer Event.
Set endpoint to the *Halted* state.
Pull endpoint from Pipe Schedule.
Advance to next Async Pipe.
```

Note 8-2:

```
Decrement the Bus Error Counter.
If Bus Error Counter = '0':
        Generate USB Transaction Error Transfer Event
        Set endpoint to the Halted state.
        Pull endpoint from Pipe Schedule.
        Advance to the next endpoint in the Pipe Schedule.
    else
        If IN or OUT endpoint, do not advance Data Toggle.
        Decrement SOPC.
        If SOPC = 0:
            Advance to the next endpoint in the Pipe Schedule.
        else
            Retry the packet.
```

Note: When retiring a TD, if its Transfer Ring is empty, pull the endpoint from the Pipe Schedule.

Table 4-8: USB3 Pipe Actions based on Endpoint Response and Residual Transfer State

Direction	Endpoint Response	Transfer State after Transaction (Bytes to transfer)	Pipe Action
IN	DP *Max Packet Size*	Not Zero	Decrement *SOPC*. If *SOPC* = 0: Advance to next endpoint. else Continue moving endpoint packets.
		Zero	Retire TD. Advance to next endpoint.
	DP Short	Don't care	Retire TD. Advance to next endpoint.
	DP(EOB = '1')	Don't care	Pull endpoint from Pipe Schedule.[46] Advance to next endpoint.
	NRDY	Don't care	Pull endpoint from Pipe Schedule. Advance to next endpoint.

[46]The assertion of EOB on a Short Packet may also retire the TD.

	Stall	Don't care	Generate *Stall Error* Transfer Event. Set the endpoint to the Halted state. Pull endpoint from schedule. Advance to next endpoint.
	DPP Error[47]	Don't care	Discard data. Decrement the *Bus Error Counter*, If *Bus Error Counter* = '0': Generate *USB Transaction Error* Transfer Event. Set endpoint to the *Halted* state. Pull endpoint from Pipe Schedule. Advance to next endpoint. else Decrement *SOPC*. If *SOPC* = 0: Advance to next endpoint.
	DPH Error[48]	Don't care	Discard data and send no acknowledgement.
	DPP exceeds *Max Packet Size* or remaining TD Transfer Size Error	Don't care	Discard data. Generate *Babble Detected Error* Transfer Event. Set endpoint to the *Halted* state. Pull endpoint from schedule. Advance to next endpoint in schedule.
IN (continued)	tHostTransactionTimeout[49] Error	Don't care	Generate *USB Transaction Error* Transfer Event. Set endpoint to the *Halted* state. Pull endpoint from schedule. Advance to next endpoint in schedule.

[47] *DPP Error* may be due to one or more of the following conditions: CRC incorrect, DPP aborted, DPP missing, ACK TP with the *Retry Data Packet* (rty) bit set, or the data length in the DPH does not match the actual data payload length.

[48] *DPH Error* may be due to one or more of the following conditions: an incorrect Device Address, the Endpoint Number and Direction does not refer to an endpoint that is part of the current configuration, or the DPH does not have an expected sequence number. A *DPH Error* may result in a *tHostTransactionTimeout* if a expected DPH is not received.

[49] Refer to section 8.13, Table 8-36 in the USB3 spec for the range of valid tHostTransactionTimeout values.

OUT	ACK TP	Not Zero	If *SOPC* exhausted: Advance to next endpoint. else Continue moving endpoint packets.
		Zero	Retire TD. Advance to next endpoint.
	ACK TP w/Rty	Don't care	Decrement the *Bus Error Counter.* If *Bus Error Counter* = '0': Generate *USB Transaction Error* Transfer Event. Set endpoint to the *Halted* state. Pull endpoint from Pipe Schedule. Advance to next endpoint. else Backup DPH sequence number to value indicated by the ACK TP *Sequence Number.* If *SOPC* exhausted: Advance to next endpoint. else Continue moving endpoint packets.
	NRDY	Don't care	Pull endpoint from schedule. Advance to next endpoint in schedule.
	Stall	Don't care	Generate *Stall Error* Transfer Event. Set the endpoint to the Halted state. Pull endpoint from Pipe Schedule. Advance to next endpoint.
	tHostTransactionTimeout[49]	Don't care	Generate *USB Transaction Error* Transfer Event. Set endpoint to the *Halted* state. Pull endpoint from Pipe Schedule. Advance to next endpoint.

	ACK TP Error[50]	Don't care	Discard.
N/A	ERDY	N/A	Place endpoint on schedule.
N/A	TP Error[51]	N/A	Discard.

Note: When retiring a TD, if its Transfer Ring is empty, pull the endpoint from the Pipe Schedule.

The xHC shall concatenate buffers referenced by TRBs in a TD, moving *Max Packet Size* transfers for all but possibly the last packet of a TD. The size of the last packet is determined by the TD Residue.

TD Residue = TD Transfer Size - (Max Packet Size *
ROUNDDOWN(TD Transfer Size / Max Packet Size))

4.14.4.1 SuperSpeed Burst Transactions

The USB3 Specification, section 8.10.2 defines *bMaxBurst* as "The number of packets an endpoint on a device can send or receive at a time without an intermediate acknowledgement packet".

For a SuperSpeed bulk endpoint, the xHC shall use *Max Burst Size* (which is set to *bMaxBurst*, refer to section 6.2.3.4) to determine the maximum number of outstanding acknowledgement packets that are allowed for an endpoint. It may also use *Max Burst Size* to identify the number of packets the endpoint should send or receive in a Service Opportunity. If more than one async endpoint has data to move, the xHC should advance to the next endpoint when *Max Burst Size* packets have been moved for an endpoint. However if there is only one endpoint with data to move in the async Pipe Schedule, then the xHC may exceed *Max Burst Size* packets to an endpoint and stream packets to/from the endpoint until either the Transfer Ring is exhausted or the device terminates the burst by asserting NumP = 0 (OUT pipe ACK TP) or EOB = '1' (IN pipe DP), or flow controls the pipe by returning an NRDY TP.

Note: Section 8.13 in the USB3 Spec states, "If the host does not see a response to a Data Transaction (either IN or OUT) within 10 µs, it shall assume that the transaction has failed and halt the endpoint. No retries shall be performed." The

[50] *ACK TP Error* may be due to one or more of the following conditions: an incorrect Device Address, the Endpoint Number and Direction does not refer to an endpoint that is part of the current configuration, or the ACK TP does not have an expected sequence number. An *ACK TP Error* may result in a *tHostTransactionTimeout* if the expected ACK TP is not received.

[51] *TP Error* may be due to one or more of the following conditions: Reserved Type or SubType, an incorrect Device Address, or the Endpoint Number and Direction does not refer to an endpoint that is part of the current configuration.

xHC shall timeout a Burst Transaction if acknowledgements for all packets of the burst are not received by 10 µs. after the last packet of the Burst Transaction is transferred. e.g. For an OUT pipe if *Max Burst Size* = 4, then the xHC shall timeout the burst if the first framing symbol of the ACK response to the last DP is not received with 10 µs. after the last framing symbol of the last DPP (4th) of the burst is transmitted.

Note: Section 8.13 in the USB3 Spec defines *tHostACKResponse* as the "Time between host reception of the last framing symbol for a DPP and the first framing symbol of an ACK response". For a Burst Transaction, the xHC shall not delay the first framing symbol of an ACK response for the first DPP of a burst more than *tHostACKResponse* (3 µs.) after the last framing symbol of the last DPP of the burst is received.

Note: When a packet retry occurs, an xHC implementation may choose to limit a Burst Transaction to *Max Burst Size* packets, which may cause a retried packet to be transferred in the next Burst Transaction, or it may choose to allow packet retries to complete in the Burst Transaction that the error occurred in, possibly extending Burst Transaction to more than *Max Burst Size* packets.

Note: If a Deferred TP or DP is received during a burst, the xHC should advance to the next endpoint in its Pipe Schedule.

- For non-ISOC endpoint, the xHC should internally flag the endpoint as being flow controlled and wait for an ERDY to place the endpoint back on the Pipe Schedule.
- For an ISOC endpoint, the xHC should terminate the current Isoch TD and advance to the next TD which will be processed during the next ESIT.

4.15 Suspend-Resume

The xHC provides an equivalent suspend and resume model as that defined for individual ports in a USB Hub. Control mechanisms are provided to allow system software to suspend and resume individual ports. The mechanisms allow the individual ports to be resumed completely via software initiation. Other control mechanisms are provided to parameterize the host controller's response (or sensitivity) to external resume events. In this discussion, host-initiated, or software initiated resumes are called *Resume Events/Actions*. Bus-initiated resume events are called *Wake-up Events*. The classes of wakeup events are:

- Remote-wakeup enabled device asserts resume signaling, similar to USB Hubs, The xHC shall always respond to explicit device resume signaling and wake up the system (if necessary).
- Port connect and disconnect and over-current events. Sensitivity to these events can be turned on or off by using the per-port control bits in the PORTSC registers.

Selective suspend is a feature supported by every PORTSC register. It is used to place specific ports into a suspend mode. This feature is used as a functional component for implementing the appropriate power management policy implemented in a particular operating system.

When system software intends to suspend the entire bus, it should selectively suspend all enabled ports, then shut off the host controller by setting the *Run/Stop* (R/S) bit in the USBCMD register to a ‘0’. The xHC can then be placed into a lower device state via the PCI power management interface (refer to Appendix A and PCI PM).

When a wake event occurs system software will eventually set the *Run/Stop* (R/S) bit to a ‘1’ and resume the suspended ports by writing a ‘0’ to their *PLS* field. Software shall not set the *Run/Stop* (R/S) bit to a ‘1’ until it is confirmed that the clock to the host controller is stable. This is usually confirmed in a system implementation in that all of the clocks in the system are stable before the CPU is restarted. So, by definition, if software is running, clocks in the system are stable and the *Run/Stop* (R/S) bit in the USBCMD register can be set to ‘1’. There are also minimum system software delays defined in the PCI PM Specification. Refer to this specification for more information.

Note: *LTSSM Clock Stopped* refers to a condition where the xHC is in a D3 state and a Root Hub port is unable to transition a link with a *Connect Detect* to the Enabled state, e.g. the LTSSM clocks are stopped. The occurrence of *LTSSM Clock Stopped* is xHC implementation dependent, e.g. it may occur only while the xHC is in the D3cold state, or it may not occur at all.

Note: Software should transition all Root Hub ports, where it has acknowledged a Connect (CCS = '1'), to the U3 or Disabled states before placing the xHC into the D3 state, and unless a Device Initiated Resume or a Disconnect occurred the port should be in the same state when Main Power is restored. Note, a port is allowed to be in the Error state when the xHC is transitioned to the D3 state.

- If the port transitioned to the Resume state, *CAS* shall be asserted when Main Power is restored.

A disconnected Root Hub port may be in a one of several states when Main Power is restored.

- If no device was attached while in D3 the port will be in the Disconnected state when power is restored.
- If a device was attached after entering D3 but before entering *LTSSM Clock Stopped*, then when power is restored the *CAS* bit shall be set and the value of the *PLS* field should be ignored.
  - If the port had been able to successfully train and transition to the U0 state before entering *LTSSM Clock Stopped* then the upstream facing port of the attached device should be in the USDPORT.Disabled state, and a USB2 Port reset (*PR* = '1') will be required to cause the USB3 port to retrain and transition to the U0 state.

- If the port was not able to successfully train and transition to the U0 state before entering *LTSSM Clock Stopped* then the upstream facing port of the attached device may be in one of many states, and USB2 or USB3 Port reset may be required to cause the USB3 port to retrain and transition to the U0 state.

- If a device was attached after entering *LTSSM Clock Stopped*, then when power is restored the *CAS* bit shall be set and the value of the *PLS* field should be ignored. The upstream facing port of the attached device should be in the USDPORT.Disabled state, and USB2 Port reset will be required to cause the USB3 port to retrain and transition to the U0 state.

If an overcurrent condition exists (OCA = '1') when Main Power is restored, the condition must be cleared before the port will be usable.

Note: Any Root Hub port that is in the *Resume* or *U3* state when the xHC is transitioned to the D0 power state shall require software to drive the port to the *U0* state. The xHC shall not automatically transition a root hub port from the *Resume* or *U3* state to the *U0* state.

4.15.1 Port Suspend

System software places individual ports into suspend mode by writing a ‘3’ into the appropriate PORTSC register *Port Link State* (PLS) field (refer to section 5.4.8). Software should only set the *PLS* field to ‘3’ when the port is in the **Enabled** state.

The xHC may evaluate a *PLS* field write immediately or wait until a microframe or frame boundary occurs. If evaluated immediately, the port is not suspended until the current transaction (if one is executing) completes. Therefore, there may be several microframes of activity on the port until the xHC evaluates the *PLS* field. The xHC shall evaluate the *PLS* field at least every frame boundary. Refer to the description of *PLS* in Table 5-26 for more information.

When the *PLS* field is written with U3 (‘3’), the status of the *PLS* bit will not change to the target U state U3 until the suspend signaling has completed to the attached device (which may be as long as 10 ms.). Software should not attempt to suspend a port unless the port reports that it is in the enabled (*PED* = ‘1’, *PLS* < ‘3’) state (refer to Section 5.4.8 for more information in *PED* and *PLS*). Note, the *Port Link State Write Strobe* (LWS) bit shall be set to ‘1’ to write the *PLS* field.

Software is required to wait for U3 transitions to complete before it puts the xHC into a low power state, and before resuming the port. Software can poll the *PLS* field for the completion of a U3 transition; however a tight polling loop may prevent any other activity on the processor, slowing the power down process. Enabling an OS timer can also slow the power down process, because the minimum OS timeout (~15 ms.) is long compared to the U3 transition time, so software either ends up hogging a CPU, or adding a significant delay to the D3 entry of the host controller. The *U3 Entry Capability* (U3C) eliminates these delays by asserting *PLC* when there is a transition of *PLS* to U3, where the assertion of *PLC* generates a Port Status Change Event. If the *U3 Entry Capability* (U3C) is supported (*U3C* = '1') in the HCCPARAMS2 register, then software may enable the assertion of *PLC* on a transition of *PLS* to U3 by setting the *U3 Entry Enable* (U3E) flag to '1' in the CONFIG register.

Note: *U3 Entry Capability* support (i.e. *U3C* = '1') shall be mandatory for all xHCI 1.1 compliant xHCs.

4.15.1.1 Selective Suspend

Software shall stop all endpoints of a device using the *Stop Endpoint Command* and setting the *Suspend* (SP) flag to '1' prior to selectively suspending a device. After the device is resumed software shall ring an endpoint's doorbell to restart it. Refer to section 3.3.8 for more information on the use of the *Stop Endpoint Command*.

4.15.1.2 Function Suspend

Software shall stop the endpoints of a device associated with the function by using the *Stop Endpoint Command* and setting the *Suspend* (SP) flag to '1' prior to issuing a SetFeature(FUNCTION_SUSPEND) request to a device. After the function is resumed software shall ring an endpoints' doorbell to restart it. Refer to section Sto for more information on the use of the *Stop Endpoint Command*.

4.15.2 Port Resume

The following subsections describe typical device initiated and host initiated resume process

4.15.2.1 Device Initiated

The following steps describe a typical <u>device initiated</u> port resume process:

1. When a port is in the **U3** state and resume signaling is detected from a device, the port transitions to the **Resume** state (*PLS* = '15') and the *Port Link State Change* (PLC) flag is set to '1'. If the assertion of *PLC* results in a '0' to '1' transition of PSCEG (4.19.2), the xHC shall generate a *Port Status Change Event*.

   Note that an LFPS Handshake[52] is required for a USB3 U3 wakeup. A device generates LFPS to initiate the resume process. The detection of LFPS while in the U3 state shall transition a USB3 port to the **Resume**

[52]Refer to section 6.9.2 in the USB3 spec for more information on the LFPS Handshake.

state[53]. The xHC shall not respond with LFPS to the device, which would allow the LFPS Handshake to complete, until directed by software.

2. Upon receipt of a *Port Status Change Event* system software evaluates the *Port ID* field to determine the port that generated the event.

3. System software then reads the PORTSC register of the port that generated the event.
   PLC = '1' and *PLS* = Resume if the event was due to a device initiated resume:

   a. For a USB3 protocol port, software shall write a '0' to the *PLS* field to direct the xHC to initiate LFPS to the device and initiate the LFPS Handshake.

   b. For a USB2 protocol port, when a resume signaling is detected from a device the xHC shall transmit the resume signaling within 1 ms (TURSM). Software shall ensure that resume is signaled for at least 20 ms (TDRSMDN). Refer to section 7.1.7.7 of the USB2 spec. Software shall start timing TDRSMDN from the notification of the transition to the *Resume* state. After TDRSMDN is complete, software shall write a '0' to the *PLS* field.

4. The completion of the resume signaling shall cause the port to transition to the **U0** state, i.e. the PORTSC register *PLS* field shall to be set to U0 ('0') and *PLC* flag to '1'. If the assertion of *PLC* results in a '0' to '1' transition of PSCEG (4.19.2), the xHC shall generate a *Port Status Change Event*.

Note: Software shall ensure that the xHC is in Run (*R/S* = '1') mode prior to transitioning a root hub port from the **Resume** to the **U0** state. This action ensures that the xHC is capable of transmitting ITPs and immediately receiving packets when the device enters the **U0** state.

4.15.2.2 Host Initiated

System software can initiate a resume on a selectively suspended port by writing the *PLS* field (refer to section 4.15.2). Software shall not attempt to resume a port that it has initiated the suspend process on, unless the port reports that it is in the suspended (PED = '1', PLS = '3') state (refer to Section 5.4.8).

If system software writes the *PLS* field with a '0' when the port is not in the suspended state (U3), but in a low power link state (e.g. U2 or U1), the port shall

[53]Refer to section 4.19.1.2.13 for more information on the **Resume** state.

generate the appropriate signaling and if successful, shall then transition to the U0 state (PLS = ‘0’).

A U3 to U0 transition of the PLS field shall cause the *Port Link State Change* (PLC) bit to transition from ‘0’ to ‘1’. If the assertion of *PLC* results in a ‘0’ to ‘1’ transition of PSCEG (Port Status Change Generation), a *Port Status Change Event* shall be generated to reflect the change in link state. If *Interrupter 0* is not masked the generation of the event will also result in an interrupt to the host.

The following steps describe a typical host initiated port resume process:

1. When a port is in the **U3** state:
    a. For a USB3 protocol port, software shall write a ‘0’ (U0) to the *PLS* field to initiate resume signaling. The port shall transition to the **U3Exit** substate and the xHC shall immediately initiate LFPS generation to the device.
    b. For a USB2 protocol port, software shall write a ‘15’ (Resume) to the *PLS* field to initiate resume signaling. The port shall transition to the **Resume** substate and the xHC shall transmit the resume signaling within 1 ms (TURSM). Software shall ensure that resume is signaled for at least 20 ms (TDRSMDN). Software shall start timing TDRSMDN from the write of ‘15’ (Resume) to *PLS*. After TDRSMDN is complete, software shall write a ‘0’ (U0) to the *PLS* field.
2. The completion of the resume signaling shall cause the port to transition from the **U3** to the **U0** state, i.e. the PORTSC register *PLS* field shall to be set to U0 (‘0’) and *PLC* flag to ‘1’. If the assertion of *PLC* results in a ‘0’ to ‘1’ transition of PSCEG (Port Status Change Generation), the xHC shall generate a *Port Status Change Event*.

4.15.2.3 Wakeup Events

An external USB event may also initiate a system level resume. The system wake-up events are defined below. When resume signaling is detected by a suspended port, a system wake-up event occurs and the port transitions to the **Resume** state.

For a USB2 protocol port:

- If the resume signaling is detected it is reflected downstream by the xHC to all enabled ports within 1 ms. (TURSM), and maintained until software transitions the port from the **Resume** state to the **U0** state.

For a USB3 protocol port:

If the resume signaling (reception of a LFPS that meets the valid t12-t10 specification in Table 6-22 of the USB3 spec) is detected, the port shall transition to the **Resume** state immediately, and the *Port Link State Change* (PLC) bit is set to a '1'.

Software may determine that the port is enabled (not suspended) by sampling the PORTSC register and observing that the *Port Enabled/Disabled* (PED) flag is '1' and the *Port Link State* (PLS) field is < '3'.

Table 4-9 summarizes the system wake-up events, defining the state of the *Port Link State* (PLS), *Current Connect Status* (CCS), *Port Enabled/*Disabled (PED), *Over-Current Active* (OCA) fields in the PORTSC register and the *Port Change Detect* (PCD) bit in the USBSTS register as function of the respective Wake Enable flag (WDE, WCE, WOE). The table values indicate the state of the fields after the respective event. The *xHC State* column indicates the response of the xHC to the system as function of its (PCIe) power state when the event occurs.

Note: A port resume is not gated by a Wake Enable flag.

Table 4-9: Behavior During System Wake-up Events

Port Status and Signaling Device State Type	Port State After Event					xHC State Note	
	PLS	CCS	PED	OCA	PCD	D0	not D0
Port is in the **Disabled** state. Resume signaling received.	No Effect					N/A	N/A
Port is in the **U3** substate. Resume signaling is received.	Resume	1	1	0	1	[10-1] [10-2]	[10-2]
A port is in a state that may detect a disconnect[54], and the port's WDE bit is '1'. A disconnect is detected.	RxDetect	0	0	0	1	[10-1] [10-2]	[10-2]
A port is in a state that may detect a disconnect[54], and the port's WDE bit is '0'. A disconnect is detected.	RxDetect	0	0	0	1	[10-1] [10-3]	[10-3]

[54]A USB2 port may detect a disconnect when the port is in the **Disabled**, **Enabled**, or **Reset** states. A USB3 port may detect a disconnect when the port is in the **Loopback**, **Compliance**, **Error**, **Polling**, **Enabled**, or **Reset** states.

Port is in the **Disconnected** state and the port's WCE bit is '1'. A connect is detected.	U0 (SS) Polling (USB2)	1	1 (SS) 0 (USB2)	0	1	[10-1] [10-2]	[10-2]
Port is in the **Disconnected** state and the port's WCE bit is '0'. A connect is detected.	U0 (SS) Polling (USB2)	1	1 (SS) 0 (USB2)	0	1	[10-1] [10-3]	[10-3]
If a port is in a state that may detect an over-current condition[55] and the port's WOE bit is '1'.An over-current condition occurs.	Disabled	0	0	1	1	[10-1] [10-2]	[10-2]
If a port is a state that may detect an over-current condition[55] and the port's WOE bit is a '0'. An over-current condition occurs.	Disabled	0	0	1	1	[10-1] [10-3]	[10-3]

4.16 Bandwidth Management

In past generations of USB host controller implementations, there was a 1:1 correspondence between a host controller interface and USB bandwidth. The xHCI diverges from this model in that it enables vendors to tailor the bandwidth available through its root hub ports to the needs of the vendor's target application space. The xHCI can support the legacy model where the bandwidth of a single USB is shared across all its root hub ports, a "bus per port" model where the full bandwidth of a USB is available on every root hub port, or any combination in between.

The determination of the bandwidth available through an xHCI is further complicated because the interface is capable of supporting multiple USB speeds, each with their own bandwidth constraints. Computation of the bandwidth available when enumerating a USB device depends on which internal

[55]A port may detect an over-current condition in any state except **Powered-off**.

Note 10-1:If the assertion of change bit results in a '0' to '1' transition of PSCEG (4.19.2), a Port Status Change Event is generated.

Note 10-2:PME# asserted if enabled (i.e. the PCI PM PMCSR PME_En bit = '1'). Note: The PCI PM PMCSR PME_Status bit shall be written with a '1' to stop asserting PME#.

Note 10-3:PME# not asserted.

USB instance of the xHCI that a root hub port is allocated to, and the bandwidth requirements of the other devices already connected to that USB instance.

An example xHC implementation may define an 8 port implementation with 1 SS, 4 HS, and 8 LS/FS USB instances, for a total of 13 independent USB instances. Or if an implementation chose to focus on performance, it may define a “bus per port”, i.e. 8 SS, 8 HS and 8 LS/FS USB instances, i.e. 24 independent USB instances.

The xHCI architecture hides the internal complexities of a host controller implementation from system software. Given the set of USB instances supported by an xHC, it is responsible for managing and allocating the available USB bandwidth. Software uses the *Configure Endpoint Command* to ask the xHC if the bandwidth required for a specific device configuration is available. The xHC is responsible for evaluating the request as a function of its internal organization and the bandwidth available on the particular USB instance that the device is attached to.

If a *Configure Endpoint Command* fails due to a *Bandwidth Error* or a *Secondary Bandwidth Error*, system software may retry the command with other endpoint settings, or issue a *Negotiate Bandwidth Command*. The *Negotiate Bandwidth Command* allows software to identify the devices with periodic endpoints attached to the same USB instance in the xHC. The *Negotiate Bandwidth Command* generates a *Bandwidth Request Event* for each device attached to the same USB instance which is currently consuming periodic bandwidth, i.e. declared Isoch or Interrupt endpoints. Using this information, software may target the reassignment of bandwidth to allow the initial device to be configured.

Refer to section 4.6.13 for more information on the *Negotiate Bandwidth Command* and section 6.4.2.4 for more information on the *Bandwidth Request Event TRB*.

A *Disable Slot Command* will cause any bandwidth allocated to the periodic endpoints of a device slot to be freed.

4.16.1 Bandwidth Negotiation

Many USB devices offer multiple configurations and/or alternate interface settings to meet a variety of bandwidth demands. For instance, a USB camera may present a dozen Alternate Interface settings that match the various resolutions and frame rates that it supports. Typically the Video Class Driver will select an interface setting that will provide the highest quality image for the user, however if this setting is rejected, because there is not enough bandwidth available, the Class Driver will attempt to set a lower quality setting that requires less bandwidth. If all alternate settings are tried and the Class Driver is still unable to enumerate the camera, it may decide to issue a *Negotiate Bandwidth Command*.

The *Negotiate Bandwidth Command* generates a *Bandwidth Request Event* for each device slot with periodic endpoints on the same USB instance.

When a *Bandwidth Request Event* is received for a device slot, system software should treat it as a request to evaluate the current bandwidth requirements of device and free some of the bandwidth if the device is able to effectively perform its tasks on a reduced bandwidth budget. There is no requirement that a device give up bandwidth due to a *Bandwidth Request Event*, however a “good citizen” will do their best to comply. To free bandwidth, the software may select another configuration or an alternate interface setting for the periodic endpoints of the device. As devices reconfigure themselves they will issue *Configure Endpoint Commands* which will free part or all of their currently assigned bandwidth. As the xHC processes the commands it shall recompute the available bandwidth of the USB instance. The *Negotiate Bandwidth* command may allow a device to enumerate that would not have been able to without it.

The *Negotiate Bandwidth Command* uses the value of the *Slot ID* field in the *Negotiate Bandwidth Command TRB* to identify the USB instance that the device requiring the bandwidth is attached to.

The *Negotiate Bandwidth* command does not block Command Ring execution, e.g the command should not wait for all BW Requests to be delivered before generating the associated Command Completion Event.

The *Negotiate Bandwidth Command* is acknowledged by the xHC with a *Success* Completion Code. *Bandwidth Request Events* shall be generated for the selected device slots. The selection of the device slots that are targeted by *Bandwidth Request Events* shall be determined by an xHC implementation specific algorithm.

After a system defined delay, the software that initiated the negotiation process may reissue the *Configure Endpoint Command* that failed, to test whether enough bandwidth has been freed to allow a successful completion.

Note: The initiator of the *Negotiate Bandwidth Command* should allow enough time for system software to receive the *Bandwidth Request Events* and to reconfigure or choose alternate interface settings for the target device, before attempting to issue a *Configure Endpoint Command*.

Whether an xHC implementation supports Bandwidth Negotiation, is identified by the *BW Negotiation Capability* (BNC) flag in the HCCPARAMS1 register.

Note: A important use of the *Negotiate Bandwidth Command* is with virtualization. It allows one VF to ask the other VFs for BW. Which means that an OS shall expect to receive a *Bandwidth Request Event* asynchronously, e.g. without having previously issued a *Negotiate Bandwidth Command*.

Refer to section Negotiate Bandwidth Command TRB (Optional Normative) for more information on the *Negotiate Bandwidth Command TRB* and *Bandwidth Request Event TRB*.

4.16.2 Bandwidth Domains

Each Bus Instance (BI) represents a “unit” bandwidth at the speed that the BI supports or a **Bandwidth Domain**. The Transaction Translator (TT) of a USB2 hub creates one or more **Secondary Bandwidth Domains** on its downstream facing ports. For a High-speed hub, a Secondary Bandwidth Domain is equivalent to a Full-speed BI, or for a SuperSpeedPlus hub, a Secondary Bandwidth Domain is equivalent to a SuperSpeed BI. The downstream facing ports of a single-TT hub creates a single Secondary Bandwidth Domain, whose bandwidth is shared across all Full- or Low-speed devices attached to the hub. A multi-TT hub creates a separate Secondary Bandwidth Domain for each downstream facing port attached to a Full- or Low-speed device.

The xHC bandwidth allocation algorithm shall comprehend Secondary Bandwidth Domains and reject a *Configure Endpoint Command* with a *Secondary Bandwidth Error* if the configuration would have exceeded the *Total Available Bandwidth* of the domain. e.g. if a Full-speed isochronous Device A that requires 60% of the FS bandwidth is attached to a HS Hub which supports a single TT and already has a FS isoch Device B attached to one of its ports that has been allocated 50% of the TT bandwidth, the configuration request for Device A will be rejected by the xHC. Note that the HS Bandwidth Domain above the hub may have plenty of bandwidth available to service the configuration.

A *Configure Endpoint Command* shall return an event with the *Completion Code* set to *Secondary Bandwidth Error* if there was insufficient bandwidth in the Secondary Bandwidth Domain to enable the configuration. Refer to section 3.3.5 for more information on the *Configure Endpoint Command*.

Software may determine the bandwidth available in a Secondary Bandwidth Domain by issuing a *Get Port Bandwidth Command* with the *Hub Slot ID* field set to the Slot ID of the target hub. Refer to section 4.6.15 for more information on the *Get Port Bandwidth Command*. Note that if the hub specified by the *Hub Slot ID* does not reside on a Secondary Bandwidth Domain boundary (e.g. the hub does not contain a TT), undefined behavior may occur, e.g. the values in the *Port Bandwidth Context* may be invalid.

Note: When evaluating a *Configure Endpoint Command*, the xHC shall check the upstream High-speed Bandwidth Domain of a hub first. If there is enough bandwidth available in the primary (HS) Bandwidth Domain then the xHC shall check the Secondary (FS) Bandwidth Domain of the hub.

4.17 Interrupters

An *Interrupter* manages events and their notification to the host. The xHCI supports up to 1024 Interrupters. The *MaxIntrs* field in HCSPARAMS1 determines the *Number of Interrupters* implemented in the xHC. Each Interrupter consists of an Interrupter Management Register, an Interrupter Moderation Register and an Event Ring. Each Interrupter shall be mapped to a single MSI or MSI-X interrupt vector. An Interrupter shall assert an interrupt if it is enabled and its associated Event Ring contains Event TRBs that require an interrupt.

IMPLEMENTATION NOTE

PCI MSI and MSI-X Interrupts

MSI-X defines a separate optional extension to basic PCI MSI functionality. Compared to MSI, MSI-X supports a larger maximum number of vectors per function, the ability for software to control aliasing when fewer vectors are allocated than requested, plus the ability for each vector to use an independent address and data value, specified by a table that resides in Memory Space. However, most of the other characteristics of MSI-X are identical to those of MSI. For more information on MSI-X, refer to the *PCI Specification*.

MSI-X maps each of the xHC Interrupters to an interrupt vector that is conveyed by xHC as a posted-write PCI Express (PCIe) transaction. Each MSI-X interrupt vector has some attributes assigned to it, such as the address and data for its posted-write message. These are described in section 5.2.8.2 that described the PCI aspects on MSI-X configuration.

Interrupters and PCI Interrupt Mechanisms

When the PCI Pin Interrupt is activated:

- Interrupter 0 may assert the INTx# pin.
- Interrupters 1 to *MaxIntrs*-1 shall be disabled.

When MSI is activated:

- If *MaxIntrs* > 32, then Interrupters 0 to 31 may each trigger a unique interrupt vector, and Interrupters 32 to *MaxIntrs*-1 shall be disabled.
- If *MaxIntrs* <= 32, then Interrupters 0 to *MaxIntrs*-1 may each trigger a unique interrupt vector.
- The MSI *Message Control* register *Multiple Message Capable* field reported by the xHC shall be equal to or less than *MaxIntrs*.

The Interrupt Vector associated with an Interrupter shall be defined as function of the value of the MSI Message Control register *Multiple Message Enable* field using the following algorithm.

Interrupt Vector = (Index of Interrupter) MODULUS (MSI Message Control:*Multiple Message Enable*)When MSI-X is activated:

- Interrupters 0 to *MaxIntrs*-1 may each trigger a unique interrupt vector. i.e. there is a 1:1 mapping of the index of an Interrupter to the index of the MSI-X vector in the MSI-X Table Structure or to the associated Pending Bit in the MSI-X PBA Structure. Refer to section 6.8.2 in the PCI spec for more information.
- The value of the MSI-X Message Control register *Table Size* field reported by the xHC shall be equal to the value of MaxIntrs.
- The allocation of MSI-X vectors is set by the enabling of the respective Interrupter using the MSI-X Enable field in the Vector Control Dword of the MSI-X Table Structure. (If Interrupter 0 is enabled, the vector defined by MSI-X Table[0] is allocated, if Interrupter 1 is enabled, the vector defined by MSI-X Table [1] is allocated, etc.).

The *Number of Interrupters* (MaxIntrs) is implementation dependent. An xHC implementation shall implement at least one Interrupter.

xHC generated interrupts to the system may be enabled by setting the *Interrupter Enable* (INTE) flag in the USBCMD register to '1'.

An xHC implementation that supports virtualization shall implement at least one Interrupter for the Physical Function and a minimum of one Interrupter per Virtual Function. Refer to section 8 for more information on virtualization.

Note: The xHC is not required to maintain event ordering across Event Rings. e.g. If events that are generated sequentially within the xHC target separate Event Rings, the events may not be placed on the respective Event Rings in the same temporal order.

4.17.1 Interrupter Mapping

An xHC implementation may support Interrupter Mapping. **Interrupter Mapping** is the ability to target an Interrupter and its Event Ring, with the Transfer Events generated by a specific Transfer Request Block.

If the *Number of Interrupters* (*MaxIntrs*) field is greater than 1, then Interrupter Mapping shall be supported.

The value of the *Interrupter Target* field in the Transfer TRB determines which Interrupter shall receive the Transfer Events generated by the respective Device Slot or Transfer TRB.

If Interrupter Mapping is not supported, the *Interrupter Target* field shall be ignored by the xHC and all Events targeted at Interrupter 0.

Valid values for a Slot Context or TRB *Interrupter Target* field are between 0 and *MaxIntrs*-1. If an *Interrupter Target* field is out of range for a TRB the behavior of the xHC shall be undefined. It is recommended that the xHC does not generate any event if this condition is detected, and let software timeouts detect the error for the endpoint. If virtualization is supported, an xHC implementation shall ensure that this “undefined behavior” does not affect another function (PF0 of VFx).

The Slot Context *Interrupter Target* value shall be checked for a valid range when a command inputs the Input Slot Context.

Refer to section 6.4.1 for more information on the *Interrupter Target* field.

This mechanism may be used to facilitate distribution of interrupts across cores in a multi-core platform.

4.17.2 Interrupt Moderation

Interrupt Moderation allows multiple events to be processed in the context of a single *Interrupt Service Request* (ISR), rather than generating an ISR for each event.

The interrupt generation that results from the assertion of the *Interrupt Pending* (IP) flag may be throttled by the settings of the *Interrupter Moderation* (IMOD) register of the associated Interrupter. The IMOD register consists of two 16-bit fields: the *Interrupt Moderation Counter* (IMODC) and the *Interrupt Moderation Interval* (IMODI).

Software may use the *IMOD* register to limit the rate of delivery of interrupts to the host CPU. This register provides a guaranteed inter-interrupt delay between the interrupts of an Interrupter asserted by the host controller, regardless of USB traffic conditions.

The following algorithm converts the inter-interrupt interval value to the common 'interrupts/sec' performance metric:

$$\text{Interrupts/sec} = (250\times10^{-9}\text{sec} \times \text{IMODI})^{-1}$$

For example, if the IMODI is programmed to 512, the host controller guarantees the host will not be interrupted by the xHC for at least 128 microseconds from the last interrupt. The maximum observable interrupt rate from the xHC should not exceed 8000 interrupts/sec.

Inversely, inter-interrupt interval value can be calculated as:

$$\text{Inter-interrupt interval} = (250\times10^{-9}\text{sec} \times \text{interrupts/sec})^{-1}$$

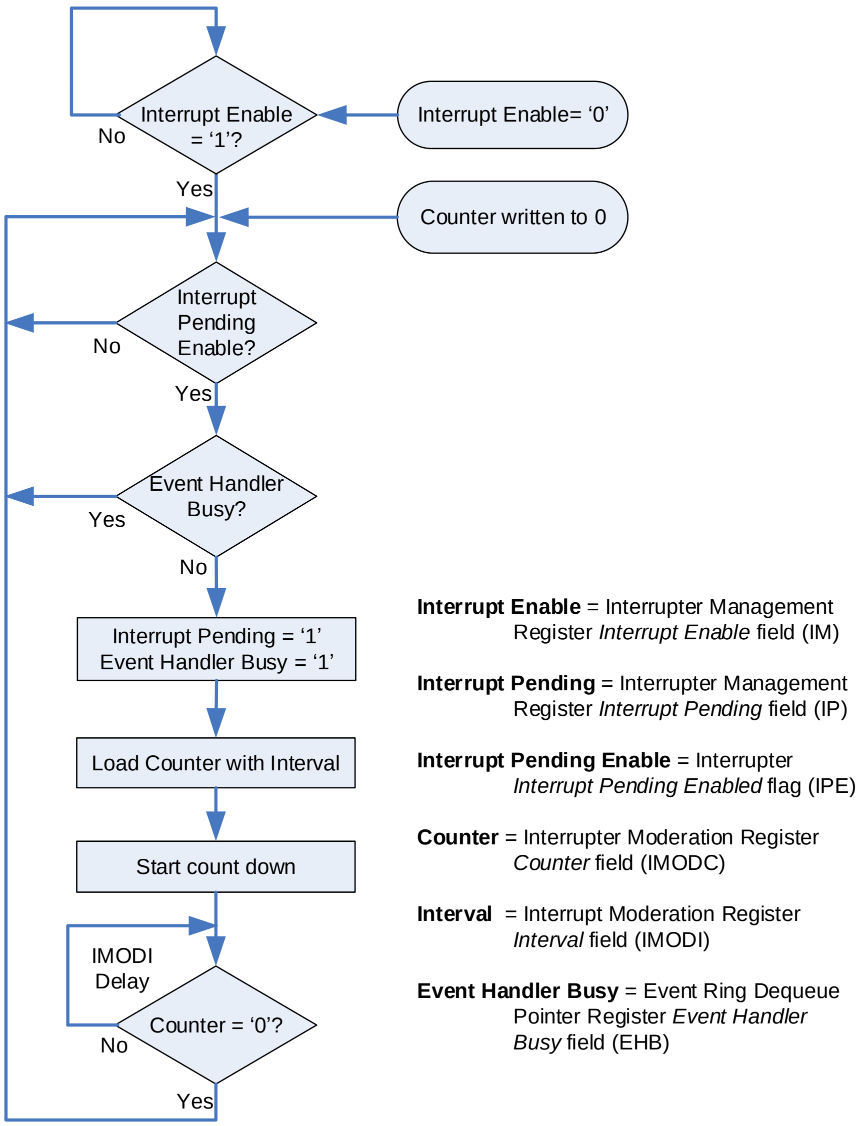
The optimal performance setting for this register is very system and configuration specific. An initial suggested range for the moderation Interval is 651-5580 (28Bh - 15CCh).

The IMODI field shall default to 4000 (1 ms.) upon initialization and reset. It may be loaded with an alternative value by software when the Interrupter is initialized.

The xHC implements interrupt moderation to reduce the number of interrupts that SW processes. The moderation scheme is based on the IMOD register and the ERDP *Event Handler Busy* (EHB) flag[56]. When an Interrupter is enabled it begins looking for two conditions: 1) *Interrupt Pending Enable* (IPE = '1') and 2) the Event Handler not busy (EHB = '0'). If these conditions are true, the *Interrupt Pending* (IP) bit in the Interrupter Management (IMAN) register and the *Event Handler Busy* (EHB) flag in the Event Ring Dequeue Pointer (ERDP) register are set to '1', IMODC is loaded with IMODI, and moderation counter starts counting down. Another interrupt message will not be asserted to the host bus by the xHC until 1) the IMODC of the associated Interrupter has counted down to '0', 2) the *Interrupt Pending Enable* is asserted (IPE = '1'), and 3) the Event Handler is not busy (EHB = '0'). When all three conditions are met, IMODC is reloaded with the value of the IMODI and the process repeats again. Refer to section 5.5.2.2 for more information on the IMOD register and the IMODC clocking rate. The interrupt flow should follow the diagram below:

[56]*EHB* enables *Interrupt Mitigation*. High-speed serial interface operation can create thousands of interrupts per second, all of which tell the system something it already knew: it has lots of TRBs to process. The EHB allows the driver to run with interrupts disabled during times of high traffic, with a corresponding decrease in system load.

Figure 4-22: Interrupt Throttle Flow Diagram



Interrupt Enable = Interrupter Management Register *Interrupt Enable* field (IM)

Interrupt Pending = Interrupter Management Register *Interrupt Pending* field (IP)

Interrupt Pending Enable = Interrupter *Interrupt Pending Enabled* flag (IPE)

Counter = Interrupter Moderation Register *Counter* field (IMODC)

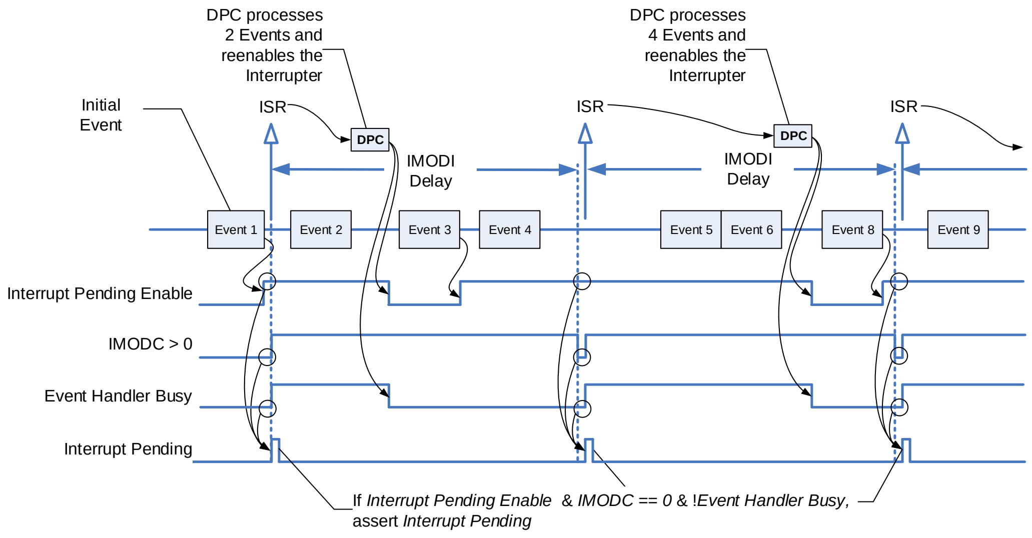
Interval = Interrupt Moderation Register *Interval* field (IMODI)

Event Handler Busy = Event Ring Dequeue Pointer Register *Event Handler Busy* field (EHB)

If PCI *Message Signaled Interrupts* (MSI or MSI-X) are enabled, then the assertion of the *Interrupt Pending* (IP) flag in Figure 4-22 generates a PCI Dword write. The *IP* flag is automatically cleared by the completion of the PCI write.

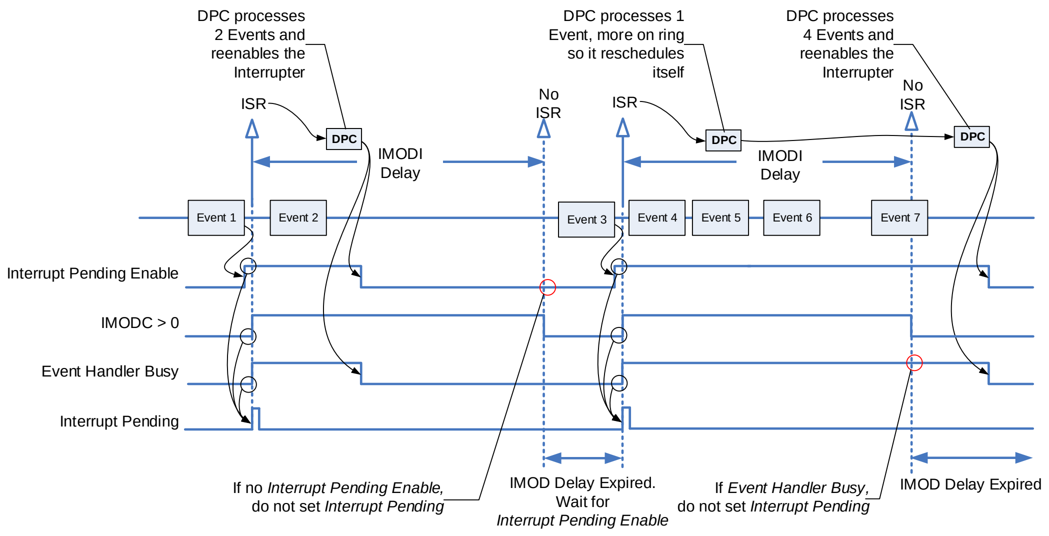
If the PCI Interrupt Pin mechanism is enabled, then the assertion of *Interrupt Pending* (IP) asserts the appropriate PCI INTx# pin. And the *IP* flag is cleared by software writing the IMAN register.

Figure 4-23: Heavy load, interrupts moderated


DPC processes 2 Events and reenables the Interrupter
DPC processes 4 Events and reenables the Interrupter
Initial Event
ISR
DPC
ISR
DPC
ISR
IMODI Delay
IMODI Delay
Event 1
Event 2
Event 3
Event 4
Event 5
Event 6
Event 8
Event 9
Interrupt Pending Enable
IMODC > 0
Event Handler Busy
Interrupt Pending
If *Interrupt Pending Enable* & *IMODC == 0* & !*Event Handler Busy*, assert *Interrupt Pending*

Under heavy load conditions (Figure 4-23), Interrupt Pending Enable (IPE) is asserted almost constantly, so if IPE = '1' when the IMODC counts down to '0' and the Event Handler is not busy (EHB = '0'), an interrupt is generated immediately, i.e. Interrupt Pending (IP) is set to '1'. When IP is asserted, the IMODC is reloaded with the IMODI and the IMODC begins counting down again. Thus, the next interrupt event will be delayed by the IMODI delay. Also note that in this example, the assertion of Interrupt Pending (IP) triggers the Interrupt Service Routine (ISR). The ISR schedules a Deferred Procedure Call (DPC) that will process the events on the Event Ring at a later time. The DPC processes events until Event Ring is empty then clears the Event Handler Busy (EHB) flag. Interrupt Pending Enable is cleared when the Event Ring goes empty, i.e. the DPC writes the Event Ring Dequeue Pointer (ERDP) register with a value that is equal to the Event Ring Enqueue Pointer.

Figure 4-24: Light load, interrupts not moderated


DPC processes 2 Events and reenables the Interrupter
ISR
DPC
IMODI Delay
No ISR
DPC processes 1 Event, more on ring so it reschedules itself
ISR
DPC
IMODI Delay
DPC processes 4 Events and reenables the Interrupter
No ISR
DPC
Event 1
Event 2
Event 3
Event 4
Event 5
Event 6
Event 7
Interrupt Pending Enable
IMODC > 0
Event Handler Busy
Interrupt Pending
If no Interrupt Pending Enable, do not set Interrupt Pending
IMOD Delay Expired. Wait for Interrupt Pending Enable
If Event Handler Busy, do not set Interrupt Pending
IMOD Delay Expired

Under light load conditions (Figure 4-24) it is desirable to fire off interrupts with minimum latency. In this case, when the IMODC counts down to '0' and no interrupts are pending (IPE = '0'), the IMODC is not reloaded with the IMODI but stays at '0'. Thus, the next assertion of *Interrupt Pending Enable* will trigger an interrupt immediately. Triggering the interrupt will also cause the IMODC to be reloaded with the IMODI and begin counting down again.

In the first case where the IMOD Delay Expires, *Interrupt Pending* (IP) is not set (so the ISR is not triggered) because the Event Ring is empty. Since IMODC = 0 when event 3 is posted, *Interrupt Pending* (IP) is asserted immediately.

In the second case, *Interrupt Pending* (IP) is not set because the Event Handler is busy (EHB = '1'). The DPC was not able to empty the Event Ring the first time it was scheduled (i.e. it only processed event 3), so it rescheduled itself to process the remaining events in the ring (i.e. event 4). While waiting for the DPC to be scheduled, events 5, 6, and 7 are posted. The rescheduled DPC processes events until Event Ring is empty then clears the *Event Handler Busy* (EHB) flag, re-enabling an immediate interrupt the next time an event is posted.

4.17.3 Interrupt Pin Support

PCI Interrupt Pins are optional. Four Interrupt Pins are supported by PCI, however PCI only allows one Interrupt Pin to be assigned to a single PCI Function. If an xHC implementation supports a PCI INTx# interrupt pin, xHC asserts its INTx# line when requesting attention from its device driver unless the xHC is enabled to use Message Signaled Interrupts (MSI, i.e. the MSI Message Control *MSI Enable* or MSI-X Message Control *MSI-X Enable* flags are true) (refer to Sections 5.2.8.1 and 5.2.8.2 for more information). Once the INTx# signal is asserted, it remains asserted until the device driver clears the *Interrupt Pending*

(IP) flag. When *Interrupt Pending* (IP) is cleared, the device deasserts its INTx# signal.

If Interrupt Pin support is enabled, then only Interrupter 0 is enabled and any other Interrupters are disabled.

The *Interrupt Pin* register in the PCI Configuration Space Header (refer to *Interrupt Pin* description in section 6.2.4 of the PCI specification) identifies which interrupt pin the device (or device function) uses. A value of 1 corresponds to INTA#, 2 corresponds to INTB#, and so on. If the xHC implementation does not use an interrupt pin it shall declare a '0' in this register.

4.17.4 Interrupter Target Identification

The target Interrupter of an event is determined in one of three ways:

1. Fixed and always the *Primary Interrupter*.
2. Defined by the *Interrupter Target* field in the TRB data structure.
3. Defined by the Slot Context *Interrupter Target* field.

Each Event TRB described in section 6.4.2 specifies which of the three methods described above it uses. The exception is the Transfer Event. There are some conditions related endpoints or transfers which are reported using a *Transfer Event TRB*, however the condition that they are reporting cannot be associated with a specific Transfer Event TRB. In these cases the Slot Context *Interrupter Target* field shall be used to identify the Interrupter that shall receive the event.

These conditions are indicated by the following *Completion Codes*:

- *USB Transaction Error* - due to detecting a Transaction Timeout while in the Stream Protocol HISPSM **Prime Pipe** state or HOSPSM **Prime Pipe** or **Start Stream End** state.
- *Stall Error* - due to detecting a STALL condition while in the Stream Protocol HISPSM **Prime Pipe** state or HOSPSM **Prime Pipe** or **Start Stream End** state.
- *Invalid Stream ID Error*.
- *Invalid Stream Type Error.*
- *Stopped - Length Invalid*. Note that the Slot Context *Interrupter Target* field is only applied to the "Stopped while waiting for more TRBs to be posted for TD" Condition in Table 4-2, not to the conditions "Stopped on Link TRB within a TD" and "Stopped on No Op TRB within a TD"
- *Ring Overrun*.
- *Ring Underrun*.

4.17.5 Interrupt Blocking

Normally, placing an Event TRB on an Event Ring causes an interrupt to be asserted to the host immediately if an Event Ring is empty or at the next interrupt threshold. However there are cases where software requires the *Completion Status* and *TRB Transfer Length* of a Transfer TRB reported by a Transfer Event TRB, but it does not want the Transfer Event to generate an interrupt. To facilitate this usage, The *Normal* and *Isoch* Transfer TRBs, and *Event Data* TRBs support a **Block Event Interrupt** (BEI) flag that allows them to place an Event TRB on an Event Ring but not assert an interrupt to the host.

An example of where the *BEI* flag can eliminate unwanted system interrupts is with Isoch transfers. For a USB microphone that declares ESIT of 1 ms. and generates 16-bit samples at a 44.1 KHz rate, software may post 10 Isoch TDs at a time to the device's Isoch IN Transfer Ring. The fractional sample rate means that over a 10 ms. period, the microphone completes 9 Isoch TDs with 44 samples (88 bytes) each, and a 10th TD with 45 samples (90 bytes). Since the number of samples per Isoch TD varies software must set the *ISP or IOC* flag in each Isoch TD to generate a Transfer Event to report the number of bytes transferred. However since software is able to schedule 10 TDs at a time, it only needs an interrupt every 10th TD. By setting the *BEI* flag in 9 of every 10 TDs, the interrupt rate due to the Isoch transfers can be reduced.

Note that software could drop the interrupt rate by adjusting the *Interrupt Moderation Interval* (IMODI) of the Interrupter, however this would affect the interrupt latency for all endpoints that shared an Event Ring. The *BEI* flag allows software to selectively reduce interrupt rates of transfers, without affecting latency sensitive transfers.

- If *BEI* = '1' in a TRB, then the event generated by the TRB is considered to be a "Blocking Event".
- If *BEI* = '0', then the event generated by the TRB is considered to be a "Non-blocking Event".
- Any TRB type that does not define a *BEI* flag always generates *Non-blocking Events*.
- If an error is detected which generates an event while processing a TRB with *BEI* = '1', then *BEI* shall be ignored and the event generated by the TRB shall be a *Non-blocking Event*.
- Any Transfer Event TRB that is not associated with a Transfer or Event Data TRB shall be a *Non-blocking Event*.

To facilitate Interrupt Blocking an *Interrupt Pending Enable* (IPE) flag may be implemented by the xHC for each Interrupter. *IPE* is an internal Interrupter flag that is not exposed through any register. Refer to section Interrupt Moderation for how *IPE* affects interrupt generation and the Interrupt Moderation mechanism.

The *IPE* flag of an Interrupter is managed as follows:

- *IPE* shall be cleared to '0':
  - When the Event Ring is initialized.
  - If the Event Ring transitions to empty.
- When an Event TRB is inserted on the Event Ring and *BEI* = '0' then:
  - *IPE* shall be set to '1'.

Note: Only *Normal, Isoch*, and *Event Data TRBs* support a *BEI* flag.

The *Interrupt Pending* (IP) flag of an Interrupter shall be managed as follows:

- When *IPE* transitions to '1':
  - If *Interrupt Moderation Counter* (IMODC) = '0' and *Event Handler Busy* (EHB) = '0', then *IP* shall be set to '1'.
- When *IMODC* transitions to '0':
  - If *EHB* = '0' and *IPE* = '1', then *IP* shall be set to '1'.
- If MSI or MSI-X interrupts are enabled, *IP* shall be cleared to '0' automatically when the PCI Dword write generated by the Interrupt assertion is complete.
- If PCI Pin Interrupts are enabled then, *IP* shall be cleared to '0' by software.

Note: A Transfer Event not associated with a Transfer TRB (i.e. a Transfer Event that uses the Slot Context *Interrupter Target*) is always a *Non-blocking Event*.

4.18 Transfer Definition and Attributes

4.18.1 No snoop

This feature is optional for PCIe implementations.

If the **Enable No Snoop** bit (Bit Location 11, Table 7-12) in the *PCI Express Capability Structure* (5.2.8) Device Control Register (PCIe spec section 7.8.4) is set, the xHC is permitted to set the No Snoop bit in the Requester Attributes of PCIe transactions it initiates that do not require hardware enforced cache coherency (refer to Section 2.2.6.5 of the PCIe spec). Note that setting this bit to '1' will not cause the xHC to set the No Snoop attribute on all PCIe transactions that it initiates. Even when this bit is '1', the xHC is only permitted to set the No Snoop attribute on a PCIe transaction when it can guarantee that the address of the transaction is not stored in any cache in the system.

If Enabled in the *PCI Express Capability Structure* and directed by software in (e.g. TRB *No Snoop* (NS) flag is set to '1'), then the xHC may set the **No Snoop** bit in the Requester Attributes of PCIe transactions it initiates that do not require hardware enforced cache coherency. Refer to Table 4-10 for recommended No Snoop behavior.

The xHC shall not assert the No Snoop attribute on PCIe transactions for memory requests that are Message Signaled Interrupts, and Message Requests (except where specifically permitted).

4.18.2 No Snoop and Relaxed Ordering for USB Traffic

SW may configure the No Snoop/Relaxed Ordering PCIe attributes for each TRB by setting the respective *No Snoop* (NS) flag in the TRB.

Table 4-10 defines the recommended behavior of the No Snoop and Relaxed Ordering PCIe Requester Attributes for PCIe transactions generated by the xHC. xHC implementations may choose other settings for these PCIe Requester Attributes. The PCIe Transaction No Snoop attribute is also conditioned for **IN Data Writes** by the TRB *No Snoop* (NS) bit.

Table 4-10: xHC Traffic Attributes

Transfer Type	No Snoop	Relaxed Ordering	Comments
TRB Read	N	Y	Command, Transfer IN or OUT
IN Data Write, TRB *No Snoop* flag = 1 TRB *No Snoop* flag = 0	 Y N	 N N	Refer to section 4.18.2.1. Snooping is dynamically controlled by the Transfer TRB *No Snoop* flag.
OUT Data Read, TRB *No Snoop* flag = 1 TRB *No Snoop* flag = 0	 Y N	 Y Y	Snooping is dynamically controlled by the Transfer TRB *No Snoop* flag.
Command Data Write	N	N	e.g. Port Bandwidth Context
TRB Write	N	N	Events
Context Read	N	Y	Any Context read, including Opaque area
Context Write	N	N	Any Context write, including Opaque area
Opaque Read	Y	Y	Scratchpad Opaque area read
Opaque Write	Y	N	Scratchpad Opaque area write

Note: "N" means that the respective Requester Attribute is not set in the PCIe Transaction. "Y" means that the respective Requester Attribute is set in the PCIe Transaction.

Section 2.2.6.4 of the PCIe spec describes the Relaxed Ordering Attribute field. And this attribute is discussed further in section 2.4 of the PCIe spec.

4.18.2.1 No Snoop option for payload

Under certain conditions, system software knows that it is safe to DMA a new data into a certain buffer without snooping. This scenario would occur when software is posting an IN buffer to the xHC that the CPU has not accessed since the last time it was owned by the xHC. This might happen if the data was transferred to an application buffer by the xHC DMA engine. In this case, software should be able to set a bit in the IN TRB indicating that the xHC should perform a "no-snoop" DMA when it eventually writes a packet to this buffer. When a non-snoop transfer is activated, the TRB will have a non-snoop flag in the TRB Control field. This is triggered by the *No Snoop* (NS) bit in the IN TRB.

4.18.2.2 No Snoop option for Scratchpad references

The Scratchpad Buffer Array and the Scratchpad Buffers that it references are exclusively owned by the xHC. To eliminate unnecessary system bus operations, the xHC should perform a "no-snoop" DMA when accessing the Scratchpad Buffer Array or Scratchpad Buffers.

4.19 Root Hub

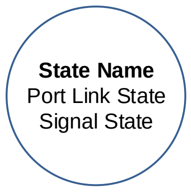
This section describes the Root Hub and Root Hub Port operational models.

The protocols supported by a xHC implementation are identified by the declared *xHCI Supported Protocol Capability* structures, Refer to section 7.2. The *xHCI Supported Protocol Capability* structures identify the number of *Port Status and Control* (PORTSC) registers supported by an xHC implementation. Refer to section 4.19.7 for more information on xHCI protocol to PORTSC register mapping.

Note: Refer to section Suspend-Resume for how to manage a port when Main Power is removed.

4.19.1 Root Hub Port State Machines

The following state machines utilize the following notation:



Where the **State Name** is an informative name defined by the xHCI spec., the *Port Link State* identifies the possible values for the PORTSC *PLS* field, and *Signal State* values are:

> *Port Power* (PP), *Current Connect Status* (CCS), *Port Enabled/Disabled* (PED), and *Port Reset* (PR), respectively, e.g. 0,0,0,0 all signals are '0'.

Note: Transitions associated with the large bubble may occur from any state defined within the bubble as long as the Conditions match.

Refer to Appendix E for state machine notation.

Note: In each state, the *Signal State* values defined by the state are forced when entering the state, so actions are not declared for changing the respective bits when transitioning from another state. e.g. If the **Disconnected** State is entered from the **Enabled** state, the CCS and PED flags are cleared. If the **Disconnected** State is entered from the **Reset** state, the CCS and PR flags are cleared. Notice that the big bubble to Disconnected state transition does define any actions related to these flags.

Note: For transition Actions: The notation **Wr(**Field Name=value**)** indicates a software write to the PORTSC register of "value" to the respective field, and Field Name=value without the "Wr()" wrapper indicates a transition of the respective field to the "value".

Note: The figures in this section are provided to illustrate state transition conditions and actions, however refer to the textual descriptions of the respective states for their explicit definition.

Note: The Root Hub Port state machines in the following subsections only references the *Port Link State Change* (PLC) flag, refer to section 4.19.2 for information on how the remaining change flags are affected by the Root Hub Port state machines.

Note: The xHCI state machines describe the exit conditions from a state and entry conditions to a state. Only conditions specifically described as an entry or exit condition shall result in a state transition, e.g. setting the *PR* flag in the Disconnected state has no effect on the state of a port because that condition is not cited in section 4.19.1.1.2.

Refer to section 5.4.8 for the details of change bit operation

4.19.1.1 USB2 Root Hub Port

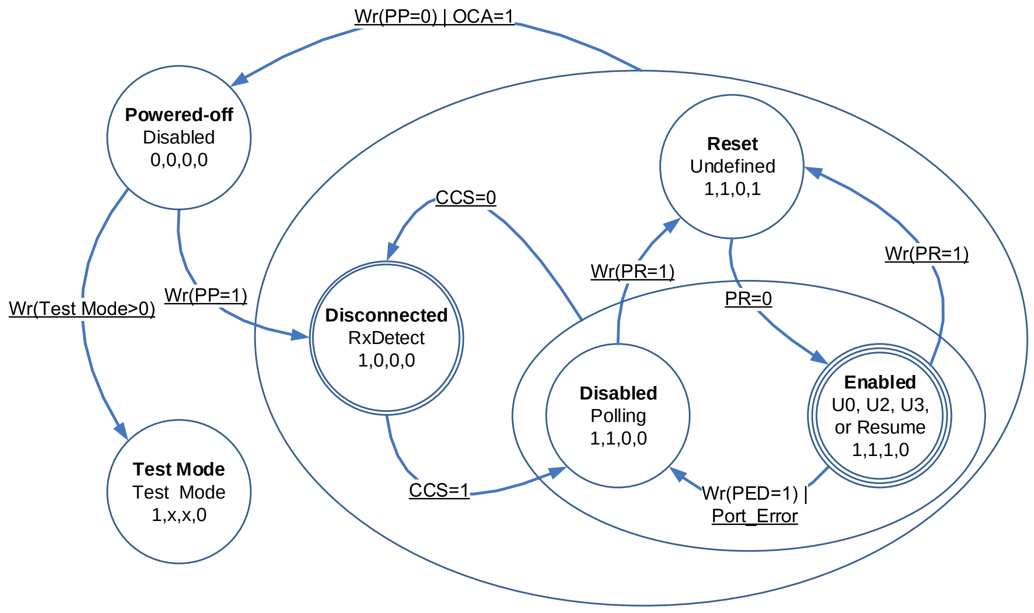


Figure 4-25: USB2 Root Hub Port State Machine

Figure 4-25 illustrates the top level transitions in a USB2 Protocol Root Hub state machine.

Note: The "Change" flags (*CSC*, *PEC*, *POCC*, *PRC*, *PLC*, and *CEC*) are set to '1' upon the detection of the respective condition. Refer to Table 5-26 for the definition of the change flags.

The initial state is **Disconnected**.

4.19.1.1.1 Powered-off

A write to the PORTSC register with *PP* set to '0' or an Over-current condition shall transition from any state to the **Powered-off** state.

A write to the PORTSC register with *PP* set to '1' shall transition from the **Powered-off** state to the **Disconnected** state.

A write to the USB2 PORTPMSC register with *Test Mode* greater than '0' shall transition from the **Powered-off** state to the **Test Mode** state.

A write to the PORTSC register with *PP* cleared to '0', or an over-current condition (OCA => '1') shall transition from the port from any state to the **Powered-off** state.

4.19.1.1.2 Disconnected

This is the initial state after initial xHC Aux Power-up or *HCRST*.

A device connect detect (CCS = '1') shall transition the port from the **Disconnected** state to the **Disabled** state and set the *CSC* flag to '1'.

A disconnect detect (*CCS* = '0') in the **Disabled** or **Enabled** state[57] shall transition the port to the **Disconnected** state, set the *CSC* flag to '1', and if *PR* or *PED* flags are set to '1', they shall be cleared to '0'.

4.19.1.1.3 Disabled

A write to the PORTSC register with *PR* set to '1' shall transition the port from the **Disabled** state to the **Reset** state.

4.19.1.1.4 Reset

When the Reset operation completes (*PR* = '0'), the port shall automatically advance to the **Enabled** state, setting *PED and PRC to* '1'.

Software shall ignore the value of the *Port Link State* (PLS) field while in the **Reset** state.

4.19.1.1.5 Test Mode

Refer to section Port Test Modes for operation of Port Test Modes.

Note: The *Current Connect Status* (CCS) and *Port Enabled/Disabled* (PED) Signal States vary as function of the selected Test Mode.

4.19.1.1.6 Enabled

While in the **Enabled** state a write to the PORTSC register with *PED* set to '1', or a *Port_Error* (refer to section 11.8.1 of the USB2 spec for conditions that may cause a Port_Error) shall transition the port from any **Enabled** substate to the **Disabled** state. If the transition was due to a Port_Error the *PEC* flag shall be set to '1'.

[57]Note that a disconnect cannot be detected by a USB2 port in the **Reset** state because the host is driving the bus.

Figure 4-26: USB2 Root Hub Port Enabled Substate Diagram

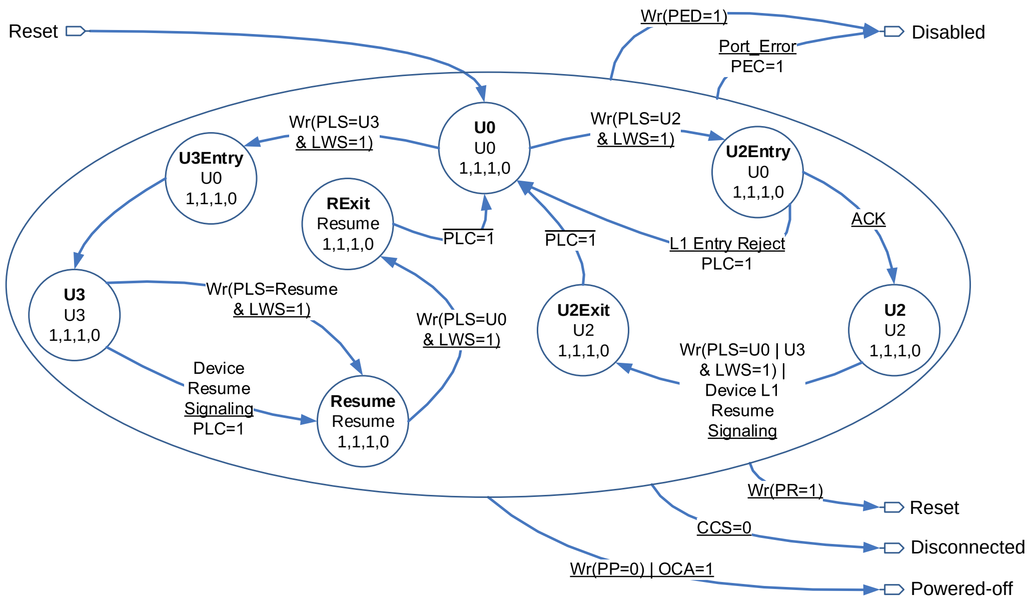


Figure 4-26 illustrates the **Enabled** substate transitions in a USB2 Protocol Root Hub state machine.

While in any of the **Enabled** substates:

- If the PORTSC register is written with *PP* = '0' or and over-current condition is detected (OCA = '1') then the respective substate shall exit to the **Powered-off** state.
- If a Disconnect condition is detected (CCS = '0') then the respective substate shall exit to the **Disconnected** state.
- If the PORTSC register is written with *PR* = '1' then the respective substate shall exit to the **Reset** state.

4.19.1.1.7 U0

Entry to the **Enabled** state always transitions to the **U0** substate.

A write to the PORTSC register with the *PLS* field set to *U2* and *LWS* set to '1' shall cause the xHC to issue an LPM transaction to the device and transition the port to the **U2Entry** substate.

A write to the PORTSC register with the *PLS* field set to *U3* and *LWS* set to '1' shall cause the xHC to suspend the device, and transition the port to the **U3Entry** substate.

If the entry to the U0 state was from the U2Exit substate due to a write to the PORTSC register with the *PLS* field set to *U3* and *LWS* set to '1' in the U2 substate, then the port shall automatically transition the port to the **U3Entry** substate, and suspend the device.

4.19.1.1.8 U2Entry

In this state the xHC shall attempt to transition the device to the L1 suspend state by issuing an LPM transaction to the device:

- If the device responds with an ACK handshake (the L1 suspend attempt was successful), the port shall set the *L1S* field to *Success* (‘1’) and transition to the **U2** substate, and the device shall enter the L1 standby state.
- If the device responds with a NYET handshake (the L1 suspend attempt was rejected by the device), the port shall set the *L1S* field to *Not Yet* (‘2’) and transition to the **U0** substate and set the *PLC* flag to ‘1’ (PLC Condition: L1 Entry Reject), and the device shall remain in the L0 state. Note that in this case there is no PLS transition, it shall remain in the U0 state.
- If the device responds with a STALL handshake (the L1 suspend attempt was not recognized by the device), the port shall set the *L1S* field to *Not Supported* (‘3’), transition to the **U0** substate, and set the *PLC* flag to ‘1’ (PLC Condition: L1 Entry Reject).
- If a Timeout occurs or a Transaction Error is detected (the L1 suspend attempt was unsuccessful), the port shall set the *L1S* field to *Timeout/Error* (‘4’), transition to the **U0** substate, and set the *PLC* flag to ‘1’ (PLC Condition: L1 Entry Reject).

Note that when the STALL, Timeout, or transaction Error cases above occur software may inspect the USB2 PORTPMSC register *L1S* field to determine the specific cause of the transition. Refer to section 4.23.5.1.1 for more information on the *L1S* result values.

Refer to sections 4.15.2 and 4.23.5 for more information on USB2 LPM operation.

4.19.1.1.9 U2

The port is in the L1Suspended state and shall remain in the **U2** substate until a Host or Device Initiated Resume occurs.

Host Initiated L1 Resume - A write to the PORTSC register with the *PLS* field set to *U0* or *U3* and *LWS* set to ‘1’ shall cause the port to initiate resume signaling to the device and transition to the **U2Exit** substate.

Device Initiated L1 Resume - If Resume Signaling is generated by the device, then the port shall transition to the **U2Exit** substate.

If the entry to the **U0** state was from the **U2Exit** substate due to a write to the PORTSC register with the *PLS* field set to *U3* and *LWS* set to '1' in the **U2** substate, then the port shall automatically transition the port to the **U3Entry** substate, and suspend the device.

4.19.1.1.10 U2Exit

When the resume signaling is complete and the device has entered the L0 state, the port shall transition to the **U0** substate and set the *PLC* flag to '1' (PLC Condition: USB2 L1 Resume complete).

4.19.1.1.11 U3Entry

In this state the xHC shall wait for transfers associated with the current microframe or other internal operations to complete before Idling the bus and suspending the device. When the enters the Idle state, the port shall transition to the **U3** substate, and if *U3C* and *U3E* = '1', set *PLC* flag to '1' (PLC Condition: U3 Entry complete).

4.19.1.1.12 U3

The port is in the Idle state (i.e. suspended) and shall remain in the **U3** state until a Host or Device Initiated Resume occurs.

Note: Section 7.1.7.6 of the USB2 specification states that devices begin to transition "into the Suspend state after they see a constant Idle state on their upstream facing bus lines for more than 3.0 ms. The device must actually be suspended, drawing only suspend current from the bus, after no more than 10 ms of bus inactivity on all its ports."

The PLS field of a USB2 Root Hub port reflects the Idle state of the port's bus lines, not whether the attached device has actually transitioned to the Suspend state.

Host Initiated Resume - A write to the PORTSC register with the *PLS* field set to *Resume* and *LWS* set to '1' shall cause the xHC to initiate resume signaling to the device and transition to the **Resume** substate.

Device Initiated Resume - If Resume Signaling is generated by the device, the port shall transition to the **Resume** substate, initiate resume signaling to the device, and set the *PLC* flag to '1' (PLC Condition: Wakeup signaling from a device).

4.19.1.1.13 Resume

A write to the PORTSC register with the *PLS* field set to *U0* and *LWS* set to '1' shall cause the port to transition to the **RExit** substate and complete the resume signaling.

Note: Software shall time the duration of the **Resume** state. Software shall remain in the **Resume** state long enough to ensure the resume sequence, as specified in the USB2 spec, completes successfully.

4.19.1.1.14 RExit

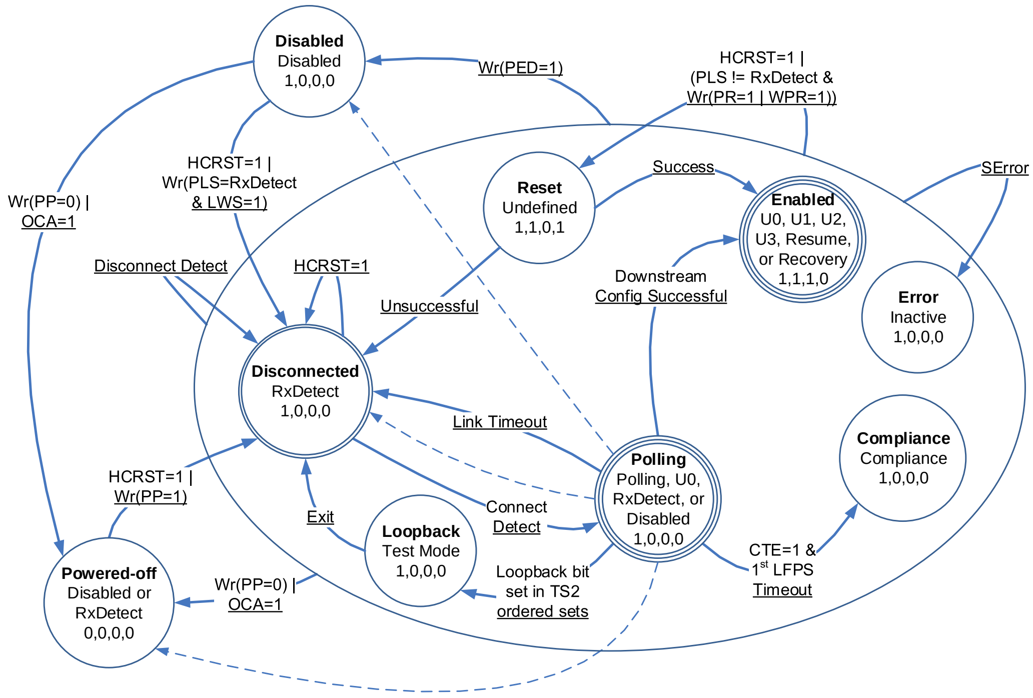
When the resume signaling is complete and the device has entered the L0 state, the port shall transition to the **U0** substate and set the *PLC* flag to ‘1’ (PLC Condition: USB2 Device Resume complete).

4.19.1.2 USB3 Root Hub Port

Figure 4-27 illustrates the top level transitions in a USB3 Protocol Root Hub state machine.

Refer to Table 5-26 for the conditions that affect the change flags.

Figure 4-27: USB3 Root Hub Port State Machine



The initial state is **Disconnected**.

Note: The dashed arrows represent optional state transitions that may occur if the Debug Capability is supported. Refer to section 4.19.1.2.4.3 for more information.

Note: Figure 4-27 does not illustrate a transition from the **Enabled** state to the **Loopback** state, which may occur. Refer to note in section 4.19.1.2.14 for additional information on transitions to the **Loopback** state.

4.19.1.2.1 Disabled

A write to the PORTSC register with the *PED* field set to ‘1’ shall transition the port, from any state except **Powered-off**, to the **Disabled** state.

A write to the PORTSC register with the *PLS* field set to *RxDetect* and *LWS* set to '1' shall transition the port to the **Disconnected** state.

A write to the USBCMD register with the *HCRST* flag set to '1' shall transition the port to the **Disconnected** state.

A write to the PORTSC register with *PP* cleared to '0' or an over-current condition (OCA = '1') shall transition the port to the **Powered-off** state.

4.19.1.2.2 Powered-off

A write to the PORTSC register with *PP* cleared to '0' shall transition from the port from any state to the **Powered-off** state.

An over-current condition (OCA = '1') shall transition from the port from any state to the **Powered-off** state, and if the *CCS*, *PR* or *PED* flags are set to '1', they shall be cleared to '0'.

A write to the PORTSC register with *PP* set to '1' shall transition the port to the **Disconnected** state.

A write to the USBCMD register with the *HCRST* flag set to '1' shall transition the port to the **Disconnected** state.

4.19.1.2.3 Disconnected

This is the initial state after initial xHC Aux Power-up.

Note: If a port has transitioned to this state from the **Powered-off** or **Disabled** states due to the assertion of *HCRST* by software, then a Hot or Warm Reset shall be issued by the port when its LTSSM enters to the Rx.Detect state or after a receiver detection in the Rx.Detect state.

Note: The completion of *Host Controller Reset* (i.e. the HCRST '1' to '0' transition) does not depend on the completion of any port activity other than entering the *Disconnected* state.

The assertion of *HCRST* = '1' shall cause the port to remain in the **Disconnected** state.

Note: An xHC implementation may assert *WPR* or *PR* to reflect the associated reset operation if *HCRST* is asserted while the port is in the **Disconnected** state.

A device *Connect Detect*[58] shall transition the port to the **Polling** state.

[58]SuperSpeed far-end receiver terminations are detected. Refer to section 6.11 in the USB3 spec.

A Disconnect Detect[59] in the any state, except **Powered-off** or **Disabled** shall transition the port to the **Disconnected** state.

4.19.1.2.4 Polling

While in the **Polling** state the port may transition between the **Training**, **CfgExcg**, and **DbC** substates.

Figure 4-28: USB3 Root Hub Port Polling Substate Diagram

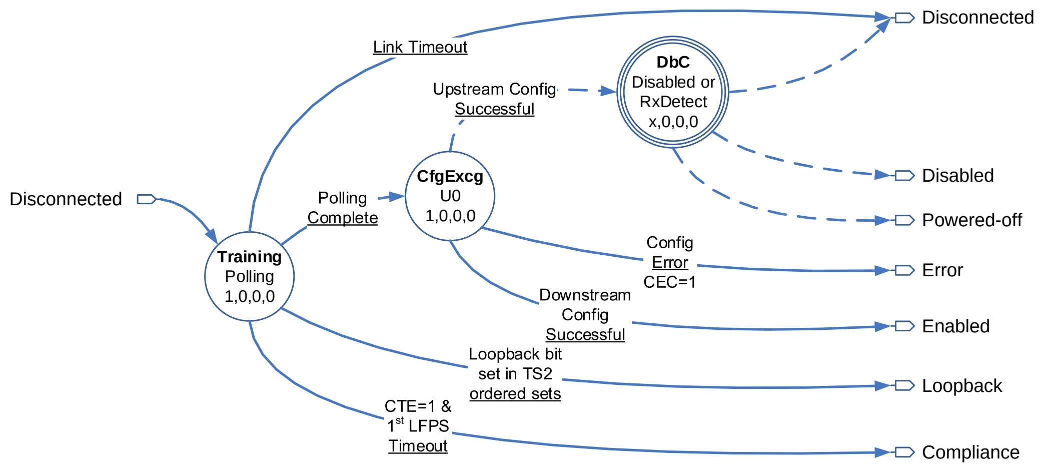


Figure 4-28 illustrates the **Polling** substate transitions in a USB3 Protocol Root Hub state machine.

Note: The dashed arrows represent optional state transitions that may occur if the Debug Capability is supported. Refer to section DbC for more information.

4.19.1.2.4.1 Training

Entry to the **Polling** state always transitions to the **Training** substate.

A Connect Detect shall cause the port to transition to the **Training** substate.

If Training completes successfully, the substate shall transition to the **CfgExcg** substate.

[59]A LTSSM transition from the any state to the Rx.Detect state due to *Removal(DS Port Only)*. Refer to section 7.5 in the USB3 spec.

If Training fails due to a Link Timeout[60] or a *Disconnect Detect*[59], the substate shall exit to the **Disconnected** state.

If the *Compliance Transition Capability* (CTC) flag in the HCCPARAMS2 register = ‘1’, then the xHC supports software control of the transition to the Compliance state and *Compliance Transition Enabled* (CTE) flags. The *CTE* flag is an internal xHCI flag associated with each Root Hub port, i.e. *CTE* is not software visible. If *CTE* = ‘1’ then the transition path to the **Compliance** substate shall be enabled, otherwise the transition is disabled. Upon Chip Hardware Reset, the assertion of *HCRST* = ‘1’, or a Warm Reset (*WPR* = ‘1’), *CTE* shall be cleared to ‘0’. Only if the port is in the **Disconnected** state, then a write to the PORTSC register with the *PLS* field set to *Compliance Mode* and *LWS* set to ‘1’ shall set *CTE* to ‘1’.

If *CTE* = ‘1’, then the detection of the first *LFPS Timeout* shall transition the substate to the **Compliance** state.

The reception of a TS2 Ordered Set with the Loopback bit set shall transition the substate to the **Loopback** state.

4.19.1.2.4.2 CfgExcg

In this state, the Port Capabilities and Port Configuration LMPs are exchanged as described in sections 8.4.5 and 8.4.6 of the USB3 spec.

If the port is successfully configured as a downstream facing port (Downstream Config Successful), the substate shall exit to the **Enabled** state.

If the port is successfully configured as an upstream facing port (Upstream Config Successful), the substate shall transition to the **DbC** substate. Note that this transition shall never occur if the xHC does not support the xHCI Debug Capability.

Note: If the xHCI Debug Capability is enabled and a Debug Host has not been detected yet, the *Direction* field of the *Port Capabilities LMP* shall be set to ‘3’ for all ports, indicating that the Root Hub port is both upstream and downstream capable. An *Upstream Config Successful* condition indicates that a downstream facing port is attached to a Root Hub port and implies that a Debug Host is attached. Once a port is mapped to the Debug Capability, all remaining ports shall assert ‘1’ (i.e. the port shall only be configured as downstream) in the *Direction* field of subsequent *Port Capabilities LMP*s. Refer to section 7.6 for more information on the operation of the xHCI Debug Capability.

Note: If a Debug Host and a Debug Target are cabled together, but the xHCI Debug Capability has not yet been enabled, then the *Direction* field of the *Port*

[60]Refer to section 7.5.4 in the USB3 spec for the LTSSM conditions that shall transition a downstream port from the Polling to the Rx.Detect state. Note, the LTSSM Rx.Detect state maps to the USB3 Port state machine **Disconnected** state.

Capabilities LMP shall be set to '1' for all ports, indicating that the Root Hub ports on both the Debug Host and the Debug Target are only downstream capable. After the link trains, a *Config Error* condition will occur because two downstream ports are connected together (i.e. an undefined Port Type selection occured), causing the port to transition to the **Error** state. If software resets the port to recover from the error, this *Config Error* scenario will repeat until the cable is disconnected or the DbC is enabled.

If the port fails to successfully configure (Config Error), the substate shall set the *CEC* flag to '1' and exit to the **Error** state.

Note: In Figure 4-28, the Upstream and Downstream "Config Successful" transitions may require the resolution of a "Tiebreaker" with the exchange of one or more *Port Capability LMPs* if the link partner also asserts the Upstream and Downstream *Direction* flags in the *Port Capability LMP* and the *Tiebreaker* fields are equal, refer to section 8.4.5 in the USB3 spec.[61]

4.19.1.2.4.3 DbC

In this substate, the port is mapped to the Debug Capability and the DbgCap substates shall emulate a port that never detects an attach

Note: This substate is optional, and shall only exist for xHC implementations that support the xHCI Debug Capability.

While in the **DbC** (Debug Capability) substate the port may transition between the **DbC Disconnected**, **DbC Disabled**, and the **DbC Powered-off** substates.

Note: Section 7.5 of the USB3 spec describes the behavior of the LTSSM for upstream and downstream facing ports. The default behavior of an xHC Root Hub is that of a downstream facing port. However, while in the **DbC** state the LTSSM of a Root Hub port is mapped to the Debug Capability and shall behave as an upstream facing port.

[61]A DbC enabled port may bias the *Port Capability LMP* exchange so that it becomes an upstream facing port by randomly choosing low Tiebreaker values, e.g. < 8.

Figure 4-29: USB3 Root Hub Port DbC Substate Diagram

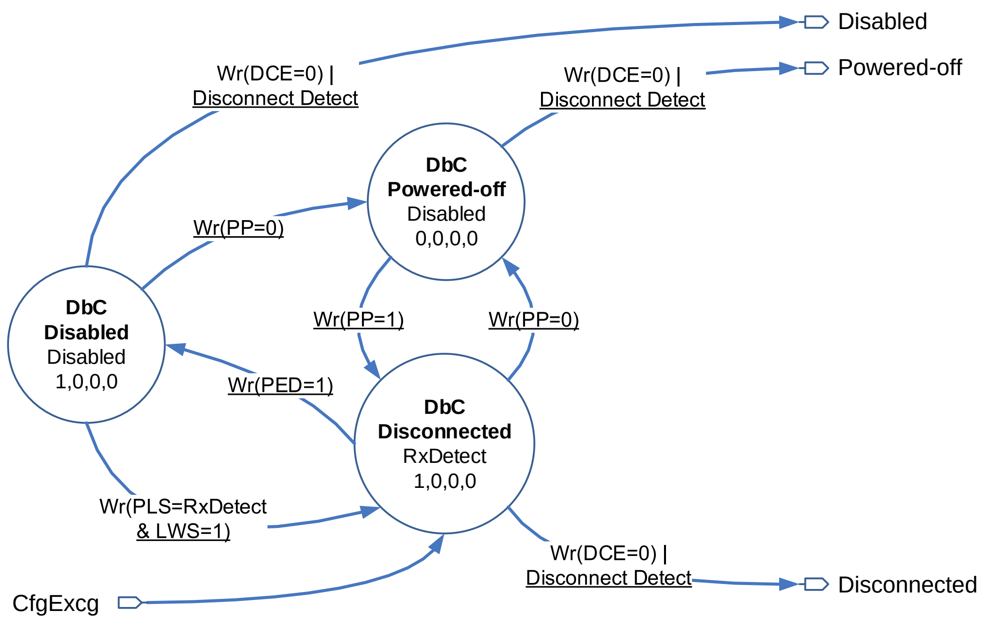


Figure 4-29 illustrates the **DbC** substate transitions in a USB3 Protocol Root Hub state machine.

4.19.1.2.4.3.1 DbC Disconnected

Entry to the **DbC** substate always transitions to the **DbC Disconnected** substate.

A write to the PORTSC register with the *PED* field set to '1' shall transition the port to the **DbC Disabled** substate.

A write to the PORTSC register with the *PP* field set to '0' shall transition the port to the **DbC Powered-off** substate.

A write to the DCCTRL register with the *DCE* field set to '0' or a *Disconnect Detect*[59], shall transition the port to the **Disconnected** state.

4.19.1.2.4.3.2 DbC Disabled

A write to the PORTSC register with the *PLS* field set to *RxDetect* and *LWS* set to '1' shall transition the port to the **DbC Disconnected** substate.

A write to the PORTSC register with the *PP* field set to '0' shall transition the port to the **DbC Powered-off** substate.

A write to the DCCTRL register with the *DCE* field set to '0' or a *Disconnect Detect*[59], shall transition the port to the **Disabled** state.

4.19.1.2.4.3.3 DbC Powered-off

A write to the PORTSC register with the *PP* field set to ‘1’ shall transition the port to the **DbC Disconnected** substate.

A write to the DCCTRL register with the *DCE* field set to ‘0’ or a *Disconnect Detect*[59], shall transition the port to the **Disconnected** state.

4.19.1.2.5 Reset

A write to the PORTSC register with *PR* or *WPR* set to ‘1’ or a write to the USBCMD register with *HCRST* set to ‘1’, shall transition the port from any state except the **Disconnected**, **Powered-off**, or **Disabled** states, to the **Reset** state.

If the Reset operation completes successfully, the port shall transition to the **Enabled** state, clearing *PR* to ‘0’ and setting *PED* to ‘1’.

If the Reset operation does not complete successfully, the port shall transition to the **Disconnected** state.

Note: If a port has transitioned to this state due to the assertion of *HCRST* by software, then a Hot or Warm Reset shall be issued by the port when its LTSSM enters to the Rx.Detect state. Depending on the link state when *HCRST* is asserted, an xHC implementation may choose to issue a Hot Reset rather than a Warm Reset to accelerate the USB recovery process.

Note: *PRC* shall be set upon exiting the **Reset** state. However the *WRC* flag shall also be set, if software set the *PR* flag and the “Hot” Reset transitioned to a Warm Reset or if software set the *WPR* flag initially to enter the **Reset** state. Refer to the note section 10.3.1.6 of the USB3 spec for more information on a Hot Reset to Warm Reset transition.

A *Disconnect Detect*[59] shall transition the port to the **Disconnected** state.

Software shall ignore the value of the *Port Link State* (PLS) field while in the **Reset** state.

4.19.1.2.6 Error

The port shall transition to the **Error** state if a serious error condition (SError) occurs while attempting to operate the link, i.e. the LTSSM transitions to the SS.Inactive state, an unsuccessful LTSSM Loopback.Exit, etc. Refer to section 10.3.1.4 “DSPORT.ERROR” of the USB3 spec for the *SError* conditions that shall cause a Root Hub port to transition to the **Error** state.

The transition to the **Error** state shall set the *PLC* flag to ‘1’ (PLC Condition: Error).

A *Disconnect Detect*[59] shall transition the port to the **Disconnected** state.

4.19.1.2.7 Compliance

A write to the PORTSC register with the *PED* field set to ‘1’ shall transition the port, to the **Disabled** state.

A write to the PORTSC register with *WPR* set to ‘1’ or the assertion of *HCRST* = ‘1’ shall transition the port to the **Reset** state.

Refer to section 4.19.1.2.2 for the conditions that shall transition the port to the **Powered-off** state.

4.19.1.2.8 Loopback

A successful Exit (LFPS handshake in the LTSSM Loopback.Exit state) or a *Disconnect Detect*[59] shall transition the port to the **Disconnected** state.

Refer to section 4.19.1.2.2 for the conditions that shall transition the port to the **Powered-off** state.

A *Timeout* in the LTSSM Loopback.Exit state (tLoopBackExitTimeout) shall transition the port to the **Error** state.

Note: Refer to note in section Recovery for additional information on transitions to the **Loopback** state.

4.19.1.2.9 Enabled

While in the **Enabled** state a the port may transition between the **U0**, **U1’**, **U2’**, **U3’** and **Recovery** substates.

Figure 4-30: USB3 Root Hub Port Enabled Substate Diagram

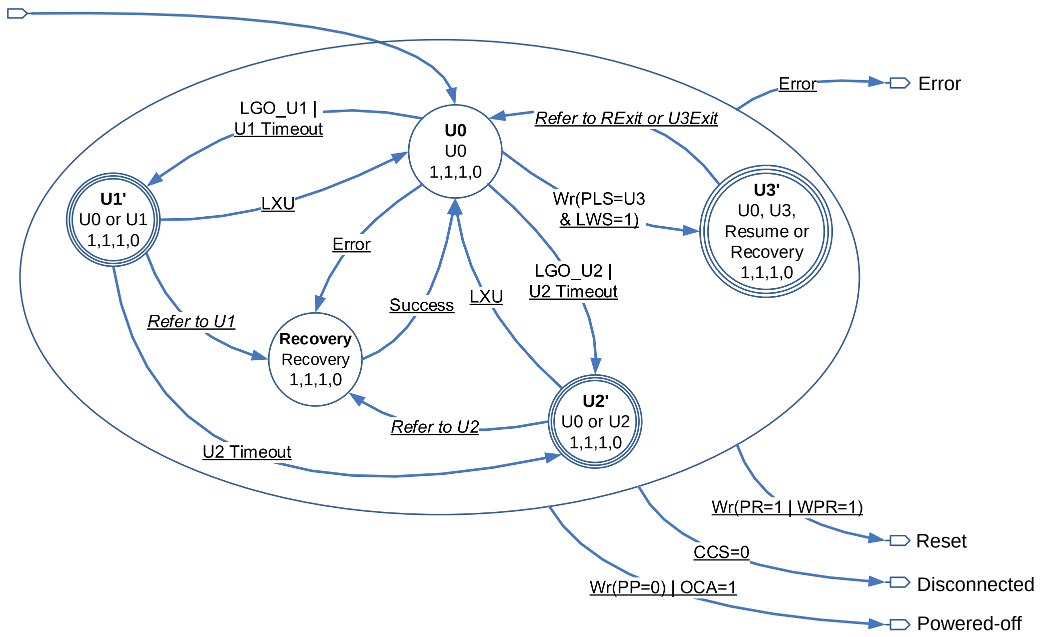


Figure 4-30 illustrates the **Enabled** substate transitions in a USB3 Protocol Root Hub state machine.

While in any **Enabled** substates:

- If the PORTSC register is written with *PP* = '0' or and over-current condition is detected (OCA = '1'), then the respective substate shall exit to the **Powered-off** state.
- If a Disconnect condition is detected (CCS = '0'), then the respective substate shall exit to the **Disconnected** state.
- If the PORTSC register is written with *PR* = '1' or *WPR* = '1', then the respective substate shall exit to the **Reset** state.
- If a condition[62] transitions the *Port Link State* (PLS) to the *Inactive* state, then the respective substate shall exit to the **Error** state.

Refer to sections 4.19.1.2.10, 4.19.1.2.11, 4.19.1.2.12, 4.19.1.2.13, and 4.19.1.2.14 for more information on the **Enabled** substates.

Note: Figure 4-30 does not illustrate a transition from the **Recovery** or **U3'** states to the **Loopback** state, however they may occur. Refer to note in section Recovery for additional information on transitions to the **Loopback** state.

[62]e.g. if a Ux_EXIT_TIMER timeout occurs in the **Recovery** state while attempting to transition from the **U1'** or **U2'** state to the **U0** state.

4.19.1.2.10 U0

Entry to the **Enabled** state always transitions to the **U0** substate.

The reception of an LGO_U1 from the link partner or a *U1 Timeout* shall cause the port to transition to the **U1'** substate.

The reception of an LGO_U2 from the link partner, or a *U2 Timeout* shall cause the port to transition to the **U2'** substate.

A write to the PORTSC register with the *PLS* field set to *U3* and *LWS* set to '1' shall cause the port to transition to the **U3'** substate.

If the entry to the U0 state was from the Recovery substate due to a write to the PORTSC register with the *PLS* field set to *U3* and *LWS* set to '1' in the U1 or U2 substate, then the port shall automatically transition the port to the **U3'** substate, and suspend the device.

The port shall transition to the **Recovery** substate if errors defined in section 7.3 of the USB3 spec occur.

4.19.1.2.11 U1'

Figure 4-31: USB3 Root Hub Port U1' Substate Diagram

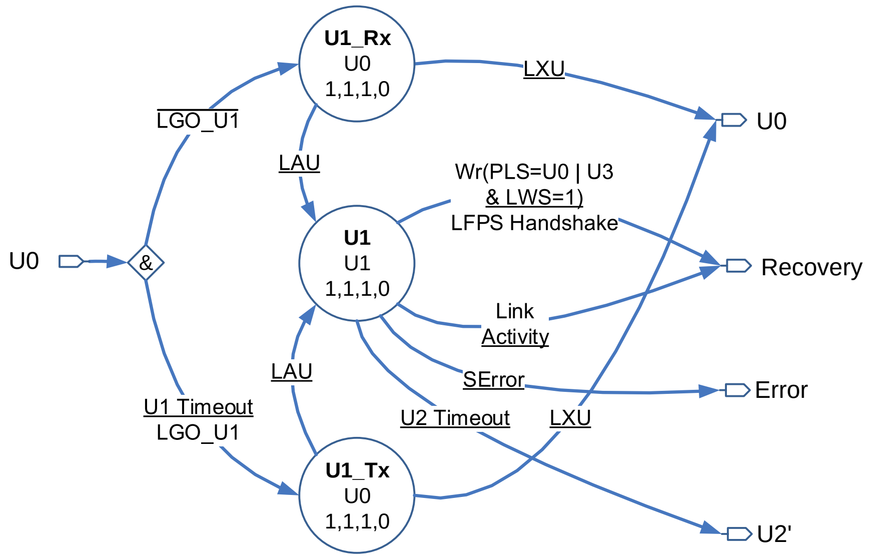


Figure 4-31 illustrates the **U1'** substate transitions in a USB3 Protocol Root Hub state machine.

If the transition to the U1' substate was due to an LGO_U1 received from the device, the xHC shall transition to the **U1_Rx** substate.

If the transition to the U1 substate was due to a *U1 Timeout*, the xHC shall transition to the **U1_Tx** substate.

Refer to sections 4.19.1.2.11.1 , 4.19.1.2.11.2 , and 4.19.1.2.11.3 for more information on the **U1'** substates.

4.19.1.2.11.1 U1_Rx

xHC implementation specific power management policies determined whether to accept or reject the LGO_U1 request. If the request is accepted the xHC shall transmit an LAU and transition to the **U1** substate. If the request is rejected the xHC shall transmit an LXU and transition to the **U0** substate.

4.19.1.2.11.2 U1_Tx

Device implementation specific power management policies determined whether the LGO_U1 request from the host shall be accepted or rejected. If the request is accepted the xHC shall receive an LAU and transition to the **U1** substate. If the request is rejected the xHC shall receive an LXU and transition to the **U0** substate.

4.19.1.2.11.3 U1

The port is in the LTSSM U1 state.

Host Initiated U1 Resume - A write to the PORTSC register with the *PLS* field set to *U0* or *U3,* and *LWS* set to '1' shall cause the xHC to initiate an LFPS Handshake with the device. If the handshake is successful, the device has entered the U0 state, the port shall exit the **U1'** substate machine and transition to the **U0** substate.

Device Initiated U1 Resume - If an LFPS Handshake is initiated by the device completes successfully, the port shall exit the **U1'** substate machine, and transition to the **U0** substate.

A *U2 Timeout* shall cause the port to transition to the **U2'** substate.

An *SError* shall cause the port to transition to the **Error** state.

4.19.1.2.12 U2'

Figure 4-32: USB3 Root Hub Port U2' Substate Diagram

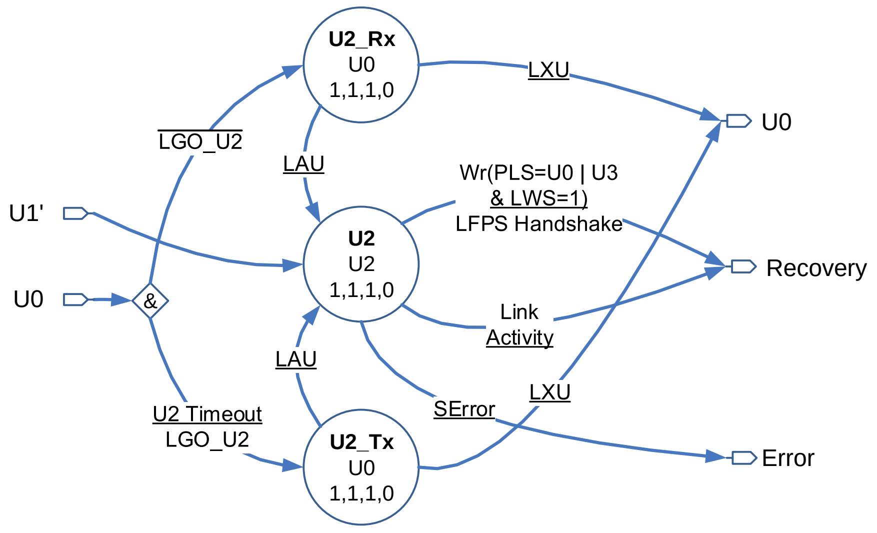


Figure 4-32 illustrates the **U2'** substate transitions in a USB3 Protocol Root Hub state machine.

If the transition to the **U2'** substate was due to an LGO_U2 received from the device, the xHC shall transition to the **U2_Rx** substate.

If the transition to the **U2'** substate was due to a *U2 Timeout*, the xHC shall transition to the **U2_Tx** substate.

If the transition to the **U2'** substate was from the **U1'** state (due to an *L2 Timeout*), the xHC shall transition to the **U2** substate.

Refer to sections 4.19.1.2.12.1 , 4.19.1.2.12.2 , and 4.19.1.2.12.3 for more information on the **U2'** substates.

4.19.1.2.12.1 U2_Rx

xHC implementation specific power management policies determined whether to accept or reject the LGO_U2 request. If the request is accepted the xHC shall transmit an LAU and transition to the **U2** substate. If the request is rejected the xHC shall transmit an LXU and transition to the **U0** substate.

4.19.1.2.12.2 U2_Tx

Device implementation specific power management policies determined whether the LGO_U2 request from the host shall be accepted or rejected. If the request is accepted the xHC shall receive an LAU and transition to the **U2** substate. If the request is rejected the xHC shall receive an LXU and transition to the **U0** substate.

4.19.1.2.12.3 U2

The port is in the LTSSM U2 state.

Host Initiated U2 Resume - A write to the PORTSC register with the *PLS* field set to *U0* or *U3*, and *LWS* set to '1' shall cause the xHC to initiate an LFPS Handshake with the device. If the handshake is successful, the device has entered the U0 state, the port shall exit the **U2'** substate machine, and transition to the **U0** substate.

Device Initiated U2 Resume - If an LFPS Handshake is initiated by the device completes successfully, the port shall exit the **U2'** substate machine, and transition to the **U0** substate.

An *SError* shall cause the port to transition to the **Error** state.

4.19.1.2.13 U3'

Figure 4-33: USB3 Root Hub Port U3' Substate Diagram

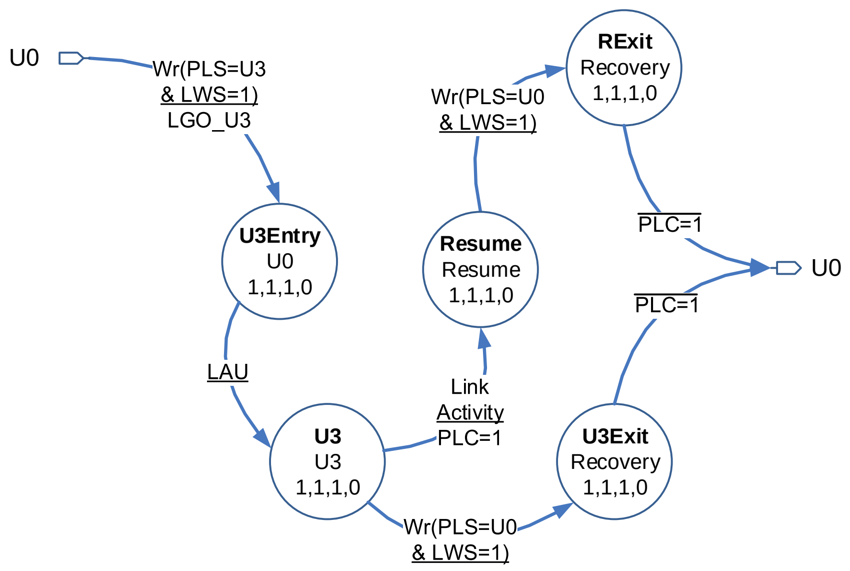


Figure 4-33 illustrates the **U3'** substate transitions in a USB3 Protocol Root Hub state machine.

Upon entry into the **U3'** substate machine transitions to the **U3Entry** substate.

Refer to sections 4.19.1.2.13.1 , 4.19.1.2.13.2 , 4.19.1.2.13.3 , and 4.19.1.2.13.4 for more information on the **U3** substates.

Note: Figure 4-33 does not illustrate a transition from the **RExit** or **U3Exit** states to the **Loopback** state, however it may occur. Refer to note in section 4.19.1.2.14 for additional information on transitions to the **Loopback** state.

4.19.1.2.13.1 U3Entry

The port shall remain in this substate until a LAU is received from the device, then transition to the **U3** substate, and if *U3C* and *U3E* = ‘1’, set *PLC* flag to ‘1’ (PLC Condition: U3 Entry complete).

4.19.1.2.13.2 U3

The port is suspended and shall remain in the **U3** substate until a Host or Device Initiated Resume occurs.

Host Initiated Resume - A write to the PORTSC register with the *PLS* field set to *U0* and *LWS* set to ‘1’ shall cause the xHC to initiate *Link Activity* (LFPS Handshake) with the device and transition to the **U3Exit** substate.

Device Initiated Resume - If *Link Activity* (LFPS Handshake) is initiated by the device, the port shall not respond, exit the **U3** substate machine, transition to the **Resume** substate, and set the *PLC* flag to ‘1’ (PLC Condition: Wakeup signaling from a device).

4.19.1.2.13.3 Resume

A write to the PORTSC register with the *PLS* field set to *U0* and *LWS* set to ‘1’ shall cause the xHC to initiate *Link Activity* (LFPS Handshake) with the device and transition to the **RExit** substate.

4.19.1.2.13.4 RExit

The LTSSM is in the *Recovery* state. Refer to section 7.5.10 in the USB3 spec.

When the handshake is successful; the port shall exit the **U3’** substate machine, transition to the **U0** substate, and set *PLC* flag to ‘1’ (PLC Condition: USB3 Device Resume completion). Note that a USB device is not allowed to reject a resume request from the host.

If a TS2 Ordered Set is received with the Loopback bit set, the port shall transition to the **Loopback** state.

If a TS2 Ordered Set is received with the Reset bit set, the port shall transition to the **Reset** state.

Note: Refer to note in section Recovery for additional information on transitions to the **Loopback** state.

4.19.1.2.13.5 U3Exit

The LTSSM is in the *Recovery* state. Refer to section 7.5.10 in the USB3 spec.

When the handshake is successful; the port shall exit the **U3’** substate machine, transition to the **U0** substate, and set *PLC* flag to ‘1’ (PLC Condition: USB3

Software Resume complete). Note that a USB device is not allowed to reject a resume request from the host.

If a TS2 Ordered Set is received with the Loopback bit set, the port shall transition to the **Loopback** state.

If a TS2 Ordered Set is received with the Reset bit set, the port shall transition to the **Reset** state.

Note: Refer to note in section Recovery for additional information on transitions to the **Loopback** state.

4.19.1.2.14 Recovery

The LTSSM is in the *Recovery* state. Refer to section 7.5.10 in the USB3 spec.

If the recovery completes successfully; the port shall transition to the **U0** substate.

If the recovery does not complete successfully; the port shall transition to the **Error** state.

If a TS2 Ordered Set is received with the Loopback bit set, the port shall transition to the **Loopback** state.

Note: The xHC USB3 Root Hub Port state machine figures do not illustrate a transition from the **Enabled**, **Recovery**, **RExit**, or **U3Exit** to **Loopback** state transition, however they may occur.

The LTSSM shall transition from the *Recovery.Idle* state to the *Loopback* state if a TS2 Ordered Set is received with the Loopback bit set. A xHC Root Hub port is a Loopback Slave. To perform loopback tests a specialized Test Device is required. The Test Device, which acts as Loopback Master, may transition a port to the **Enabled** state before transitioning the port to **Loopback** state. However, typically a Loopback Master will only assert the Loopback bit in a TS2 Ordered Set when it is initially connected, asserting an LTSSM *Polling.Idle* to *Loopback* transition.

4.19.2 Port Status Change Generation

The xHC defines a *Port Status and Control* (PORTSC) register for each Root Hub port.

There are seven *status change bits* in the PORTSC register *Connect Status Change* (CSC), *Port Enabled/Disabled Change* (PEC), *Warm Port Reset Change* (WRC), *Over-current Change* (OCC), *Port Reset Change* (PRC), *Port Link State Change* (PLC), and *Port Config Error Change* (CEC), Refer to section 5.4.8 for more information on these bits.

Root Hub port *status change bits* may be set due to hardware or software initiated conditions. When set, these bits remain set until cleared by a system software write to the PORTSC register with the appropriate *status change bit(s)* set to '1', or the assertion of a Chip Hardware Reset or *HCRST*.

When a *status change bit* is set in a PORTSC register, if the assertion of a *status change bit* results in a '0' to '1' transition of PSCEG (4.19.2), the xHC responds by generating a *Port Status Change Event* (as described in section 6.4.2.3) and/or asserting a Power Management Event (PME#). Refer to Table A-2 for more information on *Port Status Change Event* and PME# generation. The host system normally receives Root Hub port status change notifications through *Port Status Change Events*, however the "Wake on" flags in the PORTSC register can be used to manage the assertion of PME# due to port status changes. Refer to section 4.15 for more information on wake operation.

The *Connect Status Change* (CSC) bit shall be asserted if there is any connection change, i.e. connect or disconnect, i.e. a '1' to '0' or '0' to '1' transition of *CCS or CAS*.

The *Port Enabled/Disabled Change* (PEC) shall be asserted only by a USB2 protocol port when the *Port Enabled/Disabled* (PED) flag transitions to *Disabled* due to a Port_Error, i.e. a '1' to '0' transition of *PED*.

The *Warm Port Reset Change* (WRC) bit is set only when a warm reset completes, i.e. a '1' to '0' transition of *WPR*.

The *Over-current Change* (OCC) bit is set when an over-current condition is detected, i.e. a '0' to '1' transition of *OCA*.

The *Port Reset Change* (PRC) bit is set when any reset (hot or warm) completes, i.e. a '1' to '0' transition of *PR* or *WPR*.

Note: The definition of *PRC* states that it is set "when any reset processing (Warm or Hot) on this port is complete". If an over-current condition (*OCA* => '1') occurs while a port is in the **Reset** state, then reset processing is aborted and the *PR* flag shall be cleared. Hardware may or may not set the *PRC* and *WRC* flags under these conditions. If the *OCC* flag is set, then the *PRC* and *WRC* flags should be ignored by software.

The *Port Link State Change* (PLC) bit is asserted for specific *Port Link State* (PLS) field transitions. Refer to section 4.19.1 for the specific Root Hub port state transitions that will assert *PLC*.

The *Port Config Error Change* (CEC) bit is set only when a Port Configuration error is detected. Note that there is no corresponding port config status or error flag in the PORTSC register, so the assertion of CEC is the only means of flagging this error condition.

The *Port Status Change Event* reports port status changes on a per-port basis. The *Port ID* field of the *Port Status Change Event TRB* (shown in Table 88), indicates which port has experienced a status change.

System software shall acknowledge status change(s) by clearing the respective PORTSC *status change bit(s)*. The acknowledgment clears the change state for that port so future status changes may be reported.

Note: There are no coherency guarantees between a software read of PORTSC register and corresponding Events reflecting PORTSC changes, i.e., If software reads the PORTSC and sees a change bit set, there is no guarantee that the corresponding event has been written into the Event ring.

Figure 4-34 shows an example creation mechanism for *Port Status Change Event* and PME# generation.

A '0' to '1' transition of the ***Port Status Change Event Generation*** (PSCEG) signal shall cause a Port Status Change Event to be generated. PSCEG is an internal xHC variable, not directly exposed to software.

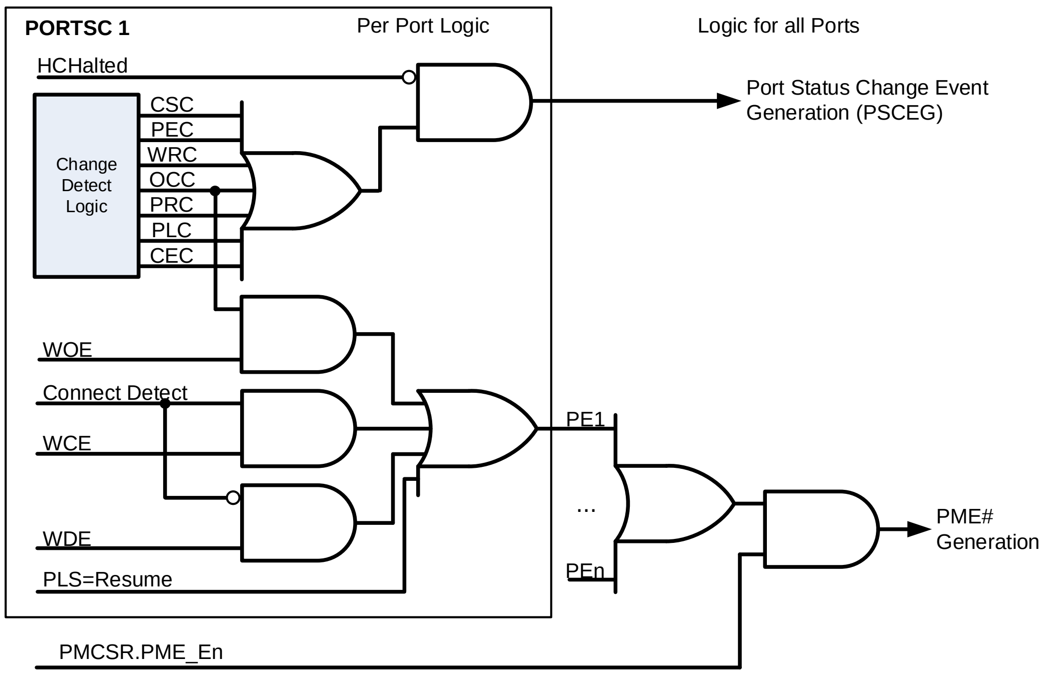
Note: The generation of a *Port Status Change Event* is triggered by the assertion of the PSCEG signal. Due to internal xHC scheduling and system delays, there will be a lag between a change bit being set and the *Port Status Change Event* that it generated being written to the Event Ring. If SW reads the PORTSC and sees a change bit set, there is no guarantee that the corresponding *Port Status Change Event* has already been written into the Event Ring.

Note: There are no ordering requirements between Transfer Events and Port Status Change Events. e.g. a due to a disconnect, transfer events for the disconnected device may be placed on an Event Ring after the *Port Status Change Event* generated by the port.

The change bits (CSC. PEC, etc.) of each port are ORed together and gated by the *HCHalted* (HCH) flag to form the *Port Status Change Event Generation* signal. A port shall generate a *Port Status Change Event* when there is '0' to '1' transition of the PSCEG signal.

The PME wake events detected by each port (PEx) are ORed together and gated by the PCI PM PMSCR.PME_En flag. Refer to Appendix A.1.1 for more information. PME# shall be asserted when there is a '0' to '1' transition of the ***PME# Generation*** signal.

Figure 4-34: Example Port Change Bit Port Status Change Event Generation



Note: A *Port Status Change Event* may be the result of multiple *status change bits* being set.

Note: *Port Status Change Events* for a port are blocked until all *status change bits* are cleared ('0'), i.e. PSCEG = '0'.

Note: Under some conditions the xHC may not be capable of generating *Port Status Change Events*, i.e. if *HCHalted* (HCH) = '1' or the Event Ring is full. If the *HCHalted* (HCH) = '0' and the Event Ring is not full, the xHC shall generate *Port Status Change Events*.

Note: For USB2 ports the *Connect Detect* signal is identical to CCS.
For USB3 ports the *Connect Detect* signal is asserted when SuperSpeed far-end receiver terminations are detected, and negated if there is a LTSSM transition from the any state to the Rx.Detect state due to Removal(DS Port Only). Refer to section 7.5 in the USB3 spec.

4.19.3 Connect Status Change Reporting

The xHC shall perform the following operations when *Port Power* is asserted (PP = '1') and a *USB Device* attach is detected on a Root Hub port:

1. The *CCS* bit in the respective PORTSC register is set to '1', indicating that a device presence has been detected.

2. The *CSC* bit in the respective PORTSC register is set to '1', indicating that a transition has been detected in the *CCS* bit.

3. If the assertion of *CSC* results in a '0' to '1' transition of PSCEG, post a *Port Status Change Event* TRB with the following field values to the *Event Ring*.

   - *TRB Type* = Port Status Change Event.
   - *Port ID* = Port Number of the Root Hub Port that detected the device attach
   - *Completion Code* = Success
   - *Cycle bit* = Current Event Ring Producer Cycle State.

When software parses the *Port Status Change Event*, it can evaluate the *Port ID* field to determine the Root Hub port that was the source of the change event. And examine the port's PORTSC register to determine that the event was generated by a *Connect Status Change* (*CSC* = '1') and that the change was an Attach (*CCS* = '1').

For a USB2 Protocol port, "device presence" is indicated by the PORTSC *PLS* field transitioning from the RxDetect to the Polling state. Software shall reset the port to transition it to the U0 state.

For a USB3 Protocol port, "device presence" is indicated by the PORTSC *PLS* field transitioning from the Polling to the U0 state.

4.19.4 Port Power

The *Port Power Control* (PPC) flag indicates whether the xHC supports port power switches.

Whether an xHC implementation supports port power switches or not, it shall automatically enable VBus on all Root Hub ports after a Chip Hardware Reset or HCRST. The initial state of an xHCI Root Hub ports shall be the **Disconnected** state, i.e. *Port Power* (PP) is asserted, and the port is waiting for signaling on the USB that indicates a device is attached.

Note: After a Chip Hardware Reset the xHC is allowed to delay the assertion of the *Port Power* (PP) flag until after the software sets *Max Device Slots Enabled* (MaxSlotsEn) field in *Configure* (CONFIG) register. This feature allows an implementation to hold off device and link power consumption until a driver is loaded.

This requirement means that Root Hub port may report a device is connected (*CCS* and *CSC* = '1') before the xHC is running (i.e. *HCHalted* (HCH) = '0'), and that when software enables the xHC and *HCHalted* (HCH) transitions to '0', *PSCEG* shall be asserted for each port with a connected device, generating a respective *Port Status Change Event*. In this case:

- A USB2 protocol port shall be in the **Disabled** state.

- A USB3 protocol port shall be in the **Enabled** state.

When *PP* = '0':

- The port is forced to the **Powered-off** state.
- *CSC* shall be asserted if the *PP* transitioned to '0' due to an over-current condition. If PP transitions to '0' for any other condition no status change flags or wake-up events shall be asserted.
- The port's receiver and transmitter are disabled, however the port's receiver terminations shall be maintained.

When *PP* transitions from '0' to '1':

- If device is not connected, then a USB2 or USB3 protocol port shall transition to the **Disconnected** state.
- If a device is connected:
    - A USB2 protocol port shall transition to the **Disabled** state.
    - A USB3 protocol port shall transition to the **Disconnected** state, detect the device and immediately transition to the **Polling** state.
        - If training is successful, the port sets the *CSC* flag to '1' and transitions to the **Enabled** state.
        - If training is not successful[63], the port transitions to the **Disconnected** state.
        - If a timeout is detected on the first LFPS handshake, the port transitions to the **Compliance** state and no change flag is set.
        - If the Loopback bit is set in a TS2 Ordered set, the port transitions to the **Loopback** state and no change flag is set.

Note: While Chip Hardware Reset or HCRST is asserted, the value of *PP* is undefined. If the xHC supports power switches (*PPC* = '1') then VBus may be deasserted during this time. *PP* (and VBus) shall be enabled immediately upon exiting the reset condition.

Note: Before the xHC driver is unloaded, the driver should clear the *Port Power* (PP) flag of all Root Hub ports to place them into the *Disabled* state and reduce port power consumption.

4.19.4.1 Enabled U0 States

There are 4 **Enabled** state **U0** pseudo-states that differ only in the values that are configured for the U1 and U2 timeouts. The *U1 Timeout* and *U2 Timeout*

[63]Refer to section 7.5.4 in the USB3 spec for the LTSSM conditions that shall transition a downstream port from the Polling to the Rx.Detect state. Note, the LTSSM Rx.Detect state maps to the USB3 Port State Machine **Disconnected** state.

values for the port default to ‘0’. The *U1 Timeout* and *U2 Timeout* values may be set by software by writing the PORTPMSC register at any time.

Each Root Hub port maintains a logical *PM Timers* for keeping track of when the U1 or U2 inactivity timeout are exceeded. The U1 or U2 timeout values may be set by software writing the *U1 Timeout* and *U2 Timeout* fields of the USB3 PORTPMSC register at any time. The PM timers are reset to ‘0’ every time the USB3 PORTPMSC register is written. The timers shall be reset every time a packet of any type except an isochronous timestamp packet is sent or received by the port’s link. The *U1 PM Timer* shall be accurate to +1/- 0 µs. The *U2 PM Timer* shall be accurate to +500/-0 µs.

The port behaves as follows for the various combinations of *U1 Timeout* and *U2 Timeout* values:

U1 Timeout = 0, *U2 Timeout* = 0

- This is the default state before the PORTPMSC register is written.
- The port’s link shall reject all U1 or U2 transition requests by the link partner.
- The PM Timers may be disabled and the PM Timer values shall be ignored.
- The port’s link shall not attempt to initiate transitions to U1 or U2.

U1 Timeout = X[64] > 0, U2 Timeout = 0

- The port’s link shall reject all U2 transition requests by the link partner.
- The PM timers shall be reset when this state is entered and the link is active.
- The port’s link shall accept U1 entry requests by its link partner unless the xHC has one or more packets/link commands to transmit on the port.
- If the *U1 Timeout* = FFh, the port shall be disabled from initiating U1 entry but shall accept U1 entry requests by the link partner unless the xHC has one or more packets/link commands to transmit on the port.
- If the *U1 Timeout* < FFh and the U1 PM Timer reaches X, the port’s link shall initiate a transition to U1. In this case the delay defined by the *U1 Timeout* field represents an amount of inactive time in U0.

U1 Timeout = 0, U2 Timeout = Y[65] > 0

- The port’s link shall reject all U1 transition requests by the link partner.
- The PM Timers shall be reset when this state is entered and the link is active.

[64]The value defined by the U1 Timeout field. Refer to Table 5-28 for U1 Timeout values.

[65]The value defined by the U2 Timeout field. Refer to Table 5-28 for U2 Timeout values.

- The port's link shall accept U2 entry requests by its link partner unless the xHC has one or more packets/link commands to transmit on the port.
- If the *U2 Timeout* = FFh, the port shall be disabled from initiating U2 entry but shall accept U2 entry requests by the link partner unless the xHC has one or more packets/link commands to transmit on the port.
- If the *U2 Timeout* < FFh and the U2 PM Timer reaches Y, the port's link shall initiate a direct transition from U0 to U2. In this case the delay defined by the *U2 Timeout* field represents an amount of inactive time in U0.

U1 Timeout =X > 0, *U2 Timeout* = Y > 0

- The PM Timers shall be reset when this state is entered and is active.
- The port's link shall accept U1 or U2 entry requests by its link partner unless the xHC has one or more packets/link commands to transmit on the port.
- If the *U1 Timeout* = FFh, the port shall be disabled from initiating U1 entry but shall accept U1 entry requests by the link partner unless the xHC has one or more packets/link commands to transmit on the port.
- If the *U1 Timeout* < FFh and the U1 PM Timer reaches X the port's link shall initiate a transition to U1.
- If the *U2 Timeout* < FFh and the U2 PM Timer reaches Y the port's link shall initiate a direct transition from U1 to U2. In this case the delay defined by the *U2 Timeout* field represents an amount of time in U1.
- If the *U2 Timeout* = FFh, the port shall be disabled from initiating U2 entry but shall accept U2 entry requests by the link partner unless the xHC has one or more packets/link commands to transmit on the port.

A port transitions to one of the Enabled U0 states (depending on the *U1 Timeout* and *U2 Timeout* values) in any of the following situations:

- From any state if software writes the PORTSC register and sets the *PLS* field to U0 ('0').
- From U1 if the link partner successfully initiates a transition to U0.
- From U2 if the link partner successfully initiates a transition to U0.
- From U1 if the xHC successfully initiates a transition to U0 after receiving a packet routed to the port.
- From U2 if the xHC successfully initiates a transition to U0 after receiving a packet routed to the port
- From an attempt to transition from the U0 to the U1 state if the downstream port's link partner rejects the transition attempt
- From an attempt to transition from the U0 to the U2 state if the downstream port's link partner rejects the transition attempt

- From U3 if software writes the PORTSC register and sets the *PLS* field to U3 ('3') and the Root Hub port received wakeup signaling while it was in U3.

4.19.5 Port Reset

Resetting a Root Hub port resets the attached USB device, and if successful; the port logic reports the speed of the attached device and transitions the port to the **Enabled** state. Whether successful or not a change bit is set ('1'). And if setting the change bit results in a '0' to '1' transition of PSCEG, then a *Port Status Change Event* shall be generated.

When system software writes the PORTSC register with the *PR* bit set to '1', the xHC shall:

1. Update the PORTSC register:
   - Set the *PR* bit ('1').
   - Clear the *PED* bit to the disabled state ('0').
2. Execute the appropriate reset signaling to the device attached to the port.

If the bus reset sequence completes successfully, the xHC shall update the PORTSC register:

- Set the *PLS* field to *U0* ('0').
- Clear the *PR* bit ('0').
- Set *PED* to the enabled state ('1').
- Set the *PRC* bit ('1').
- For a USB3 protocol port, if a Hot Reset transitioned to a Warm Reset, set the *WRC* bit ('1').
- Set *Port Speed* field to the speed of the newly attached device.

If the bus reset sequence does *NOT* complete successfully, the xHC shall update the PORTSC register:

- Set the *PLS* field to *RxDetect* ('5').
- Clear the *PR* bit ('0').
- Set the *PRC* bit ('1').
- For a USB3 protocol port, if a Hot Reset transitioned to a Warm Reset, set the *WRC* bit ('1').
- Set the *Port Speed* field to *Undefined Speed* ('0').
- Clear the *CCS* bit ('0').

If setting *PRC* results in a '0' to '1' transition of PSCEG, then generate a *Port Status Change Event* with the following field values.

- *TRB Type* = Port Status Change Event.

- *Port ID* = Port Number of the Root Hub Port that detected the Reset change transition.
- *Completion Code* = *Success*.
- *Cycle bit* = Current Event Ring Producer Cycle State.

Note: Only a USB3 protocol port may fail the bus reset sequence. USB2 protocol ports never fail the bus reset sequence.

Note: When *PR* transitions from '1' to '0', the USB device is in the "Default state" (i.e. Responding to USB Device Address 0). System software should immediately transition the device to the Address state (with an Address Device Command) or disable the port, to allow the enumeration of other newly attached USB devices.

Note: Speed detection is performed by the port hardware during the bus reset sequence, hence the *Port Speed* field of the PORTSC register shall **not** be considered valid by software until after the *PR* bit transitions from a '1' to a '0'.

Note: A "Successful Reset" is determined by the xHC hardware for the attached device.

4.19.5.1 Warm Port Reset

The USB3 specification distinguishes between "Hot" and "Warm" port reset sequences. A Warm Reset_performs all the functions of Hot Reset, e.g. transitioning a port to the **Enabled** state and resetting the USB device to the Default state, however it also resets a USB3 link, forcing the link to enter the Rx.Detect state and re-exchange link configuration information. A Warm Reset also takes longer than a Hot Reset to execute.

The operations performed during a Hot Reset are described in the section above (4.19.5). The operations performed for a Warm Reset are similar, except that software initially writes the PORTSC register with the *Warm Port Reset* (WPR) bit set to '1'. The *Port Reset* (PR) flag shall be '1' while Hot or Warm Reset is being executed. The *Port Reset Change* (PRC) flag shall be set ('1') when the reset execution is complete and *PR* transitions to '0'.

If the '1' to '0' transition of *PR* was due to a software initiated Warm Reset, or Hot Reset that transitioned to a Warm Reset because of errors[66], the *Warm Reset Change* (WRC) flag (and *PRC*) shall be asserted ('1').

Note: The PORTSC *WPR* and *WRC* bits only apply to USB3 protocol ports. The bits shall be RsvdZ for USB2 protocol ports.

[66]Refer to section 10.3.1.6 of the USB3 spec, "Note: If the port initiates a hot reset on the link and the hot reset TS1/TS2 handshake fails a warm reset is automatically tried."

4.19.6 Port Test Modes

For USB2 protocol Root Hub ports, the xHC shall implement the port test modes **Test_J_State**, **Test_K_State**, **Test_Packet**, **Test_Force_Enable**, and **Test_SE0_NAK** as described in the USB2 Specification. For USB3 protocol Root Hub ports, no test modes are supported. System software is allowed to have at most one port in test mode at a time. Placing more than one port in test mode may yield undefined results. The required, per port test sequence is:

- Disable all Device Slots.
- All ports shall be in the *Disabled* state (*PP* = ‘0’).
- Set the *Run/Stop* (R/S) bit in the USBCMD register to a ‘0’ and wait for the *HCHalted* (HCH) bit in the USBSTS register, to transition to a ‘1’. Note that an xHC implementation shall not allow port testing with the *R/S* bit set to a ‘1’.
- Set the *Port Test Control* field in the port under test PORTPMSC register to the value corresponding to the desired test mode.
    - For USB2 ports, if the selected test is **Test_Force_Enable**, then after selecting the test the *Run/Stop* (R/S) bit in the USBCMD register shall then be transitioned back to ‘1’ by software, in order to enable transmission of SOFs out of the port under test.
- When the test is complete, if the xHC is running system software shall clear the *R/S* bit and ensure the host controller is halted (*HCHalted* (HCH) bit is a ‘1’).
- Terminate and exit test mode by setting *HCRST* to a ‘1’.

4.19.7 Port Routing and Control

A USB3 hub is the logical combination of two hubs: a USB 2.0 hub and an Enhanced SuperSpeed hub, where each hub operates on a separate upstream facing connection (data bus). When a USB3 Hub is attached to a Root Hub port it may actively utilize both the USB2 and Enhanced SuperSpeed connections, depending on the speed of the devices attached to the hub’s downstream facing ports. Note that a USB Peripheral Device is required to only utilize one connection at a time.

In a USB3 hub, two independently addressable hub ports exist for each physical down stream connector; a USB2 compatible port accessed through the USB2 connection and a USB3 compatible port accessed through the SuperSpeed connection. The Root Hub of the xHCI emulates this operation by defining a Root Hub PORTSC register for each connection type; USB2 (Low-/Full-/High-Speed) or USB3 (Enhanced SuperSpeed).

Due to pin-out, power, or other implementation issues an xHC implementation may support a different number of USB2 connections than USB3. The “type” of a USB connection is defined by the protocol that it supports. The *xHCI Supported Protocol Extended Capability* (defined in section 7.2) identifies the set of Root

Hub Ports associated with a specific protocol. Refer to Table 7-11 for a list of the supported protocols.

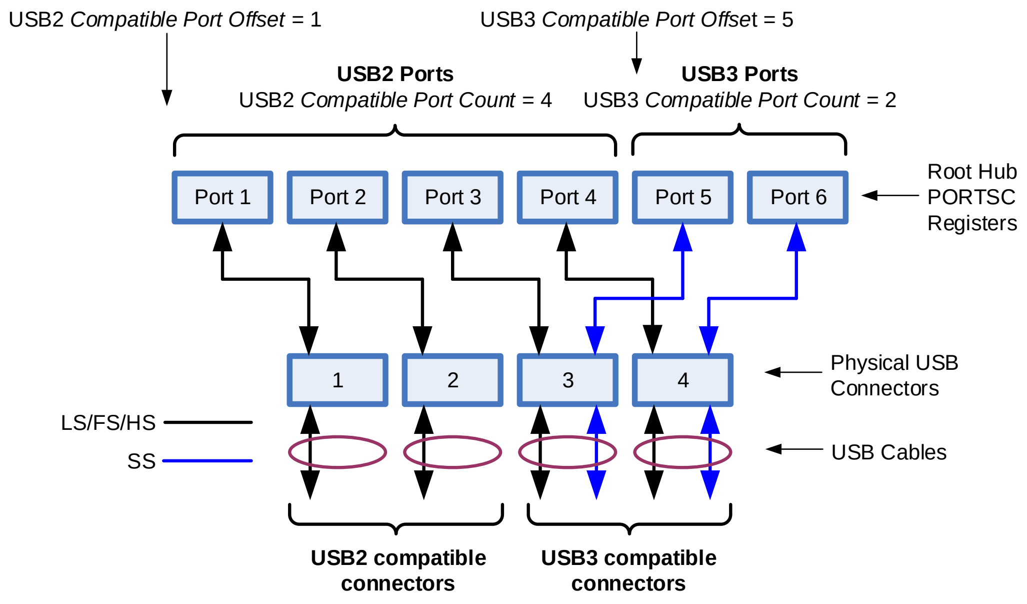
Note: A Root Hub port that supports the *USB3 protocol* is comprised of a PORTSC, a USB3 PORTPMSC, and PORTLI register (sections 5.4.8, 5.4.9.1, and 5.4.10.1), and Root Hub port that supports the *USB2 protocol* is comprised of a PORTSC and a USB2 PORTPMSC register (refer to sections 5.4.8 and 5.4.9.2).

The mapping of xHCI Root Hub Ports to the physical USB connectors of a system is defined by platform implementations and outside the scope of this specification. Refer to Appendix D for a method of mapping xHCI Root Hub ports to system USB connectors.

Note: xHC Root Hub ports are numbered from 1 to *MaxPorts*. *MaxPorts* is defined in the HCSPARAMS1 register (5.3.3).

Consider the example of an xHC implementation illustrated in Figure 4-35 that supports two protocols (USB2 and USB3) and 6 connections, where 4 connections are USB2 compatible and 2 are USB3 compatible. In this case, two *xHCI Supported Protocol Extended Capability* data structures would be declared. If the USB2 *xHCI Supported Protocol Extended Capability* data structure defined the *Compatible Port Offset* equal to '1' and the *Compatible Port Count* equal to '4', and the USB3 *xHCI Supported Protocol Extended Capability* data structure defined the *Compatible Port Offset* equal to '5' and the *Compatible Port Count* equal to '2', then Root Hub Ports 1 through 4 would reflect the attachment of USB2 devices, and Root Hub Ports 5 and 6 would reflect the attachment of USB3 devices.

Figure 4-35: Port Routing Example



USB2 Compatible Port Offset = 1
USB3 Compatible Port Offset = 5
USB2 Ports
USB2 Compatible Port Count = 4
USB3 Ports
USB3 Compatible Port Count = 2
Port 1
Port 2
Port 3
Port 4
Port 5
Port 6
Root Hub PORTSC Registers
1
2
3
4
Physical USB Connectors
LS/FS/HS
SS
USB Cables
USB2 compatible connectors
USB3 compatible connectors

IMPLEMENTATION NOTE

Port Power

Implementations shall OR together the output of the PORTSC register Port Power pins for Root Hub Ports that map to the same Physical USB Connector. Refer to section 10.10 in the USB3 spec for more information on hub port power control.

In Figure 4-35, asserting the *Port Power* flag in Root Hub Port 3 or 5 shall assert Vbus to Physical USB Connector 3, asserting the *Port Power* flag in Root Hub Port 4 or 6 shall assert Vbus to Physical USB Connector 4, etc.

4.19.8 Cold Attach Status

For USB2 protocol ports the *Current Connect Status* (CCS) flag is capable of reporting a device attach in any xHC power state. However, for USB3 protocol ports *CCS* is asserted only after the link has successfully trained and advanced to the U0 state. This is a problem if an xHC implementation is incapable of advancing a link to U0 while in the D3 state, which can occur if the LTSSM clocks required to train the link are not running. And without clocks, *CCS* cannot be used to assert PME#.

The *Cold Attach Status* (CAS) flag addresses this issue by asserting itself ('1') if:

- SuperSpeed Far-end Receiver Terminations are detected,
- The xHC is placed into the D3 state, and is in a low power state where the LTSSM and the controller clocks are stopped, or the controller is powered down (e.g. the LTSSM is unable to Train)), and
- The LTSSM is not in the Error, U3, or Disabled state.

Note that *CAS* is only asserted under these circumstances. It is not a general purpose indicator that a USB3 device is attached. Also, *CAS* does not apply to USB2 protocol ports and shall always be '0'.

A transition of CAS shall assert *Connect Status Change* (CSC).

Before software places the xHC into the D3 state it should perform the following operations:

- Halt any device activity.
- For each USB3 device that it wants to be awakened by:
    - Issue a SetFeature(FUNCTION_SUSPEND, Function Remote Wake Enable) request.
- For all connected devices:
    - Transition their Root Hub ports to the **Enabled:U3** state (suspend).

    - Set the PORTSC *Wake On Disconnect Enable* (WDE) flag, if wake on disconnect is desired.
- For all ports in the **Disconnected** state:
    - Set the PORTSC *Wake On Connect Enable* (WCE) flag, if wake on connect is desired.
- For all ports:
    - Set the PORTSC *Wake On Over-current Enable* (WOE) flag, if wake on over-current is desired.

The state of any port in the **Disconnected**, **Powered-off**, or **Disabled** state is not changed.

This approach allows wake enabled devices to wake up the system, provides suspend current to all other devices, and enables PME# to be asserted if a disconnect, connect, or overcurrent condition is detected.

When software is awaked by a PME it should:

- Turn on the Core Power Well to transition the xHC from the D3cold to the D0 state.
- Restore the Scratchpad, and all xHC register values and memory data structures that were saved before the xHC was placed in the D3cold state.
- Set the xHC running (*R/S* = '1').
- Follow the recommendations in section 4.15.2.2 for resuming any Root Hub ports that it had previously suspended.
- Check all remaining xHC Root Hub ports for *CAS* = '1' and issue a *Warm Port Reset* (WPR) to any port if it is asserted.

The assertion of *WPR* clears *CAS*.

Note: The assertion of *CCS* may also clear *CAS* if, after turning on the Core Power Well, the LTSSM of a port is able to successfully transition to the U0 state.

4.20 Scratchpad Buffers

The Scratchpad Allocation mechanism of the xHCI allows the xHC to request one or more PAGESIZE buffers of system memory for storing internal state. The PAGESIZE register is defined in section 5.4.3.

The number of pages that the xHC requires is identified by the **Max Scratchpad Buffers Hi** and **Lo** fields in the HCSPARAMS2 register (section 5.3.4). An xHC implementation may declare zero *Max Scratchpad Buffers*.

A **Scratchpad Buffer** is a PAGESIZE block of system memory located on a PAGESIZE boundary.

System software shall allocate the Scratchpad Buffer(s) before placing the xHC in to Run mode (*Run/Stop* (R/S) = '1').

The **Scratchpad Buffer Array** contains pointers to the *Scratchpad Buffers*. Entry 0 of the Device Context Base Address Array points to the *Scratchpad Buffer Array*. The *Scratchpad Buffer Array* data structure is described in section 6.6.

Features of xHC Scratchpad Allocation:

- The xHC may request multiple Scratchpad Buffers.
- When accessing a Scratchpad Buffer the xHC shall not access system memory addresses outside of the PAGESIZE memory block allocated by system software.
- System software shall not read or write a Scratchpad buffer. System software writes to the Scratchpad buffer memory may result in undefined xHC operation.
- The content of the Scratchpad Buffers shall remain intact across system power events including D3.cold if *SPR* = '1'. Refer to the *SPR* definition in Table 5-11.

The following operations take place to allocate Scratchpad Buffers to the xHC:

1. Software examines the *Max Scratchpad Buffers Hi* and *Lo* fields in the HCSPARAMS2 register.
2. Software allocates a *Scratchpad Buffer Array* with *Max Scratchpad Buffers* entries.
3. Software writes the base address of the *Scratchpad Buffer Array* to the DCBAA (Slot 0) entry.
4. For each entry in the *Scratchpad Buffer Array*:
    a. Software allocates a PAGESIZE Scratchpad Buffer.
    b. Software clears the Scratchpad Buffer to '0'.
    c. Software writes the base address of the allocated *Scratchpad Buffer* to associated entry in the *Scratchpad Buffer Array*.

Note: If the *Scratchpad Restore* (SPR) field in the HSCPARAMS2 register = '1', then the xHC shall use Scratchpad Buffers to store its internal state when executing the *Save State* operation. For the *Restore State* operation to work successfully, the content of the Scratchpad Buffers shall be intact when exiting a power down state (D3.cold). Refer to section 4.23.2 for more information.

Note: xHC references to the Scratchpad Buffer Array and Scratchpad Buffers should not snoop. Refer to section 4.18.2.2.

IMPLEMENTATION NOTE

FSC and Context handling by Save and Restore

The xHC Contexts provide public and private areas, e.g. The *EP State*, *TR Dequeue Pointer*, etc. fields in the Endpoint Context are public, and the *xHCI Reserved* (Opaque / RsvdO) areas are private. The xHCI spec leaves it as an implementation decision whether the Endpoint Context Opaque areas, the Scratchpad, or internal memory is used for caching/saving internal endpoint state.

Section 4.23.2 describes the sequence of events that should take place to save and restore the state of the xHC when suspending a system. The 1.0 specification assumes that *Stop Endpoint Commands* would push public and private endpoint state into memory, and the Save State operation would push all remaining internal xHC state into memory, so that system software could save it, and restore it later.

If an implementation uses the Scratchpad or non-volatile internal memory to cache endpoint state rather than Endpoint Contexts, then the Save State operation can flush that state to the Scratchpad and software does not have to issue *Stop Endpoint Commands* as part of the suspend process. The *Force Save Context Capability* (FSC) flag was defined to distinguish between xHCI implementations that require explicit *Stop Endpoint Commands* to all endpoints as part of the suspend process (*FSC* = ‘0’) , and those that only require *Stop Endpoint Commands* for *Running:Busy* endpoints (*FSC* = '1').

Note that some xHC implementations support FSC-like behavior but predate the definition of the flag, as a result, some OS drivers assume FSC-like behavior is supported, and do not issue *Stop Endpoint Commands* to all endpoints before executing a Save State operation. It is highly recommended that all new xHC implementations support FSC to ensure compatibility with legacy OS drivers.

4.21 PCI Express

Note: This section utilizes PCI Express (PCIe) terminology and refers to PCIe constructs (Physical Layer, Receiver Errors, Data Link or Transaction layers, etc.). Refer to the PCIe Specification for more information on Error Events and Error Reporting and Configuration Registers.

4.21.1 Configuration sharing among PCI functions

An xHC contains a single physical PCIe core interface. In Normal mode, the xHCI is designed so that all USB devices (Device Slots 0-n) appear in a single function. In Virtualization mode, the xHCI is designed to appear as distinct Virtual Functions, where each of the USB devices (Device Slots 0-n) may be mapped exclusively to a specific Virtual Function. In Normal case, the xHCI implements, amongst other registers, the PCIe device header space as described in section 5.2. In Virtualization case, the VMM implements the PCIe device header space through emulation.

4.21.2 Bus Master Enable (BME)

System software may occasionally need to disable the bus mastering capability of the xHC. In a PCI system, this is accomplished by setting the *Bus Master Enable* (BME) bit of the *Device Control Register* in PCI Configuration register space, to '0'. The xHC should be Halted, i.e. with the *Run/Stop* (R/S) bit set to '0', and *HCHalted* (HCH) verified as being '1' before system software disables bus master activity by clearing the *BME* bit. If the *BME* bit is set to '0' when the xHC is running, the xHC may treat this as a *Host Controller Error*, asserting *HCE* (‘1’) and immediately halt (*R/S* = ‘0’ and *HCH* = ‘1’). Recovery from this state will require an *HCRST*. Refer to section Internal Errors for more information.

4.22 xHCI Extended Capabilities

4.22.1 Pre-OS to OS Handoff Synchronization

A system configuration may include support in the BIOS (also referred herein as Pre-OS software) for control of the xHC. The OS Handoff Synchronization capability provides the mechanisms to allow a BIOS to enable SMI support for xHC events and also a set of registers that are used to implement a semaphore to synchronize ownership changes of the xHC. The hand-off mechanism should be clean and precise and each participant shall adhere to the protocol defined below. Failure to do so will result in two software agents believing they each have exclusive ownership of the xHC and attempt to use the controller concurrently.

The *OS Handoff Synchronization xHCI extended capability* includes two contiguous, 32-bit registers in xHC MMIO space. The first register is the *USB Legacy Support Extended Capability* register (USBLEGSUP), refer to section 7.1.1 for the field definitions. This register is a standard xHCI extended capability pointer, including an xHCI Extended Capability ID field and a link to the next xHCI extended capability.

The upper 16 bits of this register contain ownership semaphores. One semaphore is for the operating system (OS) and one is for the BIOS. These semaphores are readable and writable. These fields are in adjacent bytes, which allows each agent (OS or BIOS) to update their respective semaphore without overwriting the other ownership semaphore.

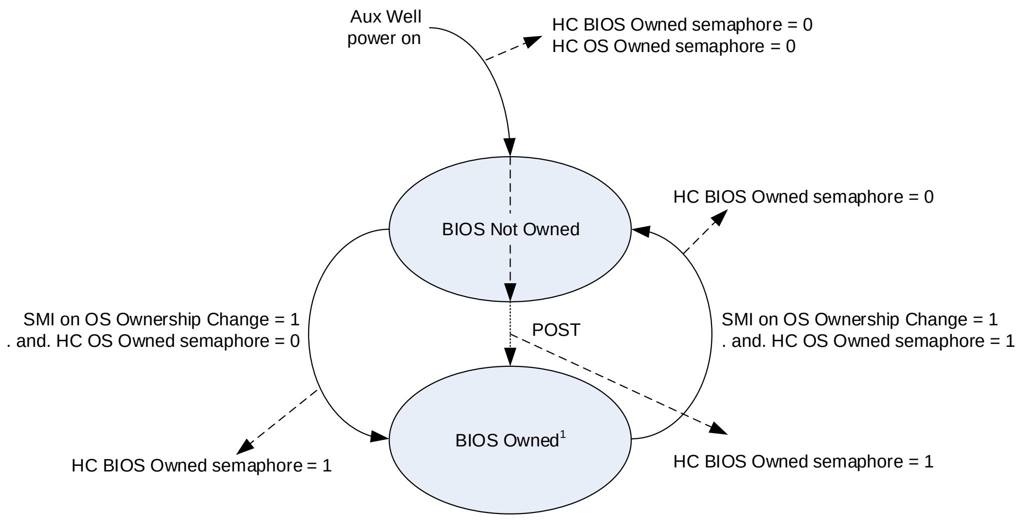
The second 32-bit register is the *USB Legacy Support Control/Status* register (USBLEGCTLSTS), refer to section 7.1.2 for the field definitions. This register defines a set of control bits that BIOS can use to enable SMIs and a set of read-only bits that shadow a subset of the bits from the USBSTS register. The specific USBSTS register bits that are shadowed represent all of the xHC events that can be detected and enabled to generate an interrupt. The USBLEGCTLSTS register provides the mechanism for BIOS to map all xHC events, all necessary reconfiguration events and OS ownership requests to SMIs.

Following are two state machines that illustrate the proper protocol (e.g. updates to the ownership semaphores) that BIOS and OS shall adhere to in order to coherently request and/or relinquish ownership of the xHC. The conventions used in these figures are:

- Solid arcs denote single or multiple events that result in a state change.
- Dotted lines with arrows indicate side effects that take place. When attached to a solid arc, interpretation is that as a result of the event, the side effect occurs.

Figure 4-36 illustrates the protocol state machine for the BIOS ownership. The OS Handoff Synchronization registers are located in the Aux Power well, so any system event that removes power from the Aux Power well will result in these registers being reset to their default values when the Aux Power well is restored.

Figure 4-36: BIOS Ownership State Machine


Aux Well power on
HC BIOS Owned semaphore = 0
HC OS Owned semaphore = 0
HC BIOS Owned semaphore = 0
BIOS Not Owned
SMI on OS Ownership Change = 1
. and. HC OS Owned semaphore = 0
POST
SMI on OS Ownership Change = 1
. and. HC OS Owned semaphore = 1
HC BIOS Owned semaphore = 1
BIOS Owned1
HC BIOS Owned semaphore = 1

Notes:
[1] The BIOS is allowed to claim control of the xHCI as a result of POST (Power On System Test) or as a result of the OS relinquishing control of the xHCI. The BIOS must never attempt to claim the xHCI once it has relinquished control.

When power is applied to the Aux Power well, the *BIOS Owned* and *OS Owned* semaphores in the USBLEGSUP go to their default values (e.g. '0's). BIOS may take ownership of the xHC by setting the *BIOS Owned* semaphore to a '1'. BIOS is only allowed to take ownership of the xHC when the *OS Owned* bit is a '0'. BIOS then may configure the SMI events it needs including the *SMI on OS Ownership Change*. The BIOS now owns the xHC, so it can configure the controller, enumerate the bus and use the devices found as necessary.

Eventually, the operating system will load. If the operating system has support for the xHC, it will need exclusive control over the xHC. The OS driver shall utilize the protocol defined in Figure 4-37 to request ownership of the xHC before it takes ownership and uses the controller. The OS driver initiates an ownership request by setting the *OS Owned* semaphore to a ‘1’. The OS waits for the *BIOS Owned* bit to go to a ‘0’ before attempting to use the xHC. The time that OS shall wait for BIOS to respond to the request for ownership should not exceed ‘1’ second. Note that there is no similar SMI-type of event defined allowing BIOS to request ownership from the OS.

If the BIOS has set *SMI on OS Ownership Enable* in the USBLEGCTLSTS register to a ‘1’, it receives an SMI when the OS Driver sets the *OS Owned* semaphore to a ‘1’ (above). BIOS observes that OS has changed the value of the *OS Owned* bit to a ‘0’, there-by notifying BIOS that it intends to relinquish control of the xHC.

Below are some recommended steps for software implementers to consider just prior to the transition of xHC ownership.

1. Gracefully pause any outstanding bus activity. (e.g. allow completion of in-flight transactions, suspend signaling, reset signaling, etc.)

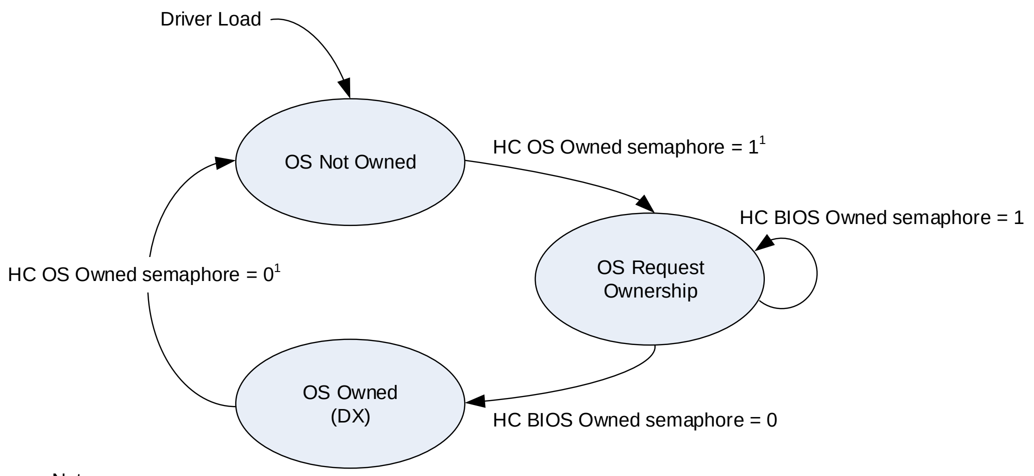
2. Disable all interrupts,

3. Save all critical state from the xHC and relevant USB devices (e.g. Human Interface Device, Mass Storage, etc.)

4. Enable “Wake” events from USB devices (e.g. Human Interface Device, Network, etc.) before suspending platform.

5. Disable all other USB root ports not enabled for wake events in step 4.

Figure 4-37: OS Ownership State Machine


Driver Load
OS Not Owned
HC OS Owned semaphore = 1[1]
HC BIOS Owned semaphore = 1
OS Request Ownership
HC OS Owned semaphore = 0[1]
OS Owned (DX)
HC BIOS Owned semaphore = 0

Notes:
[1] Modifications to the *OS Owned semaphore* results in an SMI when the *SMI on OS Ownership Enable* bit in the USBLEGCTLSTS is set to a one.

In the event that the OS driver unloads and/or wants to relinquish ownership of the xHC, it shall set the *OS Owned* semaphore to a '0'. Again, if BIOS has set *SMI on OS Ownership Enable* in the USBLEGCTLSTS register to a '1', it receives an SMI when the OS Driver sets the *OS Owned* semaphore to a '0'. The BIOS observes that the OS has relinquished control and can then take over control of the xHC as appropriate. Once system software has relinquished control of the controller, it shall then request ownership as described above.

Note that this mechanism is intended only to ensure that an exchange of ownership of the xHC can be accomplished in a very deterministic and reliable manner.

4.22.2 Debug Capability Operational Model

Refer to section 7.6.

4.22.3 Virtualization

Refer to section 8.

4.23 Power Management

This section summarizes the various power management capabilities of the xHCI.

Throughout this specification particular registers and features will be identified as requiring special consideration from a power delivery prospective. Any discussion of power delivery in this specification is with the primary objective of improving interoperability across a wide range of implementations without specifying a specific method of power delivery. The phrase “required to maintain state across power cycles”; or reference to configuration, command and status registers defined expressly for support of the host controller’s power management features will help the reader identify those constructs that require special attention.

The reader should also remain aware that common industry specifications may impose particular power delivery requirements that the design shall conform to for compliance under that industry standard.

Note: The specification and white paper references provided in this section do not represent an exhaustive list and the reader is encouraged to refer to other specifications that may be relevant to the designer’s specific implementation.

4.23.1 Power Wells

This section describes the expected feature of the Core Power and Aux Power (Auxiliary) Wells.

The power well requirements on a system board/add-in card xHC implementation include:

- A common ground plane across the entire system.
- Split voltage (i.e. Aux Power and Core Power) wells are allowed.
- The Aux Power well voltage supply shall be present whenever AC power is applied to the system (if supported).
- Core power may be switched off by the system.

Registers in the Aux Power well are reset under different conditions than the registers in the Core well. The Aux Power well, memory-space registers are initialized to their default values in the following cases:

- Initial power-up of the Aux Power well, or
- a value of ‘1’ in *HCRST* (refer to Section 5.4.1)

Note: The *USB Legacy Support Capability* registers are an exception to the Aux Power well reset rule. Refer to section Pre-OS to OS Handoff Synchronization for more information.

The Core well, memory-space registers are initialized to their default values in the following cases:

- Assertion of Chip Hardware Reset, or
- a value of ‘1’ in *HCRST*, or
- transition from the PCI PM D3hot state to the D0 state

PCI configuration-space registers implemented in the Aux Power well are reset under different conditions than the registers in the Core well. The Aux Power well, configuration-space registers are initialized to their default value in the following case:

- Initial power-up of the Aux Power well.

The Core well PCI configuration-space registers are initialized to their default values in the following cases:

- Assertion of the system (Core-well) hardware reset, or
- transition from the PCI PM D3hot state to the D0 state.

After initial power-on or HCRST (Chip Hardware Reset or via *HCRST* bit in the *USBCMD* register), all of the Operational and Runtime Registers shall be at their default values, as defined in sections 5.4 and 5.5. After a "light" hardware reset (via the *Light Host Controller Reset* (LHCRST) bit in the *USBCMD* register), only the Operational and Runtime Registers not contained in the Aux Power well shall be at their default values. And all registers in the Aux Power well shall maintain the values that had been asserted prior to asserting *Light Host Controller Reset* (LHCRST). Refer to section 5.4.1 for more information.

Exceptions to these reset conditions will be defined in the associated register section.

Note: The method for enabling or disabling the Core Power well voltage supply (e.g. to transition from a D3hot to a D3cold state) is outside the scope of this specification. Typically a platform level power control mechanism is used.

4.23.2 xHCI Power Management

When system software decides to power down the xHC with the intent of resuming operation at a later time, it shall read the xHC registers and save their state. After powering up the xHC, but before placing the xHC into Run mode (*Run/Stop* (R/S) = '1'), system software shall restore all xHC registers.

Additionally, xHC implementations maintain internal state that is not visible to software through its register set. This state shall also be saved and restored for the xHC to correctly recover from a power event, e.g. the internal Ring Cycle State (RCS) flag associated with the ERDP, the set of *Enabled* Device Slots, etc. The xHCI provides two control flags to enable this operation: *Save State* and *Restore State*. These flags reside as bits in the USBCMD register.

The *Save State* and *Restore State* flags may only be set when the xHC is Stopped (*Run/Stop* (R/S) = '0').

Required system software steps for saving xHC state and powering it down are:

1. Stop all USB activity by issuing *Stop Endpoint Commands* for *Busy* endpoints in the *Running* state. If the *Force Save Context Capability* (FSC = '0') is not supported, then *Stop Endpoint Commands* shall be issued for all *Idle* endpoints in the *Running* state as well. The *Stop Endpoint Command* causes the xHC to update the respective Endpoint or Stream Contexts in system memory, e.g. the *TR Dequeue Pointer*, *DCS, etc.* fields. Refer to Implementation Note "0".

Note: *Force Save Context Capability* support (i.e. *FSC* = '1') shall be mandatory for all xHCI 1.1 compliant xHCs.

2. Ensure that the Command Ring is in the Stopped state (*CRR* = '0') or Idle (i.e. the Command Transfer Ring is empty), and all Command Completion Events associated with them have been received.

3. Stop the controller by setting *Run/Stop* (R/S) = '0'.

Note: If FSC = '1', then software shall ensure that any Running endpoint that did not receive a Stop Endpoint Command is Idle when Run/Stop (R/S) is cleared.

4. Read the Operational and Runtime registers in the following order: USBCMD, DNCTRL, DCBAAP, CONFIG, ERSTSZ, ERSTBA, ERDP, IMAN, and IMOD and save their state.

5. Set the *Controller Save State* (CSS) flag in the USBCMD register (5.4.1) and wait for the *Save State Status* (SSS) flag in the USBSTS register (5.4.2) to transition to '0'.

Note: The Save State operation shall save all internal xHC Slot, Endpoint, Stream or other state to the memory locations described in steps 6 and 7 that is necessary for the successful restoration of xHC state, as described below.

6. If *Max Scratch Pad Buffers* is > '0' and *Scratchpad Restore* (SPR) = '1', then save an image of the Scratchpad Buffers.

7. Save a memory image of the DCBAA, Contexts and other data structures referenced by the xHC.

8. Remove Core Well power.

Note: The DCBAA and the complete tree of data structures that it references (Device Contexts, Transfer Rings, Stream Arrays, etc.), as well as the Command and Event Rings, and Scratchpad Buffers shall be preserved by system software.

Required system software steps for powering up and restoring xHC state are:

1. Enable Core Well power.

2. Restore the saved memory image of the DCBAA, Contexts and other data structures to their original physical locations in system memory, so that any addresses saved in steps 4 and 6 above reference valid objects.

3. If an image of the Scratchpad Buffers was saved, restore it.
4. Restore the Operational and Runtime Registers with their previously saved state in the following order: DNCTRL, DCBAAP, CONFIG, ERSTSZ, ERSTBA, ERDP, IMAN, and IMOD.
5. Set the *Controller Restore State* (CRS) flag in the USBCMD register (5.4.1) to ‘1’ and wait for the *Restore State Status* (RSS) flag in the USBSTS register (5.4.2) to transition to ‘0’.
6. Reinitialize the Command Ring, i.e. so its Cycle bits are consistent with the RCS value to be written to the CRCR.
7. Write the CRCR with the address and RCS value of the reinitialized Command Ring. Note that this write will cause the Command Ring to restart at the address specified by the CRCR.
8. Enable the controller by setting *Run/Stop* (R/S) = ‘1’.
9. Software shall walk the USB topology and initialize each of the xHC PORTSC, PORTPMSC, and PORTLI registers, and external hub ports attached to USB devices.
10. Restart each of the previously *Running* endpoints by ringing their doorbells.

Note: It is critical for correct xHC restore operation that all system memory data structures referenced by xHC registers when it is stopped are intact and reside at the same physical addresses when it is restarted. Software shall not modify any Contexts, data structures, or Opaque areas referenced by the xHC when it is stopped if the intent is to use the Restore State operation to restart the xHC.

Note: After a Save or Restore State operation completes, the *Save/Restore Error* (SRE) flag in the USBSTS register should be checked to ensure that the operation completed successfully.

Note: To properly restore the xHC it is critical that the registers are written (step 4) before the Restore operation is performed (step 5). The Restore operation overwrites internal default values asserted by a xHC reset.

Note: Some legacy software implementations may not follow the precise ordering of the steps described above.

The internal state of the xHC shall be valid until it enters the D3cold state. When the xHC is Stopped, software may issue a Save State operation with the expectation of subsequently placing the xHC in the D3cold state. If prior to setting the xHC into the D3cold state, software decides to restart the xHC, then a Restore State operation is not required.

4.23.2.1 Save and Restore Operations

The xHC Save and Restore State operations shall save and restore any internal state necessary to restore the xHC to the same operational state that it is was in when the previous Save was performed, irrespective of whether it uses the Scratchpad Buffer or a proprietary memory to save the state.

For example, the BIOS may save the state of xHC before it hands off the xHC to the OS, then restore that state when control of the xHC is returned to it. However, while the OS has control, it may execute its own Save and Restore State operations every time it transitions in and out of a Suspend or Hibernate states.

The Save and Restore operations may be used to accelerate the initialization process of the xHC. Rather than resetting the xHC and issuing multiple commands to bring Device Slots on line, software could take a “snapshot” of the xHC state after set of Device Slots is configured. The snapshot could then be used to bring the xHC to the same state, without having to run through the initial command sequence. This approach may be useful to quickly bring a set of permanently attached USB devices on a motherboard on line, i.e. the USB topology is fixed.

If the *Scratchpad Restore* (SPR) flag is set in the HCSPARAMS2 register, the xHC Save and Restore State operations use the Scratchpad Buffer space for storing the internal xHC state while it is powered down, and it is critical that the system maintain the integrity of the Scratchpad Buffer space across power events if the xHC is to be restored correctly. Refer to section 5.3.4 for more information.

Note: An xHC implementation is responsible for checking the saved state during a Restore State operation. If the saved state is corrupted, the *Save/Restore Error* (SRE) flag in the USBSTS register shall be set to ‘1’, the Restore operation terminated, and the *Restore State Status* (RSS) flag cleared to ‘0’.

Note: An xHC implementation shall report if the integrity of a Save/Restore State operation sequence has been compromised, i.e. after a Save State operation (*CSS* = '1') is executed, if the xHC is placed into Run mode (*R/S* = '1') before a Restore State operation (*CRS* = '1') is executed, then this condition shall be reported by the assertion of *SRE* upon the completion of the Restore State operation. This condition shall be reported only if the Aux Power well has been maintained between the Save State and Restore State operations. The reporting of this condition shall not be affected by the assertion of *HCRST* between the Save State and Restore State operations.

Note: The state of a Root Hub port is not covered by a Save or Restore operation. Refer to sections 5.4.8, 5.4.9, and 4.19 for more information on how xHC ports are managed during power events.

Note: When xHC state (e.g. Scratchpad Buffers, Contexts, Transfer Rings, etc.) is saved by system software, the data structures must be restored to the same physical addresses that they were at when they were saved, otherwise undefined behavior may occur.

This also implies that any USB devices that were attached when the Save State took place should still be attached and in the same internal state (i.e. their USB Device Address is unchanged) when the Restore State takes place. If these conditions are not true, then software is responsible for doing any "fix-ups" that may be required.

4.23.3 PCI Power Management

Refer to Appendix A.

4.23.3.1 Standard PCI Power Management

Refer to PCI and PCIe specifications.

4.23.3.2 PCI Extended Power Management

Refer to PCI Power Management (PCI PM) specification.

4.23.4 USB Power Management

4.23.4.1 USB2

Refer to USB2 Specification.

4.23.4.2 USB3

Refer to USB3 Specification.

4.23.4.3 USB Power Delivery

Unlike USB Hubs, there are no xHCI port register extensions defined for Root Hubs to support USB Power Delivery (PD). Platform level PD supported is provided through ACPI mechanisms that are outside the scope of this specification. Refer to the USB PD and ACPI Specifications for more information.

4.23.5 USB Link Power Management

The xHCI provides independent mechanisms for managing Link Power Management (LPM). One mechanism allows the xHC Root Hub ports to provide all the features defined by the USB2, USB2 LPM, and USB3 specifications for hub downstream port management. And the other mechanism, enabled through the Slot Context *Max Exit Latency* field, provides the xHC with the information it needs to most effectively schedule USB transfers, maximizing bus bandwidth utilization.

Note: xHC implementations shall support Link Power Management for all USB protocols that it supports. Refer to the *xHCI Supported Protocol Capability* (section 7.2.2.1.3) for the specific Link Power Management features supported by the xHC.

4.23.5.1 Root Hub Port LPM Support

There are two different *Link Power Management* (LPM) approaches defined by the USB specifications, one for USB3 (SuperSpeed) devices and another for USB2 (Legacy High-, Full- and Low-speed) devices. The xHCI defines mechanisms to support the Link Power Management approaches defined for both the USB3 and USB2 protocols.

Refer to the section 11 of the USB3 spec for more information on Link Power Management.

Refer to the USB2 LPM ECN and errata for more information on USB2 Link Power Management.

USB2 defines 3 'L' link states and USB3 defines 4 "U" link states.

Table 4-11: LPM State Mapping

Link State	Encoding		Description
	USB2[67]	USB3	
On	L0	U0	This is the normal link operational state. All packet communication, whether for control or data transfers, occurs in this state. A USB2 port in L0 is either actively transmitting or receiving data (L0-Active) or able to do so but not currently transmitting or receiving information (L0-Idle).
Fine-grain LPM	NA	U1	U1 is a low exit latency standby state. Refer to section 7.2.4.2 of the USB3 spec for more information. In this state the port is capable exiting to the On state in less than ~10 μS.
Coarse-grain LPM	L1[68] (Sleep)	U2	U2 is a low to medium range exit latency standby state. Refer to section 7.2.4.2 of the USB3 spec for more information. In this state the port is capable exiting to the On state in ~1 ms.

[67]This table provides USB3 extensions to Table 1-1 in the USB2 LPM ECN.

[68]The USB2 *L1* state is mapped to the USB3 *U2* state because both represent coarse-grain LPM modes, i.e. they take approximately 1 ms. to enter.

Suspend	L2	U3	This is a deep power saving state where interface (e.g., Physical Layer) power may be removed, except as needed to perform the various functions such as reset signaling, connect/disconnect detection, and wakeup. This is the formalized name for USB Suspend. Entry in to this state is nominally triggered by a command to a hub or root hub port to transition to suspend, at which point the port ceases signaling to the downstream port. This state also imposes power draw requirements (from VBUS) on the attached device. Exit from this state is via remote wake, resume signaling, reset signaling or disconnect. VBUS remains on in this state. Refer to section 11.4.1.4 of the USB3 spec for more information. Refer to Section 7.1.7.6 in the USB2 specification.
Off	L3	Not defined	In this state, the port is not capable of performing any data signaling. It corresponds to the powered-off, disconnected, and disabled states. VBUS is off in this state.

The xHCI reports the current Link State in the *Port Link State* (PLS) field of the PORTSC register. The interpretation of the *PLS* field depends on the PORTSC *Port Speed* field. If *Port Speed* reports Low-, Full-, or High-speed, then the *PLS* field shall never report a U1 state.

4.23.5.1.1 USB2 LPM Support

This section applies only if a USB2 *xHCI Supported Protocol* Capability structure (section 7.2) is declared (i.e. the *Major Revision* field = 02h).

When system software is ready to transition a USB2 port from L0 to a deeper power savings state, it writes a ‘2’ (U2) to the *Port Link State* (PLS) field, which results in setting the *L1 Status* (L1S) field to *Invalid* (‘0’), and an LPM transaction on the USB2 bus. While a USB2 link is attempting to transition to the L1 state, the *PLS* field shall continue to report the previous state (U0).

Note: The device responds to the LPM transaction with an ACK if it is ready to make the transition or a NYET if it is not currently ready to make the transition, usually because it has data pending for the host.

L1 Status (L1S) results for a LPM Transaction:

- Success - Upon receipt of an ACK, the xHC shall set the *PLS* field in the PORTSC register to the L1 state (U2) and the *L1S* field in the USB2 PORTPMSC register to *Success* (‘1’). The *Port Link Status Change* bit is not set and no *Port Status Change Event* is generated.

- Not Yet - Upon receipt of a NYET, the xHC shall set the *L1S* field in the USB2 PORTPMSC register to *Not Yet* (‘2’), set the *Port Link Status Change* (PLC) bit to ‘1’. If the assertion of *PLC* results in a ‘0’ to ‘1’ transition of PSCEG (4.19.2), a *Port Status Change Event* shall be generated for the port.
- Not Supported - A USB2 device shall transmit a STALL handshake if it does not support the requested link state (Lx, refer to the bmAttributes field of the extended LPM transaction, Table 2-3 in the USB2 LPM ECR). Upon the receipt of a STALL handshake the xHC shall set the *L1S* field in the USB2 PORTPMSC register to *Not Supported* (‘3’), disable hardware USB2 LPM (HLE = '0'), and set the *PLC* flag to ‘1’. If the assertion of *PLC* results in a ‘0’ to ‘1’ transition of PSCEG (4.19.2), a *Port Status Change Event* shall be generated for the port.
- Timeout/Error - If the xHC detects a transaction error (including timeout), it shall retry the LPM transaction up to two more times. If there are three consecutive errors then the xHC shall set the *L1S* field in the USB2 PORTPMSC register to *Timeout/Error* (‘4’), disable hardware USB2 LPM (HLE = '0'), and set the *PLC* bit to ‘1’. If the assertion of *PLC* results in a ‘0’ to ‘1’ transition of PSCEG (4.19.2), a *Port Status Change Event* shall be generated for the port.

Note: The *PLC* flag is <u>not</u> set (and an event is not generated) if the port successfully enters the L1 state. In the latter two cases above, the port asserts the *PLC* flag (PLC Condition: USB2 L1 Entry Reject), however the *PLS* field does not change, i.e. the port’s link remains in the U0 state. Software may examine the *L1 Status* (L1S) field of the PORTPMSC register when the *Port Status Change Event* is received for the port to determine the problem with entering the L1 state.

This information allows software to tune its use of the L1 state, identify misbehaving device, etc. For example, software could identify a device which consistently NAKs L1 entry but rarely moves data and notify the end user.

The xHC shall meet the following requirements:

- If the port is enabled (*PED* = ‘1’) and in the L1 (*PLS* = ‘2’) state, the xHC shall treat an L1 request (write of ‘2’ to *PLS* field) as a functional no-operation and set the *L1S* field in the USB2 PORTPMSC register to *Success* (‘1’).
- If the device detects errors in either of the token packets or does not understand the protocol extension transaction, no handshake shall be returned. In this case the xHC shall timeout and the *L1S* field in the USB2 PORTPMSC register shall be set to *Timeout/Error* (‘3’). Refer to the USB2 LPM ECR for L1 timeout details.

The L1 state may be reset to the L0 state by a software request or from the device attached to the port. To accommodate this operation, software may write a ‘0’ (U0) to the *PLS* field of a USB2 protocol port attached to a Low-, Full-, or High-Speed device that supports LPM. Refer to the definition of the *PLS* field in Table 5-26 for more information.

Note: The *Remote Wake Enable* (RWE) flag (Table 5-29) shall be used to enable or disable xHC remote wake from L1.

Note: The *L1 Device Slot* field in the USB2 PORTPMSC register references a Device Slot that is the target for the LPM Token. The *USB Device Address* field in the Slot Context of the referenced Device Slot, specifies the value of the *ADDR* field in the generated LPM Token. The *ENDP* field of the LPM Token shall be set to '0'[69].

4.23.5.1.1.1 Hardware Controlled LPM

USB 2 ports may support hardware controlled Link Power Management, as indicated by the *Hardware LPM Capability* (HLC) flag equal '1' in the USB2 *xHCI Supported Protocol* Capability structure (7.2.2.1.3.2). If Hardware USB2 LPM is supported, then the *Hardware LPM Enable* (HLE) bit in the USB2 PORTPMSC register (5.4.9.2) may be used to enable or disable it.

Each *Port Hardware LPM Control* (PORTHLPMC) register provides an inactivity timeout that shall be configured by software, the *L1 Timeout* field. Refer to section 5.4.11 for more information on the PORTHLPMC register.

Each USB2 Root Hub port maintains a logical **Link Power Management Timer** (LPM Timer) for keeping track of when the inactivity timeout is exceeded. If Hardware LPM is enabled and the LPM Timer reaches the *L1 Timeout* value, the port's link shall initiate a transition to L1.

Two methods of Hardware USB2 LPM are supported by the xHCI; HIRD and BESL. If *HLC* is set to '1' and the *BESL LPM Capability* (BLC) flag in the USB 2.0 Protocol Defined field of the USB2 *xHCI Supported Protocol Capability* structure (7.2.2.1.3.2) is cleared to '0', then the HIRD[70] LPM method is supported. If HLC is set to '1', then the PORTHLPMC register exists. And if *HLC* and *BLC* are both set to '1', then the BESL LPM method is supported.

If HIRD LPM is supported (*HLC* = '1' and *BLC* = '0'), then the Best Effort Service Latency value should be programmed by software in the BESL field of the USB2 PORTPMSC register. Refer to Table 4-12 for the encoding of the BESL field.

If BESL LPM is supported (*HLC* = '1' and *BLC* = '1') then there are two values of *Best Effort Service Latency* that should be programmed by software; the *BESL* field in the USB2 PORTPMSC register and the *BESL Deep* (BESLD) field in the USB2 PORTHLPMC register. The *BESLD* field is programmed with a value that is much larger than the *BESL* value, allowing both the host platform and the device to go into deeper low power states. The *BESL* field shall be programmed to a value smaller than the *BESLD* field for mode 1 (*HIRDM* = '1'), in which both the *BESL* and *BESLD* fields are used. Refer to Table 4-12 for the encoding of the *BESL* and *BESLD* fields.

[69]The value of ENDP is not specified in the USB2 LPM ECR, however there is errata against the ERC that clarifies the use of ENDP = '0'.
[70]Refer to Section 4.1 of the USB2 LPM spec for more information on the use of the HIRD field.

Prior to enabling hardware controlled USB2 LPM, software shall initialize the *BESL* and *RWE* fields of the USB2 PORTPMSC register and if *BLC* = ‘1’ the *BESLD, HIRD Mode* (HIRDM), and *L1 Timeout* fields of the USB2 PORTHLPMC register.

The optimal values for programming the *BESL* and *BESLD* fields depend on the overall latency characteristics of a platform. If an xHC is an integrated component of a platform, then a vendor may specify the preferred default *BESL* and *BESLD* values using the PCIe Config space *Default Best Effort Service Latency* (DBESL) and the *Default Best Effort Service Latency Deep* (DBESLD) fields. If these fields are non-zero, software may use their values to program the respective *BESL* and *BESLD* values in the xHC Port registers. Refer to Table 4-12 for the encoding of the *DBESL* and *DBESLD* fields.

If HIRD LPM is supported (*HLC* = '1' and *BLC* = '0'), the DBESL and DBESLD registers are not implemented.

Note that the hardware management mechanism may use modified *BESL*, *BESLD*, and *RWE* values in the LPM Transactions that it generates to a device.

Note that xHC is required to retain the last *BESL* duration that it used to generate a USB2 LPM transaction to a device, and to drive resume signaling for that time minus 50 µs. when waking the device.

While Hardware USB2 LPM (*HLE* = ‘1’) is enabled, software shall not modify the *BESL* or *RWE* fields of the USB2 PORTPMSC register or the *BESLD, HIRD Mode* (HIRDM), and *L1 Timeout* fields of the USB2 PORTHLPMC register, or attempt to transition the port to the L1 (*PLS* = U2) state, i.e. shall not write the PORTSC register with *PLS* = ‘2’ and *LWS* = ‘1’.

Note: BESL LMP support (i.e. *HLE* = ‘1’ and *BLC* = ‘1’) shall be mandatory for all xHCI 1.1 compliant xHCs.

Note: If Hardware USB2 LPM (*HLE* = '1') is enabled the Slot Context *Max Exit Latency* field shall be initialized to a non-zero value. Refer to section (4.23.5.2) for more information.

Note: **The port behavior described below only applies to devices that are attached to a Root Hub port.**

The port behaves as follows:

- If Hardware LPM is disabled (*HLE* = '0'), then the port's LPM Timer shall be disabled.
- When resume signaling is complete and the link transitions to L0 state, due to an xHC initiated L1 exit, or a Device Initiated L1 Exit, the *PLS* field shall be set to U0.
- The *L1 Timeout* value represents the amount of inactive time in L0 prior to initiating the transition to the L1 state (*PLS* = U2).
- If Hardware LPM is disabled (i.e. *HLE* transitions from '1' to '0') and the port's link is in the L1 (*PLS* = U2) state:

- The port shall automatically initiate a L1 exit.

Note: Software may select different values for *BESL*, *BESLD* and *L1 Timeout* based on device's class, type of endpoints, poll interval (for periodic endpoints), etc.

Note: If Hardware LPM is enabled (*HLE* = '1'), the Hardware LPM state machine automatically transitions a port in the **Enabled** state between the **U0**, **U2Entry**, **U2**, and **U2Exit** substates. Refer to the USB2 *Root Hub Port Enabled Substate Diagram* (4.19.1.1.6). The notable differences are:

- The **U0** to **U2Entry** transition is initiated by a L1 Timeout.
- For U2Enty to U0 transitions, the *PLC* flag shall not be set ('1') if a NYET response occurs. The *PLC* flag shall be set ('1') for STALL or timeout/error response
- The **U2** to **U2Exit** transition is initiated by an xHC initiated L1 exit, or a Device Initiated L1 Exit.
- The *PLC* flag shall not be set ('1') by a **U2Exit** to **U0** transition.

Note: A USB2 link transitions through L-states, and these states are reflected in the associated USB PORTSC register as U-states in the *PLS* field. Refer to Table 4-11 for the mapping of USB2 L-states to *PLS* field U-states.

The following Cases apply only if BESL LPM is supported.

Case 1: For Devices supporting Device Initiated L1 Exit

Note: For devices that support Device Initiated L1 Exit, when HW LPM is enabled (*HLE* = '1'), software should set the *RWE* bit of USB2 PORTPMSC register to '1'.

- When Hardware LPM is enabled (i.e. *HLE* transitions from '0' to '1'):
  - The port's LPM Timer shall be reset to '0' and start counting up.
- If Hardware LPM is enabled (*HLE* = '1'), then:
  - The port's LPM Timer shall be reset to '0' and start counting up, every time a data packet is sent or received by the port's link.
  - When resume signaling completes, the LPM Timer shall be reset to '0' and start counting up.
  - If *HIRD Mode* (HIRDM) = '0':
    - When the LPM Timer equals the *L1 Timeout* value:
      - The *PLS* field shall be set to *U2*.
      - The port's link shall initiate a transition to L1 by issuing an LPM Token to the device, where the HIRD[71] field of the LPM Token shall be set to the value of the PORTPMSC *BESL* field.

[71]In the USB2 LPM ECN the parameter $T_{L1HubDrvResume}$ is represented by the HIRD field in the LPM Token.

        - If the LPM Token (using the *BESL* value) is accepted by the device:
            - The LPM Timer shall be stopped.
            - The link then waits for an xHC initiated L1 exit, or a remote wake to be initiated by the device.
        - If the LPM Token is rejected by the device (NYET response):
            - The *PLS* field shall be set to *U0*.
            - The LPM Timer shall be reset to '0' and start counting up.
- *HIRD Mode* (HIRDM) = '1':
    - When the LPM Timer equals the *L1 Timeout* value:
        - The *PLS* field shall be set to *U2*.
        - The port's link shall initiate a transition to L1 by issuing an LPM Token to the device, where the HIRD field of the LPM Token shall be set to the value of the PORTHLPMC *BESLD* field.
        - If the LPM Token (using the *BESLD* value) is accepted by the device:
            - The LPM Timer shall be stopped.
            - The port then waits for an xHC initiated L1 exit, or a remote wake to be initiated by the device.
        - If the LPM Token (using *BESLD*) is rejected by the device (NYET response):
            - The port's link shall initiate a transition to L1 by issuing an LPM Token to the device, where the HIRD field of the LPM Token shall be set to the value of the PORTPMSC *BESL* field.
            - If LPM token (using *BESL*) is accepted by device:
                - The LPM Timer shall be stopped.
                - The link then waits for an xHC initiated L1 exit, or a remote wake to be initiated by the device.
            - If the LPM Token (using *BESL*) is rejected by the device (NYET response):
                - The *PLS* field shall be set to U0.
                - The LPM Timer shall be reset to '0' and start counting up.
- *HIRD Mode* (HIRDM) = '2' or '3':
    - Reserved. Undefined behavior may occur if *HIRDM* is set to a reserved value.

Note: For Bulk OUT, Interrupt OUT, an Isoch IN, an Isoch OUT EP, or a Control EP the xHC shall initiate a L1 exit if it needs to move data, i.e. a doorbell for a non-Isoch endpoint has been rung, or if an Isoch Interval has expired and an Isoch TD is available.

Note: For Interrupt IN endpoints that are actively moving data, the xHC shall initiate L1 exit when the poll interval has expired and a TD is available. For Interrupt IN endpoints which have TD available but have responded with a NAK to a poll, the xHC will not poll the device till it has data to move and has initiated an L1 exit (Remote wake).

Note: In the case of a High-speed Bulk OUT Endpoint that has returned a NYET handshake for an OUT transaction and then the Hardware LPM mechanism transitions the link to the L1 state, the xHC shall not initiate a L1 Exit (i.e. wake up the link) to do a PING transaction. The device is expected to initiate an L1 exit (Remote Wake) when it is ready to accept data.

For instance, mass storage devices are command driven, i.e. any bus activity they generate is only due to commands issued by the host. Since asynchronous notifications are not associated with these devices they typically do not support a Remote Wakeup capability. However, a rotational mass storage device's link may be inactive for 10's of milliseconds while it performs a seek operation, allowing the link to enter L1. When data is available, the disk must use the Remote Wakeup capability to return the link to the L0 state so that it can move the data and complete the command.

Case 2: For Devices not supporting Device Initiated L1 Exit

Note: For devices that do not support Remote Wakeup, if HW LPM is enabled (HLE = '1'), software should not set the RWE bit of USB2 PORTPMSC register to '1'

- When Hardware LPM is enabled (HLE transitions from '0' to '1'):
  - The port's LPM Timer shall be reset to '0' and start counting up.
- If Hardware LPM is enabled (HLE = '1'), then:
  - The port's LPM Timer shall be reset to '0' and start counting up, every time a data transfer is attempted (IN or OUT tokens).
- If HIRD Mode (HIRDM) = '0':
  - For periodic endpoints (both isochronous and interrupt), the xHC will put the link in L1 when the LPM Timer equals L1 Timeout value. The xHC will initiate an L1 exit prior to the next poll if there are pending TDs.
  - For Bulk endpoints, the LPM Timer shall be reset to '0' and start counting up, every time a data transfer is attempted (IN or OUT transaction).

Note: For all devices that do not support remote wake, the L1 Timeout value should be large enough so that L1 entry is not triggered by delays in PING retries, delays in generating IN or OUT tokens due to bandwidth sharing with high bandwidth isochronous devices, etc.

- HIRD Mode (HIRDM) = '1':
    - Software shall not set HIRD Mode to '1' when Remote Wakeup is not supported by the device
- HIRD Mode (HIRDM) = '2' or '3':
    - Reserved. Undefined behavior may occur if HIRDM is set to a reserved value.

If the *BESL LPM Capability* (Table 7-15) is supported by the xHC (*BLC* = '1'), then the xHC shall support the *BESL Duration* (as shown in the "BESL Duration BLC = 1" column of Table 4-12) and resume signaling shall be asserted by the Root Hub port for the *HIRD Duration* (as shown in the "HIRD Duration BLC = 1" column of Table 4-12).

If the *BESL LPM Capability* is not supported (*HLC* = '1' and *BLC* = '0'), i.e. the xHC implementation predates the USB2 LPM Errata, then the resume signaling shall be asserted for the *HIRD Duration* (as shown in the "HIRD Duration BLC = 0" column of Table 4-12).

Table 4-12: BESL/HIRD Encoding

BESL or BESLD Value	BESL Duration (µs) BLC = 1	HIRD Duration (µs) BLC = 1	HIRD Duration (µs) BLC = 0
0	125	75	50
1	150	100	125
2	200	150	200
3	300	250	275
4	400	350	350[72]
5	500	450	425

[72]Note: A device may NAK an LPM Token because the resume duration identified by the received LPM Token's HIRD/BESL field exceeds its resume latency requirements. Software can determine if a device supports the BESL or (legacy) HIRD interpretation of the LPM Token by inspecting the bmAttributes field of a device's DEVICE CAPABILITY:USB 2.0 EXTENSION descriptor. If BL*C* = '1' and the attached device supports HIRD (i.e. the device predates the USB2 LPM Errata), then xHC BE*SL or* BE*SLD field* values less than or equal to '4' result in an xHC resume duration that is less than or equal to the resume duration expected by the device, while values greater than '4' will exceed the device's expectations.If BLC = '0' and the attached device supports BESL, then xHC BESL *or* BESL*D fi*eld values greater or equal to '4' result in an xHC resume duration that is less than or equal to the resume duration expected by the device, while values less than '4' will exceed the device's expectations.Software should choose xHC BESL/BESLD field values that do not violate a device's resume latency requirements, e.g. not program values > '4' if BLC = '1' and a HIRD device is attached, or not program values < '4' if BLC = '0' and a BESL device is attached.

6	1000	950	500
7	2000	1950	575
8	3000	2950	650
9	4000	3950	725
10	5000	4950	800
11	6000	5950	875
12	7000	6950	925
13	8000	7950	1000
14	9000	8950	1075
15	10000	9950	1150

Note: If Hardware USB2 LPM (*HLE* = '1') is enabled, *PLC* shall not be affected by LPM state transitions, i.e. the *L1 Resume complete* (U2 -> U0) or *L1 Entry Reject* (U0 -> U0) conditions shall not assert *PLC*.

4.23.5.2 Max Exit Latency

The xHC schedules all USB data transfers. If links in the path to a USB device are in U1 or U2 state, an additional latency is incurred when accessing a device. It is not practical for the xHC to track the state of every link in the USB topology, so the *Max Exit Latency* field in the *Slot Context* identifies the worst case exit latency for the links and hubs between the xHC and the device when scheduling transfers to power managed devices.

Max Exit Latency is a software computed value, which should comprehend the following components:

1. The worst case delay to wake up all links in the path between the Root Hub port and the device if they are in their deepest allowable U state, i.e. U1 or U2.
   For SuperSpeed devices, the **Maximum Exit Latency** (MEL) described in section C.1.5.2 of the USB3 spec may be used to compute this component.

2. The minimum *Interval* value set for any Isoch endpoint of the device.

3. The worst case time it takes to transfer the Isoch data.
   Note that the value of this component may not be determined by the

largest *Max ESIT Payload* declared by a device endpoint. E.g an endpoint with a *Max ESIT Payload* of 48KB and an *Interval* of 2 microframes allows a larger *Max Exit Latency* value than an endpoint with a *Max ESIT Payload* of 24KB and an Interval of 1 microframe.
For SuperSpeed Isoch Transaction Limits refer to Appendix F.3.

A *Max Exit Latency* value of '0' indicates to the xHC that no links in the path to the device are being power managed.

For USB2 devices, if the attached device supports *Link Power Management* (as described in the USB2 LPM ECN) and LPM is enabled in the device, then the *Max Exit Latency* should be set to the value of BESL. From the BESL value, the actual maximum exit latency value ($T_{L1ExitLatency1}$) may be calculated by the xHC using the following formula. Otherwise, the *Max Exit Latency* shall be set to '0'.

$$T_{L1ExitLatency1} = BESL + T_{L1ExitDevRecovery}\ (10us.)$$

For USB3 devices, refer to section C.1.5.2 of the USB3 spec for the method of computing the value of *Max Exit Latency*.

Note: If *Max Exit Latency* = '0' and the Slot Context *Speed* field equals SuperSpeed, then the xHC may not schedule any PING TPs for endpoints associated with the Device Slot.

Note: System software sets the allowable U-states for the links in the path to a device. Software knows, based on the depth and the exit latencies of the intervening links, what the worst case time is for a PING TP to reach a device and the PING_RESPONSE TP to be returned. Software shall ensure that a device is prevented from entering a U-state where its worst case exit latency (i.e. the delay between the transmission of a PING TP and the reception of the PING_RESPONSE TP by the xHC) approaches the ESIT.

If software is going to change device or link related parameters on the bus that would result in a shorter *Max Exit Latency* value for a Device Slot, then it should change the *Max Exit Latency* value in the device's Slot Context using an *Evaluate Context Command*, before it changes any bus parameters.

If software is going to change device or link related parameters on the bus that would result in a longer *Max Exit Latency* value for a Device Slot, then it should change any bus parameters, before it changes the *Max Exit Latency* value in the device's Slot Context using an *Evaluate Context Command*.

Note: The xHC shall complete any changes to its internal Pipe Schedules before it generates a *Command Completion Event* for *Evaluate Context Command* that modifies *Max Exit Latency*.

4.23.5.2.1 No Ping Response Error

This error only applies to SuperSpeed Isoch endpoints. A *No Ping Response Error* Completion Code indicates that the xHC was unable to complete the data transfer associated with an Isoch TD within the ESIT because it did not receive a PING_RESPONSE in time.

The xHC schedules the data transfer for a SS Isoch endpoint, and if the Slot Context *Max Exit Latency* value is non-zero, it shall send a PING TP to the endpoint *Max Exit Latency* µs. before the scheduled data transfer to wake up all the links in the path. If a PING_RESPONSE TP is not received by the time the data transfer is scheduled to take place, a *No Ping Response Error* should be generated for the TD.

If the error occurs, the data associated with the TD in error shall be lost and the xHC shall advance to the next TD for the next ESIT.

In response to a *No Ping Response Error* Completion Code software should reevaluate the value assigned to *Max Exit Latency*.

Note: A *No Ping Response Error* shall utilize the Transfer Event TRB format. The *TRB Pointer* field of *No Ping Response Error* Transfer Event may be '0'. If the *TRB Pointer* = '0', then the *TRB Transfer Length* field shall be invalid.

Refer to section 6.2.2 for the definition of the Slot Context the *Max Exit Latency* field.

Refer to section C.2 in the USB3 spec for U1 and U2 Exit Latency calculation examples.

4.23.5.2.2 Max Exit Latency Too Large Error

The *Max Exit Latency Too Large Error* may be generated by an *Evaluate Context Command* or optionally by a *Configure Endpoint Command*, and informs software that the specified *Max Exit Latency* value would not allow the xHC to reliably schedule Isoch transfers for the Device Slot.

When software receives this error it knows that it can change some of the link power state options in the path to the device to less aggressive settings (which allows it to assert a smaller *Max Exit Latency* value) and retry the configuration with the same *Interval* and *Max ESIT Payload* size.

The *CMC* flag in the HCCPARAMS2 register indicates whether a *Configure Endpoint Command* is capable of generating a *Max Exit Latency Too Large Error* and the *CME* flag exists. If the *CME* flag in the CONFIG register is set to '1', then the *Command Completion Event* generated by a *Configure Endpoint Command* is allowed to assert a *Completion Code* of *Max Exit Latency Too Large Error*.

Note: In addition to waiting for the PING_RESPONSE and transferring the Isoch data, the xHC must include the *Isoch Scheduling Delay*. The *Isoch Scheduling Delay* comprehends the additional time the xHC requires to parse the PING_RESPONSE TP then enable the associated Isoch transfer, and to accommodate schedule jitter the PING and the Isoch transfer may incur within the Interval due to the other transfers that it must manage. The *Isoch Scheduling Delay* is an xHC implementation specific value. The *Max Exit Latency Too Large Error* allows the xHC to reject a proposed *Max Exit Latency* value because it could not be made to work after it evaluated the *Isoch Scheduling Delay* by the other endpoints that it had to schedule.

4.24 Host Controller Management

4.24.1 Internal Errors

The *Host Controller Error* (HCE) flag is asserted when an internal xHC error is detected that exclusively affects the xHC. When the *HCE* flag is set to '1' the xHC shall cease all activity. Software response to the assertion of *HCE* is to reset the xHC (*HCRST* = '1') and reinitialize it.

Software should implement an algorithm for checking the *HCE* flag if the xHC is not responding.

Note: *HCE* may be asserted due to a soft or hard error. An SRAM parity error while accessing an internal data structure is an example of a soft error that may assert *HCE*. However a hard error shall cause the xHC to reassert *HCE* immediately after it is reinitialized. In this case, software should employ some heuristics to prevent the case where the xHC is continually in an error-reset-reinitialize loop and report this condition to the user.

Note: *Host System Error* (HSE) shall be used to report errors detected by xHC that may affect the system as a whole. Refer to section 4.10.2.6 for more information.

4.24.2 Port to Connector Mapping

This section discusses how the xHC Root Hub registers ports shall be mapped to the **External Ports** of the xHC device, and the USB A connectors of a system, where a "system" may be a motherboard or a stand alone controller card. Consistent mapping is required to ensure that software may effectively manage the USB devices attached by the user.

4.24.2.1 Root Hub Port to External Port Assignment

This section discusses how the Root Hub registers ports shall be mapped to the **External Ports** of the xHC device.

An xHC may integrate one or more Tier[73] 2 USB 2.0 hubs. These hubs shall be referred to as **Integrated Hubs**. An *Integrated Hub* may be connected to a Root Hub port associated with a High-speed Bus Instance to provide Low-speed (LS),

[73]Refer to section 4.1.1 of the USB2 spec for more information on Tiers and USB Topologies.

Full-speed (FS), and High-speed (HS) functionality on *External Ports* presented by the xHC device or to expand the number of USB2 Protocol *External Ports*.

A USB3 hub is the logical combination of two hubs: a USB 2.0 hub and an Enhanced SuperSpeed hub. Each hub operates independently on a separate data bus. Typically, the only shared logic between the two hubs is for controlling VBus on their downstream facing ports. The paring of USB 2.0 and Enhanced SS hubs means that downstream facing ports of the USB3 hub are at the same Tier. Matched Tiers simplify the software management of the shared port power logic in a USB3 hub.

When the xHC *External Ports* associated with an *Integrated Hub* and the *External Ports* associated with a USB3 Protocol Root Hub port are assigned to the same USB connector, a mismatch is created between the Tiers presented at the connector. The USB 2.0 signal pair from the *External Hub* is at Tier 2 and the SuperSpeed signal pairs from the Root Hub port are at Tier 1. To minimize the impact on software management of power at the connector, the Tier mismatch created by *Integrated Hubs* is limited to 1.

- When *Integrated Hub*(s) are implemented:
  - Only a single *Integrated Hub* (i.e. one additional Hub Tier) shall be allowed between a xHC Root Hub port and *External Port*.
  - The only allowed USB 2.0/Enhanced SS Hub Tier mismatch case is where the USB2 Protocol *External Ports* are at Tier 2 and USB3 Protocol *External Ports* are at Tier 1.
  - The xHC vendor shall provide a description of the Root Hub port / *Integrated Hub* / *External Port* mapping. Refer to Appendix D for an example of how ACPI may be used to provide this mapping.
  - Ports of like protocols shall be grouped when defining External Port numbering. e.g. Given *n* USB2 protocol *External Ports* and *m* USB3 protocol *External Ports*, *External Ports* 1 through *n* shall be USB2 protocol ports and *External Ports* *n*+1 through *n*+*m* shall be USB3 protocol ports.
  - The USB2 xHCI Supported Protocol Capability *Integrated Hub Implemented* (IHI) flag shall be '1'.
- When *Integrated Hub*(s) are not implemented:
  - There shall be a 1:1 mapping between xHC Root Hub ports and xHC *External Ports*, where the Root Hub port 1 shall map to External Port 1, Root Hub port 2 shall map to External Port 2, and so on. This mapping means that the protocol of each Root Hub port is identical to the protocol of the respective *External Port*, as defined by the USB2 and USB3 xHCI Supported Capabilities, refer to section 7.2.
  - The USB2 xHCI Supported Protocol Capability *Integrated Hub Implemented* (IHI) flag shall be '0'.

4.24.2.2 External Port to USB Connector mapping

- This section discusses how the **External Ports** of the xHC device may be mapped to the physical **USB A connectors** of the xHC system. Consistent mapping is required to ensure that software may effectively manage the ports.

A system may incorporate USB2 or USB3 hubs that are external from the device that contains the xHC. In this section these hubs will be referred to as **Embedded Hubs**. *Embedded Hubs* may be used to expand the number of USB2, or USG3 A or C connectors presented by a system.

- When an *Embedded Hub*(s) is implemented:
  - A USB 2.0/Enhanced SS Hub Tier mismatch between the xHC *External Ports* and the USB A connectors is not allowed.
  - The system shall provide software with a description of the Root Hub port / Integrated Hub / External port / Embedded Hub / USB A or C connector mapping via ACPI or other method. Refer to Appendix D for an example of ACPI mapping.
- When an *Embedded Hub* is not implemented:
  - A system may define the mapping of xHC *External Ports* to USB connectors using ACPI or other methods.
  - Software may assume the following "default" mapping of xHC *External Port* numbers to USB connector numbers if no other method is defined by a system.

    Given *n* USB2 protocol External Ports numbered 1 to *n*, *m* USB3 protocol External Ports numbered *n*+1 to *n*+*m*, and *c* USB connectors numbered 1 to *c*; *External Ports* 1 and *n*+1 shall map to USB connector 1 to form a USB3 compatible port, *External Ports* 2 and *n*+2 shall map to USB connector 2 to form a USB3 compatible port, and so on. If there *n* is greater than *m* then there will be *m* USB3 compatible ports and *n*-*m* USB2 compatible ports, or vice versa if *m* is greater than *n.*

Note: If USB2 and USB3 protocol ports share the same over-current detection logic (whether Integrated or Embedded hub(s) are implemented or not), then an over-current condition shall assert *OCA* on both ports and transition both ports to the **Powered-off** state.

4.24.2.3 Mapping Example

Figure 4-38: Integrated Hub Example

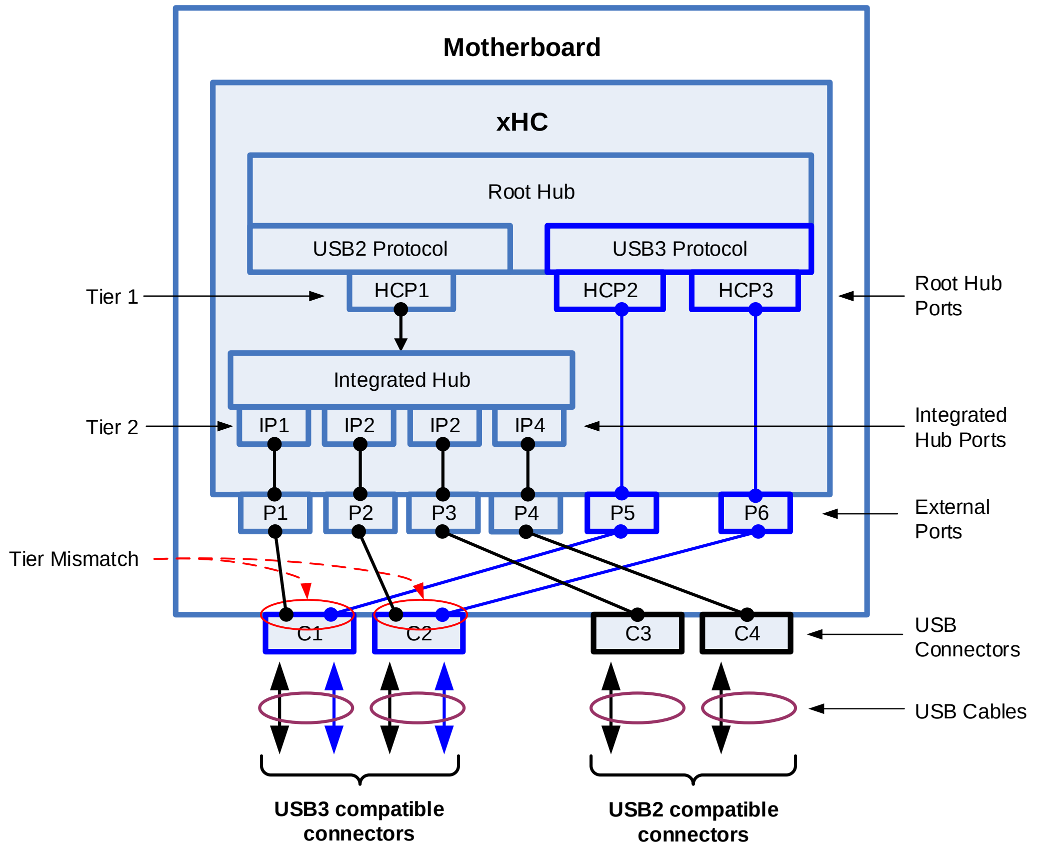


Figure 4-38 illustrates a *Integrated Hub* xHC example implementation, where:

- The motherboard presents 4 user visible connectors C1 – C4.
  - Motherboard connectors C1 and C2 support USB3 (LS/FS/HS/SS) devices.
  - Motherboard connectors C3 and C4 support USB2 (LS/FS/HS) devices.
- The xHC implements a High-speed Bus Instance associated with one USB2 Protocol Root Hub port HCP1. Note that HPC1 provides no Low- or Full-speed support.
- The xHC implements 3 Root Hub ports (HCP1 – HCP3, Tier 1), 1 USB2 Protocol and 2 USB3 Protocol.
  - Root Hub port 1 (HCP1) is attached to the HS *Integrated Hub*. The *Integrated Hub* supports 4 ports (IP1 – IP4).
    - Ports 1 to 4 (IP1-IP4, Tier 2) of the *Integrated Hub* attach to *External Ports* 1 to 4 (P1-P4), respectively.
  - Root Hub ports 2 and 3 (HCP2, HCP3) attach to *External Ports* 5 and 6 (P5, P6), respectively.

- The xHC presents 6 *External Ports* (P1 – P6).
    - *External Ports* 1 – 4 (P1 – P4) support LS/FS/HS devices.
        - P1 and P2 are attached to motherboard connectors C1 and C2, respectively, providing the LS/FS/HS support for the USB3 connectors.
        - P3 and P4 are attached to the motherboard USB2 compatible connectors C3 and C4, respectively.
    - External Ports 5 and 6 (P5, P6) are attached to motherboard connectors C1 and C2 respectively, providing the SS support for the USB3 connectors.
    - External Ports P1 through P4 present a USB2 data bus (i.e. a D+/D- signal pair). External Ports P5 and P6 present a SuperSpeed data bus (i.e. SSRx+/SSRx- and SSTx+/SSTx- signal pairs).
- The *Tier Mismatch* occurs at connectors C1 and C2 due to assigning Tier 2 *Integrated Hub* ports and Tier 1 Root Hub ports to the same USB3 connectors.

5 Register Interface

The extensible USB Host Controller contains many software accessible hardware registers. A large portion of the registers appear as Memory-mapped Host Controller Registers. Other registers may appear using non-memory address mechanisms, as in the case of a PCI or PCIe based Host Controller. For these designs it is required to implement the required registers as defined by the respective specification.

Note that the xHCI does not require support for exclusive-access mechanisms (such as PCI LOCK) for accesses to the memory-mapped register space. Therefore, if software attempts exclusive-access mechanisms to the host controller memory-mapped register space, the results are undefined.

Refer to section 3.1 for a summary of the xHCI register architecture.

Table 5-1: eXtensible Host Controller Interface Register Sets

Offset	Register Set	Size	Explanation
0 to CAPLENGTH	Capability Registers (Section 5.3)	Up to 256 Bytes	The capability registers specify the limits, restrictions, and capabilities of a host controller implementation. These values are used as parameters to the host controller driver.
CAPLENGTH to CAPLENGTH + BFFh	Operational Registers (Section 5.4)	Up to 3K Bytes	The “low-touch” operational registers are used by system software to control and monitor the operational state of the host controller.
Pointed to by the Capability Registers	Run-time Registers (Section 5.5)	Up to 32800 Bytes	The “high-touch” operational registers are used by system software to control and monitor the operational state of the host controller.
Pointed to by the Capability Registers	Doorbell Array (Section 5.6)	Up to 1K Bytes	An array of doorbells, where each 32-bit entry in the array represents a doorbell for each device attached to the host. Write the ID(s) for a specific endpoint to signal the host controller that additional work items are available.

Refer to Table 7-2 for a breakdown of the xHCI Extended Capability register sets.

Table 5-2: Register Alignment Requirement Summary

Register	Alignment in Bytes	Section
Capability Registers	Page	5.3
Operational Registers	4	5.4

Runtime Registers	PF0 = 32
	VFn = Page
		5.5

Doorbell Array	PF0 = 4
	VFn = Page
		5.6

5.1 Register Conventions

If the xHC supports 64-bit addressing (AC64 = '1'), then software should write 64-bit registers using only Qword accesses. If a system is incapable of issuing Qword accesses, then writes to the 64-bit address fields shall be performed using 2 Dword accesses; low Dword-first, high-Dword second.

If the xHC supports 32-bit addressing (AC64 = '0'), then the high Dword of registers containing 64-bit address fields are unused and software should write addresses using only Dword accesses.

Note: The *USB Legacy Support* (USBLEGSUP) Extended Capability requires support for Byte accesses for Semaphore address, refer to section 7.1.

All multi-byte register fields follow little-endian ordering; i.e. lower addresses contain the least significant parts of the field. Bytes/characters within a field shall be in little-endian order, i.e. first char of string in least significant byte, second char next significant byte, etc.

5.1.1 Attributes

The following notation is used to describe register access attributes:

Table 5-3: Register Attributes

Register Attribute	Description
HwInit	**Hardware Initialized**: Register bits are initialized by firmware or hardware mechanisms such as pin strapping or serial EEPROM. (System firmware hardware initialization is only allowed for system integrated devices.) Bits are read-only after initialization and may only be reset (for write-once by firmware) with a HCRST.
RO	**Read-only**: Register bits are read-only and may not be altered by software. Register bits may be initialized by hardware mechanisms such as pin strapping or serial EEPROM.
RW	**Read-Write**: Register bits are read-write and may be either set or cleared by software to the desired state. Note that individual bits in some read/write registers may be Read-Only.
RW1C	**Write-1-to-clear status**: Register bits indicate status when read, a set bit indicating a status event may be cleared by writing a '1'. Writing a '0' to RW1C bits has no effect.
RW1S	**Write-1-to-set status**: Register bits indicate status when read, a clear bit may be set by writing a '1'. Writing a '0' to RW1S bits has no effect.
ROS	**Sticky - Read-only**: Register bits are read-only and may not be altered by software. Where noted, registers that consume AUX Power shall preserve sticky register values when AUX Power consumption (either via AUX Power or PME Enable) is enabled. In these cases, registers are not initialized or modified by Chip Hardware Reset.
RWS	**Sticky - Read-Write**: Register bits are read-write and may be either set or cleared by software to the desired state. Where noted, registers that consume AUX Power shall preserve sticky register values when AUX Power consumption (either via AUX Power or PME Enable) is enabled. In these cases, registers are not initialized or modified by Chip Hardware Reset.
RW1CS	**Sticky - Write-1-to-clear status**: Register bits indicate status when read, a set bit indicating a status event may be cleared by writing a '1'. Writing a '0' to RW1CS bits has no effect. Where noted, registers that consume AUX Power shall preserve sticky register values when AUX Power consumption (either via AUX Power or PME Enable) is enabled. In these cases, registers are not initialized or modified by Chip Hardware Reset.
Rsvd	**Reserved**: Reserved for future RO implementations. Registers or memory that shall be treated as read-only by system software. Rsvd registers shall return '0' when read. Software shall ignore the value read from these bits.
RsvdO	**Reserved and Opaque**: Reserved for exclusive xHC use, e.g. temporary xHC workspace. Register or memory values may be modified by the xHC at any time. Software manipulation of this space may cause undetermined results. Software shall *not* write this space unless explicitly allowed by vendor specific instruction.

RsvdP	**Reserved and Preserved**: Reserved for future RW implementations. Software shall preserve the value read for writes to bits.
RsvdZ	**Reserved and Zero**: Reserved for future RW1C implementations. Software shall use '0' for writes to these bits.

Note: System software shall mask all reserved fields (Rsvd, RsvdP or RsvdZ) to '0' before evaluating a register or data structure value. This will enable current system software to run with future xHCI implementations that define the reserved fields.

Note: When a Reserved attribute (Rsvd, RsvdP, RsvdO or RsvdZ) is used to define a data structure field, system software shall set all reserved register fields to '0' when initially allocating the data structure.

Note: Registers that define "Sticky" bits shall preserve their values when the Aux Power well is enabled and the xHC is in the D3cold state. Refer to section 4.23.1 for more information on power wells and register initialization.

5.1.2 Power Well Considerations

Refer to section 4.23.1.

5.2 PCI Configuration Registers (USB)

xHCs designed for operation in PCI-based systems shall implement a PCI Configuration Space that conforms to either the PCI Specification or the PCIe Specification, as determined by the target operating environment. The implementer should refer to the appropriate specification as published by the PCI Special Interest Group (SIG) (http://www.pcisig.com)

5.2.1 Type 0 PCI Header

Figure 5-1 describes the PCI Configuration Space for an xHC. PCI-based xHCs are required to implement a PCI, Type 0 PCI device header as depicted below. xHCs are also required to implement at least the first two Base Address Registers (BAR 0 and BAR 1) to enable 64-bit addressing. These Base Address Registers are used to point to the start of the host controller's memory-mapped Input/Output (MMIO) register space.

Refer to section 6.1 of the PCI specification for detailed compliance information.

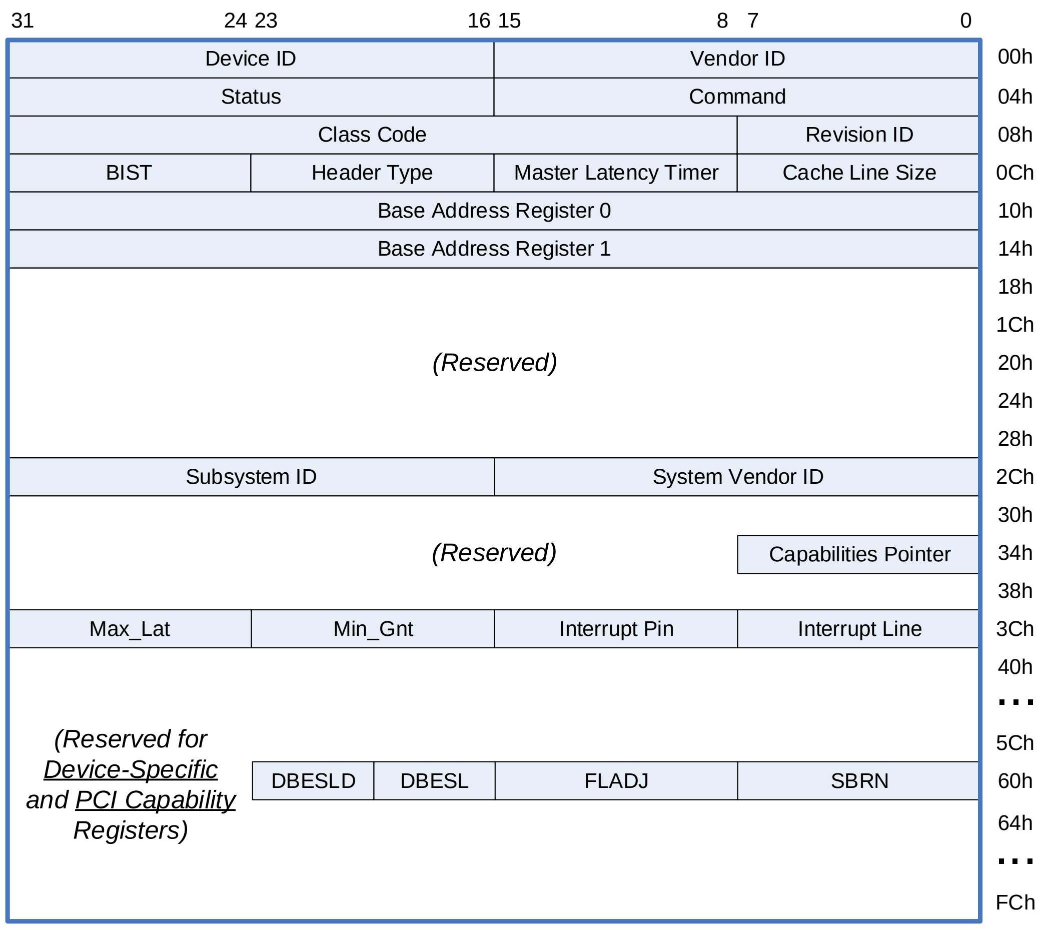
IMPLEMENTATION NOTE

BAR0 Size Allocation

If virtualization is supported, the Capability and Operational Register sets, and the Extended Capabilities may reside in a single page of virtual memory, however the RTSOFF and DBOFF Registers shall position the Runtime and Doorbell Registers to reside on their own respective virtual memory pages. The BAR0 size shall provide space that is sufficient to cover the offset between the respective register spaces (Capability, Operational, Runtime, etc.) and the register spaces themselves (e.g. a minimum of 3 virtual memory pages).

If virtualization is not supported, all xHCI register spaces may reside on a single page pointed to by the BAR0.

Figure 5-1: PCI Type 00h Configuration Space Header



Many of the fields of the PCI header space contain hardware default values, which are either fixed or, if an implementation permits, may be overridden using EEPROM, but may not be independently specified for each logical xHC instance in a platform. These fields include: Revision, Header Type, Subsystem ID, Subsystem Vendor ID, Class Code, Capability Pointer, Max Latency, and Min Grant.

The following fields are unique to each xHC instance: Device ID, Command, Status, Latency Timer, Cache Line Size[74], Memory BAR, and Interrupt Pin.

5.2.2 Class Code Register

Address Offset:	09-0Bh
Default Value:	0C0330h
Attribute:	RO
Size:	24 bits

This register contains the device programming interface information related to the Sub-Class Code and Base Class Code definition. This register also identifies the Base Class Code and the function sub-class in relation to the Base Class Code.

Table 5-4: Class Code Register (CLASSC)

Bit	Description
7:0	**Programming Interface (PI) - RO.** 30h = USB3 Host Controller that conforms to this specification.
15:8	**Sub-Class Code (SCC) - RO.** 03h = Universal Serial Bus Host Controller.
23:16	**Base Class Code (BASEC) - RO.** 0Ch = Serial Bus controller.

5.2.3 Serial Bus Release Number Register (SBRN)

Address Offset:	60h
Default Value:	Refer to Description below
Attribute:	RO
Size:	8 bits

This register contains the release of the Universal Serial Bus Specification with which this Universal Serial Bus Host Controller module is compliant.

[74]The Cache Line Size is used to align xHC DMA operations.

Table 5-5: Serial Bus Release Number Register (SBRN)

Bit	Description
7:0	**Serial Bus Specification Release Number - RO.** All other combinations are reserved. Bits[7:0] Release Number 30h Release 3.0 31h Release 3.1

5.2.4 Frame Length Adjustment Register (FLADJ)

Address Offset:	61h
Default Value:	20h
Attribute:	RWS
Size:	8 bits

This register is in the Aux Power well. This feature is used to adjust any offset from the clock source that generates the clock that drives the SOF counter. When a new value is written into these six bits, the length of the frame is adjusted for all USB buses implemented by an xHC. Its initial programmed value is system dependent based on the accuracy of hardware USB clock and is initialized by system software (typically the BIOS). This register should only be modified when the *HCHalted* (HCH) bit in the USBSTS register is ‘1’. Changing value of this register while the host controller is operating yields undefined results.

Table 5-6: Frame Length Adjustment Register (FLADJ)

Bit	Description
5:0	**Frame Length Timing Value - RWS/RsvdP.** If NFC = ‘0’, then each decimal value change to this register corresponds to 16 high-speed bit times. The SOF cycle time (number of SOF counter clock periods to generate a SOF microframe length) is equal to 59488 + value in this field. The default value is decimal 32 (20h), which gives a SOF cycle time of 60000. **Frame Length (# HS bit times) (decimal)** — **FLADJ Value (decimal)** 59488 — 0 (00h) 59504 — 1 (01h) 59520 — 2 (02h ... 59984 — 31 (1Fh) 60000 — 32 (20h) ... 60480 — 62 (3Eh) 60496 — 63 (3Fh) If *NFC* = ‘1’ then this field shall be RsvdP.
6	**No Frame Length Timing Capability (NFC) - RO**. This flag indicates whether the host controller implementation supports a *Frame Length Timing Value*. A ‘1’ in this bit indicates that the *Frame Length Timing Value* is not supported. A ‘0’ in this bit indicates that the *Frame Length Timing Value* is supported.
7	**RsvdP**.

Note: A USB3 Bus Interval Adjustment Message is used by the host to adjust its 125 µs. bus interval up to +/-13.333 µs. The FLADJ establishes the center point for this adjustment. The contents of this register are not affected by the receipt of a BUS_INTERVAL_ADJUSTMENT_MESSAGE from a USB3 device. Refer to section 8.5.6.6 in the USB3 spec.

5.2.5 Default Best Effort Service Latency (DBESL)

Address Offset:	62h
Bit Offset:	0
Default Value:	Refer to Description below
Attribute:	RO
Size:	4 bits

This register contains the optimal value for programming the PORTPMSC *Best Effort Service Latency* (BESL) field. Refer to section 4.23.5.1.1.1 for more information.

If BESL LPM is not supported (*HLC* = '0' or *BLC* = '0') then this register is reserved.

Table 5-7: Default Best Effort Service Latency (DBESL)

Bit	Description
3:0	**Default Best Effort Service Latency (DBESL) - RO**. Default = Vendor defined. If the value of this field is non-zero, it defines the recommended value for programming the PORTPMSC register *BESL* field. Refer to sections 5.4.9.2 and 4.23.5.1.1.1 for more information.

5.2.6 Default Best Effort Service Latency Deep (DBESLD)

Address Offset:	62h
Bit Offset:	4
Default Value:	Refer to Description below
Attribute:	RO
Size:	4 bits

This register contains the optimal value for programming the PORTPMSC *Best Effort Service Latency - Deep* (BESLD) field. Refer to section 4.23.5.1.1.1 for more information.

If BESL LPM is not supported (*HLC* = '0' or *BLC* = '0') then this register is reserved.

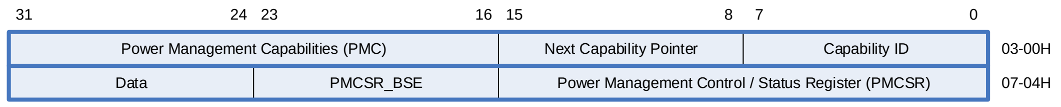
Table 5-8: Default Best Effort Service Latency - Deep (DBESLD)

Bit	Description
7:4	**Default Best Effort Service Latency Deep (DBESLD) - RO**. Default = Vendor defined. If the value of this field is non-zero, it defines the recommended value for programming the PORTPMSC register *BESLD* field. Refer to sections 5.4.9.2 and 4.23.5.1.1.1 for more information.

5.2.7 PCI Power Management Interface

Figure 5-2 is a depiction of the registers defined in the PCI Power Management Capability. xHCI compliant host controllers shall implement the PCI Power Management capability registers as defined in the PCI Specification, which is nearly identical to the structure defined in PCI PM specification, with some additional requirements. Refer to Appendix A.1 for additional xHCI operational requirements for PCI Power Management.

Figure 5-2: PCI Power Management Capability Structure


31
24
23
16
15
8
7
0
Power Management Capabilities (PMC)
Next Capability Pointer
Capability ID
03-00H
Data
PMCSR_BSE
Power Management Control / Status Register (PMCSR)
07-04H

The following section describes the PCI Power Management capability structure, which fields are required or optional for compliance, and how they are implemented by the xHC.

5.2.7.1 PCI Power Management Registers

All fields are reset on full power-up. All of the PCI PM PMCSR register fields except PME_En and PME_Status are reset on exit from D3cold state. If Aux Power is not supplied, the PME_En and PME_Status fields also reset on exit from D3cold state.

The PCI Capability List[75] is used to provide a standard way for software to find and use the PCI Power Management. Refer to section 3.2 in the PCI PM specification for the definition of Power Management Register Block.

IMPLEMENTATION NOTE

NO_SOFT_RESET

If the PCI No_Soft_Reset flag is set to '1', it will also prevent the USB from being reset when the controller transitions from to D3hot from D0. Setting the No_Soft_Reset flag has the benefit of not having to re-initialize all of the USB devices on the bus. The No_Soft_Reset flag does not have any affect on a D3cold (Core power well disabled) to D0 transition, since PERST# is required to be asserted when the main power supply is removed. Refer to section 3.2.4 PMCSR in the PCI PM specification.

5.2.8 Message Signaled Interrupts (MSI & MSI-X) Capability

Below is a depiction of the registers defined in the PCI Message Signaled Interrupt (MSI) capability. If an xHC supports PCI or PCIe it shall implement the PCI MSI and/or MSI-X capabilities as defined in the PCI Specification.

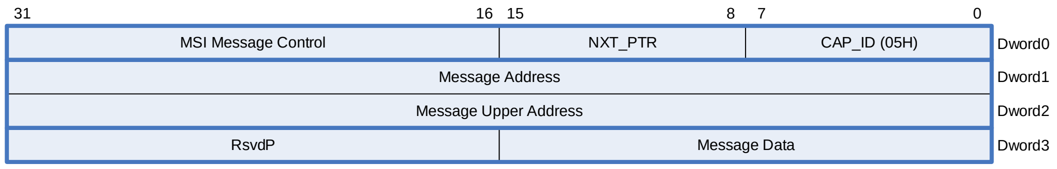
5.2.8.1 MSI configuration

The PCI Capability List is used to provide a standard way for software to find and use the PCI MSI capabilities. The following subsections describe xHC related MSI implementation issues.

[75] PCI Capability List is defined in the PCI Local Bus Specification (Section 6.7)

Figure 5-3 illustrates the Message Signaled Interrupt (MSI) Configuration capability layout, which consist of seven fields. Refer to section 6.8.1 in the PCI specification for the definition of MSI Capability Structure.

Figure 5-3: PCI MSI Configuration Capability Structure



5.2.8.2 MSI-X configuration

The MSI-X capability structure is illustrated in Figure 5-4. More than one *MSI-X Configuration Capability Structure* per function is prohibited, but a function is permitted to have both an MSI and an MSI-X capability structures.

In contrast to the MSI capability structure, which directly contains all of the control/status information for the function's vectors, the MSI-X capability structure instead points to an MSI-X Table structure and a MSI-X Pending Bit Array (PBA) structure, each residing in Memory Space.

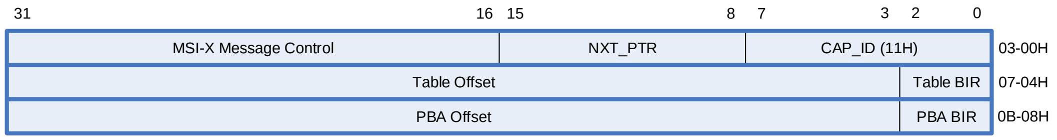
Each structure is mapped by a Base Address register (BAR) belonging to the function, located beginning at 10h in Configuration Space. A BAR Indicator register (BIR) indicates which BAR, and a Qword-aligned Offset indicates where the structure begins relative to the base address associated with the BAR. The BAR is permitted to be either 32-bit or 64-bit, but shall map Memory Space. A function is permitted to map both structures with the same BAR, or to map each structure with a different BAR.

The MSI-X Table structure typically contains multiple entries, each consisting of several fields: Message Address, Message Upper Address, Message Data, and Vector Control. Each entry is capable of specifying a unique vector.

The Pending Bit Array (PBA) structure contains the function’s Pending Bits, one per Table entry, organized as a packed array of bits within Qwords.

The last QWORD will not necessarily be fully populated.

Figure 5-4: MSI-X Configuration Capability Structure



Refer to section 6.8.2 in the PCI specification for the definition of the MSI-X Capability and Table Structures. The following subsections describe xHC related MSI-X implementation issues.

5.2.8.3 MSI-X Table

The MSI-X Capability *Table Offset* field points to the MSI-X Table. Refer to sections 6.8.2.6 through 6.8.2.9 in the PCI specification for the definition of the MSI-X Table Entry fields.

Note: The maximum number of Interrupters supported by the xHC architecture is 1024. The actual number of MSI-X Table entries required by an implementation is determined by the HCSPARAMS1 register *MaxIntrs* field.

Refer to section 5.2.8.5 for *Table Entry* addressing.

5.2.8.4 MSI-X PBA

The MSI-X Capability *PBA Offset* points to the PBA (Pending Bit Array). Refer to section 6.8.2.10 in the PCI specification for the definition of the Pending Bits for MSI-X Table Entries.

Note: The maximum number of Interrupters supported by the xHC architecture is 1024. So only one PBA Qword is implemented, and (at most) only the low order 1023 bits are implemented. The actual number of Pending bits implemented is determined by the HCSPARAMS1 register *MaxIntrs* field.

Refer to section 5.2.8.5 for *Pending Bit* addressing.

5.2.8.5 Accessing the MSI-X Table and MSI-X PBA

The MSI-X Table and MSI-X PBA are permitted to co-reside within a naturally aligned 4 KB address range, though they shall not overlap with each other.

MSI-X Table entries and Pending bits are each numbered 0 through N-1, where N-1 is indicated by the *Table Size* field in the MSI-X Message Control register. For a given arbitrary MSI-X Table entry K, its starting address can be calculated with the formula:

Entry starting address = Table base + K*16

For the associated Pending bit K, its address for Qword access and bit number within that

Qword can be calculated with the formulas:

Qword address = PBA base + (K div 64)*8

Qword bit# = K MOD 64

Software that chooses to read Pending bit K with DWORD accesses can use these formulas:

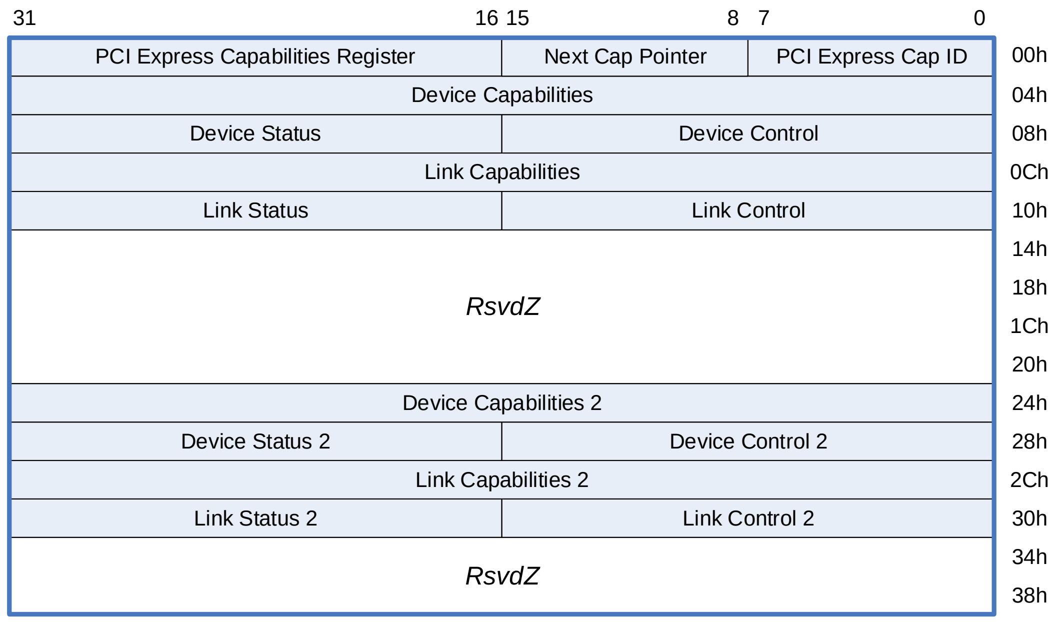
Qword address = PBA base + (K div 32)*4

Qword bit# = K

5.2.9 PCI Express Capability

The structure depicted below represents a PCI Express Capability structure that shall be implemented for any xHC designed to operate as a PCIe device within PCIe capable systems. Refer to section 7.8 of the PCIe spec, for details regarding implementation of this structure.

Figure 5-5: PCI Express Capability Structure



5.2.10 SR-IOV Extended Capability

This optional capability is only required for xHC that provides hardware support for virtualized system environments. The *Single Root I/O Virtualization and Sharing Specification* (SR-IOV) defines virtualization related extensions to the PCI Express (PCIe) specification. SR-IOV is a PCIe Extended Capability.

Refer to section 8 for details on how to implement this capability.

5.3 Host Controller Capability Registers

These registers specify the limits and capabilities of the host controller implementation.

All Capability Registers are Read-Only (RO). The offsets for these registers are all relative to the beginning of the host controller's MMIO address space. The beginning of the host controller's MMIO address space is referred to as "**Base**" throughout this document.

Table 5-9: eXtensible Host Controller Capability Registers

Base Offset	Size (Bytes)	Mnemonic	Register Name	Section
00h	1	CAPLENGTH	Capability Register Length	5.3.1
01h	1	Rsvd		
02h	2	HCIVERSION	Interface Version Number	5.3.2
04h	4	HCSPARAMS1	Structural Parameters 1	5.3.3
08h	4	HCSPARAMS2	Structural Parameters 2	5.3.4
0Ch	4	HCSPARAMS3	Structural Parameters 3	5.3.5
10h	4	HCCPARAMS1	Capability Parameters 1	5.3.6
14h	4	DBOFF	Doorbell Offset	5.3.7
18h	4	RTSOFF	Runtime Register Space Offset	5.3.8
1Ch	4	HCCPARAMS2	Capability Parameters 2	5.3.9
20h	CAPLENGTH-20h	Rsvd		

5.3.1 Capability Registers Length (CAPLENGTH)

Address:	Base + (00h)
Default Value:	Implementation Dependent
Attribute:	RO
Size:	8 bits

This register is used as an offset to add to register base to find the beginning of the Operational Register Space.

5.3.2 Host Controller Interface Version Number (HCIVERSION)

Address:	Base + (02h)
Default Value:	Implementation Dependent

Attribute:	RO
Size:	16 bits

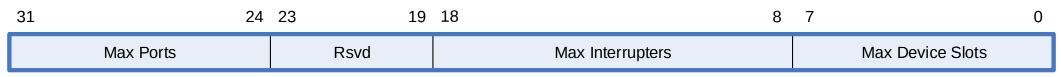
This is a two-byte register containing a BCD encoding of the xHCI specification revision number supported by this host controller. The most significant byte of this register represents a major revision and the least significant byte contains the minor revision extensions. e.g. 0100h corresponds to xHCI version 1.0.0, or 0110h corresponds to xHCI version 1.1.0, etc.

Note: Pre-release versions of the xHC shall declare the specific version of the xHCI that it was implemented against. e.g. 0090h = version 0.9.0.

5.3.3 Structural Parameters 1 (HCSPARAMS1)

Address:	Base + (04h)
Default Value:	Implementation Dependent
Attribute:	RO
Size:	32 bits

Figure 5-6: Structural Parameters 1 Register (HCSPARAMS1)



This register defines basic structural parameters supported by this xHC implementation: Number of Device Slots support, Interrupters, Root Hub ports, etc.

Table 5-10: Host Controller Structural Parameters 1 (HCSPARAMS1)

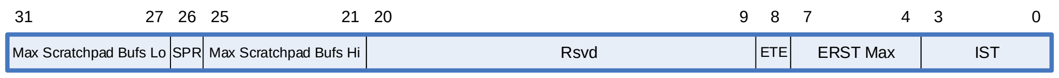
Bits	Description
7:0	**Number of Device Slots (MaxSlots).** This field specifies the maximum number of Device Context Structures and Doorbell Array entries this host controller can support. Valid values are in the range of 1 to 255. The value of '0' is reserved.
18:8	**Number of Interrupters (MaxIntrs).** This field specifies the number of Interrupters implemented on this host controller. Each Interrupter may be allocated to a MSI or MSI-X vector and controls its generation and moderation. The value of this field determines how many Interrupter Register Sets are addressable in the Runtime Register Space (refer to section 5.5). Valid values are in the range of 1h to 400h. A '0' in this field is undefined.
23:19	**Rsvd**.

31:24	**Number of Ports (MaxPorts).** This field specifies the maximum Port Number value, i.e. the highest numbered Port Register Set that are addressable in the Operational Register Space (refer to Table 5-17). Valid values are in the range of 1h to FFh. The value in this field shall reflect the maximum Port Number value assigned by an *xHCI Supported Protocol Capability*, described in section 7.2. Software shall refer to these capabilities to identify whether a specific Port Number is valid, and the protocol supported by the associated Port Register Set.

5.3.4 Structural Parameters 2 (HCSPARAMS2)

Address:	Base + (08h)
Default Value:	Implementation Dependent
Attribute:	RO
Size:	32 bits

Figure 5-7: Structural Parameters 2 Register (HCSPARAMS2)



This register defines additional xHC structural parameters.

Table 5-11: Host Controller Structural Parameters 2 (HCSPARAMS2)

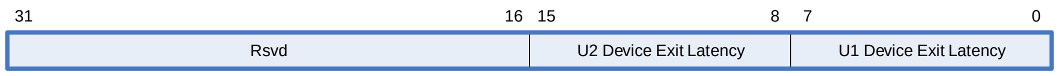
Bit	Description
0:3	**Isochronous Scheduling Threshold (IST).** Default = implementation dependent. The value in this field indicates to system software the minimum distance (in time) that it is required to stay ahead of the host controller while adding TRBs, in order to have the host controller process them at the correct time. The value shall be specified in terms of number of frames/microframes. If bit [3] of IST is cleared to '0', software can add a TRB no later than IST[2:0] Microframes before that TRB is scheduled to be executed. If bit [3] of IST is set to '1', software can add a TRB no later than IST[2:0] Frames before that TRB is scheduled to be executed. Refer to Section 4.14.2 for details on how software uses this information for scheduling isochronous transfers.
7:4	**Event Ring Segment Table Max (ERST Max).** Default = implementation dependent. Valid values are 0 – 15. This field determines the maximum value supported the *Event Ring Segment Table Base Size* registers (5.5.2.3.1), where: The maximum number of Event Ring Segment Table entries = $2^{\text{ERST Max}}$. e.g. if the ERST Max = 7, then the xHC *Event Ring Segment Table(s)* supports up to 128 entries, 15 then 32K entries, etc.

8	**Extended TBC Enable (ETE).** This flag indicates that the host controller implementation is enabled to support Transfer Burst Count (TBC) values greater that 4 in isoch TDs. When this bit is '1', the Isoch TRB *TD Size/TBC* field presents the TBC value, and the TBC/RsvdZ field is RsvdZ. When this bit is '0', the *TDSize/TCB* field presents the TD Size value, and the *TBC/RsvdZ* field presents the TBC value. This bit may be set only if ETC = '1'. Refer to section 4.11.2.3 for more information.
20:9	**Rsvd**.
25:21	**Max Scratchpad Buffers (Max Scratchpad Bufs Hi)**. Default = implementation dependent. This field indicates the high order 5 bits of the number of Scratchpad Buffers system software shall reserve for the xHC. Refer to section 4.20 for more information.
26	**Scratchpad Restore (SPR)**. Default = implementation dependent. If *Max Scratchpad Buffers* is > '0' then this flag indicates whether the xHC uses the Scratchpad Buffers for saving state when executing Save and Restore State operations. If *Max Scratchpad Buffers* is = '0' then this flag shall be '0'. Refer to section 4.23.2 for more information. A value of '1' indicates that the xHC requires the integrity of the Scratchpad Buffer space to be maintained across power events. A value of '0' indicates that the Scratchpad Buffer space may be freed and reallocated between power events.
31:27	**Max Scratchpad Buffers (Max Scratchpad Bufs Lo)**. Default = implementation dependent. Valid values for Max Scratchpad Buffers (Hi and Lo) are 0-1023. This field indicates the low order 5 bits of the number of Scratchpad Buffers system software shall reserve for the xHC. Refer to section 4.20 for more information.

5.3.5 Structural Parameters 3 (HCSPARAMS3)

Address:	Base + (0Ch)
Default Value:	Implementation Dependent
Attribute:	RO
Size:	32 bits

Figure 5-8: Structural Parameters 3 Register (HCSPARAMS3)



This register defines link exit latency related structural parameters.

Table 5-12: Host Controller Structural Parameters 3 (HCSPARAMS3)

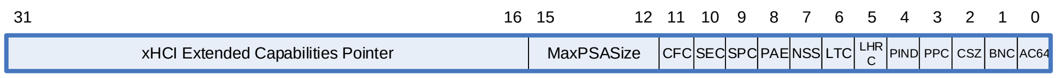
Bit	Description
7:0	**U1 Device Exit Latency.** Worst case latency to transition a root hub Port Link State (PLS) from U1 to U0. Applies to all root hub ports. The following are permissible values: **Value** — **Description** 00h — Zero 01h — Less than 1 µs 02h — Less than 2 µs. ... 0Ah — Less than 10 µs. 0B-FFh — Reserved
15:8	**Rsvd**.
31:16	**U2 Device Exit Latency**. Worst case latency to transition from U2 to U0. Applies to all root hub ports. The following are permissible values: **Value** — **Description** 0000h — Zero 0001h — Less than 1 µs. 0002h — Less than 2 µs. ... 07FFh — Less than 2047 µs. 0800-FFFFh — Reserved

5.3.6 Capability Parameters 1 (HCCPARAMS1)

Address:	Base + (10h)
Default Value:	Implementation Dependent
Attribute:	RO
Size:	32 bits

The default values for all fields in this register are implementation dependent.

Figure 5-9: Capability Parameters 1 Register (HCCPARAMS1)



This register defines optional capabilities supported by the xHCI.

Table 5-13: Host Controller Capability 1 Parameters (HCCPARAMS1)

Bits	Description
0	**64-bit Addressing Capability[76] (AC64).** This flag documents the addressing range capability of this implementation. The value of this flag determines whether the xHC has implemented the high order 32 bits of 64 bit register and data structure pointer fields. Values for this flag have the following interpretation: **Value** — **Description** 0 — 32-bit address memory pointers implemented 1 — 64-bit address memory pointers implemented If 32-bit address memory pointers are implemented, the xHC shall ignore the high order 32 bits of 64 bit data structure pointer fields, and system software shall ignore the high order 32 bits of 64 bit xHC registers.
1	**BW Negotiation Capability (BNC).** This flag identifies whether the xHC has implemented the Bandwidth Negotiation. Values for this flag have the following interpretation: **Value** — **Description** 0 — BW Negotiation not implemented 1 — BW Negotiation implemented Refer to section 4.16 for more information on Bandwidth Negotiation.
2	**Context Size (CSZ).** If this bit is set to ‘1’, then the xHC uses 64 byte Context data structures. If this bit is cleared to ‘0’, then the xHC uses 32 byte Context data structures. Note: This flag does *not* apply to Stream Contexts.
3	**Port Power Control (PPC).** This flag indicates whether the host controller implementation includes port power control. A ‘1’ in this bit indicates the ports have port power switches. A ‘0’ in this bit indicates the port do not have port power switches. The value of this flag affects the functionality of the *PP* flag in each port status and control register (refer to Section 5.4.8).
4	**Port Indicators (PIND).** This bit indicates whether the xHC root hub ports support port indicator control. When this bit is a ‘1’, the port status and control registers include a read/writeable field for controlling the state of the port indicator. Refer to Section 5.4.8 for definition of the *Port Indicator Control* field.
5	**Light HC Reset Capability (LHRC).** This flag indicates whether the host controller implementation supports a Light Host Controller Reset. A ‘1’ in this bit indicates that Light Host Controller Reset is supported. A ‘0’ in this bit indicates that Light Host Controller Reset is not supported. The value of this flag affects the functionality of the *Light Host Controller Reset* (LHCRST) flag in the USBCMD register (refer to Section 5.4.1).

[76]This is not tightly coupled with the USBBASE address register mapping control. The *64-bit Addressing Capability* (AC64) flag indicates whether the host controller can generate 64-bit addresses as a master. The USBBASE register indicates the host controller only needs to decode 32-bit addresses as a slave.

6	**Latency Tolerance Messaging Capability (LTC).** This flag indicates whether the host controller implementation supports Latency Tolerance Messaging (LTM). A '1' in this bit indicates that LTM is supported. A '0' in this bit indicates that LTM is not supported. Refer to section 4.13.1 for more information on LTM.
7	**No Secondary SID Support (NSS)**. This flag indicates whether the host controller implementation supports Secondary Stream IDs. A '1' in this bit indicates that Secondary Stream ID decoding is not supported. A '0' in this bit indicates that Secondary Stream ID decoding is supported. (refer to Sections 4.12.2 and 6.2.3).
8	**Parse All Event Data (PAE)**. This flag indicates whether the host controller implementation Parses all Event Data TRBs while advancing to the next TD after a Short Packet, or it skips all but the first Event Data TRB. A '1' in this bit indicates that all Event Data TRBs are parsed. A '0' in this bit indicates that only the first Event Data TRB is parsed (refer to section 4.10.1.1).
9	**Stopped - Short Packet Capability (SPC)**. This flag indicates that the host controller implementation is capable of generating a *Stopped - Short Packet* Completion Code. Refer to section 4.6.9 for more information.
10	**Stopped EDTLA Capability (SEC)**. This flag indicates that the host controller implementation Stream Context support a Stopped EDTLA field. Refer to sections 4.6.9, 4.12, and 6.4.4.1 for more information. *Stopped EDTLA Capability* support (i.e. *SEC* = '1') shall be mandatory for all xHCI 1.1 compliant xHCs.
11	**Contiguous Frame ID Capability (CFC)**. This flag indicates that the host controller implementation is capable of matching the Frame ID of consecutive Isoch TDs. Refer to section 4.11.2.5 for more information.
15:12	**Maximum Primary Stream Array Size (MaxPSASize)**. This fields identifies the maximum size Primary Stream Array that the xHC supports. The *Primary Stream Array* size = $2^{MaxPSASize+1}$. Valid *MaxPSASize* values are 0 to 15, where '0' indicates that Streams are not supported.
31:16	**xHCI Extended Capabilities Pointer (xECP).** This field indicates the existence of a capabilities list. The value of this field indicates a relative offset, in 32-bit words, from Base to the beginning of the first extended capability. For example, using the offset of Base is 1000h and the xECP value of 0068h, we can calculated the following effective address of the first extended capability: 1000h + (0068h << 2) -> 1000h + 01A0h -> 11A0h

5.3.7 Doorbell Offset (DBOFF)

Address:	Base + (14h)
Default Value:	Implementation Dependent
Attribute:	RO
Size:	32 bits

This register defines the offset of the Doorbell Array base address from the Base.

Figure 5-10: Doorbell Offset Register (DBOFF)

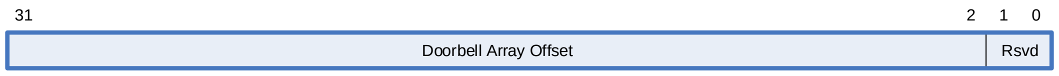


Table 5-14: Doorbell Offset Register (DBOFF)

Bit	Description
1:0	**Rsvd**.
31:2	**Doorbell Array Offset - RO.** Default = implementation dependent. This field defines the offset in Dwords of the Doorbell Array base address from the Base (i.e. the base address of the xHCI Capability register address space).

Note: Normally the Doorbell Array is Dword aligned, however if virtualization is supported by the xHC then it shall be PAGESIZE aligned. e.g. If the PAGESIZE = 4K (1000h), and the Doorbell Array is positioned at a 3 page offset from the Base, then this register shall report 0000 3000h.

5.3.8 Runtime Register Space Offset (RTSOFF)

Address:	Base + (18h)
Default Value:	Implementation Dependent
Attribute:	RO
Size:	32 bits

This register defines the offset of the xHCI Runtime Registers from the Base.

Figure 5-11: Runtime Register Space Offset Register (RTSOFF)

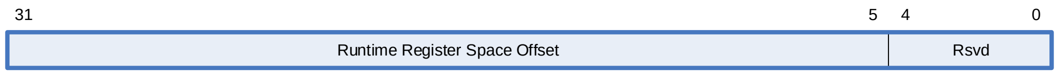


Table 5-15: Runtime Register Space Offset Register (RTSOFF)

Bit	Description
4:0	**Rsvd**.

31:5	**Runtime Register Space Offset - RO.** Default = implementation dependent. This field defines the 32-byte offset of the xHCI Runtime Registers from the Base. i.e. Runtime Register Base Address = Base + Runtime Register Set Offset.

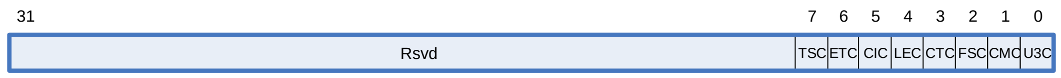
Note: Normally the Runtime Register Space is 32-byte aligned, however if virtualization is supported by the xHC then it shall be PAGESIZE aligned. e.g. If the PAGESIZE = 4K and the Runtime Register Space is positioned at a 1 page offset from the Base, then this register shall report 0000 1000h.

5.3.9 Capability Parameters 2 (HCCPARAMS2)

Address:	Base + (1Ch)
Default Value:	Implementation Dependent
Attribute:	RO
Size:	32 bits

The default values for all fields in this register are implementation dependent.

Figure 5-12: Capability Parameters Register 2 (HCCPARAMS2)



This register defines optional capabilities supported by the xHCI.

Table 5-16: Host Controller Capability Parameters 2 (HCCPARAMS2)

Bits	Description
0	**U3 Entry Capability (U3C) - RO**. This bit indicates whether the xHC Root Hub ports support port Suspend Complete notification. When this bit is '1', *PLC* shall be asserted on any transition of *PLS* to the *U3* State. Refer to section 4.15.1 for more information.
1	**Configure Endpoint Command Max Exit Latency Too Large Capability (CMC) - RO**. This bit indicates whether a *Configure Endpoint Command* is capable of generating a *Max Exit Latency Too Large Capability Error*. When this bit is '1', a *Max Exit Latency Too Large Capability Error* may be returned by a *Configure Endpoint Command*. When this bit is '0', a *Max Exit Latency Too Large Capability Error* shall not be returned by a *Configure Endpoint Command*. This capability is enabled by the *CME* flag in the USBCMD register. Refer to sections 4.23.5.2 and 5.4.1 for more information.
2	**Force Save Context Capability (FSC) - RO**. This bit indicates whether the xHC supports the *Force Save Context Capability*. When this bit is '1', the *Save State* operation shall save any cached Slot, Endpoint, Stream or other Context information to memory. Refer to Implementation Note “FSC and Context handling by Save and Restore”, and sections 4.23.2 and 5.4.1 for more information.

3	**Compliance Transition Capability (CTC) - RO**. This bit indicates whether the xHC USB3 Root Hub ports support the *Compliance Transition Enabled* (CTE) flag. When this bit is '1', USB3 Root Hub port state machine transitions to the **Compliance** substate shall be explicitly enabled software. When this bit is '0', USB3 Root Hub port state machine transitions to the **Compliance** substate are automatically enabled. Refer to section 4.19.1.2.4.1 for more information.
4	**Large ESIT Payload Capability (LEC) - RO**. This bit indicates whether the xHC supports ESIT Payloads greater than 48K bytes. When this bit is '1', ESIT Payloads greater than 48K bytes are supported. When this bit is '0', ESIT Payloads greater than 48K bytes are not supported. Refer to section 6.2.3.8 for more information.
5	**Configuration Information Capability (CIC) - RO**. This bit indicates if the xHC supports extended Configuration Information. When this bit is 1, the *Configuration Value*, *Interface Number*, and *Alternate Setting* fields in the Input Control Context are supported. When this bit is 0, the extended Input Control Context fields are not supported. Refer to section 6.2.5.1 for more information.
6	**Extended TBC Capability[77] (ETC) - RO**. This bit indicates if the TBC field in an Isoch TRB supports the definition of Burst Counts greater than 65535 bytes. When this bit is '1', the Extended EBC capability is supported by the xHC. When this bit is '0', it is not. Refer to section 4.11.2.3 for more information.
7	**Extended TBC TRB Status Capability (ETC_TSC) - RO.** This bit indicates if the TBC/TRBSts field in an Isoch TRB indicates additional information regarding TRB in the TD. When this bit is '1', the Isoch TRB TD Size/TBC field presents TBC value and TBC/TRBSts field presents the TRBSts value. When this bit is '0' then the ETC/ETE values defines the TD Size/TBC field and TBC/RsrvdZ field. This capability shall be enabled only if LEC = '1' and ETC='1'. Refer to section 4.11.2.3 for more information.
31:8	**Reserved**.

5.4 Host Controller Operational Registers

This section defines the xHCI Operational Registers.

The base address of this register space is referred to as **Operational Base**. The Operational Base shall be Dword aligned and is calculated by adding the value of the *Capability Registers Length* (CAPLENGTH) register (refer to Section 5.3.1) to the Capability Base address. All registers are multiples of 32 bits in length.

[77]The *Extended TBC Capability* (ETC) was added to enable support for *Transfer Burst Count* (TBC) values greater than 4, which are required to fully support SSP Isoch bandwidths.

Unless otherwise stated, all registers should be accessed as a 32-bit width on reads with an appropriate software mask, if needed. A software read/modify/write mechanism should be invoked for partial writes.

These registers are located at a positive offset from the Capabilities Registers (refer to Section 5.3).

Table 5-17: Host Controller Operational Registers

Offset	Mnemonic	Register Name	Section
00h	USBCMD	USB Command	5.4.1
04h	USBSTS	USB Status	5.4.2
08h	PAGESIZE	Page Size	5.4.3
0C-13h	RsvdZ		
14h	DNCTRL	Device Notification Control	5.4.4
18h	CRCR	Command Ring Control	5.4.5
20-2Fh	RsvdZ		
30h	DCBAAP	Device Context Base Address Array Pointer	5.4.6
38h	CONFIG	Configure	5.4.7
3C-3FFh	RsvdZ		
400-13FFh		Port Register Set 1-MaxPorts (refer to Table 5-18)	5.4.8, 5.4.9

Note: The *MaxPorts* value in the HCSPARAMS1 register defines the number of Port Register Sets (e.g. PORTSC, PORTPMSC, and PORTLI register sets). The PORTSC, PORTPMSC, and PORTLI register sets are grouped (consecutive Dwords). Refer to their respective sections for their addressing.

The **Offset** referenced in Table 5-17 is the offset from the beginning of the Operational Register space.

The Operational registers are located at a positive offset from the Capabilities Registers (refer to Section 5.3).

Table 5-18: Host Controller USB Port Register Set

Offset	Mnemonic	Register Name	Section
0h	PORTSC	Port Status and Control	5.4.8
4h	PORTPMSC	Port Power Management Status and Control	5.4.9
8h	PORTLI	Port Link Info	5.4.10
Ch	PORTHLPMC	Port Hardware LPM Control	5.4.11

When the Operational Registers are exposed by a Virtual Function (VF), they are emulated and managed by the VMM for the xHC instance presented by the selected VF. The VMM has full discretion as to how writes to these registers will affect the operation of a VF and the value of the read data returned by a VF, however recommendations are provided where appropriate. Refer to section 8 for more information.

5.4.1 USB Command Register (USBCMD)

Address:	Operational Base+ (00h)
Default Value:	0000 0000h
Attribute:	RO, RW (field dependent)
Size:	32 bits

The Command Register indicates the command to be executed by the serial bus host controller. Writing to the register causes a command to be executed.

Figure 5-13: USB Command Register (USBCMD)

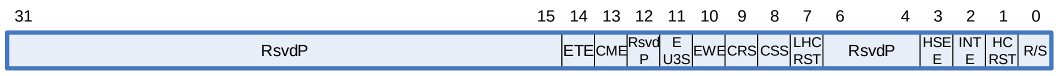


Table 5-19: USB Command Register Bit Definitions (USBCMD)

Bits	Description
0	**Run/Stop (R/S) – RW.** Default = '0'. '1' = Run. '0' = Stop. When set to a '1', the xHC proceeds with execution of the schedule. The xHC continues execution as long as this bit is set to a '1'. When this bit is cleared to '0', the xHC completes any current or queued commands or TDs, and any USB transactions associated with them, then halts. Refer to section 5.4.1.1 for more information on how *R/S* shall be managed. The xHC shall halt within 16 ms. after software clears the *Run/Stop* bit if the above conditions have been met. The *HCHalted* (HCH) bit in the USBSTS register indicates when the xHC has finished its pending pipelined transactions and has entered the stopped state. Software shall not write a '1' to this flag unless the xHC is in the Halted state (i.e. *HCH* in the USBSTS register is '1'). Doing so may yield undefined results. Writing a '0' to this flag when the xHC is in the Running state (i.e. *HCH* = '0') and any Event Rings are in the *Event Ring Full* state (refer to section 4.9.4) may result in lost events. When this register is exposed by a Virtual Function (VF), this bit only controls the run state of the xHC instance presented by the selected VF. Refer to section 8 for more information.
1	**Host Controller Reset (HCRST) – RW.** Default = '0'. This control bit is used by software to reset the host controller. The effects of this bit on the xHC and the Root Hub registers are similar to a Chip Hardware Reset. When software writes a '1' to this bit, the Host Controller resets its internal pipelines, timers, counters, state machines, etc. to their initial value. Any transaction currently in progress on the USB is immediately terminated. A USB reset shall not be driven on USB2 downstream ports, however a Hot or Warm Reset[78] shall be initiated on USB3 Root Hub downstream ports. PCI Configuration registers are not affected by this reset. All operational registers, including port registers and port state machines are set to their initial values. Software shall reinitialize the host controller as described in Section 4.2 in order to return the host controller to an operational state. This bit is cleared to '0' by the Host Controller when the reset process is complete. Software cannot terminate the reset process early by writing a '0' to this bit and shall not write any xHC Operational or Runtime registers until while *HCRST* is '1'. Note, the completion of the xHC reset process is not gated by the Root Hub port reset process. Software shall not set this bit to '1' when the *HCHalted* (HCH) bit in the USBSTS register is a '0'. Attempting to reset an actively running host controller may result in undefined behavior. When this register is exposed by a Virtual Function (VF), this bit only resets the xHC instance presented by the selected VF. Refer to section 8 for more information.
2	**Interrupter Enable (INTE) – RW.** Default = '0'. This bit provides system software with a means of enabling or disabling the host system interrupts generated by Interrupters. When this bit is a '1', then Interrupter host system interrupt generation is allowed, e.g. the xHC shall issue an interrupt at the next interrupt threshold if the host system interrupt mechanism (e.g. MSI, MSI-X, etc.) is enabled. The interrupt is acknowledged by a host system interrupt specific mechanism. When this register is exposed by a Virtual Function (VF), this bit only enables the set of Interrupters assigned to the selected VF. Refer to section 7.7.2 for more information.

[78]Depending on the link state when *HCRST* is asserted, an xHC implementation may choose to issue a Hot Reset rather than a Warm Reset to accelerate the USB recovery process.

3	**Host System Error Enable (HSEE) – RW.** Default = ‘0’. When this bit is a ‘1’, and the *HSE* bit in the USBSTS register is a ‘1’, the xHC shall assert out-of-band error signaling to the host. The signaling is acknowledged by software clearing the *HSE* bit. Refer to section 4.10.2.6 for more information. When this register is exposed by a Virtual Function (VF), the effect of the assertion of this bit on the Physical Function (PF0) is determined by the VMM. Refer to section 8 for more information.
6:4	**RsvdP**.
7	**Light Host Controller Reset (LHCRST) – RO or RW.** Optional normative. Default = ‘0’. If the *Light HC Reset Capability* (LHRC) bit in the HCCPARAMS1 register is ‘1’, then this flag allows the driver to reset the xHC without affecting the state of the ports. A system software read of this bit as ‘0’ indicates the *Light Host Controller Reset* has completed and it is safe for software to re-initialize the xHC. A software read of this bit as a ‘1’ indicates the *Light Host Controller Reset* has not yet completed. If not implemented, a read of this flag shall always return a ‘0’. All registers in the Aux Power well shall maintain the values that had been asserted prior to the *Light Host Controller Reset*. Refer to section 4.23.1 for more information. When this register is exposed by a Virtual Function (VF), this bit only generates a Light Reset to the xHC instance presented by the selected VF, e.g. Disable the VFs’ device slots and set the associated VF Run bit to Stopped. Refer to section 8 for more information.
8	**Controller Save State (CSS) - RW**. Default = ‘0’. When written by software with ‘1’ and *HCHalted* (HCH) = ‘1’, then the xHC shall save any internal state (that may be restored by a subsequent Restore State operation) and if *FSC* = '1' any cached Slot, Endpoint, Stream, or other Context information (so that software may save it). When written by software with ‘1’ and *HCHalted* (HCH) = ‘0’, or written with ‘0’, no Save State operation shall be performed. This flag always returns ‘0’ when read. Refer to the *Save State Status* (SSS) flag in the USBSTS register for information on Save State completion. Refer to section 4.23.2 for more information on xHC Save/Restore operation. Note that undefined behavior may occur if a Save State operation is initiated while *Restore State Status* (RSS) = ‘1’. When this register is exposed by a Virtual Function (VF), this bit only controls saving the state of the xHC instance presented by the selected VF. Refer to section 8 for more information.
9	**Controller Restore State (CRS) - RW**. Default = ‘0’. When set to ‘1’, and *HCHalted* (HCH) = ‘1’, then the xHC shall perform a Restore State operation and restore its internal state. When set to ‘1’ and *Run/Stop* (R/S) = ‘1’ or *HCHalted* (HCH) = ‘0’, or when cleared to ‘0’, no Restore State operation shall be performed. This flag always returns ‘0’ when read. Refer to the *Restore State Status* (RSS) flag in the USBSTS register for information on Restore State completion. Refer to section 4.23.2 for more information. Note that undefined behavior may occur if a Restore State operation is initiated while *Save State Status* (SSS) = ‘1’. When this register is exposed by a Virtual Function (VF), this bit only controls restoring the state of the xHC instance presented by the selected VF. Refer to section 8 for more information.
10	**Enable Wrap Event (EWE) - RW**. Default = ‘0’. When set to ‘1’, the xHC shall generate a MFINDEX Wrap Event every time the MFINDEX register transitions from 03FFFh to 0. When cleared to ‘0’ no MFINDEX Wrap Events are generated. Refer to section 4.14.2 for more information. When this register is exposed by a Virtual Function (VF), the generation of MFINDEX Wrap Events to VFs shall be emulated by the VMM.

11	**Enable U3 MFINDEX Stop (EU3S) - RW**. Default = '0'. When set to '1', the xHC may stop the MFINDEX counting action if all Root Hub ports are in the **U3**, **Disconnected**, **Disabled**, or **Powered-off** state. When cleared to '0' the xHC may stop the MFINDEX counting action if all Root Hub ports are in the **Disconnected**, **Disabled**, **Training**, or **Powered-off** state. Refer to section 4.14.2 for more information.
12	**RsvdP**.
13	**CEM Enable (CME) - RW**. Default = '0'. When set to '1', a *Max Exit Latency Too Large Capability Error* may be returned by a *Configure Endpoint Command*. When cleared to '0', a *Max Exit Latency Too Large Capability Error* shall not be returned by a *Configure Endpoint Command*. This bit is *Reserved* if *CMC* = '0'. Refer to section 4.23.5.2.2 for more information.
31:14	**RsvdP**.

Note: The *R/S* flag has no effect on the operation of the Debug Capability.

5.4.1.1 Run/Stop (R/S)

After *R/S* is written with a '0' by software, the xHC completes any current or queued commands or TDs (and any host initiated transactions on the USB associated with them), then halts and sets *HCH* = '1'. The time it takes for the xHC to halt depends on many things, however if many TDs are queued on Transfer Rings, then it may take a long time for the xHC to complete all outstanding work and halt.

To expedite the xHC halt process, software should ensure the following before clearing the *R/S* bit:

- All endpoints are in the *Stopped* state or *Idle* in the *Running* state, and all Transfer Events associated with them have been received.
- The Command Transfer Ring is in the *Stopped* state (*CRR* = '0') or *Idle* (i.e. the Command Transfer Ring is empty), and all Command Completion Events associated with them have been received.

Software should apply the following rules to determine when a *Busy* Transfer Ring becomes *Idle*:

- For Isoch endpoints:
  - Wait for a *Ring Underrun* or *Ring Overrun* Transfer Event or,
  - Issue a *Stop Endpoint Command* and wait for the associated *Command Completion Event*.
- For non-Isoch endpoints:
  - If the *IOC* flag is set in the last TRB on the Transfer Ring, then wait for its Transfer Event.

- If the *IOC* flag is not set in the last TRB on the Transfer Ring, then there will be no *Transfer Event* generated when the last TRB on the ring is completed, so software shall issue a *Stop Endpoint Command* and wait for the associated *Command Completion Event* and Stopped *Transfer Events*. Refer to section 4.6.9.

Note: Software shall ensure that any pending reset on a USB2 port is completed before *R/S* is cleared to '0'.

Note: The xHC is forced to halt within 16 ms. of software clearing the R/S bit to '0', irrespective of any queued Transfer or Command Ring activity. If software does not follow the "halt process" recommendations above, undefined behavior may occur, e.g. xHC commands or pending USB transactions may be lost, aborted, etc.

5.4.2 USB Status Register (USBSTS)

Address:	Operational Base + (04h)
Default Value:	0000 0001h[79]
Attribute:	RO, RW, RW1C, (field dependent)
Size:	32 bits

This register indicates pending interrupts and various states of the Host Controller. The status resulting from a transaction on the serial bus is not indicated in this register. Software sets a bit to '0' in this register by writing a '1' to it (RW1C). Refer to Section 4.17 for additional information concerning USB interrupt conditions.

Figure 5-14: USB Status Register (USBSTS)

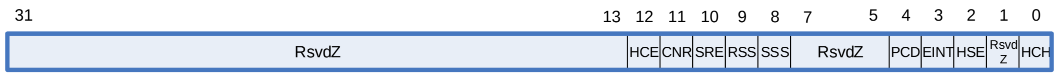


Table 5-20: USB Status Register Bit Definitions (USBSTS)

Bit	Description
0	**HCHalted (HCH) – RO**. Default = '1'. This bit is a '0' whenever the *Run/Stop* (R/S) bit is a '1'. The xHC sets this bit to '1' after it has stopped executing as a result of the *Run/Stop* (R/S) bit being cleared to '0', either by software or by the xHC hardware (e.g. internal error). If this bit is '1', then SOFs, microSOFs, or Isochronous Timestamp Packets (ITP) shall not be generated by the xHC, and any received Transaction Packet shall be dropped. When this register is exposed by a Virtual Function (VF), this bit only reflects the Halted state of the xHC instance presented by the selected VF. Refer to section 8 for more information.
1	**RsvdZ**.

[79]Note, the *CNR* flag may be asserted ('1') when the USBSTS is first examined by software.

2	**Host System Error (HSE) – RW1C.** Default = '0'. The xHC sets this bit to '1' when a serious error is detected, either internal to the xHC or during a host system access involving the xHC module. (In a PCI system, conditions that set this bit to '1' include PCI Parity error, PCI Master Abort, and PCI Target Abort.) When this error occurs, the xHC clears the *Run/Stop* (R/S) bit in the USBCMD register to prevent further execution of the scheduled TDs. If the *HSEE* bit in the USBCMD register is a '1', the xHC shall also assert out-of-band error signaling to the host. Refer to section 4.10.2.6 for more information. When this register is exposed by a Virtual Function (VF), the assertion of this bit affects all VFs and reflects the *Host System Error* state of the Physical Function (PF0). Refer to section 8 for more information.
3	**Event Interrupt (EINT) – RW1C.** Default = '0'. The xHC sets this bit to '1' when the *Interrupt Pending* (IP) bit of any Interrupter transitions from '0' to '1'. Refer to section 7.1.2 for use. Software that uses *EINT* shall clear it prior to clearing any *IP* flags. A race condition may occur if software clears the *IP* flags then clears the *EINT* flag, and between the operations another *IP* '0' to '1' transition occurs. In this case the new IP transition shall be lost. When this register is exposed by a Virtual Function (VF), this bit is the logical 'OR' of the *IP* bits for the Interrupters assigned to the selected VF. And it shall be cleared to '0' when all associated interrupter *IP* bits are cleared, i.e. all the VF's Interrupter Event Ring(s) are empty. Refer to section 8 for more information.
4	**Port Change Detect (PCD) – RW1C.** Default = '0'. The xHC sets this bit to a '1' when any port has a change bit transition from a '0' to a '1'. This bit is allowed to be maintained in the Aux Power well. Alternatively, it is also acceptable that on a D3 to D0 transition of the xHC, this bit is loaded with the OR of all of the PORTSC change bits. Refer to section 4.19.3. This bit provides system software an efficient means of determining if there has been Root Hub port activity. Refer to section 4.15.2.3 for more information. When this register is exposed by a Virtual Function (VF), the VMM determines the state of this bit as a function of the Root Hub Ports associated with the Device Slots assigned to the selected VF. Refer to section 8 for more information.
7:5	**RsvdZ**.
8	**Save State Status (SSS) - RO**. Default = '0'. When the *Controller Save State* (CSS) flag in the USBCMD register is written with '1' this bit shall be set to '1' and remain 1 while the xHC saves its internal state. When the Save State operation is complete, this bit shall be cleared to '0'. Refer to section 4.23.2 for more information. When this register is exposed by a Virtual Function (VF), the VMM determines the state of this bit as a function of the saving the state for the selected VF. Refer to section 8 for more information.
9	**Restore State Status (RSS) - RO**. Default = '0'. When the *Controller Restore State* (CRS) flag in the USBCMD register is written with '1' this bit shall be set to '1' and remain 1 while the xHC restores its internal state. When the Restore State operation is complete, this bit shall be cleared to '0'. Refer to section 4.23.2 for more information. When this register is exposed by a Virtual Function (VF), the VMM determines the state of this bit as a function of the restoring the state for the selected VF. Refer to section 8 for more information.

10	**Save/Restore Error (SRE) - RW1C**. Default = ‘0’. If an error occurs during a Save or Restore operation this bit shall be set to ‘1’. This bit shall be cleared to ‘0’ when a Save or Restore operation is initiated or when written with ‘1’. Refer to section 4.23.2 for more information. When this register is exposed by a Virtual Function (VF), the VMM determines the state of this bit as a function of the Save/Restore completion status for the selected VF. Refer to section 8 for more information.
11	**Controller Not Ready (CNR) – RO**. Default = ‘1’. ‘0’ = Ready and ‘1’ = Not Ready. Software shall not write any Doorbell or Operational register of the xHC, other than the USBSTS register, until CNR = ‘0’. This flag is set by the xHC after a Chip Hardware Reset and cleared when the xHC is ready to begin accepting register writes. This flag shall remain cleared (‘0’) until the next Chip Hardware Reset.
12	**Host Controller Error (HCE) – RO**. Default = 0. 0’ = No internal xHC error conditions exist and ‘1’ = Internal xHC error condition. This flag shall be set to indicate that an internal error condition has been detected which requires software to reset and reinitialize the xHC. Refer to section 4.24.1 for more information.
31:13	**RsvdP**.

Note: The *Event Interrupt* (EINT) and *Port Change Detect* (PCD) flags are typically only used by system software for managing the xHCI when interrupts are disabled or during an SMI.

Note: The *EINT* flag does not generate an interrupt, it is simply a logical OR of the IMAN register *IP* flag ‘0’ to ‘1’ transitions. As such, it does not need to be cleared to clear an xHC interrupt.

5.4.3 Page Size Register (PAGESIZE)

Address:	Operational Base + (08h)
Default Value:	Implementation dependent
Attribute:	RO
Size:	32 bits

Table 5-21: Page Size Register Bit Definitions (PAGESIZE)

Bit	Description
15:0	**Page Size – RO.** Default = Implementation defined. This field defines the page size supported by the xHC implementation. This xHC supports a page size of 2^(n+12) if bit n is Set. For example, if bit 0 is Set, the xHC supports 4k byte page sizes. For a Virtual Function, this register reflects the page size selected in the *System Page Size* field of the SR-IOV Extended Capability structure. For the Physical Function 0, this register reflects the implementation dependent default xHC page size. Various xHC resources reference PAGESIZE to describe their minimum alignment requirements. The maximum possible page size is 128M.
31:16	**Rsvd**.

5.4.4 Device Notification Control Register (DNCTRL)

Address:	Operational Base + (14h)
Default Value:	0000 0000h
Attribute:	RW (Writes shall be Dword)
Size:	32 bits

This register is used by software to enable or disable the reporting of the reception of specific USB Device Notification Transaction Packets. A *Notification Enable* (Nx, where x = 0 to 15) flag is defined for each of the 16 possible device notification types. If a flag is set for a specific notification type, a Device Notification Event shall be generated when the respective notification packet is received. After reset all notifications are disabled. Refer to section 6.4.2.7.

This register shall be written as a Dword. Byte writes produce undefined results.

Figure 5-15: Device Notification Control Register (DNCTRL)

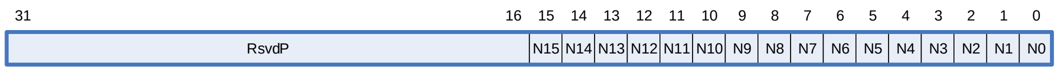


Table 5-22: Device Notification Register Bit Definitions (DNCTRL)

Bit	Description
15:0	**Notification Enable (N0-N15) – RW.** When a Notification Enable bit is set, a Device Notification Event shall be generated when a Device Notification Transaction Packet is received with the matching value in the Notification Type field. For example, setting N1 to '1' enables Device Notification Event generation if a Device Notification TP is received with its Notification Type field set to '1' (FUNCTION_WAKE), etc.
31:16	**RsvdP**.

Note: Of the currently defined USB3 Device Notification Types, only the FUNCTION_WAKE type should not be handled automatically by the xHC. Only under debug conditions would software write the DNCTRL register with a value other than 0002h. Refer to section 8.5.6 in the USB3 specification for more information on Notification Types. If new Device Notification Types are defined, software may receive them by setting the respective Notification Enable bit.

5.4.5 Command Ring Control Register (CRCR)

Address:	Operational Base + (18h)
Default Value:	0000 0000 0000 0000h
Attribute:	RW
Size:	64 bits

The Command Ring Control Register provides Command Ring control and status capabilities, and identifies the address and Cycle bit state of the Command Ring Dequeue Pointer.

Figure 5-16: Command Ring Control Register (CRCR)

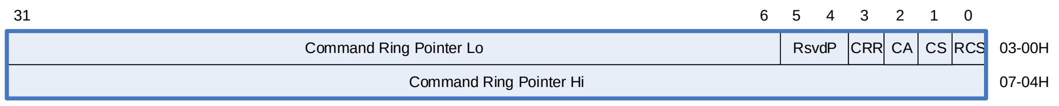


Table 5-23: Command Ring Control Register Bit Definitions (CRCR)

Bit	Description
0	**Ring Cycle State (RCS) - RW**. This bit identifies the value of the xHC *Consumer Cycle State* (CCS) flag for the TRB referenced by the *Command Ring Pointer*. Refer to section 4.9.3 for more information. Writes to this flag are ignored if *Command Ring Running* (CRR) is '1'. If the CRCR is written while the Command Ring is stopped (CRR = '0'), then the value of this flag shall be used to fetch the first Command TRB the next time the *Host Controller Doorbell* register is written with the *DB Reason* field set to *Host Controller Command*. If the CRCR is *not* written while the Command Ring is stopped (CRR = '0'), then the Command Ring shall begin fetching Command TRBs using the current value of the internal Command Ring CCS flag. Reading this flag always returns '0'.
1	**Command Stop (CS) - RW1S.** Default = '0'. Writing a '1' to this bit shall stop the operation of the Command Ring after the completion of the currently executing command, and generate a *Command Completion Event* with the *Completion Code* set to *Command Ring Stopped* and the Command TRB Pointer set to the current value of the Command Ring Dequeue Pointer. Refer to section 4.6.1.1 for more information on stopping a command. The next write to the *Host Controller Doorbell* with *DB Reason* field set to *Host Controller Command* shall restart the Command Ring operation. Writes to this flag are ignored by the xHC if *Command Ring Running* (CRR) = '0'. Reading this bit shall always return '0'.
2	**Command Abort (CA) - RW1S.** Default = '0'. Writing a '1' to this bit shall immediately terminate the currently executing command, stop the Command Ring, and generate a *Command Completion Event* with the *Completion Code* set to *Command Ring Stopped*. Refer to section 4.6.1.2 for more information on aborting a command. The next write to the *Host Controller Doorbell* with *DB Reason* field set to *Host Controller Command* shall restart the Command Ring operation. Writes to this flag are ignored by the xHC if *Command Ring Running* (CRR) = '0'. Reading this bit always returns '0'.
3	**Command Ring Running (CRR) - RO**. Default = 0. This flag is set to '1' if the *Run/Stop* (R/S) bit is '1' and the *Host Controller Doorbell* register is written with the *DB Reason* field set to *Host Controller Command. It is cleared to '0' when the* Command Ring is "stopped" after writing a '1' to the *Command Stop* (CS) or *Command Abort* (CA) flags, or if the *R/S* bit is cleared to '0'.
5:4	**RsvdP**.

63:6	**Command Ring Pointer - RW.** Default = ‘0’. This field defines high order bits of the initial value of the 64-bit Command Ring Dequeue Pointer. Writes to this field are ignored when *Command Ring Running* (CRR) = ‘1’. If the CRCR is written while the Command Ring is stopped (CRR = ‘0’), the value of this field shall be used to fetch the first Command TRB the next time the *Host Controller Doorbell* register is written with the *DB Reason* field set to *Host Controller Command*. If the CRCR is *not* written while the Command Ring is stopped (CRR = ‘0’) then the Command Ring shall begin fetching Command TRBs at the current value of the internal xHC Command Ring Dequeue Pointer. Reading this field always returns ‘0’.

Note: Refer to section 4.6 for more information on Command Ring Stop and Abort operation.

Note: Setting the *Command Stop* (CS) or *Command Abort* (CA) flags while CRR = ‘1’ shall generate a *Command Ring Stopped* Command Completion Event.

Note: Setting both the *Command Stop* (CS) and *Command Abort* (CA) flags with a single write to the CRCR while CRR = ‘1’ shall be interpreted as a Command Abort (CA) by the xHC.

Note: The Command Ring is 64 byte aligned, so the low order 6 bits of the Command Ring Pointer shall always be ‘0’.

Note: The values of the internal xHC Command Ring CCS flag and Dequeue Pointer are undefined after hardware reset, so these fields shall be initialized before setting USBCMD *Run/Stop* (R/S) to ‘1’. Refer to section 4.6.1.

Note: After asserting *Command Stop* (CS) if the Command doorbell is rung before *CRR* = ‘0’, (i.e. the ring is not fully stopped), then the behavior is undefined, e.g. the Command Ring may not restart.

5.4.6 Device Context Base Address Array Pointer Register (DCBAAP)

Address:	Operational Base + (30h)
Default Value:	0000 0000 0000 0000h
Attribute:	RW
Size:	64 bits

The Device Context Base Address Array Pointer Register identifies the base address of the Device Context Base Address Array.

The memory structure referenced by this physical memory pointer is assumed to be physically contiguous and 64-byte aligned.

Figure 5-17: Device Context Base Address Array Pointer Register (DCBAAP)

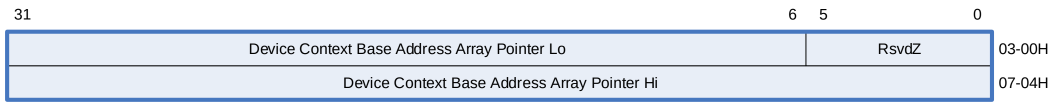

31
6
5
0
Device Context Base Address Array Pointer Lo
RsvdZ
03-00H
Device Context Base Address Array Pointer Hi
07-04H

Table 5-24: Device Context Base Address Array Pointer Register Bit Definitions (DCBAAP)

Bit	Description
5:0	**RsvdZ**.
63:6	**Device Context Base Address Array Pointer - RW. Default = '0'.** This field defines high order bits of the 64-bit base address of the Device Context Pointer Array. A table of address pointers that reference Device Context structures for the devices attached to the host.

5.4.7 Configure Register (CONFIG)

Address:	Operational Base+ (38h)
Default Value:	0000 0000h
Attribute:	RW
Size:	32 bits

This register defines runtime xHC configuration parameters.

Figure 5-18: Configure Register (CONFIG)

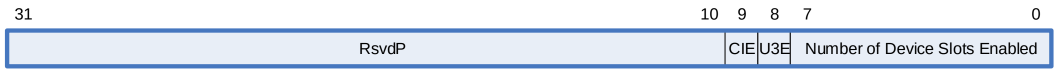


Table 5-25: Configure Register Bit Definitions (CONFIG)

Bit	Description
7:0	**Max Device Slots Enabled (MaxSlotsEn) – RW**. Default = '0'. This field specifies the maximum number of enabled Device Slots. Valid values are in the range of 0 to MaxSlots. Enabled Devices Slots are allocated contiguously. e.g. A value of 16 specifies that Device Slots 1 to 16 are active. A value of '0' disables all Device Slots. A disabled Device Slot shall not respond to Doorbell Register references. This field shall not be modified by software if the xHC is running (*Run/Stop* (R/S) = '1').

8	**U3 Entry Enable (U3E) – RW**. Default = '0'. When set to '1', the xHC shall assert the *PLC* flag ('1') when a Root Hub port transitions to the U3 State. Refer to section 4.15.1 for more information.
9	**Configuration Information Enable (CIE) - RW**. Default = '0'. When set to '1', the software shall initialize the *Configuration Value*, *Interface Number*, and *Alternate Setting* fields in the Input Control Context when it is associated with a Configure Endpoint Command. When this bit is '0', the extended Input Control Context fields are not supported. Refer to section 6.2.5.1 for more information.
31:10	**RsvdP**.

Note: Writing the *Max Device Slots Enabled* (MaxSlotsEn) field with a non-zero value, signals to the xHC that the host controller driver for the xHC is loaded. The *Run/Stop* (R/S) flag in the USBCMD register can be checked to determine if the driver is running.

Note: The value of the *Max Device Slots Enabled* (MaxSlotsEn) field may allow software to scale back its memory usage, in cases where it doesn't need to support the full number of slots supported by the xHC hardware. It may also be used by the xHC to modify internal algorithms for distributing its internal resource, i.e. More data buffering per slot, modify its endpoint scheduling algorithms, etc.

Note: If the xHC is stopped to reduce the *MaxSlotsEn* value, software shall ensure that no active Device Slots (i.e. not in the **Disabled** state) are being disabled, otherwise undefined behavior may occur. e.g. if *MaxSlotsEn* is being changed from 16 to 8, Device Slots 9 through 16 shall be in the Disabled state before *MaxSlotsEn* is changed.

5.4.8 Port Status and Control Register (PORTSC)

Address:	Operational Base + (400h + (10h * (n–1)))
	where: n = Port Number (Valid values are 1, 2, 3, … MaxPorts)
Default:	Field dependent
Attribute:	RO, RW, RW1C (field dependent)
Size	32 bits

A host controller shall implement one or more port registers. The number of port registers implemented by a particular instantiation of a host controller is documented in the HCSPARAMS1 register (Section 5.3.3). Software uses this information as an input parameter to determine how many ports need to be serviced. All ports have the structure defined below.

This register is in the Aux Power well. It is only reset by platform hardware during a cold reset or in response to a *Host Controller Reset* (HCRST). The initial conditions of a port are described in section 4.19.

Note: *Port Status Change Events* cannot be generated if the xHC is stopped (*HCHalted* (HCH) = ‘1’). Refer to section 4.19.2 for more information about change flags.

Note: Software shall ensure that the xHC is running (*HCHalted* (HCH) = ‘0’) before attempting to write to this register.

Software cannot change the state of the port unless *Port Power* (PP) is asserted (‘1’), regardless of the *Port Power Control* (PPC) capability (section 5.3.6). The host is required to have power stable to the port within 20 milliseconds of the ‘0’ to ‘1’ transition of *PP*. If PPC = ‘1’ software is responsible for waiting 20 ms. after asserting *PP*, before attempting to change the state of the port.

Note: If a port has been assigned to the Debug Capability, then the port shall not report device connected (i.e. *CCS* = ‘0’) and enabled when the Port Power Flag is ‘1’. Refer to section 7.6 for more information on the xHCI Debug Capability operation.

Figure 5-19: Port Status and Control Register (PORTSC)

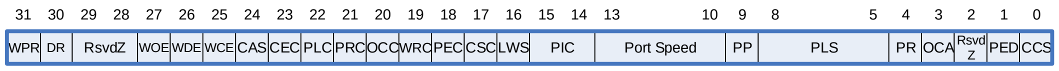


Table 5-26: Port Status and Control Register Bit Definitions (PORTSC)

Bits	Description
0	**Current Connect Status (CCS) – ROS.** Default = ‘0’. ‘1’ = A device is connected[80] to the port. ‘0’ = A device is not connected. This value reflects the current state of the port, and may not correspond directly to the event that caused the *Connect Status Change* (CSC) bit to be set to ‘1’. Refer to sections 4.19.3 and 4.19.4 for more details on the *Connect Status Change* (CSC) assertion conditions. This flag is ‘0’ if *PP* is ‘0’.

[80]For USB2 ports, *CCS* shall be asserted when the port transitions from the *Disconnected* to the *Disabled* state. Refer to section 4.19.1.1. Note that if a D- pull-up resistor is detected, then a Low-speed device is connected and CCS shall be asserted immediately (refer to section 7.1.7.3 of the USB2 spec). If a D+ pull-up resistor is detected, then a Full- or High-speed device may be connected. PED shall not be asserted until after the High-speed Detection Handshake described in section 7.1.7.5 of the USB2 spec completes and determines the speed of the device.For USB3 ports, CCS shall be asserted when the port transitions from the Polling to the Enabled state. Refer to section 4.19.1.2.

1	**Port Enabled/Disabled (PED) – RW1CS.** Default = ‘0’. ‘1’ = Enabled. ‘0’ = Disabled. Ports may only be enabled by the xHC. Software cannot enable a port by writing a ‘1’ to this flag. A port may be disabled by software writing a ‘1’ to this flag. This flag shall automatically be cleared to ‘0’ by a disconnect event or other fault condition. Note that the bit status does not change until the port state actually changes. There may be a delay in disabling or enabling a port due to other host controller or bus events. When the port is disabled (PED = ‘0’) downstream propagation of data is blocked on this port, except for reset. For USB2 protocol ports: When the port is in the **Disabled** state, software shall reset the port (*PR* = ‘1’) to transition *PED* to ‘1’ and the port to the **Enabled** state. For USB3 protocol ports: When the port is in the **Polling** state (after detecting an attach), the port shall automatically transition to the **Enabled** state and set *PED* to ‘1’ upon the completion of successful link training. When the port is in the **Disabled** state, software shall write a ‘5’ (RxDetect) to the *PLS* field to transition the port to the **Disconnected** state. Refer to section 4.19.1.2. PED shall automatically be cleared to ‘0’ when *PR* is set to ‘1’, and set to ‘1’ when *PR* transitions from ‘1’ to ‘0’ after a successful reset. Refer to Port Reset (*PR*) bit for more information on how the *PED* bit is managed. Note that when software writes this bit to a ‘1’, it shall also write a ‘0’ to the *PR* bit[81]. This flag is ‘0’ if *PP* is ‘0’.
2	**RsvdZ**.
3	**Over-current Active (OCA) – RO.** Default = ‘0’. ‘1’ = This port currently has an over-current condition. ‘0’ = This port does not have an over-current condition. This bit shall automatically transition from a ‘1’ to a ‘0’ when the over-current condition is removed.
4	**Port Reset (PR) – RW1S.** Default = ‘0’. ‘1’ = Port Reset signaling is asserted. ‘0’ = Port is not in Reset. When software writes a ‘1’ to this bit generating a ‘0’ to ‘1’ transition, the bus reset sequence is initiated[82]; USB2 protocol ports shall execute the bus reset sequence as defined in the USB2 Spec. USB3 protocol ports shall execute the Hot Reset sequence as defined in the USB3 Spec. *PR* remains set until reset signaling is completed by the root hub. Note that software shall write a ‘1’ to this flag to transition a USB2 port from the **Polling** state to the **Enabled** state. Refer to sections 4.15.2.3 and 4.19.1.1. This flag is ‘0’ if *PP* is ‘0’.

[81] The *PED* and *PR* flags are mutually exclusive. Writing the PORTSC register with *PED* and *PR* set to ‘1’ shall result in undefined behavior.

[82] A ‘0’ to ‘1’ transition of *PR* initiates a USB2 or USB3 reset signaling protocol (refer to section 7.1.7.5 in the USB2 spec and section 6.9.3 in the USB3 spec). The USB reset protocols are not designed to be interrupted or restarted before they are complete, therefore setting *PR* = ‘1’ when it is already equal to ‘1’ shall be ignored by a port to avoid possible USB reset protocol violations.

8:5	**Port Link State (PLS) – RWS.** Default = RxDetect ('5'). This field is used to power manage the port and reflects its current link state. When the port is in the **Enabled** state, system software may set the link U state by writing this field. System software may also write this field to force a **Disabled** to **Disconnected** state transition of the port. **Write Value** — **Description** 0 — The link shall transition to a U0 state from any of the U states. 2[84] — USB2 protocol ports only. The link should transition to the U2 State. 3[83] — The link shall transition to a U3 state from the U0 state. This action selectively suspends the device connected to this port. While the *Port Link State* = U3, the hub does not propagate downstream-directed traffic to this port, but the hub shall respond to resume signaling from the port. 5 — USB3 protocol ports only. If the port is in the **Disabled** state (*PLS* = Disabled, *PP* = 1), then the link shall transition to a RxDetect state and the port shall transition to the **Disconnected** state, else ignored. 10 — USB3 protocol ports only. Shall enable a link transition to the **Compliance** state, i.e. *CTE* = '1'. Refer to section 4.19.1.2.4.1 for more information. 1[84],4,6-9,11-14 — Ignored. 15 — USB2 protocol ports only. If the port is in the **U3** state (*PLS* = U3), then the link shall remain in the U3 state and the port shall transition to the **Resume** substate, else ignored. Refer to section 4.15.2 for more information. Note: The *Port Link State Write Strobe* (LWS) shall also be set to '1' to write this field. For USB2 protocol ports: Writing a value of '2' to this field shall request LPM, asserting L1 signaling on the USB2 bus. Software may read this field to determine if the transition to the U2 state was successful. Writing a value of '0' shall deassert L1 signaling on the USB. Writing a value of '1' shall have no effect. The U1 state shall never be reported by a USB2 protocol port. **Read Value** — **Meaning** 0 — Link is in the **U0** State 1 — Link is in the **U1** State 2 — Link is in the **U2** State 3 — Link is in the **U3** State (Device Suspended) 4 — Link is in the **Disabled** State[85]

[83]Refer to section 4.19.1.1.12 for more information on the U0 to U3 transition of USB2 ports.

[84]The USB3 spec allows software to issue a SetPortFeature(PORT_LINK_STATE, U1 or U2) request. These requests are strictly used for compliance testing to generate an LGO_U1 or LGO_U2 LMP. The xHCI does not support this capability directly, e.g. by writing the PORTSC register with *PLS* = U1 or U2 and *LWS* = '1' to immediately transition a Root Hub port link to a U1 or U2 state.To initiate the transition of a Root Hub port link to a U1 or U2 state, software should write the USB3 PORTPMSC register and set the U*1 Timeout* or U*2 Timeout* fields, respectively, to a value of '1'. This shall cause an LGO_U1 or LGO_U2 LMP to be generated after the respective minimum delay, which is sufficient for compliance testing.

[85]**Disabled** corresponds to the *SS.Disabled* Port Link State defined by the USB3 spec (section 10.14.2.6.1).

	5	Link is in the **RxDetect** State[86]
	6	Link is in the **Inactive** State[87]
	7	Link is in the **Polling** State
	8	Link is in the **Recovery** State
	9	Link is in the **Hot Reset** State
	10	Link is in the **Compliance Mode** State
	11	Link is in the **Test Mode**[88] State
	12-14	Reserved
	15	Link is in the **Resume** State[89]
	This field is undefined if *PP* = '0'. Note: Transitions between different states are not reflected until the transition is complete. Refer to section 4.19 for *PLS* transition conditions. Refer to sections 4.15.2 and 4.23.5 for more information on the use of this field. Refer to the USB2 LPM ECR for more information on USB link power management operation. Refer to section 7.2 for supported USB protocols.	

[86]**RxDetect** corresponds to the *Rx.Detect* Port Link State defined by the USB3 spec (section 10.14.2.6.1).

[87]**Inactive** corresponds to the *SS.Inactive* Port Link State defined by the USB3 spec (section 10.14.2.6.1).

[88]**Test Mode** indicates that the PORTPMSC *Test Mode* field of a USB2 protocol port is non-zero or a USB3 protocol port is in the *Loopback* link state, or an SSIC port is in TEST_MODE (i.e.configured to the MPHY.TEST state, refer to the SSIC spec).

[89]The **Resume** state is not defined as a Port Link State by the USB3 spec (section 10.14.2.6.1). Refer to section 4.15.2. for xHCI use of the Resume state.

9	**Port Power (PP) – RWS.** Default = '1'. This flag reflects a port's logical, power control state. Because host controllers can implement different methods of port power switching, this flag may or may not represent whether (VBus) power is actually applied to the port. When *PP* equals a '0' the port is nonfunctional and shall not report attaches, detaches, or Port Link State (PLS) changes. However, the port shall report over-current conditions when *PP* = '0' if *PPC* = '0'. After modifying *PP*, software shall read *PP* and confirm that it is reached its target state before modifying it again[90], undefined behavior may occur if this procedure is not followed. 0 = This port is in the Powered-off state. 1 = This port is not in the Powered-off state. If the *Port Power Control* (PPC) flag in the HCCPARAMS1 register is '1', then xHC has port power control switches and this bit represents the current setting of the switch ('0' = off, '1' = on). If the *Port Power Control* (PPC) flag in the HCCPARAMS1 register is '0', then xHC does not have port power control switches and each port is hard wired to power, and not affected by this bit. When an over-current condition is detected on a powered port, the xHC shall transition the *PP* bit in each affected port from a '1' to '0' (removing power from the port). Note: If this is an SSIC Port, then the DSP Disconnect process is initiated by '1' to '0' transition of *PP*. After an SSIC USP disconnect process, the port may be disabled by setting PED = 1. As noted, the SSIC spec does not define a mechanism for the USP to request DSP to be re-enabled for a subsequent re-connect. If PED is set to 1 without a prior negotiated disconnect with the USP, subsequent re-enabling of the port requires DSP to issue a WPR to bring USP back to Rx.Detect. Refer to section 5.1.2 in the SSIC Spec for more information. Refer to section 4.19.4 for more information.
13:10	**Port Speed (Port Speed) – ROS.** Default = '0'. This field identifies the speed of the connected USB Device. This field is only relevant if a device is connected (*CCS* = '1') in all other cases this field shall indicate *Undefined Speed*. Refer to section 4.19.3. (see nested table below) Note: This field is invalid on a USB2 protocol port until after the port is reset.

Value	Meaning
0	Undefined Speed
1 -15	*Protocol Speed ID* (PSI), refer to section 7.2.1 for the definition of *PSIV* field in the PSI Dword

[90]A port implementation shall initiate a Port Power change immediately when *PP* is written, however the *PP* flag may be delayed in reflecting this change, e.g. due to waiting for a port related state machine to complete reset signaling or other operation.

15:14	**Port Indicator Control (PIC) – RWS.** Default = 0. Writing to these bits has no effect if the *Port Indicators* (*PIND*) bit in the HCCPARAMS1 register is a '0'. If *PIND* bit is a '1', then the bit encodings are: **Value** / **Meaning** 0 / Port indicators are off 1 / Amber 2 / Green 3 / Undefined Refer to the USB2 Specification section 11.5.3 for a description on how these bits shall be used. This field is '0' if *PP* is '0'.
16	**Port Link State Write Strobe (LWS) – RW**. Default = '0'. When this bit is set to '1' on a write reference to this register, this flag enables writes to the *PLS* field. When '0', write data in *PLS* field is ignored. Reads to this bit return '0'.
17	**Connect Status Change (CSC) – RW1CS.** Default = '0'. '1' = Change in *CCS*. '0' = No change. This flag indicates a change has occurred in the port's *Current Connect Status* (CCS) or *Cold Attach Status* (CAS) bits. Note that this flag shall not be set if the *CCS* transition was due to software setting *PP* to '0', or the *CAS* transition was due to software setting *WPR* to '1'. The xHC sets this bit to '1' for all changes to the port device connect status[91], even if system software has not cleared an existing *Connect Status Change*. For example, the insertion status changes twice before system software has cleared the changed condition, root hub hardware will be "setting" an already-set bit (i.e., the bit will remain '1'). Software shall clear this bit by writing a '1' to it. Refer to section 4.19.2 for more information on change bit usage.
18	**Port Enabled/Disabled Change (PEC) – RW1CS.** Default = '0'. '1' = change in *PED*. '0' = No change. Note that this flag shall not be set if the *PED* transition was due to software setting *PP* to '0'. Software shall clear this bit by writing a '1' to it. Refer to section 4.19.2 for more information on change bit usage. For a USB2 protocol port, this bit shall be set to '1' only when the port is disabled due to the appropriate conditions existing at the EOF2 point (refer to section 11.8.1 of the USB2 Specification for the definition of a *Port Error*). For a USB3 protocol port, this bit shall never be set to '1'.
19	**Warm Port Reset Change (WRC) – RW1CS/RsvdZ.** Default = '0'. This bit is set when Warm Reset processing on this port completes. '0' = No change. '1' = Warm Reset complete. Note that this flag shall not be set to '1' if the Warm Reset processing was forced to terminate due to software clearing *PP* or *PED* to '0'. Software shall clear this bit by writing a '1' to it. Refer to section 4.19.5.1. Refer to section 4.19.2 for more information on change bit usage. This bit only applies to USB3 protocol ports. For USB2 protocol ports it shall be RsvdZ.

[91]The assertion of *CSC* is optional if *CCS* was cleared by the assertion of *OCA*. The assertion of *OCC* generates the necessary Port Status Change Event.

20	**Over-current Change (OCC) – RW1CS.** Default = ‘0’. This bit shall be set to a ‘1’ when there is a ‘0’ to ‘1’ or ‘1’ to ‘0’ transition of Over-current Active (OCA). Software shall clear this bit by writing a ‘1’ to it. Refer to section 4.19.2 for more information on change bit usage.
21	**Port Reset Change (PRC) – RW1CS.** Default = ‘0’. This flag is set to ‘1’ due to a '1' to '0' transition of *Port Reset* (PR). e.g. when any reset processing (Warm or Hot) on this port is complete. Note that this flag shall not be set to ‘1’ if the reset processing was forced to terminate due to software clearing *PP* or *PED* to '0'. ‘0’ = No change. ‘1’ = Reset complete. Software shall clear this bit by writing a '1' to it. Refer to section 4.19.5. Refer to section 4.19.2 for more information on change bit usage.
22	**Port Link State Change (PLC) – RW1CS.** Default = ‘0’. This flag is set to ‘1’ due to the following *PLS* transitions:

Transition	Condition
U3 -> Resume	Wakeup signaling from a device
Resume -> Recovery -> U0	Device Resume complete (USB3 protocol ports only)
Resume -> U0	Device Resume complete (USB2 protocol ports only)
U3 -> Recovery -> U0	Software Resume complete (USB3 protocol ports only)
U3 -> U0	Software Resume complete (USB2 protocol ports only)
U2 -> U0	L1 Resume complete (USB2 protocol ports only)[92]
U0 -> U0	L1 Entry Reject (USB2 protocol ports only)[92]
Any state -> Inactive	Error (USB3 protocol ports only). Note: *PLC* is asserted only on the first LTSSM *SS.Inactive.Disconnect.Detect* to *SS.Inactive.Quiet* substate transition after entering the SS.Inactive state[93].
Any State -> U3	U3 Entry complete. Note: *PLC* is asserted only if *U3E* = ‘1’[94].

Note that this flag shall not be set if the *PLS* transition was due to software setting *PP* to ‘0’. Refer to section 4.23.5 for more information. '0' = No change. '1' = Link Status Changed. Software shall clear this bit by writing a '1' to it. Refer to “PLC Condition:” references in section 4.19.1 for the specific port state

[92]*PLC* shall not be set if an *L1 Resume Complete* or *L1 Entry Reject* condition was due to HW initiated LPM transitions, i.e. while *HLE* = ‘1’. Refer to section 4.23.5.1.1 for more information on USB2 LPM support.

[93]The *Any state* -> *Inactive* transition shall assert *PLS* only when an attached device has entered the Inactive state. If a device is disconnected when the link is in U0, the *PLS* will transition through the U0->Recovery->Inactive->RxDetect states. This requirement eliminates the assertion of *PLC* due the Recovery->SS.Inactive transition of a disconnect.

[94]Refer to section 4.15.1 for more information.

	transitions that set this flag. Refer to section 4.19.2 for more information on change bit usage.
23	**Port Config Error Change (CEC) – RW1CS/RsvdZ**. Default = ‘0’. This flag indicates that the port failed to configure its link partner. 0 = No change. 1 = Port Config Error detected. Software shall clear this bit by writing a '1' to it. Refer to section 4.19.2 for more information on change bit usage. Note: This flag is valid only for USB3 protocol ports. For USB2 protocol ports this bit shall be RsvdZ.
24	**Cold Attach Status (CAS) – RO.** Default = ‘0’. ‘1’ = Far-end Receiver Terminations were detected in the Disconnected state and the Root Hub Port State Machine was unable to advance to the Enabled state. Refer to sections 4.19.8 for more details on the *Cold Attach Status* (CAS) assertion conditions. Software shall clear this bit by writing a '1' to *WPR* or the xHC shall clear this bit if *CCS* transitions to ‘1’. This flag is ‘0’ if *PP* is ‘0’ or for USB2 protocol ports.
25	**Wake on Connect Enable (WCE) – RWS.** Default = ‘0’. Writing this bit to a ‘1’ enables the port to be sensitive to device connects as system wake-up events[95]. Refer to section 4.15 for operational model.
26	**Wake on Disconnect Enable (WDE) – RWS.** Default = ‘0’. Writing this bit to a ‘1’ enables the port to be sensitive to device disconnects as system wake-up events[95]. Refer to section 4.15 for operational model.
27	**Wake on Over-current Enable (WOE) – RWS.** Default = ‘0’. Writing this bit to a ‘1’ enables the port to be sensitive to over-current conditions as system wake-up events[95]. Refer to section 4.15 for operational model.
29:28	**RsvdZ**.
30	**Device Removable[96] (DR) - RO**. This flag indicates if this port has a removable device attached. ‘1’ = Device is non-removable. ‘0’ = Device is removable.
31	**Warm Port Reset (WPR) – RW1S/RsvdZ**. Default = ‘0’. When software writes a ‘1’ to this bit, the Warm Reset sequence as defined in the USB3 Specification is initiated and the *PR* flag is set to ‘1’. Once initiated, the *PR*, *PRC*, and *WRC* flags shall reflect the progress of the Warm Reset sequence. This flag shall always return ‘0’ when read. Refer to section 4.19.5.1. This flag only applies to USB3 protocol ports. For USB2 protocol ports it shall be RsvdZ.

[95]If host software sets this bit to a ‘1’ when the port is not enabled (i.e. *PED* = ‘0’) the results are undefined.

[96]The *DR* field mimics the function of the USB Hub Descriptor *DeviceRemovable* flag for xHC Root Hub ports. Refer to section 10.12.2.1 in the USB3 spec for more information.

5.4.8.1 USB2 to USB3 Port State Mapping

Figure 10-9 in the USB3 Specification describes the Downstream Facing Hub Port State Machine (DFHPSM) of a USB3 hub port. Each DSPORT state specifies the associated *Port Link State* (PLS) value presented by a port.

Figure 11-10 in the USB2 Specification describes the Downstream Facing Hub Port State Machine of a USB2 hub port. Table 5-27 enumerates the Downstream Facing Hub Port State Machine states defined in section 11.5.1 of the USB2 spec and maps them to their equivalent xHCI *Port Link State* (PLS) values.

Table 5-27: USB2 to USB3 Port Link State Mapping

USB2 State	USB3 Port Link State
Not Configured	N/A[97]
Powered-off	Disabled
Disconnected	RxDetect
Disabled	Polling[98]
Resetting	Undefined
Enabled	U0
Transmit	U0
TransmitR	U0
Suspended	U3
Resuming	Resume
SendEOR	Preserves previous PLS state.[99]
Restart_S	N/A[100]

[97]USB2 State does not apply to Root Hub ports.

[98]In this case *PP* and *CCS* = '1', and *PE* and *PR* = '0' for a USB2 port. This state is approximately equivalent to the USB3 DSPORT.Polling state defined in Figure 10-9, section 10.3 of the USB3 spec, where a connected device has been detected but the port is not enabled. This state is only presented by USB2 protocol ports. Refer to section 4.15.2.3.

[99]i.e. U0 if entered from Enabled, Resume if entered from Resuming or L1Resuming.

[100]Section 11.5.1.12 of the USB2 spec "Restart_S" describes a state that applies to the DFHPSM when implemented as USB hub with an Upstream Receiver, as such, this state does not apply to a Root Hub port.

Restart_E	N/A[101]
WLPM[102]	U0
L1Suspend[102]	U2
L1Resuming[102]	Resume

5.4.9 Port PM Status and Control Register (PORTPMSC)

Address:	Operational Base + (404h + (10h * (n-1)))
	where: n = Port Number (Valid values are 1, 2, 3, … MaxPorts)
Default:	0000 0000h
Attribute:	RWS
Size	32 bits

The definition of the fields in the PORTPMSC register depend on the USB protocol supported by the port.

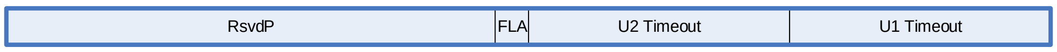
This register is in the Aux Power well. It is only reset by platform hardware during a cold reset or in response to a *Host Controller Reset* (HCRST).

5.4.9.1 USB3 Protocol PORTPMSC Definition

The *USB3 Port Power Management Status and Control* register controls the SuperSpeed USB link U-State timeouts.

Refer to the section 11 of the USB3 spec for more information on Link Power Management.

Figure 5-20: USB3 Port Power Management Status and Control Register (PORTPMSC)



[101]Section 11.5.1.13 of the USB2 spec "Restart_E" describes a state that applies to the DFHPSM when implemented as USB hub with an Upstream Receiver, as such, this state does not apply to a Root Hub port.

[102]USB2 Link Power Management state. Refer to USB2 LPM Figure 4-11.

Table 5-28: USB3 Port Power Management Status and Control Register Bit Definitions (PORTPMSC)

Bit	Description
7:0	**U1 Timeout – RWS.** Default = '0'. Timeout value for U1 inactivity timer. If equal to FFh, the port is disabled from initiating U1 entry. This field shall be set to '0' by the assertion of *PR* to '1'. Refer to section 4.19.4.1 for more information on *U1 Timeout* operation. The following are permissible values: Value Description 00h Zero (default) 01h 1 µs. 02h 2 µs. ... 7Fh 127 µs. 80h–FEh Reserved FFh Infinite
15:8	**U2 Timeout – RWS.** Default = '0'. Timeout value for U2 inactivity timer. If equal to FFh, the port is disabled from initiating U2 entry. This field shall be set to '0' by the assertion of *PR* to '1'. Refer to section 4.19.4.1 for more information on *U2 Timeout* operation. The following are permissible values: Value Description 00h Zero (default) 01h 256 µs 02h 512 µs ... FEh 65,024 ms FFh Infinite A *U2 Inactivity Timeout LMP* shall be sent by the xHC to the device connected on this port when this field is written. Refer to Sections 8.4.3 and 10.4.2.10 of the USB3 specification for more details.
16	**Force Link PM Accept (FLA) - RW**. Default = '0'. When this bit is set to '1', the port shall generate a *Set Link Function* LMP with the Force_LinkPM_Accept bit asserted ('1'). When this bit is cleared to '0', the port shall generate a *Set Link Function* LMP with the Force_LinkPM_Accept bit de-asserted ('0'). This flag shall be set to '0' by the assertion of *PR* to '1' or when *CCS* = transitions from '0' to '1'. Writes to this flag have no effect if *PP* = '0'. The *Set Link Function LMP* is sent by the xHC to the device connected on this port when this bit transitions from '0' to '1' or '1' to '0'. Refer to Sections 8.4.2 and 10.14.2.2 of the USB3 specification for more details. Improper use of the SS Force_LinkPM_Accept functionality can impact the performance of the link significantly. This bit shall only be used for compliance and testing purposes. Software shall ensure that there are no pending packets at the link level before setting this bit. This flag is '0' if *PP* is '0'.
31:17	**RsvdP**.

Refer to the section 10.4.2.1 of the USB3 spec for more information on U1 and U2 Timeouts.

5.4.9.2 USB2 Protocol PORTPMSC Definition

The *USB2 Port Power Management Status and Control* register provides the USB2 LPM parameters necessary for the xHC to generate a LPM Token to the downstream device.

Refer to section 4.23.5.1 for more information on xHCI Link Power Management features.

Refer to the USB2 LPM ECR for more information on USB2 Link Power Management.

Figure 5-21: USB2 Port Power Management Status and Control Register (PORTPMSC)

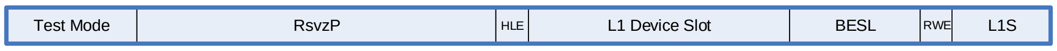


Table 5-29: USB2 Port Power Management Status and Control Register Bit Definitions (PORTPMSC)

Bit	Description
2:0	**L1 Status (L1S) - RO**. Default = 0. This field is used by software to determine whether an L1-based suspend request (LPM transaction) was successful, specifically: **Value Meaning** 0 Invalid - This field shall be ignored by software 1 Success - Port successfully transitioned to L1 (ACK) 2 Not Yet - Device is unable to enter L1 at this time (NYET) 3 Not Supported - Device does not support L1 transitions (STALL) 4 Timeout/Error - Device failed to respond to the LPM Transaction or an error occurred 5-7 Reserved The value of this field is only valid when the port resides in the L0 or L1 state (*PLS* = '0' or '2'). Refer to section 4.23.5.1.1 for more information.
3	**Remote Wake Enable (RWE) - RW**. Default = '0'. System software sets this flag to enable or disable the device for remote wake from L1. The value of this flag shall temporarily (while in L1) override the current setting of the Remote Wake feature set by the standard Set/ClearFeature() commands defined in Universal Serial Bus Specification, revision 2.0, Chapter 9.

7:4	**Best Effort Service Latency (BESL) - RW**. Default = '0'. System software sets this field to indicate to the recipient device how long the xHC will drive resume if it (the xHC) initiates an exit from L1. The *BESL* value encoding is defined in Table 13. Note that the BESL field is used by both software and hardware controlled LPM. Refer to section 4.23.5.1.1 for more information on *BESL* use. Refer to section 5.2.5 for information on how *DBESL* may be used to establish an initial value for *BESL*.
15:8	**L1 Device Slot - RW**. Default = '0'. System software sets this field to indicate the ID of the Device Slot associated with the device directly attached to the Root Hub port. A value of '0' indicates no device is present. The xHC uses this field to lookup information necessary to generate the LPM Token packet.
16	**Hardware LPM Enable (HLE) - RW**. Default = '0'. If this bit is set to '1', then hardware controlled LPM shall be enabled for this port. Refer to section 4.23.5.1.1.1. If the USB2 *Hardware LPM Capability* is not supported (*HLC* = '0') this field shall be RsvdZ. Note the BESL LMP Capability support (i.e. *HLE* = '1' and *BLC* = '1') shall be mandatory for all xHCI 1.1 compliant xHCs.
27:17	**RsvdP**.
31:28	**Port Test Control (Test Mode) – RW.** Default = '0'. When this field is '0', the port is NOT operating in a test mode. A non-zero value indicates that it is operating in test mode and the specific test mode is indicated by the specific value. A non-zero Port Test Control value is only valid to a port that is in the *Powered-Off* state (PLS = *Disabled*). If the port is not in this state, the xHC shall respond with the *Port Test Control* field set to *Port Test Control Error*. Refer to section 4.19.6 for the operational model for using these test modes. The encoding of the Test Mode bits for a USB2 protocol port are: **Value Test Mode** 0 Test mode not enabled 1 Test J_STATE 2 Test K_STATE 3 Test SE0_NAK 4 Test Packet 5 Test FORCE_ENABLE 6-14 Reserved. 15 Port Test Control Error. Refer to the sections 7.1.20 and 11.24.2.13 of the USB2 spec for more information on Test Modes.

Note: All fields in this register apply only to the device attached to and immediately downstream of the associated Root Hub port. It is the responsibility of system software to ensure the *L1 Device Slot* field is consistent with the selected port.

Note: *L0* and *L1* refer to the USB 2.0 "Line" states referred to in the USB2 LPM ECR. These "Line" states map to the xHCI *Port Link States* (PLS) U0 and U2, respectively.

Note: Due to similar exit latencies (~1ms.), the USB 2.0 *L1* state is mapped to the USB3 *U2* state.

Note: The *L1 Device Slot* field provides the device address for generating USB2 LPM transactions to the device attached to the Root Hub port.

5.4.10 Port Link Info Register (PORTLI)

Address:	Operational Base + (408h + (10h * (n-1)))
	where: n = Port Number (Valid values are 1, 2, 3, … MaxPorts)
Default:	0000 0000h
Attribute:	RO
Size	32 bits

The definition of the fields in the PORTLI register depend on the USB protocol supported by the port.

5.4.10.1 USB3 Protocol PORTLI Definition

The *USB3 Port Link Info* register reports the Link Error Count.

Refer to the section 10.14.2.5 of the USB3 spec for more information on Link error count reporting.

Figure 5-22: USB3 Port Link Info Register (PORTLI)

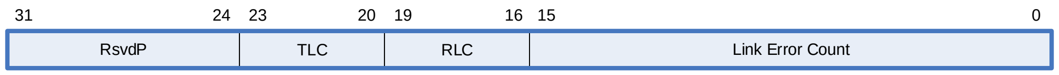


Table 5-30: USB3 Port Link Info Register Bit Definitions (PORTLI)

Bit	Description
15:0	**Link Error Count – RO.** Default = '0'. This field returns the number of link errors detected by the port. This value shall be reset to '0' by the assertion of a Chip Hardware Reset, *HCRST*, when *PR* transitions from '1' to '0', or when *CCS* = transitions from '0' to '1'.
19:16	**Rx Lane Count (RLC) - RO.** Default = '0'. This field that identifies the number of Receive Lanes negotiated by the port. This is a "zero-based" value, where 0 to 15 represents Lane Counts of 1 to 16, respectively. This value is valid only when *CCS* = '1'. *RLC* shall equal '0' for a simplex Sublink. Refer to section 7.2.1 for more information.
23:20	**Tx Lane Count (TLC) - RO.** Default = '0'. This field that identifies the number of Transmit Lanes negotiated by the port. This is a "zero-based" value, where 0 to 15 represents Lane Counts of 1 to 16, respectively. This value is valid only when *CCS* = '1'. *TLC* shall equal '0' for a simplex Sublink. Refer to section 7.2.1 for more information.

31:24	**RsvdP**.

5.4.10.2 USB2 Protocol PORTLI Definition

The *USB2 Port Link Info* register is reserved and shall be treated as RsvdP by software.

5.4.11 Port Hardware LPM Control Register (PORTHLPMC)

Address:	Operational Base + (40Ch + (10h * (n-1)))
	where: n = Port Number (Valid values are 1, 2, 3, … MaxPorts)
Default:	0000 0000h
Attribute:	RWS
Size	32 bits

The definition of the fields in the PORTHLPMC register depend on the USB protocol supported by the port.

This register is in the Aux Power well. It is only reset by platform hardware during a cold reset or in response to a *Host Controller Reset* (HCRST).

5.4.11.1 USB3 Protocol PORTHLPMC Definition

The *USB3 Port Hardware LPM Control* register is reserved and shall be treated as RsvdP by software.

5.4.11.2 USB2 Protocol PORTHLPMC Definition

The optional normative *USB2 Port Hardware LPM Control* register provides the USB2 LPM parameters necessary for the xHC to automatically generate a LPM Token to the downstream device. If LPM is not supported (*HLC* = '0') then this register is reserved. Refer to section 4.23.5.1.1.1 for more information.

Figure 5-23: USB2 Port Hardware LPM Control Register (PORTHLPMC)

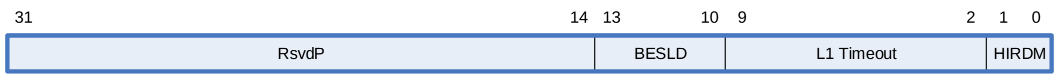


Table 5-31: USB2 Port Hardware LPM Control Register Bit Definitions (PORTHLPMC)

Bit	Description
1:0	**Host Initiated Resume Duration Mode (HIRDM) - RWS**. Default = 0h. Indicates which HIRD value should be used. The following are permissible values: **Value Description** 0 Initiate L1 using *BESL* only on timeout. (default) 1 Initiate L1 using *BESLD* on timeout. If rejected by device, initiate L1 using *BESL*. 3-2 Reserved.
9:2	**L1 Timeout – RWS.** Default = 00h. Timeout value for the L1 inactivity timer (LPM Timer). This field shall be set to 00h by the assertion of *PR* to '1'. Refer to section 4.23.5.1.1.1 for more information on *L1 Timeout* operation. The following are permissible values: **Value Description** 00h 128 µs. (default) 01h 256 µs. 02h 512 µs. 03h 768 µs. ... FFh 65,280 µs.
13:10	**Best Effort Service Latency Deep (BESLD) - RWS**. Default = '0'. System software sets this field to indicate to the recipient device how long the xHC will drive resume on an exit from U2. Refer to section 4.23.5.1.1.1 for more information on *BESLD* use. The *BESLD* value encoding is defined in Table 13. Refer to section 5.2.6 for information on how *DBESLD* may be used to establish an initial value for *BESLD*.
31:14	**RsvdP**.

Refer to Table 4-11 for the mapping of USB2 L-states to U-states.

5.5 Host Controller Runtime Registers

This section defines the xHCI Runtime Register space. The base address of this register space is referred to as **Runtime Base**. The Runtime Base shall be 32-byte aligned and is calculated by adding the value *Runtime Register Space Offset* register (refer to Section 5.3.8) to the Capability *Base* address. All Runtime registers are multiples of 32 bits in length.

Unless otherwise stated, all registers should be accessed with Dword references on reads, with an appropriate software mask if needed. A software read/modify/write mechanism should be invoked for partial writes.

Software should write registers containing a Qword address field using only Qword references. If a system is incapable of issuing Qword references, then writes to the Qword address fields shall be performed using 2 Dword references; low Dword-first, high-Dword second.

Table 5-32: Host Controller Runtime Registers

Offset	Mnemonic	Register Name
0000h	MFINDEX	Microframe Index
001Fh:0004h	RsvdZ	
0020h	IR0	Interrupter Register Set 0
…	…	…
8000h	IR1023	Interrupter Register Set 1023

The Offset referenced in Table 5-32 is the offset from the beginning of the Runtime Register space.

5.5.1 Microframe Index Register (MFINDEX)

Address:	Runtime Base
Default Value:	0000 0000h
Attribute:	RO
Size:	32 bits

This register is used by the system software to determine the current periodic frame. The register value is incremented every 125 microseconds (once each microframe).

This register is only incremented while *Run/Stop* (R/S) = '1'.

The value of this register affects the SOF value generated by USB2 Bus Instances. Refer to section 4.14.2 for details. Also see Figure 4-21.

Figure 5-24: Microframe Index Register (MFINDEX)

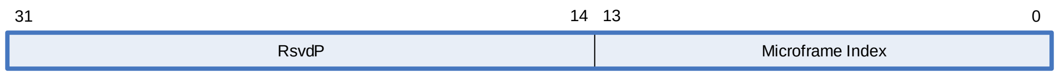


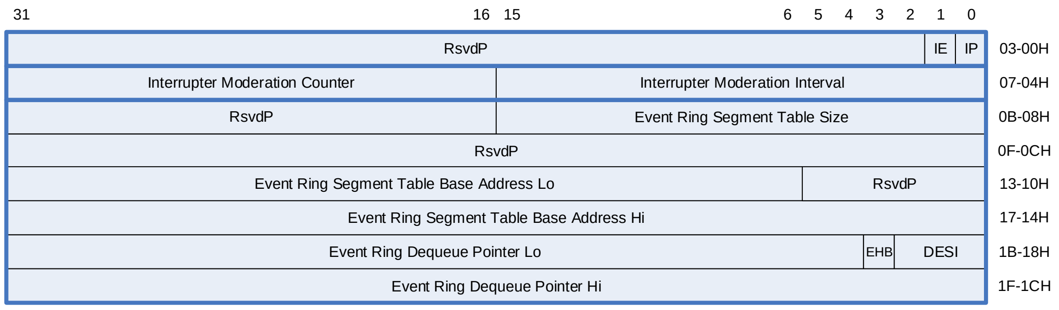
Table 5-33: Microframe Index Register Bit Definitions (MFINDEX)

Bit	Description
13:0	**Microframe Index – RO.** The value in this register increments at the end of each microframe (e.g. 125us.). Bits [13:3] may be used to determine the current 1ms. Frame Index.
31:14	**RsvdZ**.

5.5.2 Interrupter Register Set

The Interrupter logic consists of an *Interrupter Management Register*, an *Interrupter Moderation Register*, and the *Event Ring Registers*. A one to one mapping is defined for Interrupter to MSI-X vector. Up to 1024 Interrupters are supported.

Figure 5-25: Interrupter Register Set


31
16
15
6
5
4
3
2
1
0
RsvdP
IE
IP
03-00H
Interrupter Moderation Counter
Interrupter Moderation Interval
07-04H
RsvdP
Event Ring Segment Table Size
0B-08H
RsvdP
0F-0CH
Event Ring Segment Table Base Address Lo
RsvdP
13-10H
Event Ring Segment Table Base Address Hi
17-14H
Event Ring Dequeue Pointer Lo
EHB
DESI
1B-18H
Event Ring Dequeue Pointer Hi
1F-1CH

Refer to section 4.9.4.3 for a discussion of Primary and Secondary Interrupters and Event Rings.

Note: All registers of the Primary Interrupter shall be initialized before setting the *Run/Stop* (RS) flag in the USBCMD register to ‘1’. Secondary Interrupters may be initialized after *RS* = ‘1’, however all Secondary Interrupter registers shall be initialized before an event that targets them is generated. Not following these rules, shall result in undefined xHC behavior.

Table 5-34: Interrupter Registers

Offset	Size (bits)	Mnemonic	Register Name	Section
00h	32	IMAN	Interrupter Management	5.5.2.1
04h	32	IMOD	Interrupter Moderation	5.5.2.2
08h	32	ERSTSZ	Event Ring Segment Table Size	5.5.2.3.1

0Ch	32	RsvdP		
10h	64	ERSTBA	Event Ring Segment Table Base Address	5.5.2.3.2
18h	64	ERDP	Event Ring Dequeue Pointer	5.5.2.3.3

5.5.2.1 Interrupter Management Register (IMAN)

Address:	Runtime Base + 020h + (32 * Interrupter) where: *Interrupter* is 0, 1, 2, 3, ... 1023
Default Value:	0000 0000h
Attribute:	RW
Size:	32 bits

The Interrupter Management register allows system software to enable, disable, and detect xHC interrupts.

Table 5-35: Interrupter Management Register Bit Definitions (IMAN)

Bit	Description
0	**Interrupt Pending (IP) - RW1C.** Default = '0'. This flag represents the current state of the Interrupter. If *IP* = '1', an interrupt is pending for this Interrupter. A '0' value indicates that no interrupt is pending for the Interrupter. Refer to section 4.17.3 for the conditions that modify the state of this flag.
1	**Interrupt Enable (IE) – RW**. Default = '0'. This flag specifies whether the Interrupter is capable of generating an interrupt. When this bit and the IP bit are set ('1'), the Interrupter shall generate an interrupt when the Interrupter Moderation Counter reaches '0'. If this bit is '0', then the Interrupter is prohibited from generating interrupts.
31:2	**RsvdP**.

Note: In systems that do not support MSI or MSI-X, the *IP* bit may be cleared by writing a '1' to it. Most systems have write buffers that minimize overhead, but this may require a read operation to guarantee that the write has been flushed from posted buffers.

Refer to section 4.17.2 for more information.

5.5.2.2 Interrupter Moderation Register (IMOD)

Address:	Runtime Base + 024h + (32 * Interrupter) where: *Interrupter* is 0, 1, 2, 3, ... 1023

Default Value:	Field dependent
Attribute:	RW
Size:	32 bits

The Interrupter Moderation Register controls the “interrupt moderation” feature of an Interrupter, allowing system software to throttle the interrupt rate generated by the xHC.

Table 5-36: Interrupter Moderation Register (IMOD)

Bit	Description
15:0	**Interrupt Moderation Interval (IMODI) – RW.** Default = ‘4000’ (~1ms). Minimum inter-interrupt interval. The interval is specified in 250ns increments. A value of ‘0’ disables interrupt throttling logic and interrupts shall be generated immediately if *IP* = ‘0’, *EHB* = ‘0’, and the Event Ring is not empty.
31:16	**Interrupt Moderation Counter (IMODC) – RW.** Default = undefined. Down counter. Loaded with the IMODI value whenever *IP* is cleared to ‘0’, counts down to ‘0’, and stops. The associated interrupt shall be signaled whenever this counter is ‘0’, the Event Ring is not empty, the *IE* and *IP* flags = ‘1’, and *EHB* = ‘0’. This counter may be directly written by software at any time to alter the interrupt rate.

Software may use this register to pace (or even out) the delivery of interrupts to the host CPU. This register provides a guaranteed inter-interrupt delay between interrupts asserted by the xHC, regardless of USB traffic conditions. To independently validate configuration settings, software may use the following algorithm to convert the inter-interrupt *Interval* value to the common 'interrupts/sec' performance metric:

$$\text{interrupts/sec} = 1/(250\times10^{-9}\text{sec} \times \mathit{Interval})$$

For example, if the interval is programmed to 500, the xHC guarantees the CPU will not be interrupted by it for 125 microseconds from the last interrupt. The maximum observable interrupt rate from the xHC should never exceed 8000 interrupts/sec.

Inversely, inter-interrupt interval value can be calculated as:

$$\text{inter-interrupt interval} = (250\times10^{-9}\text{sec} \times \text{interrupts/sec}) -1$$

The optimal performance setting for this register is very system and configuration specific.

Refer to section 4.17.2 for more information.

5.5.2.3 Event Ring Registers

Refer to section 4.9.4 for more information in Event Ring management. Refer to section 6.5 for more information on the *Event Ring Segment Table* and its entries.

5.5.2.3.1 Event Ring Segment Table Size Register (ERSTSZ)

Address:	Runtime Base + 028h + (32 * Interrupter) where: *Interrupter* is 0, 1, 2, 3, ... 1023
Default Value:	0000 0000h
Attribute:	RW
Size:	32 bits

The *Event Ring Segment Table Size Register* defines the number of segments supported by the Event Ring Segment Table.

Table 5-37: Event Ring Segment Table Size Register Bit Definitions (ERSTS

Bit	Description
15:0	**Event Ring Segment Table Size – RW.** Default = '0'. This field identifies the number of valid Event Ring Segment Table entries in the Event Ring Segment Table pointed to by the *Event Ring Segment Table Base Address* register. The maximum value supported by an xHC implementation for this register is defined by the *ERST Max* field in the HCSPARAMS2 register (5.3.4). For Secondary Interrupters: Writing a value of '0' to this field disables the Event Ring. Any events targeted at this Event Ring when it is disabled shall result in undefined behavior of the Event Ring. For the Primary Interrupter: Writing a value of '0' to this field shall result in undefined behavior of the Event Ring. The Primary Event Ring cannot be disabled.
31:16	**RsvdP**.

Note: The *Event Ring Segment Table Size* may be set to any value up to *ERST Max*, however software shall allocate a buffer for the Event Ring Segment Table that rounds up its size to the nearest 64B boundary to allow full cache-line accesses.

5.5.2.3.2 Event Ring Segment Table Base Address Register (ERSTBA)

Address:	Runtime Base + 030h + (32 * Interrupter) where: *Interrupter* is 0, 1, 2, 3, ... 1023
Default Value:	0000 0000 0000 0000h
Attribute:	RW
Size:	64 bits

The *Event Ring Segment Table Base Address Register* identifies the start address of the Event Ring Segment Table (ERST). Refer to section 6.5 for the definition of an ERST entry.

Table 5-38: Event Ring Segment Table Base Address Register Bit Definitions (ERSTBA)

Bit	Description
5:0	**RsvdP**.
63:6	**Event Ring Segment Table Base Address Register – RW.** Default = '0'. This field defines the high order bits of the start address of the Event Ring Segment Table. Writing this register sets the Event Ring State Machine:EREP Advancement to the Start state. Refer to Figure 4-12 for more information. For Secondary Interrupters: This field may be modified at any time. For the Primary Interrupter: This field shall not be modified if *HCHalted* (HCH) = '0'.

Note: Refer to section 5.1for register 64-bit address write conventions.

5.5.2.3.3 Event Ring Dequeue Pointer Register (ERDP)

Address:	Runtime Base + 038h + (32 * Interrupter) where: *Interrupter* is 0, 1, 2, 3, … 1023
Default Value:	0000 0000 0000 0000h
Attribute:	RW
Size:	64 bits

The Event Ring Dequeue Pointer Register is written by software to define the Event Ring Dequeue Pointer location to the xHC. Software updates this pointer when it is finished the evaluation of an Event(s) on the Event Ring.

Table 5-39: Event Ring Dequeue Pointer Register Bit Definitions (ERDP)

Bit	Description
2:0	**Dequeue ERST Segment Index (DESI) – RW**. Default = '0'. This field may be used by the xHC to accelerate checking the Event Ring full condition. This field is written with the low order 3 bits of the offset of the ERST entry which defines the Event Ring segment that the Event Ring Dequeue Pointer resides in. Refer to section 6.5 for the definition of an ERST entry.
3	**Event Handler Busy (EHB) - RW1C**. Default = '0'. This flag shall be set to '1' when the *IP* bit is set to '1' and cleared to '0' by software when the Dequeue Pointer register is written. Refer to section 4.17.2 for more information.
5:4	RsvdP

63:6	**Event Ring Dequeue Pointer - RW.** Default = ‘0’. This field defines the high order bits of the 64-bit address of the current Event Ring Dequeue Pointer.

Dequeue ERST Segment Index **(DESI) usage:**

When software finishes processing an Event TRB, it will write the address of that Event TRB to the ERDP. Before enqueuing an Event, the xHC shall check that space is available on the Event Ring. This check can be skipped if the xHC is currently enqueuing Event TRBs in a different ERST segment than the one that software is using to dequeue Events.

To enable this optimization, software provides a hint to the xHC by writing the *Dequeue ERST Segment Index* (DESI) with the low order bits of the index of the segment that the ERDP resides in when it writes the ERDP. The xHC may compare this value with the ERST Segment Index of the Enqueue Pointer to determine whether it should check for an Event Ring Full condition.

E.g. Consider an ERST that defines multiple segments (ERSTSZ > 1), and software is dequeuing an Event TRB in the 1st segment of the ERST. In this case, the *Dequeue ERST Segment Index* (DESI) field shall be written with the value of ‘0’ (i.e. the index of the associated Event Ring Segment Table Entry data structure). If the Dequeue Pointer references an Event TRB in the 2nd segment, then the *Dequeue ERST Segment Index* (DESI) field shall be written with the value of ‘1’, and so on.

Note: If the ERSTSZ is > 8, then the *Dequeue ERST Segment Index* (DESI) shall provide an alias of the actual ERST Segment that was written. e.g *ERST Segment Index*(2:0).

Note: Software shall not write *ERDP* consecutively with the same value unless it is a FULL to EMPTY advancement of the Event Ring.

5.6 Doorbell Registers

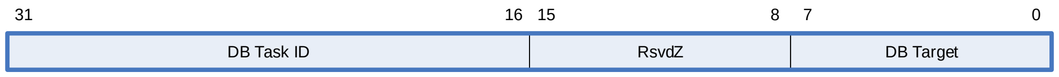
The *Doorbell Array* is organized as an array of up to 256 Doorbell Registers. One 32-bit *Doorbell Register* is defined in the array for each *Device Slot*. System software utilizes the *Doorbell Register* to notify the xHC that it has *Device Slot* related work for the xHC to perform.

The number of Doorbell Registers implemented by a particular instantiation of a host controller is documented in the *Number of Device Slots* (MaxSlots) field of the HCSPARAMS1 register (section 5.3.3).

These registers are pointed to by the *Doorbell Offset Register* (DBOFF) in the xHC Capability register space. The Doorbell Array base address shall be Dword aligned and is calculated by adding the value in the DBOFF register (section 5.3.7) to “Base” (the base address of the xHCI Capability register address space).

Refer to section 4.7 for more information on Doorbell registers.

Figure 5-26: Doorbell Register



All registers are 32 bits in length. Software should read and write these registers using only Dword accesses.

Note: Software shall not write the Doorbell of an endpoint until after it has issued a *Configure Endpoint Command* for the endpoint and received a successful *Command Completion Event*.

Table 5-40: Doorbell Register Bit Field Definitions (DB)

Bit	Description
7:0	**DB Target – RW.** Doorbell Target. This field defines the target of the doorbell reference. The table below defines the xHC notification that is generated by ringing the doorbell. Note that Doorbell Register 0 is dedicated to Command Ring and decodes this field differently than the other Doorbell Registers. Device Context Doorbells (1-255) **Value Definition** 0 Reserved 1 Control EP 0 Enqueue Pointer Update 2 EP 1 OUT Enqueue Pointer Update 3 EP 1 IN Enqueue Pointer Update 4 EP 2 OUT Enqueue Pointer Update 5 EP 2 IN Enqueue Pointer Update … … 30 EP 15 OUT Enqueue Pointer Update 31 EP 15 IN Enqueue Pointer Update 32:247 Reserved 248:255 Vendor Defined Host Controller Doorbell (0) **Value Definition** 0 Command Doorbell 1:247 Reserved 248:255 Vendor Defined This field returns ‘0’ when read and should be treated as “undefined” by software.

	When the *Command Doorbell* is written, the *DB Stream ID* field shall be cleared to '0'.
15:8	**RsvdZ**.
31:16	**DB Stream ID - RW.** Doorbell Stream ID. If the endpoint of a Device Context Doorbell defines Streams, then this field shall be used to identify which Stream of the endpoint the doorbell reference is targeting. System software is responsible for ensuring that the value written to this field is valid. If the endpoint defines Streams (*MaxPStreams* > 0), then 0, 65535 (No Stream) and 65534 (Prime) are reserved Stream ID values and shall not be written to this field. If the endpoint does not define Streams (*MaxPStreams* = 0) and a non-'0' value is written to this field, the doorbell reference shall be ignored. This field only applies to *Device Context Doorbells* and shall be cleared to '0' for *Host Controller Command Doorbells*. This field returns '0' when read.

Note: If virtualization is supported, an xHC implementation shall ensure that an invalid values do not affect another function (PF0 of VFx).

6 Data Structures

This section defines the interface data structures used to communicate control, status and data between HCD (software) and the eXtensible Host Controller (hardware). The data structure definitions in this chapter support a 32-bit or 64-bit memory buffer address space. The interface consists of Transfer Request Buffers (TRBs) that are managed in TRB Rings.

All transfer types (Isoch, Interrupt, Control, and Bulk) utilize the same basic TRB structure. TRBs also support Scatter/Gather operations for Data Page concatenation in systems that employ Virtual Memory.

TRBs are optimized to reduce the total memory footprint of the schedule and to reduce (on average) the number of memory accesses needed to execute a USB transaction.

Table 6-1 identifies the Max Size and alignment requirements of the various xHCI data structures. Note that software shall ensure that no interface data structure with a Max Size less than or equal to 64KB spans a 64KB boundary, and that no interface data structure with a Max Size less than or equal to PAGESIZE spans a PAGESIZE boundary.

The data structures defined in this chapter are (from the host controller's perspective) a mix of read-only and read/writable fields. Software shall preserve the read-only fields on all data structure writes.

Note: Refer to notes at the end of section 5.1.1 for a description of the Reserved field (RsvdZ, RsvdO, etc.) use in data structures.

Note: Whenever possible, software should read and write xHCI data structures as "cache line" operations.

All multi-byte data structure fields follow little-endian ordering; i.e. lower addresses contain the least significant parts of the field. Bytes/characters within a field shall be in little-endian order, i.e. first char of string in least significant byte, second char next significant byte, etc.

Table 6-1: Data Structure Max Size, Boundary, and Alignment Requirement Summary

Data Structure	Max Size in Bytes	Boundary Requirement[103]	Alignment in Bytes	Section
Device Context Base Address Array	2048	PAGESIZE	64	6.1
Device Context	2048	PAGESIZE	64	6.2.1
Input Control Context	64	PAGESIZE	64	6.2.5.1
Slot Context	64	PAGESIZE	32	6.2.2
Endpoint Context	64	PAGESIZE	32	6.2.3
Stream Context	16	PAGESIZE	16	6.2.4.1
Stream Array (Linear)	1M	None	16	6.2.4
Stream Array (Pri/Sec)	4K[104]	PAGESIZE	16	6.2.4
Transfer Ring segments	64K	64KB	16	4.9.2
Command Ring segments	64K	64KB	64	4.9.3
Event Ring segments	64K	64KB	64	4.9.4
Event Ring Segment Table	512K	None	64	6.5
Scratchpad Buffer Array	248	PAGESIZE	64	6.6
Scratchpad Buffers	PAGESIZE	PAGESIZE	Page	4.20

[103]Boundary which data structure shall not span.

[104]Using the Primary/Secondary Stream Array mechanism described in section 4.12.2, Stream Arrays may be limited to 4KB while allowing access to approximately 64K stream IDs.

6.1 Device Context Base Address Array

The Device Context Base Address Array (DCBAA) data structure is used to associate an xHCI Device Slot with its respective Device Context data structure. The Device Context Base Address Array entry associated with each allocated Device Slot shall contain a 64-bit pointer to the base of the associated Device Context. Refer to section 3.2.1 for more information.

System software initializes the *Device Context Base Address Array* to ‘0’, and updates individual entries when the respective *Device Slot* is allocated. The xHC reads an entry in the *Device Context* after a doorbell associated with the entries’ *Device Slot* is rung.

The *Device Context Base Address Array* shall be indexed by the Device Slot ID.

The *Device Context Base Address Array* shall be aligned to a 64 byte boundary.

The *Device Context Base Address Array* shall be physically contiguous within a page.

The *Device Context Base Address Array* shall contain MaxSlotsEn + 1 entries. The maximum size of the *Device Context Base Address Array* is 256 64-bit entries, or 2K Bytes.

Software shall set *Device Context Base Address Array* entries for unallocated Device Slots to ‘0’.

Software shall set *Device Context Base Address Array* entries for allocated Device Slots to point to the *Device Context* data structure associated with the device.

System software shall not modify a *Device Context Base Address Array* entry while the respective Device Slot is enabled.

The address of the *Device Context Base Address Array* shall be written to the *Device Context Base Address Array Pointer Register* (DCBAAP, refer to section 5.4.6) before the xHC is placed into “run” mode (*R/S* = ‘1’).

The *Device Context Base Address Array* data structure is also used to reference the *Scratchpad Buffer Array* data structure. Refer to section 4.20 for more information on Scratchpad Buffer allocation.

If the *Max Scratchpad Buffers* field of the HCSPARAMS2 register is > ‘0’, then the first entry (entry_0) in the DCBAA shall contain a pointer to the *Scratchpad Buffer Array.* If the *Max Scratchpad Buffers* field of the HCSPARAMS2 register is = ‘0’, then the first entry (entry_0) in the DCBAA is reserved and shall be cleared to ‘0’ by software.

Individual elements of the *Device Context Base Address Array* are defined in Table 6-2 and Table 6-3.

Table 6-2: Device Context Base Address Array Element 1-n Field Bit Definitions

Bit	Description
5:0	**RsvdZ**.
63:6	**Device Context Base Address – RW. Default = '0'.** This field contains a pointer to a *Device Context* data structure. *Device Context* data structure is aligned on a 64 byte boundary; hence the low order 6 bits are reserved and always cleared to '0' when initialized by software.

Table 6-3: Device Context Base Address Array Element 0 Field Bit Definitions

Bit	Description
5:0	**RsvdZ**.
63:6	**Scratchpad Buffer Array Base Address – RW. Default = '0'.** This field contains the high order bits of a 64-bit pointer to a *Scratchpad Buffer Array* data structure. *Scratchpad Buffers* are aligned on a Page Size boundary; hence the low order bits are reserved and always cleared to '0' when initialized by software. The number of low order bits cleared to '0' depend on the value of the Page Size register.

Note: The xHCI shall not access the *Device Context Base Address Array* entry associated with a Device Slot that is in the **Enabled** state prior to receiving the first *Address Device Command* for the slot, or a Device Slot that is in the **Disabled** state.

6.2 Contexts

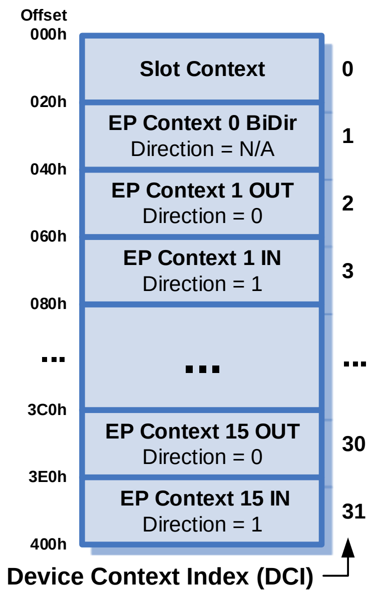
xHC **Contexts** are data structures that act as containers for state information. In some cases a Context may contain other Contexts.

Note: Software shall not modify *Contexts* "owned" by the xHC unless specifically stated.

6.2.1 Device Context

The *Device Context* data structure consists of up to 32 entries. The first entry (entry_0) is the *Slot Context* data structure and the remaining entries are *Endpoint Context* data structures. The *Context Entries* field in the *Slot Context* identifies the number of entries in the *Device Context*. Refer to section Slot for the definition of the *Slot Context* data structure. Refer to section Endpoint Context for the definition of the *Endpoint Context* data structure.

Figure 6-1: Device Context Data Structure



The *Device Context* data structure is used in the xHCI architecture as Output by the xHC to report device configuration and state information to system software. The Device Context data structure is pointed to by an entry in the *Device Context Base Address Array* (refer to section 6.1).

The *Device Context Index* (DCI) is used to reference the respective element of the *Device Context* data structure.

All unused entries of the Device Context shall be initialized to ‘0’ by software.

Note: Figure 6-1 illustrates offsets with 32 byte *Device Context* data structures. i.e. the *Context Size* (CSZ) field in the HCCPARAMS1 register = '0'. If the *Context Size* (CSZ) field = '1' then the *Device Context* data structures consume 64 bytes each. The offsets shall be 040h for the EP Context 0, 080h for EP Context 1, and so on.

Note: Ownership of the Output *Device Context* data structure is passed to the xHC when software rings the Command Ring doorbell for the first *Address Device Command* issued to a Device Slot after an Enable Slot Command, i.e. the first transition of the Slot from the *Enabled* to the *Default* or *Addressed* state. Software shall initialize the Output Device Context to 0 prior to the execution of the first Address Device Command.

Ownership of the *Device Context* data structure is passed back to software when the Device Slot transitions to the *Disabled* state.

Software shall not write the *Device Context* data structure while the xHC has ownership of it. This means that software shall not attempt to allocate an *Input Context* data structure that overlaps or overlays an Output *Device Context* that is owned by the xHC.

6.2.2 Slot Context

The *Slot Context* data structure defines information that applies to a device as a whole.

Note: Unless otherwise stated: **As Input**, all fields of the Slot Context shall be initialized to the appropriate value by software before issuing a command. **As Output**, the xHC shall update each field to reflect the current value that it is using.

Refer to section 4.5.2 for more information on Slot Context initialization.

Figure 6-2: Slot Context Data Structure

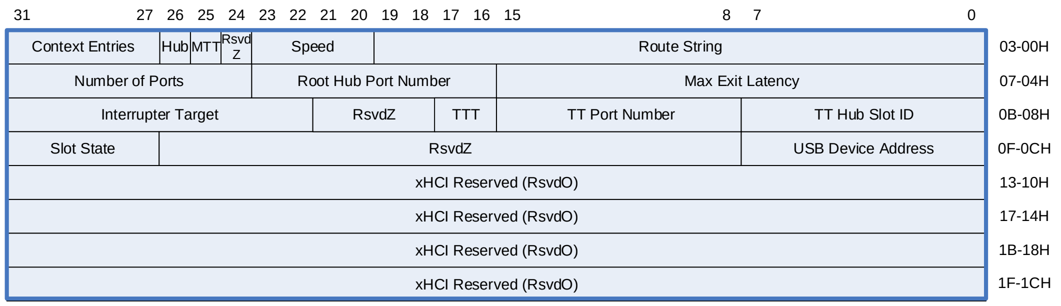


Table 6-4: Offset 00h – Slot Context Field Definitions

Bits	Description
19:0	**Route String.** This field is used by hubs to route packets to the correct downstream port. The format of the Route String is defined in section 8.9 the USB3 specification. As Input, this field shall be set for *all* USB devices, irrespective of their speed, to indicate their location in the USB topology[105].
23:20	**Speed.** This field indicates the speed of the device. Refer to the PORTSC *Port Speed* field in Table 5-26 for the definition of the valid values.
24	**RsvdZ**.

[105]If HS or FS hub in the path supports more than 14 ports the associated *Route String Port* field shall be set to 15.

25	**Multi-TT (MTT)**[106]. This flag is set to '1' by software if this is a High-speed hub (Speed = '3' and Hub = '1') that supports Multiple TTs and the Multiple TT Interface has been enabled by software, or if this is a Low-/Full-speed device (Speed = '1' or '2', and Hub = '0') and connected to the xHC through a parent[107] High-speed hub that supports Multiple TTs and the Multiple TT Interface of the parent hub has been enabled by software, or '0' if not.
26	**Hub.** This flag is set to '1' by software if this device is a USB hub, or '0' if it is a USB function.
31:27	**Context Entries.** This field identifies the index of the last valid Endpoint Context within this Device Context structure. The value of '0' is Reserved and is not a valid entry for this field. Valid entries for this field shall be in the range of 1-31. This field indicates the size of the Device Context structure. For example, ((Context Entries+1) * 32 bytes) = Total bytes for this structure. Note, Output Context Entries values are written by the xHC, and Input Context Entries values are written by software.

Table 6-5: Offset 04h – Slot Context Field Definitions

Bits	Description
15:0	**Max Exit Latency.** The Maximum Exit Latency is in microseconds, and indicates the worst case time it takes to wake up all the links in the path to the device, given the current USB link level power management settings. Refer to section 4.23.5.2 for more information on the use of this field.
23:16	**Root Hub Port Number**. This field identifies the *Root Hub Port Number* used to access the USB device. Refer to section 4.19.7 for port numbering information. Note: Ports are numbered from 1 to MaxPorts.
31:24	**Number of Ports**. If this device is a hub (*Hub* = '1'), then this field is set by software to identify the number of downstream facing ports supported by the hub. Refer to the bNbrPorts field description in the Hub Descriptor (Table 11-13) of the USB2 spec. If this device is not a hub (*Hub* = '0'), then this field shall be '0'.

[106] Software shall issue a Set Interface request to select the Multi-TT Interface of the hub prior to issuing any transactions to devices attached to the hub.

[107] A "parent High-speed hub" is the hub whose downstream facing port isolates the High-speed signaling environment from the Low-/Full-speed signaling environment for a device.

Table 6-6: Offset 08h – Slot Context Field Definitions

Bits	Description
7:0	**TT Hub Slot ID**. If this device is Low-/Full-speed and connected through a High-speed hub, then this field shall contain the Slot ID of the parent High-speed hub[108]. If this device is attached to a Root Hub port or it is not Low-/Full-speed then this field shall be '0'.
15:8	**TT Port Number**. If this device is Low-/Full-speed and connected through a High-speed hub, then this field contains the number of the downstream facing port of the parent High-speed[108] hub. If this device is attached to a Root Hub port or it is not Low-/Full-speed then this field shall be '0'.
17:16	**TT Think Time (TTT)**. If this is a High-speed hub (*Hub* = '1' and *Speed* = *High-Speed*), then this field shall be set by software to identify the time the TT of the hub requires to proceed to the next full-/low-speed transaction. Value Think Time 0 TT requires at most 8 FS bit times of inter-transaction gap on a full-/low-speed downstream bus. 1 TT requires at most 16 FS bit times. 2 TT requires at most 24 FS bit times. 3 TT requires at most 32 FS bit times. Refer to the TT Think Time sub-field of the wHubCharacteristics field description in the Hub Descriptor (Table 11-13) and section 11.18.2 of the USB2 spec for more information on TT Think Time. If this device is not a High-speed hub (*Hub* = '0' or Speed != High-speed), then this field shall be '0'.
21:18	**RsvdZ**.
31:22	**Interrupter Target**. This field defines the index of the Interrupter that will receive *Bandwidth Request Events* and *Device Notification Events* generated by this slot, or when a *Ring Underrun* or *Ring Overrun* condition is reported (refer to section 4.10.3.1). Valid values are between 0 and *MaxIntrs*-1.

[108] A "parent High-speed hub" is the hub whose downstream facing port isolates the High-speed signaling environment from the Low-/Full-speed signaling environment for a device.

Table 6-7: Offset 0Ch – Slot Context Field Definitions

Bits	Description
7:0	**USB Device Address.** This field identifies the address assigned to the USB device by the xHC, and is set upon the successful completion of a *Set Address Command*. Refer to the USB2 spec for a more detailed description. As Output, this field is invalid if the *Slot State* = Disabled or Default. As Input, software shall initialize the field to '0'.
26:8	**RsvdZ**.
31:27	**Slot State.** This field is updated by the xHC when a Device Slot transitions from one state to another. Value Slot State 0 Disabled/Enabled 1 Default 2 Addressed 3 Configured 31-4 Reserved Slot States are defined in section 4.5.3. As Output, since software initializes all fields of the Device Context data structure to '0', this field shall initially indicate the *Disabled* state. As Input, software shall initialize the field to '0'. Refer to section 4.5.3 for more information on Slot State.

Note: The remaining bytes (10-1Fh) within the Slot Context are dedicated for exclusive use by the xHC and shall be treated by system software as Reserved and Opaque (RsvdO).

Note: Figure 6-2 illustrates a 32 byte Slot Context. i.e. the Context Size (CSZ) field in the HCCPARAMS1 register = '0'. If the Context Size (CSZ) field = '1' then each Slot Context data structure consumes 64 bytes, where bytes 32 to 63 are also xHCI Reserved (RsvdO).

Note: The *Speed*, *TT Hub Slot ID* and *TT Port Number* are used to construct the Split Transaction token to the parent hub's Transaction Translator. Refer to section 4.3.7 for more information on these fields.

Note: Depending on the internal organization of an xHC implementation, the *USB Device Address* may not be unique across all *Slot Contexts*, however the *USB Device Address/Root Hub Port Number* combination shall be.

Note: The value of *Max Exit Latency* shall depend on the link states that software has allowed the links in the path to go to. This value is used by the xHC for generating PINGs for periodic endpoints. Its value does not need to be modified when the device is placed on the U3 state because the expectation is that all periodic endpoints of the device are stopped before the device is placed in U3 state, e.g. no Pings will be generated if the periodic Transfer Rings are empty.

6.2.2.1 Address Device Command Usage

The Input *Slot Context* is considered "valid" by the *Address Device Command* if: 1) the *Route String* field defines a valid route string, 2) the *Speed* field identifies the speed of the device, 3) the *Context Entries* field is set to '1' (i.e. Only the Control Endpoint Context is valid), 4) the value of the *Root Hub Port Number* field is between 1 and *MaxPorts*, 5) if the device is LS/FS and connected through a HS hub, then the *TT Hub Slot ID* field references a Device Slot that is assigned to the HS hub, the *MTT* field indicates whether the HS hub supports Multi-TTs, and the *TT Port Number* field indicates the correct TT port number on the HS hub, else these fields are cleared to '0', 6) the *Interrupter Target* field set to a valid value, and 7) all other fields are cleared to '0'.

Prior to the first command execution, a 'valid' Output *Slot Context* for the first *Address Device Command* issued for a Device Slot requires that the value of the *Slot State* field shall be equal to *Disabled* and all other *Slot Context* fields should be cleared to '0'. Refer to section 4.6.5 for more information on valid *Slot Context* field values.

Any Output *Slot Context* is 'valid' for subsequent *Address Device Commands* because all fields of the Output *Slot Context* are overwritten by the xHC.

6.2.2.2 Configure Endpoint Command Usage

A 'valid' Input *Slot Context* for a *Configure Endpoint Command* requires the *Context Entries* field to be initialized to the index of the last valid *Endpoint Context* that is defined by the target configuration. The *Hub* field shall also be initialized. If *Hub* = '1' and *Speed* = *High-Speed*, then the *TT Think Time* (TTT) and *Multi-TT* (MTT) fields shall be initialized. Refer to Table 6-4 and Table 6-5 for the specific initialization values of these fields. If *Hub* = '1', then the *Number of Ports* field shall be initialized, else *Number of Ports* = '0'. Refer to section 4.6.6 for more information on the *Configure Endpoint Command*.

Prior to command execution, a 'valid' Output *Slot Context* for a *Configure Endpoint Command* requires the *Slot State* field to be in the *Addressed* or *Configured* state. If the *Slot State* is not in the *Addressed* or *Configured* state a *Context State Error* shall be generated. The Output *Context Entries* and *Slot State* fields may be updated by the xHC due to a *Configure Endpoint Command*. If the Input *Hub* field = '1', then the Output *Hub* and *Number of Ports* field shall be initialized. If Input *Hub* = '1' and *Speed* = *High-Speed*, then the Output *TT Think Time* (TTT) and *Multi-TT* (MTT) fields shall be initialized.

6.2.2.3 Evaluate Context Command Usage

A 'valid' Input *Slot Context* for an *Evaluate Context Command* requires the *Interrupter Target* and *Max Exit Latency* fields to be initialized. Only these fields shall be evaluated when the xHC receives an *Evaluate Context Command* that flags the *Slot Context* (i.e. *Add Context 0* flag set to ‘1’). Refer to section 4.6.7 for more information on the *Evaluate Context Command*.

Prior to command execution, a 'valid' Output *Slot Context* for an *Evaluate Context Command* requires the *Slot State* field to be in the *Default*, *Addressed* or *Configured* state. If the *Slot State* is not in the *Default*, *Addressed* or *Configured* state a *Context State Error* shall be generated. Only the Output *Interrupter Target* and *Max Exit Latency* fields are updated by the *Evaluate Context Command*.

6.2.2.4 Reset Device Command Usage

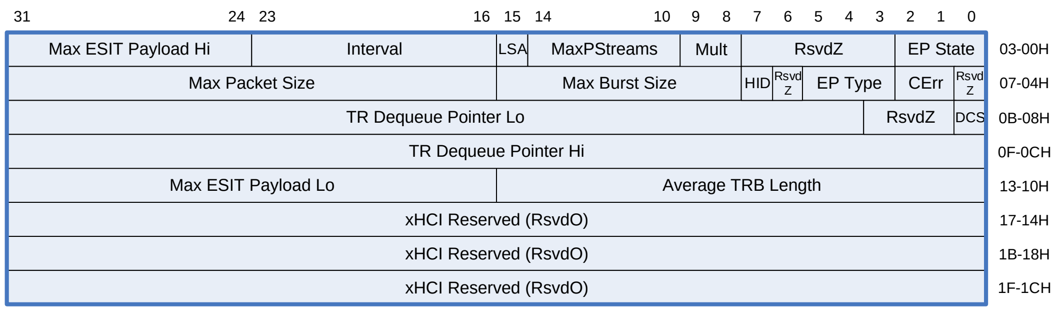
Upon the completion of *Reset Device Command*, the Output Slot Context *Route String* and *Root Hub Port Number* fields shall contain the same values that they contained prior to the execution of the *Reset Device Command*. The *Context Entries* field shall be set to '1' (indicating that only the Default Control Endpoint is operational). And the *Slot State* field shall be set to the Default state. All other fields shall be cleared to '0'.

6.2.3 Endpoint Context

The *Endpoint Context* data structure defines information that applies to a specific endpoint.

Note: Unless otherwise stated: **As Input**, all fields of the Endpoint Context shall be initialized to the appropriate value by software before issuing a command. **As Output**, the xHC shall update each field to reflect the current value that it is using.

Figure 6-3: Endpoint Context Data Structure



31 – 24	23 – 16	15	14 – 10	9 – 8	7	6	5 – 3	2 – 1	0	
Max ESIT Payload Hi	Interval	LSA	MaxPStreams	Mult	RsvdZ			EP State		03-00H
Max Packet Size		Max Burst Size			HID	RsvdZ	EP Type	CErr	RsvdZ	07-04H
TR Dequeue Pointer Lo								RsvdZ	DCS	0B-08H
TR Dequeue Pointer Hi										0F-0CH
Max ESIT Payload Lo		Average TRB Length								13-10H
xHCI Reserved (RsvdO)										17-14H
xHCI Reserved (RsvdO)										1B-18H
xHCI Reserved (RsvdO)										1F-1CH

Table 6-8: Offset 00h – Endpoint Context Field Definitions

Bits	Description
2:0	**Endpoint State (EP State).** The Endpoint State identifies the current operational state of the endpoint. **Value Definition** 0 Disabled The endpoint is not operational 1 Running The endpoint is operational, either waiting for a doorbell ring or processing TDs. 2 HaltedThe endpoint is halted due to a Halt condition detected on the USB. SW shall issue *Reset Endpoint Command* to recover from the Halt condition and transition to the *Stopped* state. SW may manipulate the Transfer Ring while in this state. 3 Stopped The endpoint is not running due to a *Stop Endpoint Command* or recovering from a Halt condition. SW may manipulate the Transfer Ring while in this state. 4 Error The endpoint is not running due to a *TRB Error*. SW may manipulate the Transfer Ring while in this state. 5-7 Reserved As Output, a *Running* to *Halted* transition is forced by the xHC if a STALL condition is detected on the endpoint. A *Running* to *Error* transition is forced by the xHC if a *TRB Error* condition is detected. As Input, this field is initialized to '0' by software. Refer to section 4.8.3 for more information on Endpoint State.
7:3	**RsvdZ**.
9:8	**Mult**. If *LEC* = '0', then this field indicates the maximum number of bursts within an Interval that this endpoint supports, where the valid range of values is '0' to '2', where '0' = 1 burst, '1' = 2 bursts, etc.[109] This field shall be '0' for all endpoint types except for SS Isochronous. If *LEC* = '1', then this field shall be RsvdZ and *Mult* is calculated as: (*Max ESIT Payload* / *Max Packet Size* / *Max Burst Size*) rounded up to the nearest integer value.

[109]Note that there is no requirement that *Max Burst Size* must equal 16 if *Mult* is greater than 0.

14:10	**Max Primary Streams (MaxPStreams)**. This field identifies the maximum number of *Primary Stream IDs* this endpoint supports. Valid values are defined below. If the value of this field is '0', then the *TR Dequeue Pointer* field shall point to a *Transfer Ring*. If this field is > '0' then the *TR Dequeue Pointer* field shall point to a *Primary Stream Context Array*. Refer to section 4.12 for more information. A value of '0' indicates that Streams are not supported by this endpoint and the Endpoint Context *TR Dequeue Pointer* field references a Transfer Ring. A value of '1' to '15' indicates that the *Primary Stream ID Width* is MaxPstreams+1 and the *Primary Stream Array* contains $2^{\text{MaxPStreams+1}}$ entries. For SS Bulk endpoints, the range of valid values for this field is defined by the *MaxPSASize* field in the HCCPARAMS1 register (refer to Table 5-13). This field shall be '0' for all SS Control, Isoch, and Interrupt endpoints, and for all non-SS endpoints.
15	**Linear Stream Array (LSA)**. This field identifies how a Stream ID shall be interpreted. Setting this bit to a value of '1' shall disable Secondary Stream Arrays and a Stream ID shall be interpreted as a linear index into the Primary Stream Array, where valid values for MaxPStreams are '1' to '15'. A value of '0' shall enable Secondary Stream Arrays, where the low order (MaxPStreams+1) bits of a Stream ID shall be interpreted as a linear index into the Primary Stream Array, where valid values for MaxPStreams are '1' to '7'. And the high order bits of a Stream ID shall be interpreted as a linear index into the Secondary Stream Array. If *MaxPStreams* = '0', this field RsvdZ. Refer to section 4.12.2 for more information.
23:16	**Interval**. The period between consecutive requests to a USB endpoint to send or receive data. Expressed in 125 µs. increments. The period is calculated as 125 µs. * 2^{Interval}; e.g., an Interval value of 0 means a period of 125 µs. (2^0 = 1 * 125 µs.), a value of 1 means a period of 250 µs. (2^1 = 2 * 125 µs.), a value of 4 means a period of 2 ms. (2^4 = 16 * 125 µs.), etc. Refer to Table 6-12 for legal *Interval* field values. See further discussion of this field below. Refer to section 6.2.3.6 for more information.
31:24	**Max Endpoint Service Time Interval Payload High (Max ESIT Payload Hi)**. If *LEC* = '1', then this field indicates the high order 8 bits of the *Max ESIT Payload* value. If *LEC* = '0', then this field shall be RsvdZ. Refer to section 6.2.3.8 for more information.

Table 6-9: Offset 04h – Endpoint Context Field Definitions

Bits	Description
0	**RsvdZ**.

2:1	**Error Count (CErr)**[110]**.** This field defines a 2-bit down count, which identifies the number of consecutive USB Bus Errors allowed while executing a TD. If this field is programmed with a non-zero value when the Endpoint Context is initialized, the xHC loads this value into an internal *Bus Error Counter* before executing a USB transaction and decrements it if the transaction fails. If the *Bus Error Counter* counts from '1' to '0', the xHC ceases execution of the TRB, sets the endpoint to the *Halted* state, and generates a *USB Transaction Error Event* for the TRB that caused the internal *Bus Error Counter* to decrement to '0'. If system software programs this field to '0', the xHC shall not count errors for TRBs on the Endpoint's Transfer Ring and there shall be no limit on the number of TRB retries. Refer to section 4.10.2.7 for more information on the operation of the *Bus Error Counter*. Note: *CErr* does not apply to Isoch endpoints and shall be set to '0' if *EP Type = Isoch Out* ('1') or *Isoch In* ('5').
5:3	**Endpoint Type (EP Type).** This field identifies whether an Endpoint Context is Valid, and if so, what type of endpoint the context defines. **Value Endpoint Type Direction** 0 Not Valid N/A 1 Isoch Out 2 Bulk Out 3 Interrupt Out 4 Control Bidirectional 5 Isoch In 6 Bulk In 7 Interrupt In
6	**RsvdZ**.
7	**Host Initiate Disable (HID)**. This field affects Stream enabled endpoints, allowing the *Host Initiated* Stream selection feature to be disabled for the endpoint. Setting this bit to a value of '1' shall disable the Host Initiated Stream selection feature. A value of '0' will enable normal Stream operation. Refer to section 4.12.1.1 for more information.
15:8	**Max Burst Size**. This field indicates to the xHC the maximum number of consecutive USB transactions that should be executed per scheduling opportunity. This is a "zero-based" value, where 0 to 15 represents burst sizes of 1 to 16, respectively. Refer to section 6.2.3.4 for more information.
31:16	**Max Packet Size.** This field indicates the maximum packet size in bytes that this endpoint is capable of sending or receiving when configured. Refer to section 6.2.3.5 for more information.

[110]Software should set *CErr* to '3' for normal operations. The values of '1' or '2' should be avoided during normal operation because they will reduce transfer reliability. The value of '0' is typically only used for test or debug.Note that the xHCI handles CErr differently than the EHCI did.EHCI – if software programs a value of '1' or '2', that value will apply only for the first load of the EHCI Bus Error Counter. And all subsequent reloads of the EHCI Bus Error Counter wi*l*l use '3'. If software programmed '0', then the EHCI will leave it at '0' and disable error counting.xHCI – the Bus *Error Counter is* always reloaded with the value of CErr*, wh*ich means transactions will be less robust (e.g. devices may halt due intermittent errors more frequently) if CErr = '1' or '2'.

Table 6-10: Offset 08h – Endpoint Context Field Definitions

Bits	Description
0	**Dequeue Cycle State (DCS)**. This bit identifies the value of the xHC *Consumer Cycle State* (CCS) flag for the TRB referenced by the *TR Dequeue Pointer*. Refer to section 4.9.2 for more information. This field shall be '0' if *MaxPStreams* > '0'.
3:1	**RsvdZ**.
63:4	**TR Dequeue Pointer.** As Input, this field represents the high order bits of the 64-bit base address of a *Transfer Ring* or a *Stream Context Array* associated with this endpoint. If *MaxPStreams* = '0' then this field shall point to a *Transfer Ring*. If *MaxPStreams* > '0' then this field shall point to a *Stream Context Array.* As Output, if *MaxPStreams* = '0' this field shall be used by the xHC to store the value of the Dequeue Pointer when the endpoint enters the *Halted* or *Stopped* states, and the value of the this field shall be undefined when the endpoint is not in the *Halted* or *Stopped* states. if *MaxPStreams* > '0' then this field shall point to a *Stream Context Array.* The memory structure referenced by this physical memory pointer shall be aligned to a 16-byte boundary.

Table 6-11: Offset 10h – Endpoint Context Field Definition

Bits	Description
15:0	**Average TRB Length**. This field represents the average *Length* of the TRBs executed by this endpoint. The value of this field shall be greater than '0'. Refer to section 4.14.1.1 and the implementation note *TRB Lengths and System Bus Bandwidth* for more information. The xHC shall use this parameter to calculate system bus bandwidth requirements.
31:16	**Max Endpoint Service Time Interval Payload Low (Max ESIT Payload Lo)**. This field indicates the low order 16 bits of the *Max ESIT Payload*. The *Max ESIT Payload* represents the total number of bytes this endpoint will transfer during an ESIT. This field is only valid for periodic endpoints. Refer to section 6.2.3.8 for more information.

Note: The remaining bytes (14-1Fh) within the *Endpoint Context* are dedicated for exclusive use by the xHC and shall be treated by system software as Reserved and Opaque (RsvdO).

Note: Figure 6-3 illustrates a 32 byte *Endpoint Context*. i.e. the *Context Size* (CSZ) field in the HCCPARAMS1 register = '0'. If the *Context Size* (CSZ) field = '1' then each *Endpoint Context* data structure consumes 64 bytes, where bytes 32 to 63 are xHCI Reserved (RsvdO).

Note: The requirement that *TD Fragments* shall not span Transfer Ring Segments places a lower limit on the value of *Average TRB Length*. E.g. a 4KB Transfer Ring Segment may describe up to 256 TRBs, where the last TRB of the segment is a Link TRB. If the MBP is 16K, then the 16KB payload defined by a TD Fragment may not be contain more than 255 Transfer TRBs, which means that software shall not specify an *Average TRB Length* value less than 65B. Larger Transfer Ring Segments allow smaller *Average TRB Length* values. Refer to section 4.11.7.1.

Note: Software shall set *Average TRB Length* to ‘8’ for control endpoints.

6.2.3.1 Address Device Command Usage

The Endpoint 0 Context (DCI = 1) is the only Endpoint Context of an Input Context or Device Context referenced by the Address Device Command. All other Endpoint Contexts (DCI = 2-31) are ignored by the Address Device Command.

The Input Endpoint 0 Context is considered “valid” by the Address Device Command if: 1) the EP Type field = Control, 2) the values of the Max Packet Size, Max Burst Size, and the Interval are considered within range for endpoint type and the speed of the device, 3) the TR Dequeue Pointer field points to a valid Transfer Ring, 4) the DCS field = ‘1’, 5) the MaxPStreams field = ‘0’, and 6) all other fields are within the valid range of values.

Note: The *Max Packet Size* field of the Control *Endpoint Context 0* shall be set by system software to the **default** max packet size for the endpoint as function of the devices’ speed. e.g. 8 bytes for a Low/Full-speed device etc. After the Device Descriptor is read from the device using the *default Max Packet Size*, software may issue an *Evaluate Context Command* to inform the xHC of the actual *Max Packet Size* for the control endpoint if it is different than the default value.

After the first *Address Device Command* execution, any Output *Endpoint Context* is 'valid' for an *Address Device Command* because all fields of the Output *Endpoint Context* are over written by the command.

6.2.3.2 Configure Endpoint Command Usage

The *Configure Endpoint Command* does not reference the Input or Output *Endpoint 0 Context* (DCI = 1). Any other *Endpoint Context* (DCI = 2-31) may be referenced by the *Configure Endpoint Command*.

An Input *Endpoint Context* is considered “valid” by the *Configure Endpoint Command* if the *Add Context* flag is ‘1’ and: 1) the values of the *Max Packet Size*, *Max Burst Size*, and the *Interval* are considered within range for endpoint type and the speed of the device, 2) if *MaxPStreams* > 0, then the *TR Dequeue Pointer* field points to an array of valid *Stream Contexts*, or if *MaxPStreams* = 0, then the *TR Dequeue Pointer* field points to a Transfer Ring, 3) the *EP State* field = *Disabled*, and 4) all other fields are within their valid range of values.

6.2.3.3 Evaluate Context Command Usage

A 'valid' Input *Endpoint Context* for an *Evaluate Context Command* requires that if the *Add Context* flag (A1) for Default Control Endpoint is set to '1', the *Max Packet Size* field shall be evaluated. Endpoint Contexts 2 through 31 shall not be evaluated by the Evaluate Context Command. Refer to section 4.6.7 for more information on the *Evaluate Context Command*.

Prior to command execution, a 'valid' Output *Endpoint Context* for an *Evaluate Context Command* requires the *Endpoint State* (EP State) field to be in the *Running:Idle* sub-state or the *Stopped* state. If the respective context is not in one of these states when the command is executed, undefined behavior may occur.

After the completion of the *Evaluate Context Command*, the updated field values will be used by the xHC for the next transfer performed by the respective endpoint. It is system software's responsibility to coordinate the execution of *Evaluate Context Commands* with Transfer Ring operations.

6.2.3.4 Max Burst Size

The *Max Burst Size* * *Mult* identifies the maximum number of USB transactions that will be executed by the xHC per Transfer Ring scheduling opportunity.

For all Low-/Full-Speed endpoints this field shall be cleared to '0'.

For High-Speed control and bulk endpoints this field shall be cleared to '0'.

For High-Speed isochronous and interrupt endpoints this field shall be set to the *number of additional transaction opportunities per microframe*, i.e. the value defined in bits 12:11 of the USB2 Endpoint Descriptor *wMaxPacketSize* field. Refer to section 9.6.6 of the USB2 Specification.

For SuperSpeed endpoints this field shall be set to the value defined in the *bMaxBurst* field of the SuperSpeed Endpoint Companion Descriptor. Refer to section 9.6.7 of the USB3 Specification.

Refer to section 4.14.4.1 for more information on the use *Max Burst Size*.

6.2.3.5 Max Packet Size

The Max Packet Size field identifies the maximum number of bytes that shall be moved per USB packet. If Max Burst Size is greater than 0, then a High-bandwidth endpoint is defined and a USB transaction may contain up to Max Burst Size+1 packets.

This field shall be set to the value defined in bits 10:0 of the USB Endpoint Descriptor *wMaxPacketSize* field. Note that the *Max Packet Size* field is not encoded the same as the USB *wMaxPacketSize* field Max Packet Size (e.g. as a base 2 multiple), but as a linear byte count value.

6.2.3.6 Interval

The Interval field defines the Interval for polling endpoint for data transfers, expressed in 125 µs units. The periodic interval defined by the Endpoint Context *Interval* field is computed as 125µs * 2^{Interval}, where Interval = 0 to 15.

For high-speed bulk and high-speed control OUT endpoints:

- The Interval shall specify the maximum NAK rate of the endpoint.
- A value of 0 indicates the endpoint never NAKs.
- Other values indicate at most 1 NAK each *Interval* number of microframes.

Refer to Table 6-8 for the definition of the *Interval* field.

Refer to Table 6-12 for the range of valid *Interval* values.

For SuperSpeedPlus and SuperSpeed bulk and control endpoints, the *Interval* field shall not be used by the xHC.

For all other endpoint types and speeds, system software shall translate the *bInterval* field in the USB Endpoint Descriptor to the appropriate value for this field.

Table 6-12: Endpoint Type vs. Interval Calculation

Endpoint	bInterval Range	Time Range	Time Computation	Endpoint Context Valid *Interval* range
FS/LS Interrupt	1 - 255	1 - 255 ms.	bInterval * 1ms.[111]	3-10
FS Isoch	1 - 16	1 - 32,768 ms.	$2^{\text{bInterval-1}}$ * 1ms.	3-18
SSP, SS or HS Interrupt or Isoch	1 - 16	125 µs. - 4,096 ms.	$2^{\text{bInterval-1}}$ * 125 µs.	0-15

6.2.3.7 Reset Device Command Usage

Upon the completion of *Reset Device Command*, the *Output Default Control Endpoint Context* (DCI = '1') *Max Packet Size*, *EP Type*, *CErr*, *TR Dequeue Pointer*, and *Average TRB Length* fields shall contain the same values that they contained

[111]For FS/LS Interrupt endpoints software shall round the computed value of Endpoint Context *Interval* field down to the nearest base 2 multiple of bInterval * 8.

prior to the execution of the *Reset Device Command*. And the EP State field shall be set to the *Running* state. All other fields shall be cleared to '0'.

6.2.3.8 Max ESIT Payload

The *Max ESIT Payload* represents the total number of bytes this endpoint will transfer during an ESIT. With the introduction of USB Gen 2 speed data rates (SSP), the *Max ESIT Payload* values exceeded 64K. The *Large ESIT Payload Capability* (LEC) flag in the HCCPARAMS2 register indicates if an xHC implementation is capable of supporting *Max ESIT Payload* values greater than 48K bytes.

If *LEC* = '0', then the largest value the xHC supports for the *Max ESIT Payload* is 48K bytes. Note that only devices attached to SSP or faster USB3 Root Hub ports may support *Max ESIT Payload* values greater than 48KB.

If *LEC* = '1', then the largest value the xHC supports for the *Max ESIT Payload* is 16MB-1 bytes.

Refer to section 4.14.2 for the definition of an "ESIT" and more information related to setting the value of *Max ESIT Payload*.

For periodic endpoints, the *Max ESIT Payload* is used by the xHC to reserve the bus transfer time for the endpoint in its Pipe Schedule.

6.2.4 Stream Context Array

The xHCI supports hierarchal Stream Context Arrays. Refer to section 4.12 for more information on their use. A *Stream Context Array* contains *Stream Context* data structures. Entries are addressed by a *Stream ID*. Steam ID 0 is reserved and does not reference a Transfer Ring or another Stream Context Array.

6.2.4.1 Stream Context

The *Stream Context* data structure defines information that applies to a specific Stream associated with an endpoint.

Note: Unless otherwise stated: **As Input**, all fields of the Stream Context shall be initialized to the appropriate value by software before issuing a command. **As Output**, the xHC shall update each field to reflect the current value that it is using.

Figure 6-4: Stream Context Data Structure

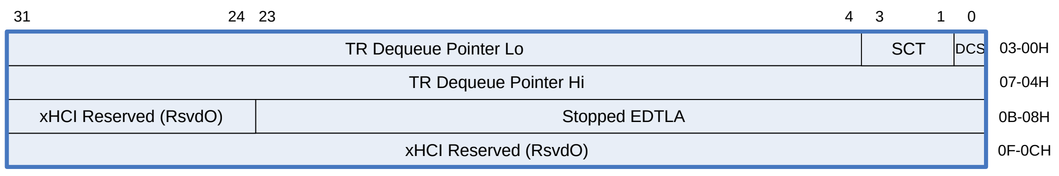


Table 6-13: Offset 00h and 04h – Stream Context Field Definitions

Bits	Description
0	**Dequeue Cycle State (DCS)**. This bit identifies the value of the xHC *Consumer Cycle State* (CCS) flag for the TRB referenced by the *TR Dequeue Pointer*. Refer to section 4.9.2 for more information.
3:1	**Stream Context Type (SCT)**. This field identifies whether the Stream Context is a member of a Primary or Secondary Stream Context Array, if the *TR Dequeue Pointer* field references a Transfer Ring or a Stream Context Array, and if a Stream Context Array is referenced, the size of the array. Value Stream Array Type / Dequeue Ptr / Secondary Stream Array Size 0 Secondary Transfer Ring N/A 1 Primary Transfer Ring N/A 2 Primary SSA 8 3 Primary SSA 16 4 Primary SSA 32 5 Primary SSA 64 6 Primary SSA 128 7 Primary SSA 256 Refer to section 4.12.2.1 for more information.
63:4	**TR Dequeue Pointer.** This field represents the high order bits of the 64-bit base address of the TRB ring or Stream Context Array associated with this Stream. The memory structure referenced by this physical memory pointer shall be aligned to a 16-byte boundary. This field is initialized by software and shall be overwritten by the xHC to save the value of the Dequeue Pointer when the endpoint enters the *Halted* or *Stopped* states. The value of the this field shall be undefined when the endpoint is not in the *Halted* or *Stopped* states.

Table 6-14: Offset 08h and 0Ch – Stream Context Field Definitions

Bits	Description
23:0	**Stopped EDTLA**. If the *Stopped EDTLA Capability* (SEC) field in the CCSPARAMS register = ‘1’, then this field shall identify the value of the EDTLA when the Stream is in the Stopped State. If *SEC* = ‘0’, then this field shall be *RsvdO*. Refer to sections 4.6.9, 4.12, and 5.3.6 for more information.
63:24	**RsvdO**.

Note: The *Context Size* (CSZ) field in the HCCPARAMS1 register does *not* apply to Stream Context data structures, they are always 16 bytes in size.

Note: A "valid" *Stream Context* requires:

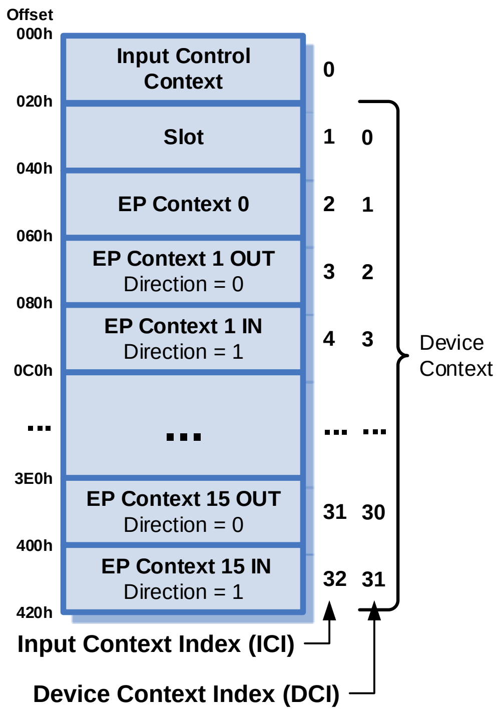
- The *TR Dequeue Pointer* is '0', i.e. no Transfer Ring or Stream Context is assigned yet.
- The *TR Dequeue Pointer* points to a valid Transfer Ring and the DCS flag represents the Cycle State of the segment referenced by the TR Dequeue Pointer.
- The *TR Dequeue Pointer* points to a valid Stream Array.

6.2.5 Input Context

The *Input Context* data structure specifies the endpoints and the operations to be performed on those endpoints by the *Address Device*, *Configure Endpoint*, and *Evaluate Context Commands*. Refer to section 4.6 for more information on these commands.

The *Input Context* is pointed to by an *Input Context Pointer* field of a *Address Device*, *Configure Endpoint*, and *Evaluate Context Command* TRBs. The *Input Context* is an array of up to 33 context data structure entries.

Figure 6-5: Input Context



The first entry (offset 000h) of the *Input Context* shall be the *Input Control Context* data structure. The remaining entries shall be organized identically to the *Device Context* data structures. Refer to section 6.2.5.1 for the definition of the *Input Control Context* data structure. Refer to section 6.2 for the definition of the *Device Context* and its data structures.

If the *Add Context* flag is set for an entry in the *Input Context*, then the entry shall be initialized appropriately by software. All other entries of the *Input Context* are ignored by the xHC. The *Add Context* and *Drop Context* flag indices are calculated identically to the *Device Context Index* (DCI) described in section 4.5.1 for the Device Context portion of the Input Context. e.g. EP context 1 OUT maps to D2 and A2, and so on, up to EP 15 IN mapping to D31 and A31.

Note: Figure 6-5 illustrates offsets with 32 byte *Input Control Context* data structures. i.e. the *Context Size* (CSZ) field in the HCCPARAMS1 register = '0'. If the *Context Size* (CSZ) field = '1' then the *Input Control Context* data structures consume 64 bytes each. The offsets shall be 040h for the Slot Context, 080h for EP Context 0, and so on.

Note: The *Input Context* shall be physically contiguous within a page.

6.2.5.1 Input Control Context

The *Input Control Context* data structure defines which Device Context data structures are affected by a command and the operations to be performed on those contexts.

Figure 6-6: Input Control Context

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	RsvdZ		03-00H
A31	A30	A29	A28	A27	A26	A25	A24	A23	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	07-04H
RsvdZ																																0B-08H
RsvdZ																																0F-0CH
RsvdZ																																13-10H
RsvdZ																																17-14H
RsvdZ																																1B-18H
RsvdZ								Alternate Setting								Interface Number								Configuration Value								1F-1CH

Note: Figure 6-6 illustrates a 32 byte *Input Control Context* data structure. i.e. the *Context Size* (CSZ) field in the HCCPARAMS1 register = '0'. If the *Context Size* (CSZ) field = '1' then the *Input Control Context* data structure consumes 64 bytes, where bytes 32 to 63 are RsvdZ.

Table 6-15: Offset 00h – Input Control Context Field Definitions

Bits	Description
1:0	**RsvdZ**.

31:2	**Drop Context flags (D2 - D31).** These single bit fields identify which Device Context data structures should be disabled by command. If set to '1', the respective Endpoint Context shall be disabled. If cleared to '0', the Endpoint Context is ignored.

Table 6-16: Offset 04h – Input Control Context Field Definitions

Bits	Description
31:0	**Add Context flags (A0 - A31).** These single bit fields identify which Device Context data structures shall be evaluated and/or enabled by a command. If set to '1', the respective Context shall be evaluated. If cleared to '0', the Context is ignored.

Table 6-17: Offset 1Ch – Input Control Context Field Definitions

Bits	Description
7:0	**Configuration Value.** If *CIC* = '1', *CIE* = '1', and this Input Context is associated with a Configure Endpoint Command, then this field contains the value of the Standard Configuration Descriptor bConfigurationValue field associated with the command, otherwise the this field shall be cleared to '0'.
15:8	**Interface Number.** If *CIC* = '1', *CIE* = '1', this Input Context is associated with a Configure Endpoint Command, and the command was issued due to a SET_INTERFACE request, then this field contains the value of the Standard Interface Descriptor bInterfaceNumber field associated with the command, otherwise the this field shall be cleared to '0'.
23:16	**Alternate Setting.** If *CIC* = '1', *CIE* = '1', this Input Context is associated with a Configure Endpoint Command, and the command was issued due to a SET_INTERFACE request, then this field contains the value of the Standard Interface Descriptor bAlternateSetting field associated with the command, otherwise the this field shall be cleared to '0'.
31:24	**RsvdZ**.

Note: The specific operations to be performed on a context by a command as a function of the *Drop Context* and *Add Context* flag settings are defined in detail in section 4.6.

Note: The fields in this data structure shall not be modified by software from the time the command is placed on the Command Ring until the associated Command Completion Event is received.

Note: The *Add Context* and *Delete Context* flag indices are calculated identically to the *Device Context Index* (DCI) described in section 4.5.1 for the Device Context portion of the Input Context. e.g. EP context 1 OUT maps to D2 and A2, and so on, up to EP 15 IN mapping to D31 and A31.

The *Add Context* and *Delete Context* flag indices relative to *Input Context* are calculated as follows:

The *Input Context Index* (ICI) (refer to Figure 6-5) of the *Input Control Context* is 0.
The ICI of the *Slot Context* is 1.
For the remaining Input Context indices 2-31, the following rules apply:

1) For Isoch, Interrupt, or Bulk type endpoints the ICI is calculated using the *Endpoint Number* and *Direction* with the following formula;

ICI = ((Endpoint Number * 2) + 1 + Direction,
where Direction = ‘0’ for OUT endpoints and ‘1’ for IN endpoints.

2) For Control type endpoints, including the Default Control Endpoint:

ICI = (Endpoint Number + 1) * 2.

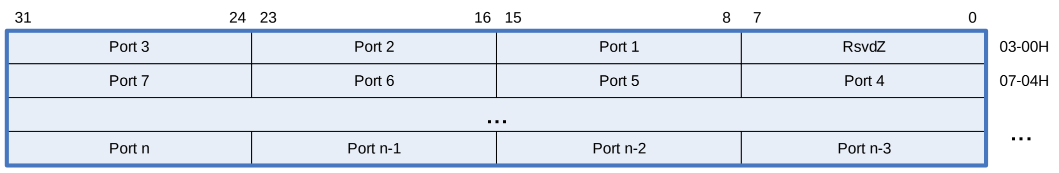
Note: The extended Configuration Information fields *Configuration Value*, *Interface Number*, and *Alternate Setting* shall be initialized by software if the *Configuration Information Enable* (CIE) flag is set to ‘1’. Support for the extended Configuration Information fields is required by all 1.1 compliant xHCI drivers.

Note: If the *Configuration Information Capability* of an xHC is enabled (*CIE* = ‘1’) the system software shall ensure that each Configure Endpoint Command to the xHCI represents the endpoint changes due to exactly one SET_CONFIGURATION or SET_ALTERNATIVE)_INTERFACE request to a USB device.

6.2.6 Port Bandwidth Context

The *Port Bandwidth Context* data structure is used to provide system software with the percentage of periodic bandwidth available on each Root Hub Port, at the Speed indicated by the Device Speed field of the *Get Port Bandwidth Command*. Software allocates the Context data structure and the xHC updates it during the execution of a *Get Port Bandwidth Command*. Refer to section 4.6.15 for more information.

Figure 6-7: Port Bandwidth Context



Note: Figure 6-7 illustrates a generic *Port Bandwidth Context* data structure. System sizes this data structure as a function of the number of Root Hub ports supported by the xHC (i.e. MaxPorts). Software shall round up the size of the buffer to the nearest 8-byte boundary.

Table 6-18: Offset 00h – Port Bandwidth Context Field Definitions

Bits	Description
7:0	**RsvdZ**.
15:8	**Port 1 Bandwidth (Port 1).** Percentage of Total Available Bandwidth available on Port 1.
23:16	**Port 2 Bandwidth (Port 2).** Percentage of Total Available Bandwidth on Port 2.
31:24	**Port 3 Bandwidth (Port 3).** Percentage of Total Available Bandwidth on Port 3.

Table 6-19: Offset n-03h – Port Bandwidth Context Field Definitions

Bits	Description
7:0	**Port n-3 Bandwidth (Port n-3).** Percentage of Total Available Bandwidth on Port n-3.
15:8	**Port n-2 Bandwidth (Port n-2).** Percentage of Total Available Bandwidth on Port n-2.
23:16	**Port n-1 Bandwidth (Port n-1).** Percentage of Total Available Bandwidth on Port n-1.
31:24	**Port n Bandwidth (Port n).** Percentage of Total Available Bandwidth on Port n.

Note: Refer to section 4.14 for the definition of “Total Available Bandwidth”.

Note: The range of valid values depends on the value of the Dev Speed field in the Get Port Bandwidth Command. 0 to 80% for HS, and 0 to 90% for SS and FS. Refer to section 4.14.2 for more information.

Note: The *Port* fields of the *Port Bandwidth Context* shall report decimal percentage values in hex, i.e. 0Ah = 10%, 50h = 80%, etc.

6.3 TRB Ring

A TRB Ring is an array of TRB (Transfer Request Block) structures, which is used by the xHCI as a circular queue to communicate with the host. Refer to section 4.9 for a detailed description of Ring operation.

6.4 Transfer Request Block (TRB)

The Transfer Request Block is the basic building block upon which all xHC USB transfers are constructed. All Transfer Request Blocks shall be aligned on a 16-byte boundary.

Each TRB has the basic format described in section 4.11.1. TRBs are used for all transactions performed by an xHC, which includes commands sent to the host controller, events generated by the host controller, and transactions associated with USB endpoints.

Note: Vendor defined TRBs are shall support the *TRB Type* and *Cycle bit* fields.

6.4.1 Transfer TRBs

A Transfer TRB shall be found on a **Transfer Ring**. A Work Item on a Transfer Ring is called a **Transfer Descriptor** (TD) and is comprised of one or more Transfer TRB data structures. This section describes the transfer related TRBs.

Note: If a zero-length transfer is specified, the *Data Buffer Pointer* field is ignored by the xHC, irrespective of the state of the *IDT* flag.

Note: Data buffers referenced by Transfer TRBs shall not span 64KB boundaries. If a physical data buffer spans a 64KB boundary, software shall chain multiple TRBs to describe the buffer.

6.4.1.1 Normal TRB

A *Normal TRB* is used in several ways; exclusively on Bulk and Interrupt Transfer Rings for normal and Scatter/Gather operations, to define additional data buffers for Fine and Coarse Grain Scatter/Gather operations on Isoch Transfer Rings, and to define the Data stage information for Control Transfer Rings. Refer to section 4.11.2.1 for information on the use of *Normal TRBs*. Refer to section 3.2.8 for an overview of xHCI scatter/gather support.

Figure 6-8: Normal TRB

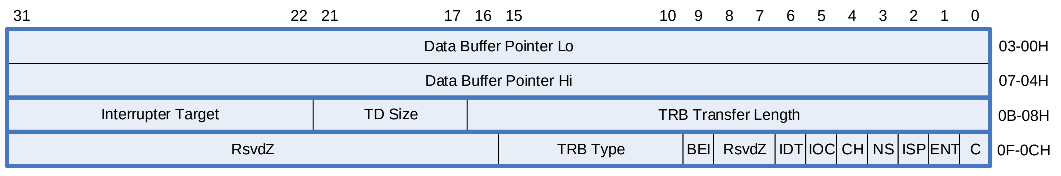


Table 6-20: Offset 00h and 04h – Normal TRB Field Definitions

Bits	Description
63:0	**Data Buffer Pointer Hi and Lo**. These fields represent the 64-bit address of the TRB data area for this transaction or 8 bytes of immediate data. The *Immediate Data* (IDT) control flag selects this option for each Normal TRB. The memory structure referenced by this physical memory pointer is allowed to begin on a byte address boundary. However, user may find other alignments, such as 64-byte or 128-byte alignments, to be more efficient and provide better performance.

Table 6-21: Offset 08h – Normal TRB Field Definitions

Bits	Description
16:0	**TRB Transfer Length.** For an OUT, this field defines the number of data bytes the xHC shall send during the execution of this TRB. If the value of this field is '0' when the xHC fetches this TRB, the xHC shall execute a zero-length transaction. Note: If a zero-length transfer is specified, the *Data Buffer Pointer* field is ignored by the xHC, irrespective of the state of the *IDT* flag. Refer to section 4.9.1 for more information on zero-length Transfer TRB handling. For an IN, the value of the field identifies the size of the data buffer referenced by the Data Buffer Pointer, i.e. the number of bytes the host expects the endpoint to deliver. Valid values are 0 to 64K.
21:17	**TD Size**. This field provides an indicator of the number of packets remaining in the TD. Refer to section 4.10.2.4 for how this value is calculated.
31:22	**Interrupter Target**. This field defines the index of the Interrupter that will receive events generated by this TRB. Valid values are between 0 and *MaxIntrs*-1.

Table 6-22: Offset 0Ch – Normal TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of the Transfer ring.
1	**Evaluate Next TRB (ENT)**. If this flag is '1' the xHC shall fetch and evaluate the next TRB before saving the endpoint state. Refer to section 4.12.3 for more information.

2	**Interrupt-on Short Packet (ISP).** If this flag is '1' and a *Short Packet* is encountered for this TRB (i.e., less than the amount specified in *TRB Transfer Length*), then a Transfer Event TRB shall be generated with its Completion Code set to *Short Packet*. The *TRB Transfer Length* field in the Transfer Event TRB shall reflect the residual number of bytes not transferred into the associated data buffer. In either case, when a *Short Packet* is encountered, the TRB shall be retired without error and the xHC shall advance to the next Transfer Descriptor (TD). Note that if the ISP and IOC flags are both '1' and a Short Packet is detected, then only one Transfer Event TRB shall be queued to the Event Ring. Also refer to section 4.10.1.1.
3	**No Snoop (NS).** When set to '1', the xHC is permitted to set the No Snoop bit in the Requester Attributes of the PCIe transactions it initiates if the PCIe configuration Enable No Snoop flag is also set. When cleared to '0', the xHC is not permitted to set PCIe packet No Snoop Requester Attribute. Refer to section 4.18.1 for more information. NOTE: If software sets this bit, then it is responsible for maintaining cache consistency.
4	**Chain bit (CH).** Set to '1' by software to associate this TRB with the next TRB on the Ring. A Transfer Descriptor (TD) is defined as one or more TRBs. The *Chain bit* is used to identify the TRBs that comprise a TD. The *Chain bit* is always '0' in the last TRB of a TD.
5	**Interrupt On Completion (IOC).** If this bit is set to '1', it specifies that when this TRB completes, the Host Controller shall notify the system of the completion by placing an Transfer Event TRB on the Event ring and asserting an interrupt to the host at the next interrupt threshold. Note that the interrupt assertion may be blocked for the Transfer Event by *BEI*. Refer to sections 4.10.4 and 4.17.5.
6	**Immediate Data (IDT).** If this bit is set to '1', it specifies that the *Data Buffer Pointer* field of this TRB contains data, not a pointer, and the *Length* field shall contain a value between '0' and '8' to indicate the number of valid bytes from offset 0 in the TRB that should be used as data. Note: If the IDT flag is set in one Transfer TRB of a TD, then it shall be the only Transfer TRB of the TD. An Event Data TRB may be included in the TD. Failure to follow this rule may result in undefined xHC operation. Note: *IDT* shall not be set ('1') for TRBs on endpoints that define a *Max Packet Size* < 8 bytes, or on IN endpoints.
8:7	**RsvdZ**.
9	**Block Event Interrupt (BEI).** If this bit is set to '1' and *IOC* = '1', then the Transfer Event generated by *IOC* shall *not* assert an interrupt to the host at the next interrupt threshold. Refer to section 4.17.5.
15:10	**TRB Type.** This shall be set to *Normal TRB* type. Refer to Table 6-86 for the definition of the valid Transfer TRB type IDs.
31:16	**RsvdZ**.

6.4.1.2 Control TRBs

Control transfers require two or three TDs to define them: a *Setup Stage TD* followed by an *Status Stage TD*, if a data stage is required for the transfer an optional *Data Stage TD* will reside between the Setup Stage and Status Stage TDs. This sections defines the TRBs that comprise the respective TDs. Refer to section 4.11.2.2 for more information on xHCI control transfers.

Note: The IOC flag should only be set in the *Status Stage TRB* of a Control transfer.

6.4.1.2.1 Setup Stage TRB

A *Setup Stage TRB* is created by system software to initiate a USB Setup packet on a control endpoint. Refer to section 3.2.9 for more information on Setup Stage TRBs and the operation of control endpoints. Also refer to section 8.5.3 in the USB2 spec. for a description of "Control Transfers".

Figure 6-9: Setup Stage TRB

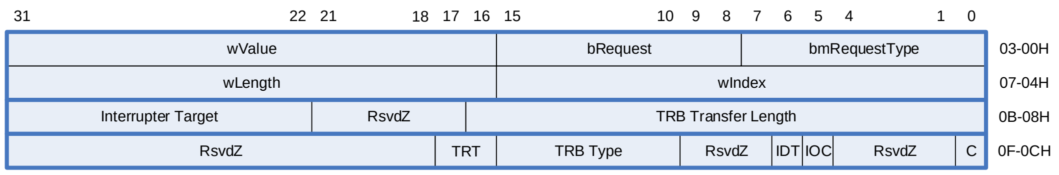


Table 6-23: Offset 00h – Setup Stage TRB Field Definitions

Bits	Description
7:0	**bmRequestType.** Refer to Table 9-2 "Format of Setup Data" in the USB2 or USB3 specification.
15:8	**bRequest.** Refer to Table 9-2 "Format of Setup Data" in the USB2 or USB3 specification.
31:16	**wValue.** Refer to Table 9-2 "Format of Setup Data" in the USB2 or USB3 specification.

Table 6-24: Offset 04h – Setup Stage TRB Field Definitions

Bits	Description
15:0	**wIndex.** Refer to Table 9-2 "Format of Setup Data" in the USB2 or USB3 specification.
31:16	**wLength.** Refer to Table 9-2 "Format of Setup Data" in the USB2 or USB3 specification.

Table 6-25: Offset 08h – Setup Stage TRB Field Definitions

Bits	Description
16:0	**TRB Transfer Length.** Always 8.
21:17	**RsvdZ**.
31:22	**Interrupter Target**. This field defines the index of the Interrupter that will receive events generated by this TRB. Valid values are between 0 and *MaxIntrs*-1.

Table 6-26: Offset 0Ch – Setup Stage TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue point of a Transfer ring.
4:1	**RsvdZ**.
5	**Interrupt On Completion (IOC).** If this bit is set to '1', it specifies that when this TRB completes, the Host Controller shall notify the system of the completion by placing an Event TRB on the Event ring and sending an interrupt at the next interrupt threshold. Refer to section 4.10.4.
6	**Immediate Data (IDT).** This bit shall be set to '1' in a Setup Stage TRB. It specifies that the Parameter component of this TRB contains Setup Data.
9:7	**RsvdZ**.
15:10	**TRB Type.** This field is set to *Setup Stage TRB* type. Refer to Table 6-86 for the definition of the Type TRB IDs.
17:16	**Transfer Type (TRT).** This field indicates the type and direction of the control transfer. Value Definition 0 No Data Stage 1 Reserved 2 OUT Data Stage 3 IN Data Stage Refer to section 4.11.2.2 for more information on the use of *TRT*.
31:18	**RsvdZ**.

6.4.1.2.2 Data Stage TRB

A *Data Stage TRB* is used generate the Data stage transaction of a USB Control transfer. Refer to section 3.2.9 for more information on Control transfers and the operation of control endpoints. Also refer to section 8.5.3 in the USB2 spec. for a description of "Control Transfers".

Figure 6-10: Data Stage TRB

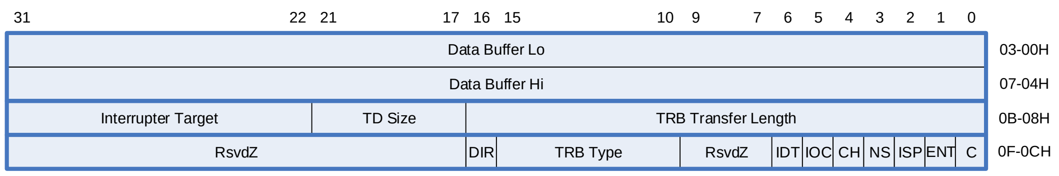


Table 6-27: Offset 00h and 04h – Data Stage TRB Field Definitions

Bits	Description
63:0	**Data Buffer Pointer Hi and Lo.** These fields represent the 64-bit address of the Data buffer area for this transaction The memory structure referenced by this physical memory pointer is allowed to begin on a byte address boundary. However, user may find other alignments, such as 64-byte or 128-byte alignments, to be more efficient and provide better performance.

Table 6-28: Offset 08h – Data Stage TRB Field Definitions

Bits	Description
16:0	**TRB Transfer Length.** For an OUT, this field is the number of data bytes the xHC will send during the execution of this TRB. For an IN, the initial value of the field identifies the size of the data buffer referenced by the Data Buffer Pointer, i.e. the number of bytes the host expects the endpoint to deliver. Valid values are 1 to 64K.
21:17	**TD Size**. This field provides an indicator of the number of packets remaining in the TD. Refer to section 4.11.2.4 for how this value is calculated.
31:22	**Interrupter Target**. This field defines the index of the Interrupter that will receive events generated by this TRB. Valid values are between 0 and *MaxIntrs*-1.

Table 6-29: Offset 0Ch – Data Stage TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of the Transfer ring.
1	**Evaluate Next TRB (ENT)**. If this flag is '1' the xHC shall fetch and evaluate the next TRB before saving the endpoint state. Refer to section 4.12.3 for more information.
2	**Interrupt-on Short Packet (ISP).** If this flag is '1' and a *Short Packet* is encountered for this TRB (i.e., less than the amount specified in *TRB Transfer Length*), then a Transfer Event TRB shall be generated with its Completion Code set to *Short Packet*. The *TRB Transfer Length* field in the Transfer Event TRB shall reflect the residual number of bytes not transferred into the associated data buffer. In either case, when a *Short Packet* is encountered, the TRB shall be retired without error and the xHC shall advance to the Status Stage TD. Note: if the ISP and IOC flags are both '1' and a Short Packet is detected, then only one Transfer Event TRB shall be queued to the Event Ring. Also refer to section 4.10.1.1.
3	**No Snoop (NS).** When set to '1', the xHC is permitted to set the No Snoop bit in the Requester Attributes of the PCIe transactions it initiates if the PCIe configuration Enable No Snoop flag is also set. When cleared to '0', the xHC is not permitted to set PCIe packet No Snoop Requester Attribute. Refer to section 4.18.1 for more information. NOTE: If software sets this bit, then it is responsible for maintaining cache consistency.
4	**Chain bit (CH).** Set to '1' by software to associate this TRB with the next TRB on the Ring. A Data Stage TD is defined as a Data Stage TRB followed by zero or more Normal TRBs. The *Chain bit* is used to identify a multi-TRB Data Stage TD. The *Chain bit* is always '0' in the last TRB of a Data Stage TD.
5	**Interrupt On Completion (IOC).** If this bit is set to '1', it specifies that when this TRB completes, the Host Controller shall notify the system of the completion by placing an Event TRB on the Event ring and asserting an interrupt to the host at the next interrupt threshold. Refer to section 4.10.4.
6	**Immediate Data (IDT).** If this bit is set to '1', it specifies that the *Data Buffer Pointer* field of this TRB contains data, not a pointer. If *IDT* = '1', the *Length* field shall contain a value between 1 and 8 to indicate the number of valid bytes from offset 0 in the TRB that should be used as data. Note: If the *IDT* flag is set in one Data Stage TRB of a TD, then it shall be the only Transfer TRB of the TD. An Event Data TRB may also be included in the TD. Failure to follow this rule may result in undefined xHC operation.
9:7	**RsvdZ**.
15:10	**TRB Type.** This shall be set to *Data Stage TRB* type. Refer to Table 6-86 for the definition of the valid Transfer TRB type IDs.

16	**Direction (DIR).** This bit indicates the direction of the data transfer as defined in the *Data State TRB* Direction column of Table 7. If cleared to '0', the data stage transfer direction is OUT (Write Data). If set to '1', the data stage transfer direction is IN (Read Data). Refer to section 4.11.2.2 for more information on the use of *DIR*.
31:17	**RsvdZ**.

6.4.1.2.3 Status Stage TRB

A *Status Stage TRB* is used to generate the Status stage transaction of a USB Control transfer. Refer to section 3.2.9 for more information on Control transfers and the operation of control endpoints.

Figure 6-11: Status Stage TRB

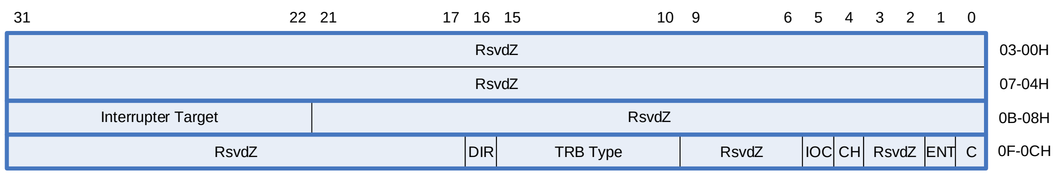


Table 6-30: Offset 08h – Status Stage TRB Field Definitions

Bits	Description
21:0	**RsvdZ**.
31:22	**Interrupter Target**. This field defines the index of the Interrupter that will receive events generated by this TRB. Valid values are between 0 and *MaxIntrs*-1.

Table 6-31: Offset 0Ch – Status Stage TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of the Transfer ring.
1	**Evaluate Next TRB (ENT)**. If this flag is '1' the xHC shall fetch and evaluate the next TRB before saving the endpoint state. Refer to section 4.12.3 for more information.

3:2	**RsvdZ**.
4	**Chain bit (CH).** Set to ‘1’ by software to associate this TRB with the next TRB on the Ring. A Status Stage TD is defined as a Status Stage TRB followed by zero or one Event Data TRB. The *Chain bit* is used to identify a multi-TRB Status Stage TD. The *Chain bit* is always ‘0’ in the last TRB of a Status Stage TD.
5	**Interrupt On Completion (IOC)**. If this bit is set to ‘1’, it specifies that when this TRB completes, the Host Controller shall notify the system of the completion by placing an Event TRB on the Event ring and asserting an interrupt to the host at the next interrupt threshold. Refer to section 4.10.4.
9:6	**RsvdZ**.
15:10	**TRB Type**. This field shall be set to *Status Stage TRB* type. Refer to Table 6-86 for the definition of the valid Transfer TRB type IDs.
16	**Direction (DIR)**. This bit indicates the direction of the data transfer as defined in the *Status State TRB* Direction column of Table 7. If cleared to ‘0’, the status stage transfer direction is OUT (Host-to-device). If set to ‘1’, the status stage transfer direction is IN (Device-to-host). Refer to section 4.11.2.2 for more information on the use of *DIR*.
31:17	**RsvdZ**.

A Transfer Event generated by this TRB shall reflect the status state response from the USB device.

6.4.1.3 Isoch TRB

An *Isoch TRB* defines isochronous data transfers. Refer to section 3.2.11 for more information on *Isoch TRBs* and the operation of isochronous endpoints.

Figure 6-12: Isoch TRB

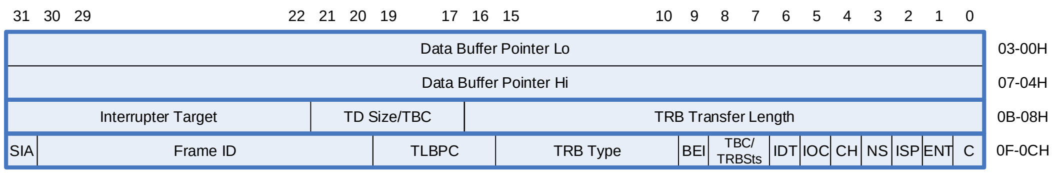


Table 6-32: Offset 00h and 04h – Isoch TRB Field Definitions

Bits	Description
63:0	**Data Buffer Pointer Hi and Lo.** This field represents the 64-bit address of the TRB data area for this transaction or 8 bytes of immediate data. The *Immediate Data* (IDT) control flag selects this option for each Isoch TRB. The memory structure referenced by this physical memory pointer is allowed to begin on a byte address boundary. However, user may find other alignments, such as 64-byte or 128-byte alignments, to be more efficient and provide better performance.

Table 6-33: Offset 08h – Isoch TRB Field Definitions

Bits	Description
16:0	**TRB Transfer Length.** For an OUT, this field is the number of data bytes the host controller will send during the execution of this TRB. For an IN, the initial value of the field is the number of bytes the host expects the endpoint to deliver, i.e. the number of bytes the host expects the endpoint to deliver. Refer to section 4.9.1 for more information on zero-length Transfer TRB handling. Valid values are 0 to 64K.
21:17	**TD Size/TBC**. If *ETE* = '0', then this field defines the *TD Size*, which provides an indicator of the number of bytes remaining in the TD. Refer to section 4.11.2.4 for how this value is calculated. If *ETE* = '1', then this field defines the **Transfer Burst Count (TBC)**, which identifies the number of bursts - 1 that shall be required to move this Isoch TD. All bursts except the last shall transfer *Max Burst Size* packets. The last burst shall transfer *TLBPC* + 1 packets. Refer to section 4.11.2.3 for more information.
31:22	**Interrupter Target**. This field defines the index of the Interrupter that will receive events generated by this TRB. Valid values are between 0 and *MaxIntrs*-1.

Table 6-34: Offset 0Ch – Isoch TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue point of a Transfer ring.
1	**Evaluate Next TRB (ENT)**. If this flag is '1' the xHC shall fetch and evaluate the next TRB before saving the endpoint state. Refer to section 4.12.3 for more information.

2	**Interrupt-on Short Packet (ISP).** If this flag is '1' and a *Short Packet* is encountered for this TRB (i.e., less than the amount specified in *TRB Transfer Length*), then a Transfer Event TRB shall be generated with the with its Completion Status set to *Short Packet*. In either case when a *Short Packet* is encountered, the TRB shall be retired without error and the xHC shall advance to the next Transfer Descriptor (TD). Also refer to section 4.10.1.1. Note: if the ISP and IOC flags are both '1' and a *Short Packet* is detected, then only one Transfer Event TRB shall be queued to the Event Ring.
3	**No Snoop (NS).** When set to '1', the xHC is permitted to set the No Snoop bit in the Requester Attributes of the PCIe transactions it initiates if the PCIe configuration Enable No Snoop flag is also set. When cleared to '0', the xHC is not permitted to set PCIe packet No Snoop Requester Attribute. Refer to section 4.18.1 for more information. NOTE: If software sets this bit, then it is responsible for maintaining cache consistency.
4	**Chain bit (CH).** Set to '1' by software to associate this TRB with the next TRB on the Ring. An Isoch Transfer Descriptor is defined as an Isoch TRB followed by zero or more Normal TRBs. The *Chain bit* is used to identify the TRBs that comprise the TD. The *Chain bit* is always '0' in the last TRB of an Isoch TD.
5	**Interrupt On Completion (IOC).** If this bit is set to '1', it specifies that when this TRB completes, the Host Controller shall notify the system of the completion by placing an Event TRB on the Event ring and sending an interrupt at the next interrupt threshold. Refer to section 4.10.4.
6	**Immediate Data (IDT).** If this bit is set to '1', it specifies that the *Data Buffer Pointer* field of this TRB contains data, not a pointer, and the *Length* field shall contain a value between '0' and '8' to indicate the number of valid bytes from offset 0 in the TRB that should be used as data. Note: If the *IDT* flag is set in one Transfer TRB of a TD, then it shall be the only Transfer TRB the TD. An Event Data, Link TRB may also be included in the TD. Failure to follow this rule may result in undefined xHC operation. Note: The *IDT* flag shall not be set ('1') for TRBs on endpoints that define a Max Packet Size < 8 bytes, or on IN endpoints.
8:7	**Transfer Burst Count (TBC/TRBSts).** If *ETE* = '0', then this field identifies number of bursts - 1 that shall be required to move this Isoch TD. All bursts except the last shall transfer *Max Burst Size* packets. The last burst shall transfer *TLBPC* + 1 packets. Refer to section 4.11.2.3 for more information. If *ETE* = '1' and ETC_TSC='1' and ETC='1', then this field is set to 1 to explicitly indicate that it is the last Transfer TRB of the TD. Other values for TRBSts are reserved. If ETE='1' and ETC_TSC='0' and ETC='1', then this field shall be *RsvdZ*.
9	**Block Event Interrupt (BEI).** If this bit is set to '1' and *IOC* = '1', then the Transfer Event generated by *IOC* shall *not* assert an interrupt to the host at the next interrupt threshold. Refer to section 4.17.5.
15:10	**TRB Type.** This field is set to *Isoch TRB* type. Refer to Table 6-86 for the definition of the Type TRB IDs.

19:16	**Transfer Last Burst Packet Count (TLBPC).** This field indent if i es number of packets -1 that shall be in the last burst of this Isoch TD, e.g. '0' = 1 packet, '1' = 2 packets, etc. Refer to section 4.11.2.3 for more information.
30:20	**Frame ID**. The value in this field identifies the target 1ms. frame that the Interval associated with this Isochronous Transfer Descriptor will start on. Bits [13:3] of the *Microframe Index* field of the MFINDEX register may be used to determine the current periodic frame. This field is ignored by the xHC if the *Start Isoch ASAP* flag is set ('1'). For more information on the programming of this field refer to section 4.11.2.5.
31	**Start Isoch ASAP (SIA)**. If this flag is set ('1'), the Frame ID is ignored and the Isoch TD is scheduled as soon as possible. If this flag is cleared ('0'), the Frame ID is valid and the Isoch TD is scheduled the next time there is a match between the Frame ID and the Frame Index portion (bits 13:3) of the Microframe Index (MFINDEX) register. Refer to Figure 4-21. For more information refer to section 4.11.2.3.

6.4.1.4 No Op TRB

The *No Op TRB* provides a simple means for verifying the operation of the basic Transfer Ring mechanisms offered by the xHCI. It may be inserted on a Transfer Ring to generate a *Transfer Event*.

Note: Consecutive *No Op TRBs* may impact xHC performance and should be avoided by software. Refer to section 4.11.7 for more information on *No Op TRB* placement rules.

Figure 6-13: No Op TRB

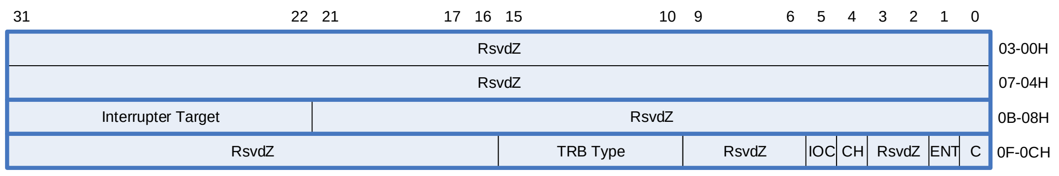


Table 6-35: Offset 08h – No Op TRB Field Definitions

Bits	Description
21:0	**RsvdZ**.
31:22	**Interrupter Target**. This field defines the index of the Interrupter that will receive Transfer Events generated by this TRB. Valid values are between 0 and *MaxIntrs*-1.

Table 6-36: Offset 0Ch – No Op TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Transfer Ring.
1	**Evaluate Next TRB (ENT)**. If this flag is '1' the xHC shall fetch and evaluate the next TRB before saving the endpoint state. Refer to section 4.12.3 for more information.
3:2	**RsvdZ**.
4	**Chain bit (CH).** Set to '1' by software to associate this TRB with the next TRB on the Ring. A Transfer Descriptor (TD) is defined as one or more TRBs. The *Chain bit* is used to identify the TRBs that comprise a TD. The *Chain bit* is always '0' in the last TRB of a TD.
5	**Interrupt On Completion (IOC).** If this bit is set to '1', it specifies that when this TRB completes, the Host Controller shall notify the system of the completion by placing a Transfer Event TRB on the Event ring and sending an interrupt at the next interrupt threshold. Refer to section 4.10.4.
9:6	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the No Op TRB type ID.
31:16	**RsvdZ**.

6.4.2 Event TRBs

Event TRBs shall be found on an **Event Ring**. A Work Item on an Event Ring is called an **Event Descriptor** (ED). An ED shall be comprised of only one Event TRB data structure. This section describes the event related TRBs.

6.4.2.1 Transfer Event TRB

A *Transfer Event* provides the completion status associated with a Transfer TRB. Refer to section 4.11.3.1 for more information on the use and operation of *Transfer Events*.

Note: The Primary Event Ring (0) or a Secondary Event Ring may receive a Transfer Event TRB. Normally the xHC shall use the *Interrupter Target* field of the originating Transfer TRB to determine the Event Ring that shall receive this event. Refer to section 4.17.4 for the exception cases, which use the Slot Context *Interrupter Target* field.

Figure 6-14: Transfer Event TRB

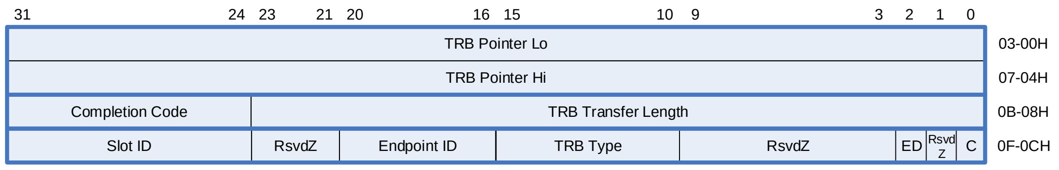


Table 6-37: Offset 00h and 04h – Transfer Event TRB Field Definitions

Bits	Description
63:0	**TRB Pointer Hi and Lo.** This field represents the 64-bit address of the TRB that generated this event or 64 bits of Event Data if the *ED* flag is '1'. If a TRB memory structure is referenced by this field (*ED* = '0'), then it shall be physical memory pointer aligned on a 16-byte boundary, i.e. bits 0 through 3 of the address are '0'.

Table 6-38: Offset 08h – Transfer Event TRB Field Definitions

Bits	Description
23:0	**TRB Transfer Length**. This field shall reflect the residual number of bytes not transferred. For an OUT, this field shall indicate the value of the *Length* field of the Transfer TRB, minus the data bytes that were successfully transmitted. A successful OUT transfer shall return a *Length* of '0'. For an IN, this field shall indicate the value of the *TRB Transfer Length* field of the Transfer TRB, minus the data bytes that were successfully received. If the device terminates the receive transfer with a Short Packet, then this field shall indicate the difference between the expected transfer size (defined by the Transfer TRB) and the actual number of bytes received. If the receive transfer completed with an error, then this field shall indicate the difference between the expected transfer size and the number of bytes successfully received. If the *Event Data* flag is '0' the legal range of values is 0 to 10000h. If the *Event Data* flag is '1' or the *Condition Code* is *Stopped - Short Packet*, then this field shall be set to the value of the *Event Data Transfer Length Accumulator* (EDTLA). Refer to section 4.11.5.2 for a description of EDTLA.
31:24	**Completion Code.** This field encodes the completion status that can be identified by a TRB. Refer to section 6.4.5 for an enumerated list of possible error conditions.

Table 6-39: Offset 0Ch – Transfer Event TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Dequeue Pointer of an Event Ring.
1	**RsvdZ**.
2	**Event Data (ED)**. When set to '1', the event was generated by an Event Data TRB and the Parameter Component (*TRB Pointer* field) contains a 64-bit value provided by the Event Data TRB. If cleared to '0', the Parameter Component (*TRB Pointer* field) contains a pointer to the TRB that generated this event. Refer to section 4.11.5.2 for more information.
9:3	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the *Transfer Event TRB* type ID.
20:16	**Endpoint ID**. The ID of the Endpoint that generated the event. This value is used as an index in the Device Context to select the Endpoint Context associated with this event.
23:21	**RsvdZ**.
31:24	**Slot ID.** The ID of the Device Slot that generated the event. This is value is used as an index in the *Device Context Base Address Array* to select the *Device Context* of the source device.

Note: For multi-TRB TDs, if *ED* = '0', the *TRB Transfer Length* only reflects the number of bytes transferred for the buffer associated with the Transfer TRB pointed to by the Transfer Event, *not* the total bytes transferred for the TD.

Note: A *Ring Overrun* or *Ring Underrun Event* utilizes a *Transfer Event TRB* to report the error. In this case, the *TRB Pointer* field is invalid.

Note: If an error occurs during the execution of a Transfer TRB that does not have its *IOC* or *ISP* flags set, a Transfer Event shall be generated for the error and the Transfer Event shall point to the offending TRB. Refer to sections 4.10.1 and 4.10.2 for more information on handling errors related to Transfer TRBs.

Note: *CStream* is not valid until a Streams endpoint transitions to the *Start Stream* state for the first time. A *Transfer Event* generated by a *Stop Endpoint Command* shall report '0' in the *TRB Pointer* and *TRB Length* fields if the command is executed and *CStream* is invalid. Refer to section 4.12.1.

6.4.2.2 Command Completion Event TRB

A *Command Completion Event TRB* shall be generated by the xHC when a command completes on the Command Ring. Refer to section 4.11.4 for more information on the use of *Command Completion Events*.

Note: The Primary Event Ring (0) shall receive all Command Completion Events.

Figure 6-15: Command Completion Event TRB

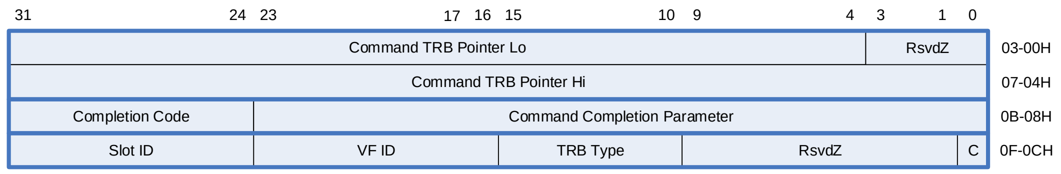


Table 6-40: Offset 00h and 04h – Command Completion Event TRB Field Definition

Bits	Description
3:0	**RsvdZ**.
63:4	**Command TRB Pointer Hi and Lo.** This field represents the high order bits of the 64-bit address of the Command TRB that generated this event. Note that this field is not valid for some *Completion Code* values. Refer to Table 6-85 for specific cases. The memory structure referenced by this physical memory pointer shall be aligned on a 16-byte address boundary.

Table 6-41: Offset 08h – Command Completion Event TRB Field Definitions

Bits	Description
23:0	**Command Completion Parameter**. This field may optionally be set by a command. Refer to section 4.6.6.1 for specific usage. If a command does not utilize this field it shall be treated as *RsvdZ*.
31:24	**Completion Code.** This field encodes the completion status of the command that generated the event. Refer to the respective command definition for a list of the possible Completion Codes associated with the command. Refer to section 6.4.5 for an enumerated list of possible error conditions.

Table 6-42: Offset 0Ch – Command Completion Event TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Dequeue Pointer of an Event Ring.
9:1	**RsvdZ**.

15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the *Command Completion Event TRB* type ID.
23:16	**VF ID.** The ID of the Virtual Function that generated the event. Note that this field is valid only if Virtual Functions are enabled. If they are not enabled this field shall be cleared to '0'.
31:24	**Slot ID.** The Slot ID field shall be updated by the xHC to reflect the slot associated with the command that generated the event, with the following exceptions: - The Slot ID shall be cleared to '0' for *No Op, Set Latency Tolerance Value, Get Port Bandwidth,* and *Force Event Commands*. - The Slot ID shall be set to the ID of the newly allocated Device Slot for the *Enable Slot Command.* - The value of Slot ID shall be vendor defined when generated by a vendor defined command. This value is used as an index in the Device Context Base Address Array to select the Device Context of the source device. If this Event is due to a Host Controller Command, then this field shall be cleared to '0'.

Note: All commands for a Device Slot or VF are executed in order.

Note: All Vendor Defined Event TRBs shall support the *Completion Code*, *Cycle bit*, and *TRB Type* fields. The remaining fields and reserved areas may be vendor defined/allocated.

6.4.2.3 Port Status Change Event TRB

A Port Status Change Event TRB shall be generated by the xHC any time there is a '0' to '1' transition of the Port Status Change Event Generation (PSCEG) variable, e.g. a status change bit transitions to a non-zero value (CSC, PEC, OCC, etc.). Refer to section 4.19.2 for more information on the use and generation of the Port Status Change Event. Refer to section 5.4.8 for more information on the port status change bits.

Note: The Primary Event Ring (0) shall receive all Port Status Change Events.

Figure 6-16: Port Status Change Event TRB

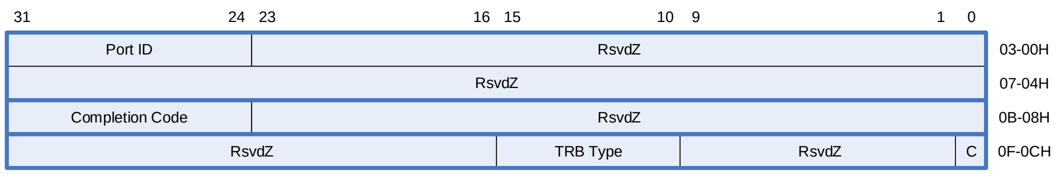


Table 6-43: Offset 00h – Port Status Change Event TRB Field Definitions

Bits	Description
23:0	**RsvdZ**.
31:24	**Port ID.** The Port Number of the Root Hub Port that generated this event.

Table 6-44: Offset 08h – Port Status Change Event TRB Field Definitions

Bits	Description
23:0	**RsvdZ**.
31:24	**Completion Code.** This field encodes the completion status that can be identified by a TRB. The *Completion Code* field shall be set to *Success*.

Table 6-45: Offset 0Ch – Port Status Change Event TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Dequeue Pointer of an Event Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Port Status Change Event TRB type ID.
31:16	**RsvdZ**.

6.4.2.4 Bandwidth Request Event TRB

A *Bandwidth Event TRB* shall be generated by the xHC when the Negotiate Bandwidth Command is received. Refer to section 4.6.13 for more information on Bandwidth Request Events.

Note: The Primary Event Ring (0) or a Secondary Event Ring may receive a Bandwidth Request Event TRB. The xHC shall use the *Interrupter Target* field of the Slot

Context indexed by the Bandwidth Request Event TRB *Slot ID* field to determine the Event Ring that shall receive the event.

Figure 6-17: Bandwidth Request Event TRB

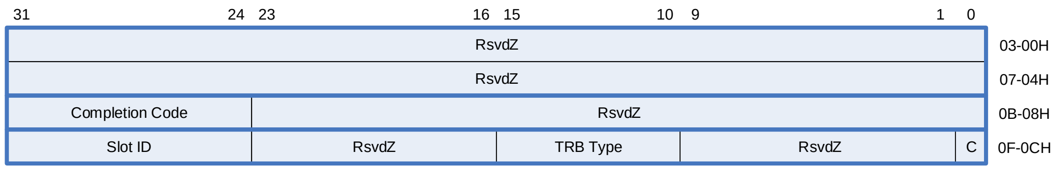


Table 6-46: Offset 08h – Bandwidth Request Event TRB Field Definitions

Bits	Description
23:0	**RsvdZ**.
31:24	**Completion Code.** This field encodes the completion status that can be identified by a TRB. The *Completion Code* field shall always be set to *Success* for a *Bandwidth Request Event*.

Table 6-47: Offset 0Ch – Bandwidth Request Event TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Dequeue Pointer of an Event Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Bandwidth Request TRB type ID.
23:16	**RsvdZ**.
31:24	**Slot ID.** The ID of the Device Slot that should evaluate its bandwidth requirements. This is value is used as an index in the Device Context Base Address Array to select the Device Context of the source device.

6.4.2.5 Doorbell Event TRB

A *Doorbell Event TRB* shall be generated by the xHC when an emulated doorbell is written in a VF. A doorbell is emulated if the *Slot Emulated* bit is set to ‘1’ for the respective *VF Device Slot Assignment Register*. Refer to section 7.7.3.

Note: The Primary Event Ring (0) shall receive all Doorbell Events.

Figure 6-18: Doorbell Event TRB

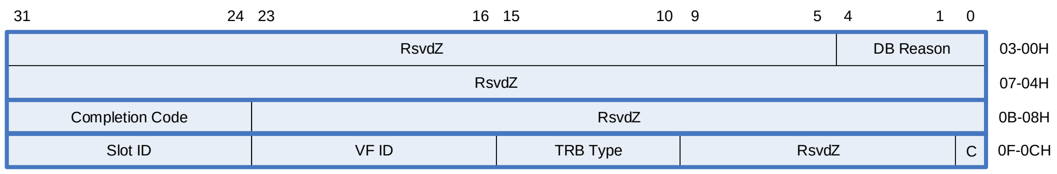


Table 6-48: Offset 00h – Doorbell Event TRB Field Definitions

Bits	Description
4:0	**DB Reason.** This field contains the value written to the DB Target field of the associated Doorbell.
31:5	**RsvdZ**.

Table 6-49: Offset 08h – Doorbell Event TRB Field Definitions

Bits	Description
23:0	**RsvdZ**.
31:24	**Completion Code.** This field encodes the completion status that can be identified by a TRB. The *Completion Code* field shall always be set to *Success* for a *Doorbell Event*.

Table 6-50: Offset 0Ch – Doorbell Event TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Dequeue Pointer of an Event Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Doorbell Event TRB type ID.
23:16	**VF ID.** The ID of the Virtual Function that generated the event.

31:24	**Slot ID.** The ID of the Device Slot that generated the event. This is value is used as an index in the Device Context Base Address Array to select the Device Context of the source device. If this Event is due to a Host Controller Command, then this field shall be cleared to '0'.

6.4.2.6 Host Controller Event TRB

A Host Controller Event TRB is a generic TRB, used to report xHC state changes and Error conditions.

Note: The Primary Event Ring (0) or a Secondary Event Ring may receive a Host Controller Event TRB, e.g. *Event Ring Full Error*.

Figure 6-19: Host Controller Event TRB

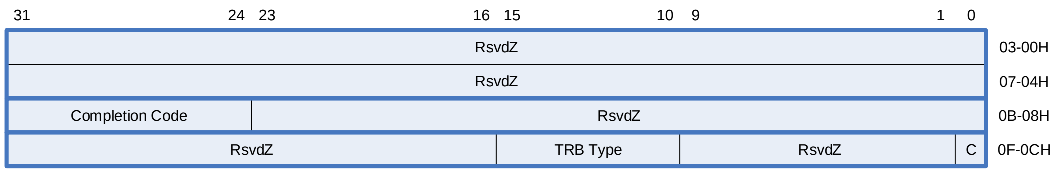


Table 6-51: Offset 08h – Host Controller Event TRB Field Definitions

Bits	Description
23:0	**RsvdZ**.
31:24	**Completion Code.** This field encodes the completion status that can be identified by a TRB. Refer to section 6.4.5 for an enumerated list of possible completion code values.

Table 6-52: Offset 0Ch – Host Controller Event TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Dequeue Pointer of an Event Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Host Controller Event TRB type ID.

31:16	**RsvdZ**.

6.4.2.7 Device Notification Event TRB

A *Device Notification Event TRB* is used to report the information received in USB Device Notification (DEV_NOTIFICATION) Transaction Packets from USB Devices. Refer to section 4.13 for more information on Device Notifications.

Note: The Primary Event Ring (0) or a Secondary Event Ring may receive a Device Notification Event TRB. If enabled in the DNCTRL register (5.4.4), the xHC shall use the *Interrupter Target* field of the Slot Context indexed by the Device Notification Event TRB *Slot ID* field to determine the Event Ring that shall receive the event.

Figure 6-20: Device Notification Event TRB

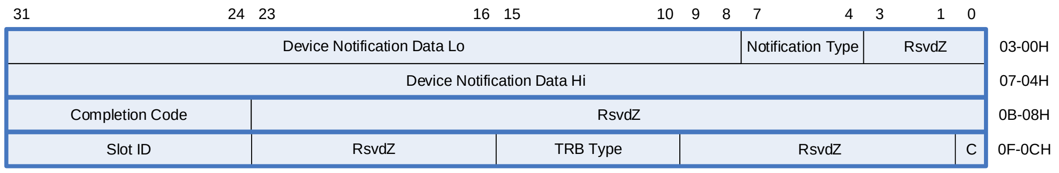


Table 6-53: Offset 00h and 04h – Device Notification Event TRB Field Definitions

Bits	Description
3:0	**RsvdZ**.
7:4	**Notification Type**. This field reports the value of the *Notification Type* field of the received USB Device Notification Transaction Packet.
63:8	**Device Notification Data.** This field reports the value of bytes 05h through 0Bh of the received USB Device Notification Transaction Packet (DNTP), i.e. Device Notification Event (DNE) TRB byte 01h = DNTP byte 05h,..., DNE TRB byte 07h = DNTP byte 0Bh.

Table 6-54: Offset 08h – Device Notification Event TRB Field Definitions

Bits	Description
23:0	**RsvdZ**.
31:24	**Completion Code.** This field encodes the completion status of the TRB, and shall always be set to *Success*. Refer to section 6.4.5 for an enumerated list of the completion code values.

Table 6-55: Offset 0Ch – Device Notification Event TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Dequeue Pointer of an Event Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Device Notification Event TRB type ID.
23:16	**RsvdZ**.
31:24	**Slot ID.** The ID of the Device Slot that generated the event. This value is used as an index in the Device Context Base Address Array to select the Device Context of the source device.

6.4.2.8 MFINDEX Wrap Event TRB

A *MFINDEX Wrap Event TRB* may be used by software to report when the MFINDEX register wrap from 0x3FFFh to 0. Refer to section 4.12.2 for more information.

Note: The Primary Event Ring (0) shall receive all MFINDEX Wrap Events.

Figure 6-21: MFINDEX Wrap Event TRB

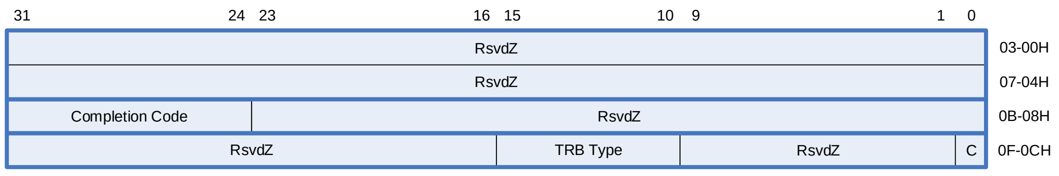


Table 6-56: Offset 08h – MFINDEX Wrap Event TRB Field Definitions

Bits	Description
23:0	**RsvdZ**.
31:24	**Completion Code.** This field encodes the completion status of the TRB, and shall always be set to *Success*. Refer to section 6.4.5 for an enumerated list of the completion code values.

Table 6-57: Offset 0Ch – MFINDEX Wrap Event TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Dequeue Pointer of an Event Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the MFINDEX Wrap Event TRB type ID.
31:16	**RsvdZ**.

6.4.3 Command TRBs

A Command TRB shall be found on a **Command Ring**. A Work Item on a Command Ring is called a **Command Descriptor** (CD) and is comprised of a single Command TRB. This section describes the command related TRBs.

Note: Data buffers referenced by Command TRBs shall not span PAGESIZE boundaries.

6.4.3.1 No Op Command TRB

The *No Op Command TRB* provides a simple means for verifying the operation of the Command Ring mechanisms offered by the xHCI. Refer to section 4.6.2 for more information.

Figure 6-22: No Op Command TRB

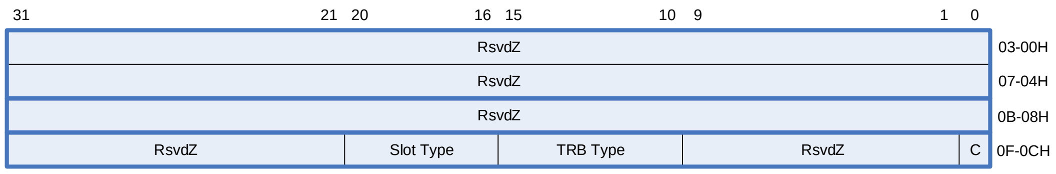


Table 6-58: Offset 0Ch – No Op Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the No Op Command TRB type ID.
31:16	**RsvdZ**.

6.4.3.2 Enable Slot Command TRB

The *Enable Slot Command TRB* causes the xHC to select an available Device Slot and return the ID of the selected slot to the host in a Command Completion Event. Refer to section 4.6.3 for more information.

Figure 6-23: Enable Slot Command TRB

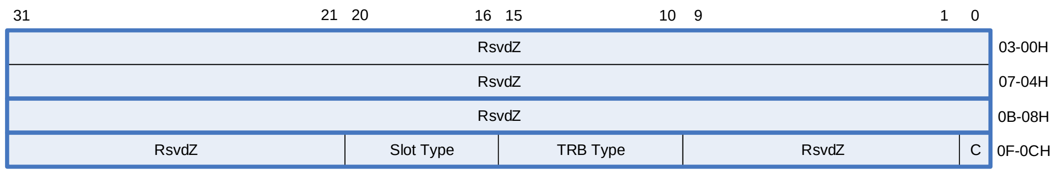


Table 6-59: Offset 0Ch – Enable Slot Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
9:1	**RsvdZ**.

15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Enable Slot Command TRB type ID.
20:16	**Slot Type**. This field identifies the type of Slot that will be enabled by this command. Refer to Table 7-9 for more information on the usage of *Slot Type*.
31:21	**RsvdZ**.

6.4.3.3 Disable Slot Command TRB

The *Disable Slot Command TRB* releases any bandwidth assigned to the disabled slot and frees any internal xHC resources assigned to the slot. Refer to section 4.6.4 for more information.

Figure 6-24: Disable Slot Command TRB

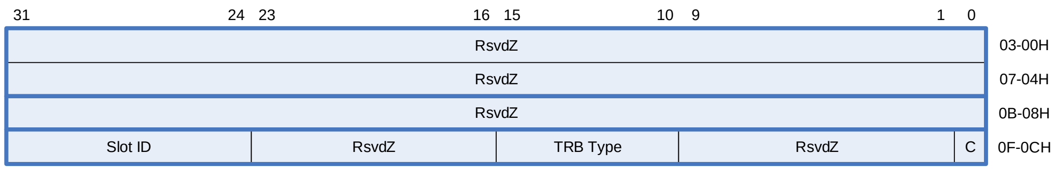


Table 6-60: Offset 0Ch – Disable Slot Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Disable Slot Command TRB type ID.
23:16	**RsvdZ**.
31:24	**Slot ID.** The ID of the Device Slot to disable.

6.4.3.4 Address Device Command TRB

The *Address Device Command TRB* transitions the selected Device Context from the *Default* to the *Addressed* state and causes the xHC to select an address for the USB device in the Default State and issue a SET_ADDRESS request to the USB device. Refer to section 4.6.5 for more information.

Figure 6-25: Address Device Command TRB

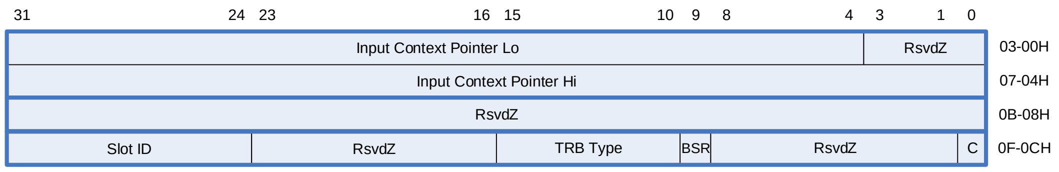


Table 6-61: Offset 00h and 04h – Address Device Command TRB Field Definitions

Bits	Description
3:0	**RsvdZ**.
63:4	**Input Context Pointer Hi and Lo.** This field represents the high order bits of the 64-bit base address of the *Input Context* data structure associated with this command. Refer to section 6.2.5 for more information on the *Input Context* data structure. The memory structure referenced by this physical memory pointer shall be aligned on a 16-byte address boundary.

Table 6-62: Offset 0Ch – Address Device Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
8:1	**RsvdZ**.
9	**Block Set Address Request (BSR)**. When this flag is set to '0' the *Address Device Command* shall generate a USB SET_ADDRESS request to the device. When this flag is set to '1' the *Address Device Command* shall not generate a USB SET_ADDRESS request. Refer to section 4.6.5 for more information on the use of this flag.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Address Device Command TRB type ID.
23:16	**RsvdZ**.
31:24	**Slot ID.** The ID of the Device Slot that is the target of this command.

6.4.3.5 Configure Endpoint Command TRB

The *Configure Endpoint Command TRB* evaluates the bandwidth and resource requirements of the endpoints selected by the command. Refer to section 4.6.6 for more information.

Figure 6-26: Configure Endpoint Command TRB

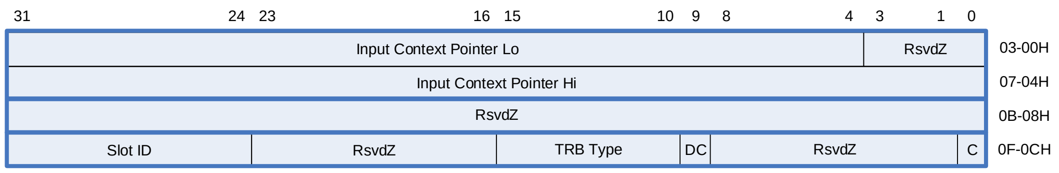


Table 6-63: Offset 00h and 04h – Configure Endpoint Command TRB Field Definitions

Bits	Description
3:0	**RsvdZ**.
63:4	**Input Context Pointer Hi and Lo.** This field represents the high order bits of the 64-bit base address of the *Input Context* data structure associated with this event. Refer to section 6.2.5 for more information on the *Input Context* data structure. The memory structure referenced by this physical memory pointer shall be aligned on a 16-byte address boundary.

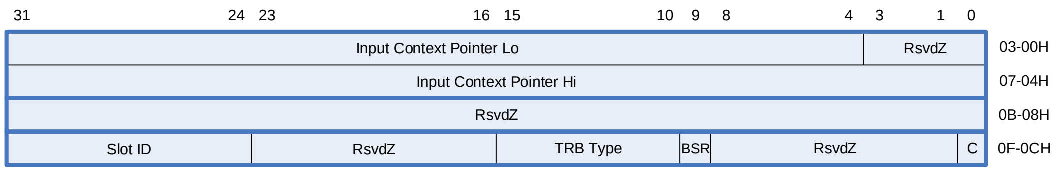
Table 6-64: Offset 0Ch – Configure Endpoint Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
8:1	**RsvdZ**.
9	**Deconfigure (DC)**. Set to '1' by software to "deconfigure" the Device Slot. If the *DC* flag = '1', the *Input Context Pointer* field is ignored by the xHC.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Configure Endpoint Command TRB type ID.
23:16	**RsvdZ**.
31:24	**Slot ID.** The ID of the Device Slot that is the target of this command.

6.4.3.6 Evaluate Context Command TRB

The *Evaluate Context Command TRB* is used by system software to inform the xHC that the selected Context data structures in the Device Context have been modified by system software and that the xHC shall evaluate any changes. Refer to the Slot and Endpoint Context data structure descriptions (sections 6.2.2.3 and 6.2.3.3, respectively) for more information on how the xHC applies this command. Refer to section 4.6.7 for more information.

Figure 6-27: Evaluate Context Command TRB



The *Evaluate Context Command TRB* uses the same format as the *Address Device Command TRB*, with the following exceptions: 1) the *TRB Type* field is set to the *Evaluate Context Command* TRB type ID, and 2) the *BSR* field is not used. Refer to Table 6-62 for the definitions of the remaining fields in the *Address Device Command* Control component.

6.4.3.7 Reset Endpoint Command TRB

The *Reset Endpoint Command TRB* is used by system software to reset a specified Transfer Ring. Refer to section 4.6.8 for more information.

Figure 6-28: Reset Endpoint Command TRB

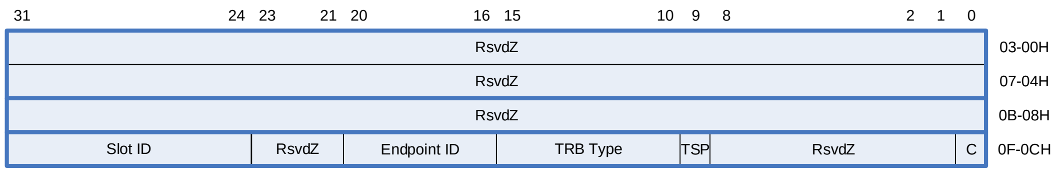


Table 6-65: Offset 0Ch – Reset Endpoint Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
8:1	**RsvdZ**.

9	**Transfer State Preserve (TSP).** Set to '1' by software if the Reset operation does not affect the current transfer state of the endpoint. Cleared to '0' by software if the Reset operation resets the current transfer state of the endpoint, i.e. The Data Toggle of a USB2 device or the Sequence Number of a USB3 device is cleared to '0'. Also refer to section 4.6.8.1.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the *Reset Endpoint Command TRB* type ID.
20:16	**Endpoint ID.** This field identifies the DCI of the endpoint to be reset.
23:21	RsvdZ.
31:24	**Slot ID.** The ID of the Device Slot.

6.4.3.8 Stop Endpoint Command TRB

The *Stop Endpoint Command TRB* command allows software to stop the xHC execution of the TDs on a Transfer Ring and temporarily take ownership of TDs that had previously been passed to the xHC. Refer to section 4.6.9 for more information.

Figure 6-29: Stop Endpoint Command TRB

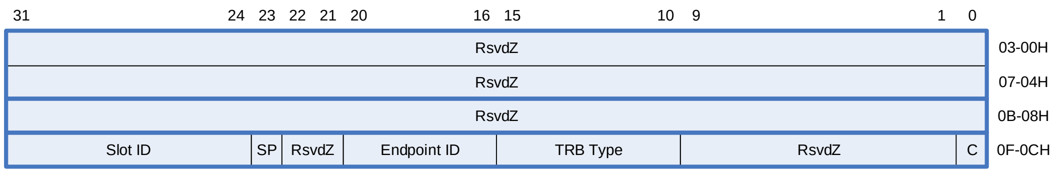


Table 6-66: Offset 0Ch – Stop Endpoint Command TRB Field Definitions

Bits	Description
0	**Cycle (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Stop Endpoint Command TRB type ID.
20:16	**Endpoint ID.** This field identifies the DCI of the endpoint to be stopped. Valid values are '1' to Output Slot Context *Context Entries*.
22:21	**RsvdZ**.

23	**Suspend (SP)**. When '1' this bit indicates that the *Stop Endpoint Command* is being issued to stop activity on an endpoint that is about to be suspended, and the endpoint shall be stopped for at least 10 ms. The xHC may use this information to power manage the endpoint hardware resources. Refer to section 4.15 for more information.
31:24	**Slot ID.** The ID of the Device Slot.

In order to assure proper USB device operation, software shall wait for at least 10 ms. after a port indicates that it is suspended (PLS = '3') before initiating a port resume.

6.4.3.9 Set TR Dequeue Pointer Command TRB

The *Set TR Dequeue Pointer Command TRB* is used by system software to modify the *TR Dequeue Pointer* and *DCS* fields of an Endpoint or Stream Context. Refer to section 4.6.10 for more information.

Figure 6-30: Set TR Dequeue Pointer Command TRB

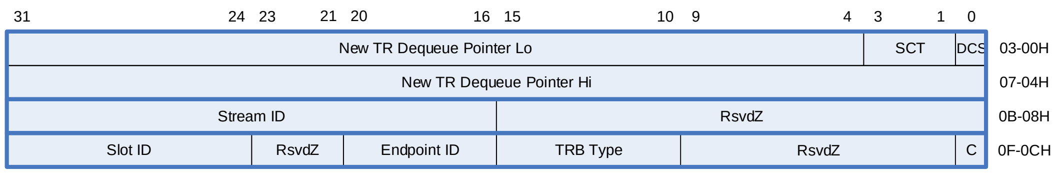


Table 6-67: Offset 00h and 04h – Set TR Dequeue Pointer Command TRB Field Definitions

Bits	Description
0	**Dequeue Cycle State (DCS)**. This bit identifies the value of the xHC Consumer Cycle State (CCS) flag for the TRB referenced by the *TR Dequeue Pointer*.
3:1	**Stream Context Type (SCT)**. If the *Stream ID* field is non-zero, this field identifies the type of the Stream Context, otherwise this field shall be '0'. Refer to section Table 6-13 for the definition the *SCT* field values.
63:4	**New TR Dequeue Pointer Hi and Lo.** This field represents the high order bits of the 64-bit base address to be written to the *TR Dequeue Pointer* field in the target Endpoint or Stream Context. The memory structure referenced by this physical memory pointer shall be aligned on a 16-byte address boundary.

Table 6-68: Offset 08h – Set TR Dequeue Pointer Command TRB Field Definitions

Bits	Description
15:0	**RsvdZ**.
31:16	**Stream ID.** If Streams are enabled for this endpoint, this field identifies the Stream Context that will receive the new *TR Dequeue Pointer*. Refer to section 4.12.2.1 for the bounds checking that the xHC shall perform on this value.

Table 6-69: Offset 0Ch – Set TR Dequeue Pointer Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Set TR Dequeue Pointer Command TRB type ID.
20:16	**Endpoint ID.** This field identifies the DCI of the endpoint that is the target of this command. If Streams are not enabled for the endpoint, the Endpoint Context will receive the new *TR Dequeue Pointer*.
23:21	RsvdZ.
31:24	**Slot ID.** The ID of the Device Slot.

Note: This command shall not be issued by software unless the target Transfer Ring is in the *Error* or *Stopped* state or if it is a Streams endpoint and the target Stream ID is active.

6.4.3.10 Reset Device Command TRB

The *Reset Device Command TRB* is used by software to inform the xHC that a USB device has been Reset. The reset operation sets the device slot to the *Default* state, sets the Device Address to '0', and disables all endpoints except for the Default Control Endpoint. Refer to section 4.6.11 for more information.

Figure 6-31: Reset Device Command TRB

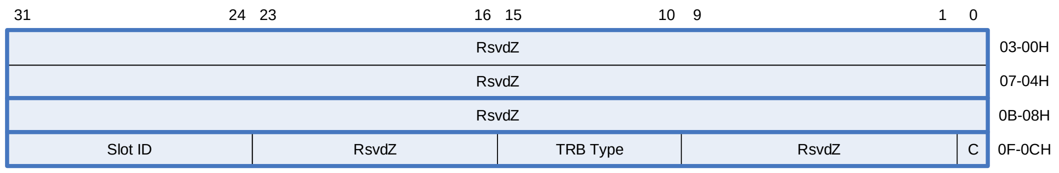


Table 6-70: Offset 0Ch – Reset Device Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Reset Device Command TRB type ID.
23:16	**RsvdZ**.
31:24	Slot ID. The ID of the Device Slot that is being reset.

6.4.3.11 Force Event Command TRB (Optional Normative)

The *Force Event Command TRB* allows a VMM to inject an Event TRB on the Event Ring of a selected VF. VMMs utilize this command when emulating a USB device to a VM. Refer to section 8 for more information on virtualization. Refer to section 4.6.12 for more information.

Figure 6-32: Force Event Command TRB

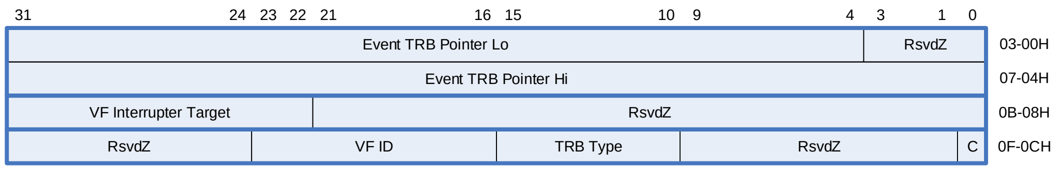


Table 6-71: Offset 00h and 04h – Force Event Command TRB Field Definitions

Bits	Description
3:0	**RsvdZ**.

63:4	**Event TRB Pointer Hi and Lo.** This field represents the high order bits of the 64-bit address of the Event TRB that will be posted to the target Event Ring. The memory structure referenced by this physical memory pointer shall be aligned on a 16-byte address boundary.

Table 6-72: Offset 08h – Force Event Command TRB Field Definitions

Bits	Description
21:0	**RsvdP**.
31:22	**VF Interrupter Target.** This field shall indicate the ID of the Interrupter, whose Event Ring will receive the forced event. The Interrupter ID is the virtual value used by the target VF (based on the *Interrupter Offset* field of the *VF Interrupter Range Register*), not a physical value. Refer to section 7.7.2 for more information on virtual Interrupter mapping.

Table 6-73: Offset 0Ch – Force Event Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Force Event Command TRB type ID.
23:16	**VF ID.** The ID of the Virtual Function who's Event Ring will receive this Event.
31:24	**RsvdZ**.

6.4.3.12 Negotiate Bandwidth Command TRB (Optional Normative)

The Negotiate Bandwidth Command TRB is used by system software to initiate Bandwidth Request Events to periodic endpoints. This command may be used to recover unused USB bandwidth from the system. Refer to section 4.6.3 for more information.

The Negotiate Bandwidth Command TRB uses the same format as the Disable Slot Command (6.4.3.3), with the exception that the TRB Type field is set to the Negotiate Bandwidth Command TRB type ID, and the Slot ID is set to the ID of the slot that requires the bandwidth negotiation. Refer to Table 6-60 for the definitions of the remaining fields in the Negotiate Bandwidth Command Control component.

6.4.3.13 Set Latency Tolerance Value (LTV) Command TRB (Optional Normative)

The *Set LTV Command TRB* provides a simple means for host software to provide a single Best Effort Latency Tolerance (BELT) value. This command is optional normative, however it shall be supported if the xHC also supports a corresponding host interconnect LTM mechanism. Refer to sections 4.6.14 and 4.13.1 for more information.

Figure 6-33: Set Latency Tolerance Value Command TRB

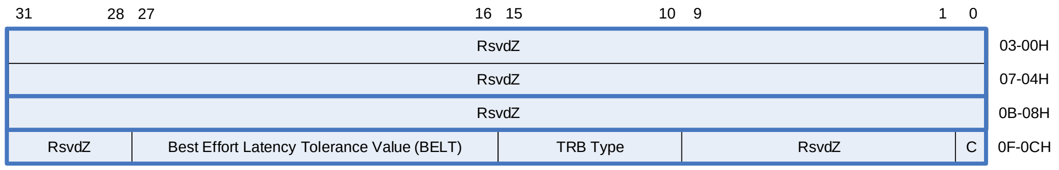


Table 6-74: Offset 0Ch – Set Latency Tolerance Value Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Set Latency Tolerance Value Command TRB type ID.
27:16	**Best Effort Latency Tolerance Value**. The Best Effort Latency Tolerance (BELT) value provided by software. This value shall be formatted as defined in the section of the USB3 Specification describing Device Notification (DEV_NOTIFICATION) Transaction Packet (TP).
31:28	**RsvdZ**.

6.4.3.14 Get Port Bandwidth Command TRB

The *Get Port Bandwidth Command TRB* provides a means for host software to identify the bandwidth available on xHC Root Hub Ports. Refer to section 4.6.15 for more information.

Figure 6-34: Get Port Bandwidth Command TRB

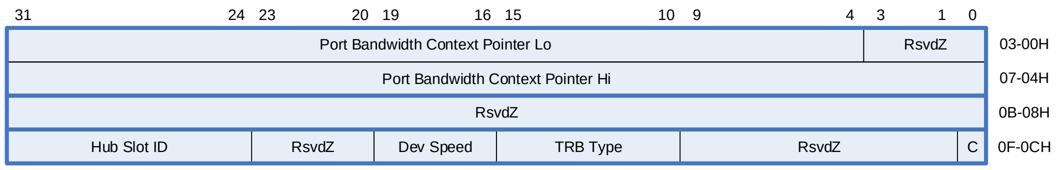


Table 6-75: Offset 00h and 04h – Get Port Bandwidth Command TRB Field Definitions

Bits	Description
3:0	**RsvdZ**.
63:4	**Port Bandwidth Context Pointer Hi and Lo.** This field represents the high order bits of the 64-bit address of the Port Bandwidth Context data structure that will receive the Port Bandwidth information. The memory structure referenced by this physical memory pointer shall be aligned on a 16-byte address boundary.

Table 6-76: Offset 0Ch – Get Port Bandwidth Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer of a Command Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Get Port Bandwidth Command TRB type ID.
19:16	**Dev Speed.** The bus speed of interest. Refer to the *Port Speed* field in Table 5-26 for a definition of the allowed values. Note: The *Undefined* and *Reserved* Speeds are invalid values for this field.
23:20	**RsvdZ**.
31:24	**Hub Slot ID**. This field identifies the hub ports that the bandwidth information shall be returned for. A value of '0' shall update the Port Bandwidth Context with the Root Hub port bandwidth information. If this field is set to the Slot ID of a High-speed hub, the Port Bandwidth Context shall be updated with that port's bandwidth information. This field is ignored if *SBD* = '0'. Refer to section 4.16.2 for more information on the use of this field.

6.4.3.15 Force Header Command TRB

A *Force Header Command TRB* is used to generate a USB Transaction or Link Management Packet to a USB Device. Refer to section 4.6.16 for more information.

Figure 6-35: Force Header Command TRB

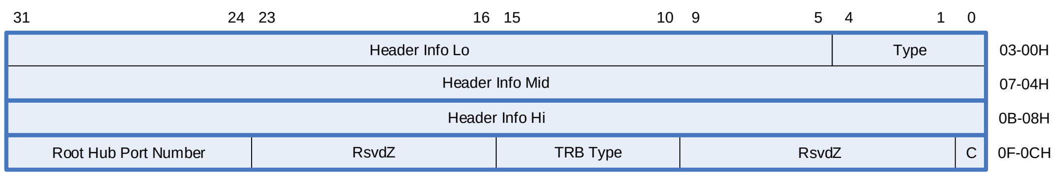


Table 6-77: Offset 00h, 04h, and 08h – Force Header Command TRB Field Definitions

Bits	Description
4:0	Packet Type (Type). This field identifies the packet type. Refer to section 8.3.1.2 in the USB3 specification for valid values.
95:5	**Header Info.** This field defines the value of bytes 00h through 0Bh of the transmitted USB Transaction or Link Management Packet. Refer to Section 8 in the USB3 specification for the definition of this field.

Table 6-78: Offset 0Ch – Force Header Command TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Dequeue Pointer of an Event Ring.
9:1	**RsvdZ**.
15:10	**TRB Type.** This field identifies the type of the TRB. Refer to Table 6-86 for the definition of the Force Header Command TRB type ID.
23:16	**RsvdZ**.
31:24	**Root Hub Port Number.** This field identifies the number of the Root Hub Port that the header packet shall be issued to. Refer to section 4.19.7 for port numbering information.

6.4.4 Other TRBs

6.4.4.1 Link TRB

A *Link TRB* provides support for non-contiguous TRB Rings. Refer to section 4.11.5.1 for more information on *Link TRBs* and the operation of non-contiguous TRB Rings.

Note: Consecutive *Link TRBs* may impact xHC performance and should be avoided by software. Refer to section 4.11.7 for more information on *Link TRB* placement rules.

Figure 6-36: Link TRB

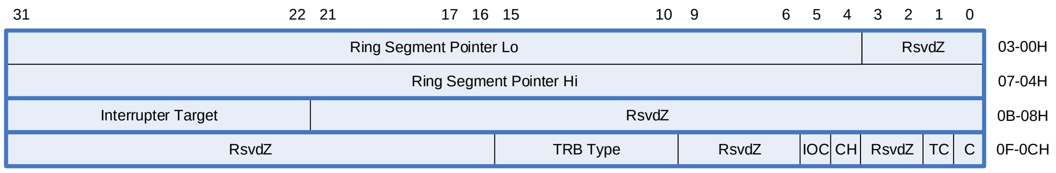


Table 6-79: Offset 00h and 04h – Link TRB Field Definitions

Bits	Description
3:0	**RsvdZ.** Ring Segments are TRB aligned (16 Byte boundaries).
63:4	**Ring Segment Pointer Hi and Lo.** These fields represent the high order bits of the 64-bit base address of the next Ring Segment. The memory structure referenced by this physical memory pointer shall begin on a 16-byte address boundary.

Table 6-80: Offset 08h – Link TRB Field Definitions

Bits	Description
21:0	**RsvdZ**.
31:22	**Interrupter Target**. This field defines the index of the Interrupter that will receive Transfer Events generated by this TRB. Valid values are between 0 and *MaxIntrs*-1. This field is ignored by the xHC on Command Rings.

Table 6-81: Offset 0Ch – Link TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is used to mark the Enqueue Pointer location of a Transfer or Command Ring.
1	**Toggle Cycle (TC).** When set to '1', the xHC shall toggle its interpretation of the Cycle bit. When cleared to '0', the xHC shall continue to the next segment using its current interpretation of the Cycle bit.
3:2	**RsvdZ**.
4	**Chain bit (CH).** Set to '1' by software to associate this TRB with the next TRB on the Ring. A Transfer Descriptor (TD) is defined as one or more TRBs. The *Chain bit* is used to identify the TRBs that comprise a TD. Refer to section 4.11.7 for more information on Link TRB placement within a TD. On a Command Ring this bit is ignored by the xHC.
5	**Interrupt On Completion (IOC).** If this bit is set to '1', it specifies that when this TRB completes, the Host Controller shall notify the system of the completion by placing an Event TRB on the Event ring and sending an interrupt at the next interrupt threshold.
9:6	**RsvdZ**.
15:10	**TRB Type.** This field is set to *Link TRB* type. Refer to Table 6-86 for the definition of the Type TRB IDs.
31:16	**RsvdZ**.

6.4.4.2 Event Data TRB

An *Event Data TRB* allows system software to generate a software defined event and specify the Parameter field of the generated Transfer Event.

Note: When applying Event Data TRBs to control transfer: 1) An Event Data TRB may be inserted at the end of a Data Stage TD in order to report the accumulated transfer length of a multi-TRB TD. 2) An Event Data TRB may be inserted at the end of a Status Stage TD in order to provide Event Data associated with the control transfer completion.

Refer to section 4.11.5.2 for more information.

Figure 6-37: Event Data TRB

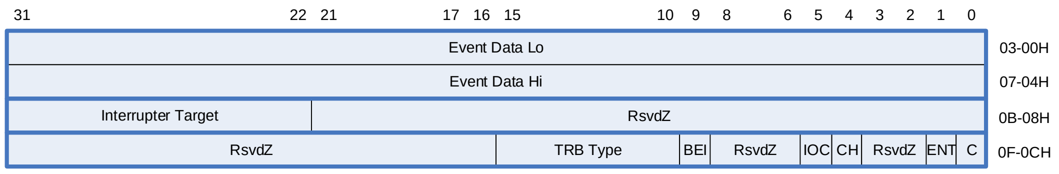


Table 6-82: Offset 00h and 04h – Event Data TRB Field Definitions

Bits	Description
63:0	**Event Data Hi and Lo.** This field represents the 64-bit value that shall be copied to the TRB Pointer field (Parameter Component) of the Transfer Event TRB.

Table 6-83: Offset 08h – Event Data TRB Field Definitions

Bits	Description
21:0	**RsvdZ**.
31:22	**Interrupter Target**. This field defines the index of the Interrupter that will receive Transfer Events generated by this TRB. Valid values are between 0 and *MaxIntrs*-1.

Table 6-84: Offset 0Ch – Event Data TRB Field Definitions

Bits	Description
0	**Cycle bit (C).** This bit is ignored by the xHC in a Link TRB.
1	**Evaluate Next TRB (ENT)**. If this flag is '1' the xHC shall fetch and evaluate the next TRB before saving the endpoint state. Refer to section 4.12.3 for more information.
3:2	**RsvdZ**.
4	**Chain bit (CH).** Set to '1' by software to associate this TRB with the next TRB on the Transfer Ring. The Chain bit is used to identify the TRBs that comprise a TD. The Chain bit is always '0' in the last TRB of a TD.

5	**Interrupt On Completion (IOC).** If this bit is set to '1', it specifies that when this TRB completes, the Host Controller shall notify the system of the completion by placing an Event TRB on the Event ring and sending an interrupt at the next interrupt threshold.
8:6	**RsvdZ**.
9	**Block Event Interrupt (BEI).** If this bit is set to '1' and *IOC* = '1', then the Transfer Event generated by *IOC* shall not assert an interrupt to the host at the next interrupt threshold. Refer to section 4.17.5.
15:10	**TRB Type.** This field is set to *Event Data TRB* type. Refer to Table 6-86 for the definition of the Type TRB IDs.
31:16	**RsvdZ**.

6.4.5 TRB Completion Codes

The following TRB Completion Status codes will be asserted by the Host Controller during status update if the associated error condition is detected.

Table 6-85: TRB Completion Code Definitions

Value	Definition	Description
0	Invalid	Indicates that the Completion Code field has not been updated by the TRB producer.
1	Success	Indicates successful completion of the TRB operation.
2	Data Buffer Error	Indicates that the Host Controller is unable to keep up with the reception of incoming data (overrun) or is unable to supply data fast enough during transmission (underrun). Section 4.10.2.5 defines the requirements of the host controller when a *Data Buffer Error* occurs.
3	Babble Detected Error	Asserted when "babbling" is detected during the transaction generated by this TRB[113].
4	USB Transaction Error	Asserted in the case where the host did not receive a valid response from the device (Timeout, CRC, Bad PID, unexpected NYET, etc.).
5	TRB Error	Asserted when a TRB parameter error condition (e.g., out of range or invalid parameter) is detected in a TRB. Refer to section 4.10.2.2 for examples.

6	Stall Error	Asserted when a Stall condition (e.g., a Stall PID received from a device) is detected for a TRB. Refer to section 4.10.2.1 for more information on Stalls. This code also indicates that the USB device has an error that prevents it from completing a command issued through a Control endpoint. Refer to section 8.5.3.1 of the USB2 specification for more information.
7	Resource Error	Asserted by a *Configure Endpoint Command* or an *Address Device Command* if there are not adequate xHC resources available to successfully complete the command. Refer to sections 4.6.5 and 4.6.6 for more information.
8	Bandwidth Error	Asserted by a Configure Endpoint Command if periodic endpoints are declared and the xHC is not able to allocate the required Bandwidth. Refer to section 4.16 for more information.
9	No Slots Available Error	Asserted if a adding one more device would result in the host controller to exceed the maximum *Number of Device Slots* (MaxSlots) for this implementation. Refer to section 4.6.3 for more information.
10	Invalid Stream Type Error	Asserted if a invalid *Stream Context Type* (SCT) value is detected. Refer to section 4.12.2.1 for more information.
11	Slot Not Enabled Error	Asserted if a command is issued to a Device Slot that is in the *Disabled* state. The Slot ID is reported.
12	Endpoint Not Enabled Error	Asserted if a doorbell is rung for an endpoint that is in the *Disabled* state. The Slot ID and error Endpoint ID are reported. Also refer to section 4.7.
13	Short Packet	Asserted if the number of bytes received was less than the TD Transfer Size.
14	Ring Underrun	Asserted in a Transfer Event TRB if the Transfer Ring is empty when an enabled Isoch endpoint is scheduled to transmit data. Refer to section 4.10.3.1. Note that the Transfer Event *TRB Pointer* field is not valid when this condition is indicated and should be ignored by software.
15	Ring Overrun	Asserted in a Transfer Event TRB if the Transfer Ring is empty when an enabled Isoch endpoint is scheduled to receive data. Refer to section 4.10.3.1. Note that the Transfer Event *TRB Pointer* field is not valid when this condition is indicated and should be ignored by software.
16	VF Event Ring Full Error	Asserted by a Force Event command if the target VF's Event Ring is full. Refer to section 4.9.4 for more information. Note that the Transfer Event *TRB Pointer* field is not valid when this error is indicated and should be ignored by software.

17	Parameter Error	Asserted by a command if a Context parameter is invalid.
18	Bandwidth Overrun Error	Asserted during an Isoch transfer if the TD exceeds the bandwidth allocated to the endpoint.
19	Context State Error	Asserted if a command is issued to transition from an illegal context state.
20	No Ping Response Error	Asserted if the xHC was unable to complete a periodic data transfer associated within the ESIT, because it did not receive a PING_RESPONSE in time. Refer to section 4.23.5.2.1 for more information.
21	Event Ring Full Error	Asserted if the Event Ring is full, the xHC is unable to post an Event to the ring (refer to section 4.9.4). This error is reported in a Host Controller Event TRB.
22	Incompatible Device Error	Asserted if the xHC detects a problem with a device that does not allow it to be successfully accessed. e.g. due to a device compliance or compatibility problem. This error may be returned by any command or transfer, and is fatal as far as the Slot is concerned. Software shall issue a *Disable Slot Command* to recover[114,112].
23	Missed Service Error	Asserted if the xHC was unable to service a Isochronous endpoint within the *Interval* time (ESIT). Refer to sections 4.9.4 and 4.10.3.2 for more information.
24	Command Ring Stopped	Asserted in a *Command Completion Event* due to a Command Stop (CS) operation. Refer to section 4.6 for more information.
25	Command Aborted	Asserted in a *Command Completion Event* of an aborted command if the command was terminated by a Command Abort (CA) operation. Refer to section 4.6 for more information.
26	Stopped	Asserted in a *Transfer Event* if the transfer was terminated by a *Stop Endpoint Command*. Refer to section 4.6.9 for more information.
27	Stopped - Length Invalid	Asserted in a *Transfer Event* if the transfer was terminated by a *Stop Endpoint Command* and the Transfer Event *TRB Transfer Length* field is invalid. Refer to section 4.6.9 for more information.

[112]USB system software stacks commonly support a number of "Quirk" devices. A *Quirk* device is any device that is not compliant with the USB spec and requires software or the xHC to make a compliance exception to support it. An *Incompatible Device Error* should be generated if the xHC detects a *Quirk* device that it does not support.

28	Stopped - Short Packet	Asserted in a *Transfer Event* if the transfer was terminated by a *Stop Endpoint Command*, and the transfer was stopped after Short Packet conditions were met, but before the end of the TD was reached. The Transfer Event *TRB Transfer Length* field shall contain the value of the EDTLA. Refer to section 4.6.9 for more information on the Stop Endpoint Command, section 4.10.1.1 for Short Transfer information, and section 4.11.5.2 for EDTLA information.
29	Max Exit Latency Too Large Error	Asserted by the *Evaluate Context Command* if the proposed *Max Exit Latency* would not allow the periodic endpoints of the Device Slot to be scheduled. Refer to sections 4.23.5.2.2 and 4.6.6.1.
30	Reserved	
31	Isoch Buffer Overrun	Asserted if the data buffer defined by an Isoch TD on an IN endpoint is less than the Max ESIT Payload in size and the device attempts to send more data than it can hold[113]. Refer to sections 4.14.2.1.1 and 4.14.2.1.3.
32	Event Lost Error	Asserted if the xHC internal event overrun condition. If the condition is due to TD related events, then the endpoint shall be halted. The conditions that generate this error are xHC implementation specific[114]. Refer to section 4.10.1.
33	Undefined Error	May be reported by an event when other error codes do not apply. The conditions that assert this condition code are xHC implementation specific. Refer to section 4.11.6 for more information. An *Undefined Error* shall be treated as a fatal error by software.
34	Invalid Stream ID Error	Asserted if a invalid Stream ID is received. Refer to section 4.12.2.1 for more information.
35	Secondary Bandwidth Error	Asserted by a Configure Endpoint Command if periodic endpoints are declared and the xHC is not able to allocate the required Bandwidth due to a Secondary Bandwidth Domain. Refer to section 4.16 for more information.
36	Split Transaction Error	Asserted if an error is detected on a USB2 protocol endpoint for a split transaction. Refer to section 4.10.3.3.

[113]When a TD Babble condition occurs on non-Isoch endpoints it generates a *Babble Detected Error* and halts the endpoint. However for Isoch endpoints, a TD Babble condition generates an *Isoch Buffer Overrun* and does not halt the endpoint.

[114]Refer to the xHC vendor data sheet for more information on the possible sources of this error.

37-191	Reserved	
192-223	Vendor Defined Error	Asserted by a vendor to indicate an error condition has occurred. Refer to vendor documentation to identify specific error condition(s). If software does not recognize the code, it shall interpret this range of vendor defined values as a *Undefined Error* condition. Refer to section 4.11.6 for more information.
224-255	Vendor Defined Info	Asserted by a vendor for informational purposes. Refer to vendor documentation to identify specific information reported. If software does not recognize the code, it shall interpret this range of vendor defined values as a *Success* condition code. Refer to section 4.11.6 for more information.

If multiple error conditions occur during the execution of a TRB only the first detected condition will be reported.

6.4.6 TRB Types

TRB Types fall into three categories; Command, Event, or Transfer. These categories relate to the TRB Ring that specific TRB(s) may appear on. Table 6-86 identifies the specific TRB Types that are "Allowed" on each Ring type.

Note: In Table 6-86 the *ID* values are uniquely assigned to each TRB Type, however to conserve IDs as new TRB Types are defined in the future the same ID value may identify different TRB types as a function of Ring type. e.g. a new TRB that is only allowed on a Command Ring may use ID = 2.

Table 6-86: TRB Type Definitions

Allowed TRB Types			ID	TRB Name
Command Ring	Event Ring	Transfer Ring		
			0	Reserved
		Allowed	1	Normal
		Allowed	2	Setup Stage
		Allowed	3	Data Stage
		Allowed	4	Status Stage

		Allowed	5	Isoch
Allowed		Allowed	6	Link
		Allowed	7	Event Data
		Allowed	8	No-Op
Allowed			9	Enable Slot Command
Allowed			10	Disable Slot Command
Allowed			11	Address Device Command
Allowed			12	Configure Endpoint Command
Allowed			13	Evaluate Context Command
Allowed			14	Reset Endpoint Command
Allowed			15	Stop Endpoint Command
Allowed			16	Set TR Dequeue Pointer Command
Allowed			17	Reset Device Command
Allowed			18	Force Event Command *(Optional, used with virtualization only)*
Allowed			19	Negotiate Bandwidth Command *(Optional)*
Allowed			20	Set Latency Tolerance Value Command (*Optional*)
Allowed			21	Get Port Bandwidth Command
Allowed			22	Force Header Command
Allowed			23	No Op Command
			24-31	Reserved
	Allowed		32	Transfer Event
	Allowed		33	Command Completion Event
	Allowed		34	Port Status Change Event
	Allowed		35	Bandwidth Request Event *(Optional)*

	Allowed		36	Doorbell Event *(Optional, used with virtualization only)*
	Allowed		37	Host Controller Event
	Allowed		38	Device Notification Event
	Allowed		39	MFINDEX Wrap Event
			40-47	Reserved
Optional	Optional	Optional	48-63	Vendor Defined

Note: Only the TRB Types specifically "Allowed" in the **Command Ring** column of Table 6-86 shall be executed on a Command Ring by the xHC. All other TRB types found on a Command Ring shall generate a Command Completion Event with the Completion Code set to *TRB Error*, the *Command TRB Pointer* set to the address of the TRB in error, and the *Slot ID* field cleared to '0'.

Note: Only the TRB Types specifically "Allowed" in the **Event Ring** column of Table 6-86 shall be generated on an Event Ring by the xHC.

Note: Only the TRB Types specifically "Allowed" in the **Transfer Ring** column of Table 6-86 shall be executed on a Transfer Ring by the xHC. All other TRB types found on a Transfer Ring shall generate a Transfer Event with the Completion Code set to *TRB Error*, the *TRB Pointer* set to the address of the TRB in error, and the *Slot ID* and *Endpoint ID* fields should reflect the *Slot ID* and *Endpoint ID* of the Transfer Ring in error.

Note: The IDs available for the **Vendor Defined** TRB types shall be assigned by the xHC vendor. System software shall qualify all Vendor Defined TRB type IDs with the *Vendor ID* and *Device ID* fields in the PCI Configuration Space Header. If the xHC is not based on PCI, then the xHC vendor shall provide an alternate means of identifying the Vendor and Device Type to system software.

System software should provide interface extensions that allow vendor access to proprietary xHC vendor defined features through the xHCD.

Table 6-87 defines the allowable Transfer Ring TRB Types as function of endpoint type.

Table 6-87: Allowed TRB Type as function of Endpoint Type

Allowed TRB Types				Transfer Ring TRB Type
Isoch	Interrupt	Control	Bulk	
Allowed	Allowed	Allowed	Allowed	Normal
		Allowed		Setup Stage
		Allowed		Data Stage
		Allowed		Status Stage
Allowed				Isoch
Allowed	Allowed	Allowed	Allowed	Link
Allowed	Allowed	Allowed	Allowed	Event Data
Allowed	Allowed	Allowed	Allowed	No-Op
Optional	Optional	Optional	Optional	Vendor Defined

Note: If the xHC detects a disallowed TRB type on a Transfer Ring, it shall generate Transfer Event for the TD with the *TRB Error* completion code set and set the state of the ring to *Error*.

Table 6-88 defines the allowable Transfer Ring TRB Types as function of Transaction type.

Table 6-88: Allowed TRB Types as function of Transfer Descriptor Type

Transfer Descriptor Type	Allowed TRB Types
Isoch	Isoch, Normal, Event Data, No Op
Interrupt	Normal, Event Data, No Op
Control	Setup Stage, Data Stage, Status Stage, Normal, Event Data, No Op
Bulk	Normal, Event Data, No Op
Vendor Defined	Vendor Defined, Event Data, No Op

Note: If the xHC detects a disallowed TRB type on a Transfer Ring, it shall generate Transfer Event for the TD with the *TRB Error* completion code set and set the state of the endpoint to *Error*.

6.5 Event Ring Segment Table

The *Event Ring Segment Table* (ERST) is used to define multi-segment Event Rings and to enable runtime expansion and shrinking of the Event Ring. The location of the Event Ring Segment Table is defined by the *Event Ring Segment Table Base Address Register* (5.5.2.3.2). The size of the Event Ring Segment Table is defined by the *Event Ring Segment Table Base Size Register* (5.5.2.3.1).

This section defines the properties of a single Event Ring Segment Table element. Refer to section 4.9.4 for more information.

Figure 6-38: Event Ring Segment Table Entry

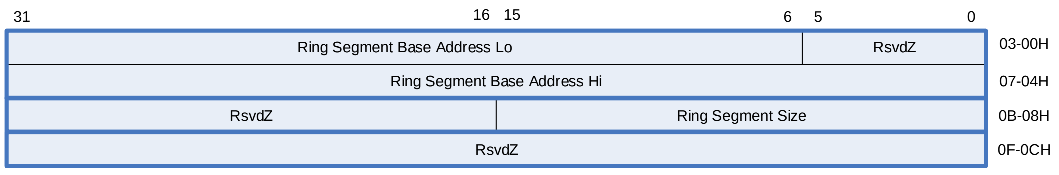


Table 6-89: Offset 00 and 04 – Event Ring Segment Table Entry Field Definitions

Bits	Description
5:0	RsvdZ.
63:6	**Ring Segment Base Address Hi and Lo.** These fields represent the high order bits of the 64-bit base address of the Event Ring Segment. The memory structure referenced by this physical memory pointer shall begin on a 64-byte address boundary.

Table 6-90: Offset 08 – Event Ring Segment Table Entry Field Definitions

Bits	Description
15:0	**Ring Segment Size.** This field defines the number of TRBs supported by the ring segment, Valid values for this field are 16 to 4096, i.e. an Event Ring segment shall contain at least 16 entries.
32:16	**RsvdZ**.

Note: The *Ring Segment Size* may be set to any value from 16 to 4096, however software shall allocate a buffer for the Event Ring Segment that rounds up its size to the nearest 64B boundary to allow full cache-line accesses.

6.6 Scratchpad Buffer Array

The *Scratchpad Buffer Array* is used to define the locations of statically allocated memory pages that are available for the private use of the xHC.

The location of the *Scratchpad Buffer Array* is defined by entry 0 of the *Device Context Base Address Array* (6.1).

The size of the *Scratchpad Buffer Array* is defined by the *Max Scratchpad Buffers* field in the HCSPARAMS2 Register (5.3.4).

Table 6-91 defines the properties of a single *Scratchpad Buffer Array* element. All elements in the *Scratchpad Buffer Array* are identical. Refer to section 4.20 for more information.

Table 6-91: Scratchpad Buffer Array Element Field Bit Definitions

Bit	Description
11:0	**RsvdZ**.
PSZ:12	**RsvdZ**. Valid values for PSZ are 12 to 20, depending on the value of PAGESIZE. Note if PAGESIZE = 4K, then this field is zero bits wide. Refer to section 6.6.1 for how PSZ is calculated. If PSZ = 12, then no bits are reserved by this field.
63:PSZ	**Scratchpad Buffer Base Address – RW. Default = ‘0’**. This field contains bits 63 to PSZ of a pointer to a *Scratchpad Buffer*. The actual number of bits used for the *Scratchpad Buffer Base Address* field depends on the value of the PAGESIZE register. If PAGESIZE = 4K then bits 31-12 of the *Scratchpad Buffer Base Address* field are valid, if PAGESIZE = 8K then bits 31-13 of the *Scratchpad Buffer Base Address* field are valid, and so on. Valid values for PSZ are 12 to 20.

6.6.1 PSZ

The *Page Size* register determines the low-order boundary of the *Scratchpad Buffer Base Address* field of a *Scratchpad Buffer Array Element*. This boundary is referred to as “**PSZ**”. The calculation of the PSZ bit offset equals the *Page Size* bit offset + 12. For example, if the Page Size register defines a 4K system page size, then the bit offset of PSZ = 12, if the Page Size register defines a 16K system page size, then the bit offset of PSZ = 14.

7 xHCI Extended Capabilities

The xHC exports xHCI-specific extended capabilities utilizing a method similar to the PCI extended capabilities. If an xHC implements any extended capabilities, it specifies a non-zero value in the *xHCI Extended Capabilities Pointer (xECP)* field of the HCCPARAMS1 register (5.3.6). This value is an offset into xHC MMIO space from the *Base*, where the *Base* is the beginning of the host controller's MMIO address space. Each capability register has the format illustrated in Table 7-1.

Table 7-1: Format of xHCI Extended Capability Pointer Register

Bit	Description
7:0	**Capability ID – RO.** This field identifies the xHCI Extended capability. Refer to Table 7-2 for a list of the valid xHCI extended capabilities.
15:8	**Next xHCI Extended Capability Pointer – RO.** This field points to the xHC MMIO space offset of the next xHCI extended capability pointer. A value of 00h indicates the end of the extended capability list. A non-zero value in this register indicates a relative offset, in Dwords, from this Dword to the beginning of the next extended capability. For example, assuming an effective address of this data structure is 350h and assuming a pointer value of 068h, we can calculate the following effective address: 350h + (068h << 2) -> 350h + 1A0h -> 4F0h
31:16	**Capability Specific.** The definition and attributes of these bits depends on the specific capability.

Table 7-2: xHCI Extended Capability Codes

ID	Name	Description	Size	Section
0	Reserved			
1	USB Legacy Support	This capability provides the xHCI Pre-OS to OS Handoff Synchronization support capability.	8B	7.1
2	Supported Protocol	This capability enumerates the protocols and revisions supported by this xHC. At least one of these capability structures is required for all xHC implementations.	12B	7.2
3	Extended Power Management	This capability is required for all xHC non-PCI implementations.	Refer to PCI PM spec.	7.3

4	I/O Virtualization	This capability is optional-normative for xHC implementations that require hardware virtualization support.	Up to 1280B	7.7
5	Message Interrupt	Either this or the *xHCI Extended Message Interrupt* capability is required for all xHC non-PCI implementations.	Refer to PCI spec.	7.5
6	Local Memory	This capability is optional-normative for xHC implementations that require local memory support.	Up to 4TB	7.8
7-9	Reserved			
10	USB Debug Capability	This capability is optional-normative for xHC implementations and describes the xHCI USB Debug Capability.	56B	7.6
11-16	Reserved			
17	Extended Message Interrupt	Either this or the *xHCI Message Interrupt* capability is required for all xHC non-PCI implementations.	Refer to PCI spec.	7.4
18-191	Reserved			
192-255	Vendor Defined	These IDs are available for vendor specific extensions to the xHCI.	Vendor defined	

7.1 USB Legacy Support Capability

The USB Legacy Support provided by the xHC is optional normative functionality that is applicable to pre-OS software (BIOS) and the operating system for the coordination of ownership of the xHC.

This capability is chained through the xHCI Extended Capabilities Pointer (xECP) field and resides in MMIO space.

Table 7-3: HC Extended Capability Registers

Configuration Offset	Mnemonic	Register	Power Well	Register Access
xECP+0h	USBLEGSUP	USB Legacy Support Capability Register	Aux Power	RO, RWS
xECP+4h	USBLEGCTLSTS	USB Legacy Support Control and Status Register	Aux Power	RWS, RW1CS

The xECP field is in the HCCPARAMS1 register, refer to Section 5.3.6.

Note: The *USB Legacy Support Capability* registers reside in the Aux Power well. Refer to section 4.22.1 for reset conditions.

7.1.1 USB Legacy Support Capability (USBLEGSUP)

Offset:	xECP + 00h
Default Value:	Implementation Dependent
Attribute:	RO, RW
Size:	32 bits

This register is an xHCI extended capability register. It includes a specific function section and a pointer to the next xHCI Extended Capability. This register is used by pre-OS software (BIOS) and the operating system to coordinate ownership of the xHC. This register is in the Aux Power well.

Table 7-4: USB Legacy Support Extended Capability (USBLEGSUP)

Bit	Description
7:0	**Capability ID – RO.** This field identifies the extended capability. Refer to Table 7-2 for the value that identifies the capability as Legacy Support. This extended capability requires one additional 32-bit register for control/status information (USBLEGCTLSTS), and this register is located at offset xECP+04h.
15:8	**Next Capability Pointer - RO**. This field indicates the location of the next capability with respect to the effective address of this capability. Refer to Table 7-1 for more information on this field.
16	**HC BIOS Owned Semaphore – RW.** Default = ‘0’. The BIOS sets this bit to establish ownership of the xHC. System BIOS will set this bit to a ‘0’ in response to a request for ownership of the xHC by system software.
23:17	**RsvdP**.

24	**HC OS Owned Semaphore – RW.** Default = '0'. System software sets this bit to request ownership of the xHC. Ownership is obtained when this bit reads as '1' and the *HC BIOS Owned Semaphore* bit reads as '0'.
31:25	**RsvdP**.

Note: To support the BIOS's and OS's ability to modify the Owned Semaphores independently,
Byte (8-bit) accesses shall be supported by this register.

7.1.2 USB Legacy Support Control/Status (USBLEGCTLSTS)

Offset:	xECP + 04h
Default Value:	0000 0000h
Attribute:	RO, RW, RW1C
Size:	32 bits

Pre-OS (BIOS) software uses this register to enable System Management Interrupts (SMIs) for every xHCI/USB event it needs to track. Bits [21:16] of this register are simply shadow bit of USBSTS register [5:0]. This register is in the Aux Power well.

Table 7-5: USB Legacy Support Control/Status (USBLEGCTLSTS)

Bit	Description
0	**USB SMI Enable – RW.** Default = '0'. When this bit is a '1', and the *SMI on Event Interrupt* bit (below) in this register is a '1', the host controller will issue an SMI immediately.
3:1	**RsvdP**.
4	**SMI on Host System Error Enable – RW.** Default = '0'. When this bit is a '1', and the *SMI on Host System Error* bit (below) in this register is a '1', the host controller will issue an SMI immediately.
12:5	**RsvdP**.
13	**SMI on OS Ownership Enable – RW.** Default = '0'. When this bit is a '1' AND the OS Ownership Change bit is '1', the host controller will issue an SMI.
14	**SMI on PCI Command Enable – RW.** Default = '0'. When this bit is '1' and SMI on PCI Command is '1', then the host controller will issue an SMI.
15	**SMI on BAR Enable – RW. Default** = '0'. When this bit is '1' and SMI on BAR is '1', then the host controller will issue an SMI.

16	**SMI on Event Interrupt – RO.** Default = ‘0’. Shadow bit of *Event Interrupt* (EINT) bit in the USBSTS register. Refer to Section 5.4.2 for definition. This bit follows the state the *Event Interrupt* (EINT) bit in the USBSTS register, e.g. it automatically clears when EINT clears or set when EINT is set.
19:17	**RsvdP**.
20	**SMI on Host System Error – RO**. Default = ‘0’. Shadow bit of *Host System Error* (HSE) bit in the USBSTS register refer to Section 5.4.2 for definition and effects of the events associated with this bit being set to ‘1’. To clear this bit to a ‘0’, system software shall write a ‘1’ to the *Host System Error* (HSE) bit in the USBSTS register.
28:21	**RsvdZ**.
29	**SMI on OS Ownership Change – RW1C.** Default = ‘0’. This bit is set to ‘1’ whenever the *HC OS Owned Semaphore* bit in the USBLEGSUP register transitions from ‘1’ to a ‘0’ or ‘0’ to a ‘1’.
30	**SMI on PCI Command – RW1C.** Default = ‘0’. This bit is set to ‘1’ whenever the PCI Command Register is written.
31	**SMI on BAR – RW1C.** Default = ‘0’. This bit is set to ‘1’ whenever the Base Address Register (BAR) is written.

Note: For all enable register bits, ‘1’ = Enabled, ‘0’ = Disabled.

Note: SMI – System Management Interrupt.

Note: BAR – Base Address Register.

Note: MSE – Memory Space Enable.

Note: SMI’s are independent of the interrupt threshold value.

7.2 xHCI Supported Protocol Capability

At least one of these capability structures is required for all xHCI implementations. More than one may be defined for implementations that support more that one bus protocol. Refer to section 4.19.7 for more information.

Figure 7-1: xHCI Supported Protocol Capability

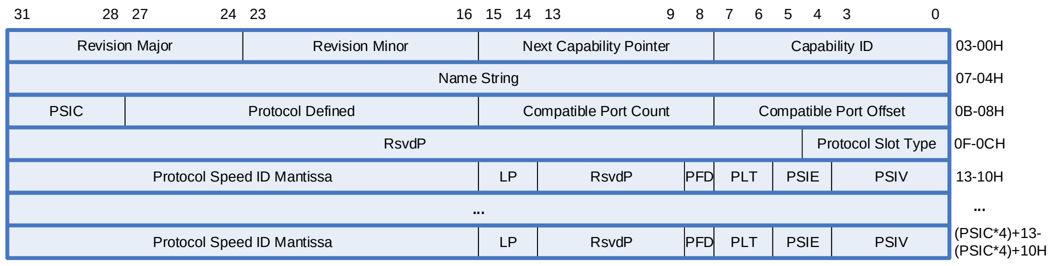


Table 7-6: Offset 00h - xHCI Supported Protocol Capability Field Definitions

Bits	Description
7:0	**Capability ID – RO.** Refer to Table 7-2 for the value that identifies the capability as Supported Protocol.
15:8	**Next Capability Pointer – RO.** This field indicates the location of the next capability with respect to the effective address of this capability. Refer to Table 7-1 for more information on this field.
23:16	**Minor Revision – RO.** Minor Specification Release Number in Binary-Coded Decimal (i.e., version x.10 is 10h). This field identifies the minor release number component of the specification with which the xHC is compliant.
31:24	**Major Revision – RO.** Major Specification Release Number in Binary-Coded Decimal (i.e., version 3.x is 03h). This field identifies the major release number component of the specification with which the xHC is compliant.

Table 7-7: Offset 04h - xHCI Supported Protocol Capability Field Definitions

Bits	Description
31:0	**Name String – RO.** This field is a mnemonic name string that references the specification with which the xHC is compliant. Four ASCII characters may be defined. Allowed characters are: alphanumeric, space, and underscore. Alpha characters are case sensitive. Refer to section 7.2.2 for defined values.

Table 7-8: Offset 08h - xHCI Supported Protocol Capability Field Definitions

Bits	Description
7:0	**Compatible Port Offset – RO.** This field specifies the starting Port Number of Root Hub Ports that support this protocol. Valid values are '1' to MaxPorts.
15:8	**Compatible Port Count – RO.** This field identifies the number of consecutive Root Hub Ports (starting at the *Compatible Port Offset*) that support this protocol. Valid values are 1 to MaxPorts.
27:16	**Protocol Defined.** This field is reserved for protocol specific definitions. Refer to section 7.2.2.1.3.
31:28	**Protocol Speed ID Count (PSIC) – RO**. This field indicates the number of *Protocol Speed ID* (PSI) Dwords that the *xHCI Supported Protocol Capability* data structure contains. If this field is non-zero, then all speeds supported by the protocol shall be defined using *PSI* Dwords, i.e. no *implied* Speed ID mappings apply. Refer to section 7.2.2 and its subsections for protocol specific requirements related to this field.

Note: An xHCI Supported Protocol Capability shall not reference a Root Hub port number referenced by another xHCI Supported Protocol Capability.

Table 7-9: Offset 0Ch - xHCI Supported Protocol Capability Field Definitions

Bits	Description
4:0	**Protocol Slot Type[115] – RO**. This field specifies the *Slot Type* value which may be specified when allocating Device Slots that support this protocol. Valid values are '0' to '31'. Refer to sections 4.6.3 and 7.2.2.1.4.
31:5	**RsvdP**.

[115]The value of the *Protocol Slot Type* field declared by a *xHCI Supported Protocol Capability* structure is unique to an xHC implementation. Software shall not assume a fixed mapping of the *Protocol Slot Type* value to a specific type of Supported Protocol.Note that for compatibility reasons, the Pr*otocol Slot Type* value of '0' is the exception to this rule and reserved for the USB Protocol De*vice Slot* type.

7.2.1 Protocol Speed ID (PSI)

Protocol Speed ID (PSI) Dwords immediately follow the Dword at offset 10h in an *xHCI Supported Protocol Capability* data structure. Table 7-10 defines the fields of a *PSI* Dword.

Table 7-10: Offset 10h to (PSIC*4)+10h - xHCI Supported Protocol Capability Field Definitions

Bits	Description
3:0	**Protocol Speed ID Value (PSIV) – RO**. If a device is attached that operates at the bit rate defined by this *PSI* Dword, then the value of this field shall be reported in the *Port Speed* field of PORTSC register (5.4.8) of a compatible port. Note, the *PSIV* value of '0' is reserved and shall not be defined by a PSI.
5:4	**Protocol Speed ID Exponent (PSIE) – RO**. This field defines the base 10 exponent times 3, that shall be applied to the *Protocol Speed ID Mantissa* when calculating the maximum bit rate represented by this *PSI* Dword. PSIE Value Bit Rate 0 Bits per second 1 Kb/s 2 Mb/s 3 Gb/s
7:6	**PSI Type (PLT) – RO**. This field identifies whether the *PSI* Dword defines a symmetric or asymmetric bit rate, and if asymmetric, then this field also indicates if this Dword defines the receive or transmit bit rate. Note that the Asymmetric PSI Dwords shall be paired, i.e. an Rx immediately followed by a Tx, and both Dwords shall define the same value for the *PSIV*. PLT Value Bit Rate Note 0 Symmetric Single PSI Dword 1 Reserved 2 Asymmetric Rx Paired with Asymmetric Tx PSI Dword 3 Asymmetric Tx Immediately follows Rx Asymmetric PSI Dword
8	**PSI Full-duplex (PFD) – RO**. If this bit is '1' the link is full-duplex (dual-simplex), and if '0' the link is half-duplex (simplex).
13:9	**RsvdP**.
15:14	Link Protocol (LP) - RO. if *xHCI Protocol Extended Capability:Major Revision* = 03h, then this field identifies the link-level protocol supported by the ports associated with this PSI Dword. Refer to section 8.5.6.7 in the USB3 spec for more information. If *xHCI Protocol Extended Capability:Major Revision* = 02h, then this field shall be '0', and the link protocol (LS, FS, or HS) depends on the reported link speed. **LP Value** **Protocol** 0 SuperSpeed 1 SuperSpeedPlus 3-2 Reserved

31:16	**Protocol Speed ID Mantissa (PSIM) – RO**. This field defines the mantissa that shall be applied to the *PSIE* when calculating the maximum bit rate represented by this *PSI* Dword.

Note: An xHC implementation that employs an Integrated Hub to provide USB Full-speed and Low-speed support and only provided a USB 2.0 High-speed BI may define a USB2 *xHCI Supported Protocol Capability* data structure with a single *PSI* Dword (*PSIC* = 1), where the *PSI* Dword at offset 0Ch would define *PSIV* = 3, *PLT* = 0, *PFD* = 0, *PSIE* = 2, and *PSIM* = 480.

Note: If the PSI Exponent (PSIE) and Mantissa (PSIM) fields do not allow the exact definition of a protocol's bit rate, then the PSIM should be rounded to the closest value.

Note: The "symmetry" of a port is determined by the current *PSI Type* (PLT). To determine the current *PSI Type*, inspect the value reported by the PORTSC *Speed* field. If the *Speed* value refers to a PSI Dword whose *PSI Type* = *Symmetric*, then the receive and transmit speed and lane counts are identical, i.e. the PSI Dword defines the speed of the port and the USB3 PORTLI *RLC* and *TLC* fields shall report identical values. If the *Speed* value refers to a PSI Dword whose *PSI Type* = *Asymmetric*, then the receive and transmit speeds and/or the lane counts of the port may be different. The PSI Dword with *PLT* = *Asymmetric Rx* identifies the speed of the ports' receive path and the USB3 PORTLI *RLC* identifies the lane count of the receive path, and the PSI Dword with *PLT* = *Asymmetric Tx* identifies the speed of the ports' transmit path and the USB3 PORTLI *TLC* field identifies the lane count of the transmit path. An Asymmetric port may report the same speed in both directions, but different lane counts. Refer to section 5.4.10.1 for more information on the PORTLI register.

7.2.2 Supported Protocols

Table 7-11 lists the Supported Protocols defined in this specification.

Table 7-11: xHCI Supported Protocols

Name String	Major Revision	Minor Revision[116]	Specification Reference
"USB " or 20425355h	03h	10h	USB 3.1 specification (USB3)
"USB " or 20425355h	03h	00h	USB 3.0 specification (USB3)

[116] The Major and Minor Revision fields implement the same BCD format as described in Section 9.6.1 of the spec for the bcdUSB field.

"USB " or 20425355h	02h	00h	USB 2.0 specification (USB2)

Note: One xHCI Supported Protocol Capability shall define a *Compatible Port Offset* of '1'.

Note: Gaps are allowed in the port numbers assigned by xHCI Supported Protocol Capabilities, e.g. the *Compatible Port Offset* of a xHCI Supported Protocol Capability may not be equal to the sum of the *Compatible Port Offset* and *Compatible Port Count* fields of the previous xHCI Supported Protocol Capability.

Note: Multiple xHCI Supported Protocol Capabilities of the same type (i.e. identical Name String, Major Revision, Minor Revision) may be declared by an xHCI implementation, however the port numbers assigned by them shall not overlap.

Note: Undefined behavior may occur if software references Root Hub port numbers not defined by xHCI Supported Protocol Capabilities.

Note: The *Major Revision* and *Minor Revision* fields contain a BCD version number. The value of the *Major Revision* field is JJh and the value of the *Minor Revision* field is MNh for version JJ.M.N, where JJ = major revision number, M - minor version number, N = sub-minor version number, e.g. version 2.1.3 is represented with the value 0213 and version 3.1 is represented with a value of 0310h. The intent is to follow the USB3 spec (section 9.6.1) defined format for the Standard Device Descriptor bcdUSB field.

7.2.2.1 USB Protocols

The following subsection define *xHCI Supported Protocol Capability* extensions that are specific to USB protocols.

Note: The set of ports defined by a USB3 xHCI Supported Protocol Capability shall not overlap those defined by a USB2 xHCI Supported Protocol Capability, and vice versa.

Note: To support USB3 device certification requirements for USB 2 user attached devices, USB 2.0 and USB 3.x Supported Protocol Capabilities shall be declared if any USB3 connectors are associated with xHCI Root Hub ports that enable user attached devices. Refer to sections 11.1 and 11.3 in the USB3 spec.

PSI Dwords shall be used to define the bit rate associated with an SSIC Profile. Table 7-12 provides an example of values that define an SSIC implementation capable of supporting HS-GEAR 1, 2, or 3 and Rate Series A or B speeds in each GEAR. Also notice that in each case the protocol on the wire is USB3 and that the SSIC-*g*B-L*l* (i.e. Series B) *PSIM* values are rounded to the nearest value. Refer to section 2.2.1 in the SSIC Spec for more information.

Table 7-12: Example SSIC PSI Dword values

SSIC Profile	Bit Rate (Mb/s)	Protocol[117]	*PSI* Dword values				
			PSIV	PLT	PFD	PSIE	PSIM
SSIC-G1A-L1	1248	USB 3.0	1	0	1	2	1248
SSIC-G2A-L1	2496	USB 3.0	2	0	1	2	2496
SSIC-G3A-L1	4992	USB 3.0	3	0	1	2	4992
SSIC-G1B-L1	1457.6	USB 3.0	4	0	1	2	1458
SSIC-G2B-L1	2915.2	USB 3.0	5	0	1	2	2915
SSIC-G3B-L1	5830.4	USB 3.0	6	0	1	2	5830

7.2.2.1.1 Default USB Speed ID Mapping

The following default mappings apply to the USB2 and USB3 protocols.

Table 7-13: Default USB Speed ID Mapping

Default Speed ID Value[118]	Definition	Bit Rate	Protocol	Equivalent *PSI* Dword values			
				PLT	PFD	PSIE	PSIM
1	Full-speed	12 MB/s	USB 2.0	0	0	2	12
2	Low-speed	1.5 Mb/s	USB 2.0	0	0	1	1500
3	High-speed	480 Mb/s	USB 2.0	0	0	2	480
4	SuperSpeed	5 Gb/s	USB 3.x	0	1	3	5
5	SuperSpeedPlus	10 Gb/s	USB 3.1	0	1	3	10

[117]Refer to the SSIC spec for the specific protocol requirements of SSIC ports, e.g. and SSIC port may support a SuperSpeed protocol (i.e. 3.0), an Enhanced SuperSpeed protocol, e,g, 3.1, etc.

[118] The Default Speed ID Values shall be presented in PORTSC *Port Speed* field only if no PSI Dwords are defined (PSIC = ‘0’).

7.2.2.1.2 Protocol Speed ID Count (PSIC) field

USB *xHCI Supported Protocol Capability* data structures may define *PSIC* = ‘0’ field under the following conditions:

- For a **USB 3.1** x*HCI Supported Protocol Capability* data structure (i.e. *Name String* - 20425355h, *Major Revision* = 03h, and Minor Revision = 10h) a PSIC value of ‘0’ implies that only the default SuperSpeed and SuperSpeedPlus bit rates are supported. Refer to Table 7-13 for default USB 3.1 Speed ID mappings.
- For a **USB 3.0** *xHCI Supported Protocol Capability* data structure (i.e. *Name String* = 20425355h, *Major Revision* = 03h, and *Minor Revision* = 00h) a *PSIC* value of ‘0’ implies that only the default SuperSpeed bit rate is supported. Refer to Table 7-13 for default USB 3.0 Speed ID mappings.
- For a **USB 2.0** *xHCI Supported Protocol Capability* data structure (i.e. *Name String* = 20425355h, *Major Revision* = 02h, and *Minor Revision* = 00h) a *PSIC* value of ‘0’ implies that the default Full-speed, Low-speed, and High-speed bit rates are supported. Refer to Table 7-13 for default USB 2.0 Speed ID mappings.
- Only these two protocols/revisions support implied mappings. All other protocols or revisions of these protocols and SSIC ports shall define a non-zero *PSIC* value.

7.2.2.1.3 Protocol Defined field

The *Protocol Defined* field only applies to the specific protocol referenced by its *xHCI Supported Protocol Capability*. This section identifies how the *Protocol Defined* field applies to each of the protocols defined in this specification.

7.2.2.1.3.1 USB3

The following Protocol Defined fields are defined by a USB3 xHCI Supported Protocol Capability.

All USB3 ports shall support Link Power Management.

Figure 7-2: USB3 Protocol Defined fields

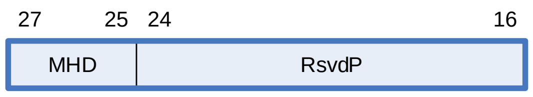


Table 7-14: USB3 Protocol Defined Field Definitions

Bits	Description
24:16	**RsvdP**.
27:25	**Hub Depth (MHD) - RO**. Default = Implementation dependent. If this field is '0', then the standard USB3 hub depth constrains apply, if this field is > '0', then it indicates the maximum hub depth supported by the USB3 ports.

7.2.2.1.3.2 USB2

The following Protocol Defined fields are defined by a USB 2.0 xHCI Supported Protocol Capability.

Figure 7-3: USB 2.0 Protocol Defined fields

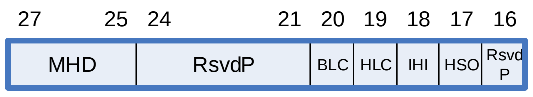


Table 7-15: USB 2.0 Protocol Defined Field Definitions

Bits	Description
16	**RsvdP**.
17	**High-speed Only (HSO) - RO**. Default = Implementation dependent. If this bit is cleared to '0', the USB2 ports described by this capability are Low-, Full-, and High-speed capable. If this bit is set to '1', the USB2 ports described by this capability are High-speed only, e.g. the ports don't support Low- or Full-speed operation. High-speed only implementations may introduce a "Tier mismatch", refer to section 4.24.2 for more information.
18	**Integrated Hub Implemented (IHI) - RO**. Default = Implementation dependent. If this bit is cleared to '0', the Root Hub to External xHC port mapping adheres to the default mapping described in section 4.24.2.1. If this bit is set to '1', the Root Hub to External xHC port mapping does not adhere to the default mapping described in section 4.24.2.1, and an ACPI or other mechanism is required to define the mapping.
19	**Hardware LPM Capability (HLC) - RO**. Default = Implementation dependent. If this bit is set to '1', the ports described by this xHCI Supported Protocol Capability support hardware controlled USB2 Link Power Management. Refer to section 4.23.5.1.1.1.
20	**BESL LPM Capability[119] (BLC) - RO**. Default = Implementation dependent. If this bit is set to '1', the ports described by this xHCI Supported Protocol Capability shall apply BESL timing to *BESL* and *BESLD* fields of the PORTPMSC and PORTHLPMC registers, as defined in Table 13. If this bit is cleared to '0', the ports described by this xHCI Supported Protocol Capability shall apply HIRD timing to *BESL* and *BESLD* fields of the PORTPMSC and PORTHLPMC registers, as defined in Table 13. Refer to section 4.23.5.1.1.1 for more information. Note the BESL LMP Capability support (i.e. *HLE* = '1' and *BLC* = '1') shall be mandatory for all xHCI 1.1 compliant xHCs.

[119]In 2007, an ECN to the USB spec defined the "USB 2.0 Link Power Management Addendum". This ECN added the concept of an LPM Token and *Host Initiated Resume Duration* (HIRD) to the USB2 spec to support better link power management. And in 2011, the "Errata for USB 2.0 ECN: Link Power Management (LPM) - 7/2007" was

24:20	**RsvdP**.
27:25	**Hub Depth (MHD) - RO**. Default = Implementation dependent. If this field is '0', then the standard USB2 hub depth constrains apply, if this field is > '0', then it indicates the maximum hub depth supported by the USB2 ports.

7.2.2.1.4 Protocol Slot Type Field

The Protocol Slot Type field of a USB3 or USB2 xHCI Supported Protocol Capability shall be set to '0'.

7.3 HCI Extended Power Management Capability

This capability is required for all xHC implementations that do not support PCI based system interfaces.

The *xHCI Extended Power Management Capability* shall utilize the format of the *Power Management Register Block Definition* defined in section 3.2 of the PCI PM Specification with the following exception. For xHCI the definition of the "Next Capability Pointer" register field is modified from the PCI definition. A non-zero value in the "Next Capability Pointer" register indicates a relative offset, in 32-bit words, from this 32-bit word to the beginning of the first extended capability.

Note: Refer to section 5.2.7 for details on register definition and structure organization.

7.4 xHCI Extended Message Interrupt Capability

Either this capability or the *xHCI Message Interrupt Capability* is required for all xHC implementations that do not support PCI based system interfaces. The choice is xHC implementation dependent.

The *xHCI Extended Message Interrupt Capability* shall utilize the format of the *MSI-X Capability and Table* Structures defined in section 6.8.2 of the PCI Specification with the following exception. For xHCI the definition of the "Next Capability Pointer" register field is modified from the PCI definition. A non-zero value in the "Next Capability Pointer" register indicates a relative offset, in 32-bit words, from this 32-bit word to the beginning of the first extended capability.

generated to address some shortcomings of the original ECN, which redefined the HIRD field of the LPM Token to be Best Effort Service Latency (BESL). The *BESL LPM Capability* flag in the xHCI Supported Protocol Capability identifies whether an xHCI implementation supports the pre- or post-errata USB2.0 LPM definition.A key aspect of the LPM Errata is that it makes a distinction between the Bes*t Effort Service Latency* that a device should expect, and the H*ost Initiated Resume Delay* that will be signaled on the bus to exit the L1 state.

Note: Refer to section 5.2.8 for details on register definition and structure organization.

7.5 xHCI Message Interrupt Capability

Either this capability or the *xHCI Extended Message Interrupt Capability* is required for all xHC implementations that do not support PCI based system interfaces. The choice is xHC implementation dependent.

The *xHCI Message Interrupt Capability* shall utilize the format of the *MSI Capability* Structure defined in section 6.8.1 of the PCI Specification with the following exception. For xHCI the definition of the "Next Capability Pointer" register field is modified from the PCI definition. A non-zero value in the "Next Capability Pointer" register indicates a relative offset, in 32-bit words, from this 32-bit word to the beginning of the first extended capability.

Note: Refer to section 5.2.8 for details on register definition and structure organization.

7.6 Debug Capability (DbC)

The USB **Debug Capability** provided by the xHC is optional functionality that enables low-level system debug over USB. The xHCI debugging capability provides a means of connecting two systems where one system is a **Debug Host** and the other a **Debug Target** (System Under Test).

This section describes the xHCI USB Debug Capability used by a Debug Target to present a **Debug Device** to a Debug Host. A Debug Device is fully compliant with the USB Framework. A Debug Device provides the equivalent of a very high performance full-duplex serial link between a Debug Host and a Debug Target.

The USB Debug Capability provides an interface that is completely independent of the xHCI interface described in the other sections of this specification. This section describes the required implementation and behavior of a USB3 Debug Capability as part of an xHCI compatible controller. Specific features of the xHCI USB Debug Capability are:

- The interface provided by the xHCI USB Debug Capability is independent of the standard xHCI interface utilized by the Operating System, e.g The USBCMD register *R/S* flag has no effect on the operation of the Debug Capability.
- If *DbC System Bus Reset* (SBR) = '0', then a Chip Hardware Reset or the assertion of *Host Controller Reset* (HCRST = '1') or *Light Host Controller Reset* (LHCRST = '1') shall reset the Debug Capability, or optionally if *SBR* = '1', then a Chip Hardware Reset, a System Bus (e.g. the assertion of PCI RST#), or a transition from the PCI PM D3hot state to the D0 state shall reset the DbC.
- Only works with a SuperSpeed capable host.

- The Debug Capability is automatically assigned to the first xHCI Root Hub Port on that detects an attach of the downstream facing port of a SuperSpeed capable Root Hub or an external Hub.
- The Root Hub port assigned to the Debug Capability appears through the xHCI as a fully functional Root Hub port that never sees a device attach.
- The Debug Capability is operational anytime the port is not suspended AND the host controller is in D0 power state.
- The Debug Capability works through standard USB3 Hubs, allowing large numbers of systems to be debugged with a single host.
- High bandwidth data transfers are supported.

This capability is chained through the xHCI Extended Capabilities Pointer (xECP) field and resides in MMIO space.

Wherever possible, the Debug Capability attempts to reuse logic blocks defined for the xHCI architecture. For instance, the operation and definition of the Debug Capability Event Ring Management register block is identical to the xHCI Event Ring Registers defined in section 5.5.2.3, except that it provides an Event Ring that is dedicated to the Debug Capability.

Because the Debug Capability presents a “device side” interface to USB, which is used to manage the upstream facing port of a device rather than the downstream facing port of a Root Hub, some of the register definitions in the Debug Capability may appear to be very similar to those in the xHCI, however they may have subtle differences to support “device side” operation. e.g. Many of the fields in the Debug Capability DCPORTSC Register are named the same as fields in the xHCI PORTSC register, however they work differently because the DCPORTSC register shall manage “device side” operation.

The Debug Capability also utilizes xHCI Endpoint Context data structures, however their organization is different than the xHCI’s.

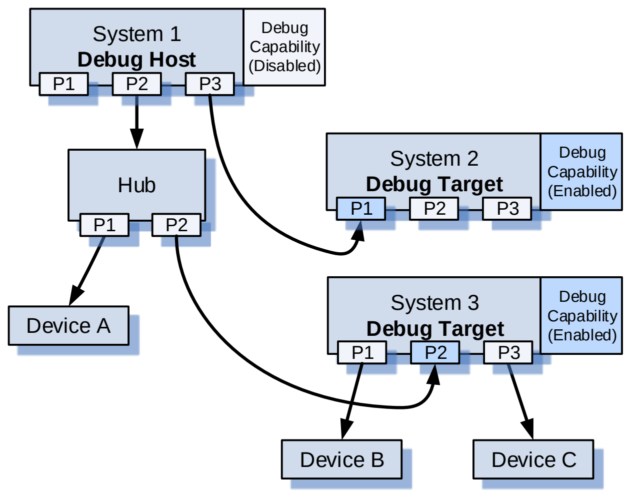
Note: Keep the “device side” difference of the Debug Capability in mind when reading the register definitions in the following sections.

7.6.1 Debugging Topologies

A Debug Target enumerates as “normal” USB device to the Debug Host, allowing a Debug Host to access a Debug Target through the standard USB software stack. Multiple Debug Targets may be attached to a Debug Host. Debug Targets may be connected to any downstream facing port below a Debug Host (i.e. anywhere in the fabric, refer to Figure 7-4). A Debug Target may only connect to a Debug Host through a Root Hub port of the target. Connection of a Debug

Target to a Debug Host through the ports of an external hub controlled by the Debug Target is not supported.

Figure 7-4: Example Debugging Topology



In the example illustrated by Figure 7-4, System 1 is the Debug Host. It is attached to two Debug Targets; Systems 2 and 3. Port 1 (P1) of System 2 is attached to a Root Hub port of System 1 and Port 2 (P2) of System 3 is attached to the downstream facing port of a Hub controlled by System 1. Note that other (non-Debug Target) USB devices may also be attached to a Debug Host or Target system. Device A is attached to System 1, and Devices B and C are attached to System 3. All 3 systems support xHCI Debug Capability hardware, software distinguishes a Debug Target from a Debug Host by enabling the Debug Capability on Targets.

The Debug Host provides a USB Debug Capability class driver, which will manage Debug Targets when they are enumerated and provide an API for debugger applications.

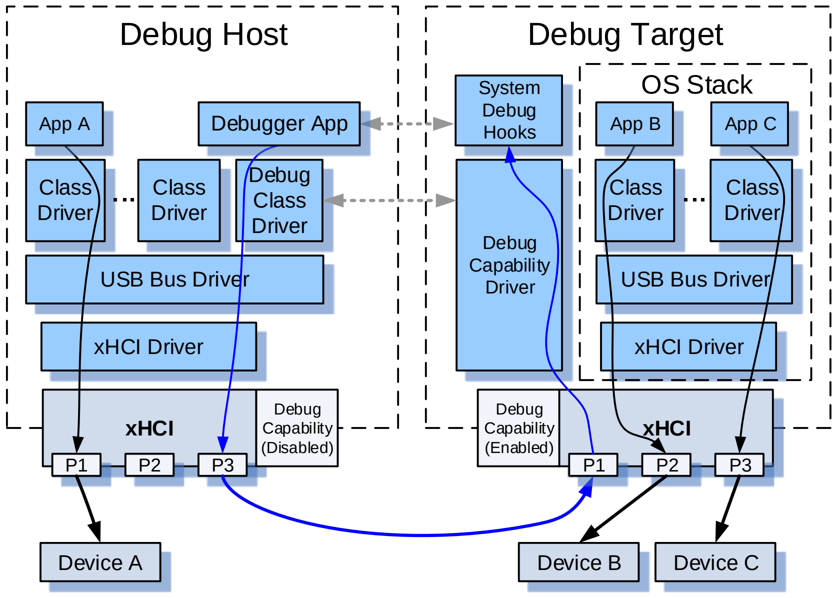
The Debug Target provides software to manage communications between the Debug Device and the Debug Host. The Debug Target software interfaces to the xHCI Debug Capability to manage Debug Device emulation and service Debug Device Class specific requests from the Debug Host.

Note: A Debug Target may only expose its USB Debug Capability through a Root Hub port. A Debug Target *cannot* connect to a Debug Host through the downstream facing port of a hub owned by the Debug Target.

7.6.2 Debug Stacks

Figure 7-5 shows an example of the software stacks in the Debug Host and Debug Target, and their relationships.

Figure 7-5: Example Debug Software Stacks



In Figure 7-5, the Debug Host provides a Debug Class Driver which communicates with the System Debug Hooks in the Debug Target, through the Debug Capability (blue path).

On the Debug Target, the Debug Capability Driver is completely independent of the OS Stack (USB Bus Driver, xHCI driver, etc.). The Debug Capability Driver is expected to be loaded immediately after POST so that the OS stack can be debugged. The Debug Capability Driver manages the xHCI Debug Capability register set, and the standard USB OS stack manages all non-Debug USB devices attached to the system.

On the Debug Host, the xHCI Debug Capability is disabled and there is no driver associated with it. And the standard USB OS stack manages all USB devices attached to the system, including the Debug device presented by the Debug Capability Driver on the Debug Target.

The user interface through which a programmer enables a system's xHCI USB Debug Capability or its features are outside the scope of this specification. The Debug Device Class is defined in section 7.6.10.

7.6.2.1 Debug Software Startup

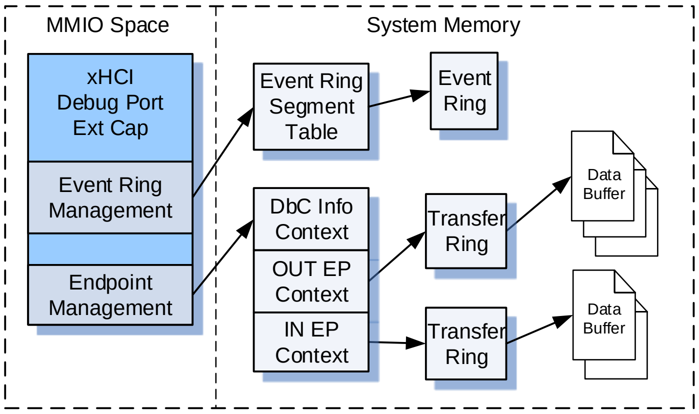
There are two general cases for debug software startup: 1) when the xHC has not been initialized by the system host controller driver, and 2) when the xHC has been initialized by the system host controller driver. Debug software generally knows what case it has to deal with (typically case 1), but can do further determination by examining the *MaxSlotsEn* field in the xHC *CONFIG* register. If the *MaxSlotsEn* field is non-zero, then the system host controller driver has already initialized the xHC. Generic startup procedures for the two cases are the same. Other than being linked into the xHCI Extended Capabilities list the Debug Capability is able to function completely independently of the xHCI interface used by a system host controller driver. As such, it can be initialized before or after the system host controller driver loads. The only effects that the system host controller driver sees is that one of its Root Hub ports will never generate a Port Status Change Event for a connect, and that port shall report no bandwidth available when querying for bandwidth with a *Get Port Bandwidth Command*.

7.6.3 Memory Map

The xHCI Debug Capability register set resides in the xHCI MMIO space. The MMIO space is located through the xHCI Extended Capability chain.

A variety of data structures required by the Debug Capability reside in System Memory and are accessed by the xHC DMA mechanisms. The DbC Structure contains pointers to the memory based data structures that it utilizes.

Figure 7-6: Debug Capability Memory Map



7.6.3.1 ERST and Event Ring

The Debug Capability supports a dedicated Event Ring, which is managed through an *Event Ring Segment Table* data structure. The format and use of the Debug Capability's *Event Ring Segment Table* data structure is identical to the xHCI Event Ring mechanism described in section 6.5. And the Event Ring Segments referenced by the Debug Capability Event Ring Segment Table work identically to those described in section 4.9.3. More information on the use of the Event Ring data structures by the Debug Capability is described in section Event Generation.

The Event Ring Segment Table is pointed to by the *Debug Capability Event Ring Segment Table Base Address Register* described in section Event Ring Segment Table Base Address Register (ERSTBA). The number of entries in the *Event Ring Segment Table* is defined by the *Debug Capability Event Ring Segment Table Size Register* described in section Event Ring Segment Table Size Register (ERSTSZ).

7.6.3.2 Endpoint Contexts and Transfer Rings

The Debug Capability maps all its endpoints to two Transfer Rings. *Endpoint Context* data structures (as described in section 6.2.3) are used to define and manage these Transfer Rings. The Debug Capability *Endpoint Contexts* are organized as a two element array, where element '0' defines an OUT Transfer Ring and the element '1' defines an IN Transfer Ring.

The IN and OUT Bulk endpoints presented by a Debug Device to a Debug Host are cross-coupled to the two OUT and IN Transfer Rings, respectively. This is because the USB Debug Device presented by the Debug Capability shall output data when it receives an IN TP from the Debug Host, and it shall input data when it receives an OUT DP from the Debug Host.

The Debug Capability *Endpoint Contexts* are contained in the *Debug Capability Context* data structure (7.6.9) which is pointed to by the *Debug Capability Context Pointer Register* described in section 7.6.8.7.

Note: xHCI power management effects the DbC. Software should shut down all DbC activity prior to transitioning the xHC a D3 state. If not, undefined behavior may occur.

Software shall initialize the fields of the Endpoint Context as follows:

Max Packet Size = 1024.

Max Burst Size = *Debug Max Burst Size*[120].

EP Type = 2 for the OUT Bulk endpoint and 6 for the IN Bulk endpoint.

TR Dequeue Pointer = for the OUT Bulk endpoint, a pointer to the Transfer Ring that will contain data to be sent to the Debug Host, and for the IN Bulk endpoint, a pointer to the Transfer Ring that will contain buffers which will receive data from the Debug Host

Average TRB Length = initialized to software defined value.

All other fields shall be initialized to '0'.

The Endpoint Context *Interval*, *LSA*, *MaxPStreams*, *Mult*, *HID*, *CErr*, FE, and *Max ESIT Payload* fields do not apply to the DbC.

[120] Note that a DbC implementation may utilize a smaller *Max Burst Size* than set by software.

The *EP State* field shall be updated as described in section 4.8.3.

The DbC shall update the Endpoint Context *TR Dequeue Pointer* field, if the *HOT* or *HIT* flags are set to '1', the DbC Port State Machine exits the DbC-Configured state, or if *SBR* = ‘0’ and *HCRST* is set to '1'.

7.6.4 Operational Model

This section describes the general operational model for the xHCI **Debug Capability** (DbC) interface. This model is managed by the xHCI Debug Capability driver. Each significant operational feature of the Debug Capability is discussed in a separate subsection. Each subsection presents the operational model requirements for the Debug Capability hardware. Where appropriate, recommended system software operational models for features are also presented.

The xHCI Debug Capability Structure (or DbC Structure) is located using the methods described at the beginning of section 7. The DbC Structure (section Debug Capability Structure) defines a set of registers that Debug Target software uses to emulate USB Debug Device to a Debug Host.

The DbC Structure is divided into seven register sets; Capability, Doorbell, Event Ring Management, Control, Status, Port Management, and Endpoint Management. The Capability registers allow the DbC to be linked into the xHCI’s list of Extended Capabilities and define static features of the DbC. The Doorbell and Endpoint Management registers are used to define and manage the Control and Bulk pipes presented by the DbC. The Event Ring Management, Control, and Status registers provide the Debug Capability driver with the means to track and manage the execution of DbC operations.

Note: The DbC shall respond with a ACK TP to a SetFeature(FUNCTION_SUSPEND) Setup Stage request.

7.6.4.1 Debug Capability Initialization

Typically the DbC will be initialized and enabled prior to the Operating System loading on the target system, however it may be enabled at any time. In this section “software” refers to the code that manages the DbC.

In order to initialize the DbC software should perform the following steps:

- Allocate and initialize all DbC memory data structures
  - The DbC *Event Ring Segment Table* and the Event Ring Segments that it points to.
  - The DbC IN and OUT *Endpoint Contexts* and the Transfer Rings that they point to.

- Initialize the *Debug Capability Event Ring Segment Table Size Register* (DCERSTSZ) with number of entries in the *Event Ring Segment Table*.
- Initialize the *Debug Capability Event Ring Segment Table Base Address Register* (DCERSTBA) with the physical memory address of the *Event Ring Segment Table*.
- Initialize the *Debug Capability Event Ring Segment Table Dequeue Pointer Register* (DCERDP) with the physical memory address of the Event Ring Segment pointed to by *Event Ring Segment Table* entry 0.
- Initialize the *Debug Capability Context Pointer* (DCCP) with the physical memory address of the *Debug Capability Context*.
- Set the *Debug Capability Enable* (DCE) bit to '1' in the *Debug Capability Control Register* (DCCTRL).

At this point, the Debug Capability is initialized, the Root Hub ports are looking for an attached Debug Host, and the DCPORTSC register is enabled to report a Debug Host connection.

When a Debug Host connection is detected, a *Port Status Change Event* will be generated on the DbC Event Ring.

To detect the Debug Host connection, or any event generated by the DbC, software shall periodically poll the *Event Ring Not Empty* bit in the *Debug Capability Status Register* (DCST), or evaluate the DbC Event Ring for change in the Event Ring Enqueue Pointer (i.e. a *Cycle* bit change, refer to section 4.9.4 for more information on the Event Ring Enqueue Pointer).

After the Debug Host connection is detected, software shall wait for the Debug Device to be configured by the Debug Host. The transition of the *DbC Run* (DCR) bit to '1' indicates the successful configuration of the Debug Device.

Software shall impose a timeout between the detection of the Debug Host connection and the *DbC Run* transition to '1'. If the *DbC Run* transition takes too long, software may toggle the *DCE* bit to disable then re-enable the DbC to retry the Debug Device enumeration process.

Note: If a Debug Host attempts to attach to a Debug Target before the *DCE* flag is set, both ends of the link shall transition to the *Inactive* state. So a Debug Host should periodically issue a Warm Reset to ports that are *Inactive* to enable a connection to the DbC of the Debug Target.

Note: If the OS code that is being debugged resets the xHC (e.g. asserts HCRST), then the Debug Capability will also be reset. This condition may be detected by the Debug Capability Driver if *DCE* = '0', after having previously been enabled (set to '1'). If this condition occurs, the *Debug Capability Driver* is required to re-initialize the Debug Capability to continue communication with the Debug Host.

Note: The Debug Capability registers should not be accessed while the *Controller Not Ready* (CNR) bit is set.

7.6.4.2 Event Generation

There are four DCPORTSC *status change bits* in the DCPORTSC register *Connect Status Change* (CSC), *Port Reset Change* (PRC), *Port Link State Change* (PLC), and *Port Config Error Change* (CEC), refer to section 7.6.8.6 for more information on these bits.

DCPORTSC *status change bits* may be set due to hardware or software initiated conditions. When set, these bits remain set until cleared by a system software write to the DCPORTSC register with the appropriate *status change bit(s)* set to ‘1’, a Chip Hardware Reset, or disabling the Debug Capability (*DCE* = ‘0’).

All DCPORTSC *status change bits* are ORed together to form an internal Debug Capability *DCPORT Port Status Change Event Generation* variable (DCPSCEG). When a DCPORTSC *status change bit* is set, if the assertion of a *status change bit* results in a ‘0’ to ‘1’ transition of DCPSCEG, the Debug Capability responds by generating a *Port Status Change Event* (as described in section 6.4.2.3).

The *Port ID* field of the *Port Status Change Event TRB* (shown in Figure 6-16) is always ‘0’ for *Port Status Change Events* found on the Debug Capability’s Event Ring.

System software shall acknowledge Debug Capability status change(s) by clearing the respective DCPORTSC *status change bit(s)*. The acknowledgment clears the change state so future status changes may reported.

Note: DbC Event Ring management is performed identically to xHCI Event Ring management, as described in section 4.9.4.

Note: Possible *Completion Codes* for DbC Transfer Event are *Success*, *Stall Error*, *USB Transaction Error*, *Babble Detected Error*[121], *TRB Error*, *Short Packet*, *Undefined Error*, *Event Ring Full Error*, and *Vendor Defined Error* (refer to Table 6-85).

7.6.4.3 Halted DbC Endpoints

If a bulk endpoint is transferring data when its *Halt Out Transfer Ring* (HOT) or *Halt In Transfer Ring* (HIT) flags is set to '1', the following actions shall occur:

[121]Section 8.11.3 of the USB3 spec defines a possible cause of a DPP Error as “Data length in the DPH does not match the actual data payload length”, i.e. a Packet Babble condition. And Table 8-27 states that if a device detects a DPP Error it shall “Discard DP, send an ACK TP with the sequence number of the DP expected (thereby indicating that the DP was not received), the Retry bit set and the number of DPs that the device can receive for this endpoint.” So for a USB3 device, a Packet Babble condition, is not fatal. The USB3 spec is silent in how a device should interpret a TD Babble condition. A DbC shall not generate *Babble Detected Error* due to a Packet Babble condition, however if a TD Babble condition is detected, it may treat it as fatal, generating a *Babble Detected Error* and STALLing the endpoint, or “silently”, i.e. sending an ACK TP with the sequence number of the DP expected and the Retry bit set, then waiting for the host to resend the DP in error. Refer to section 4.10.2.4 for more information on Babble Errors.

- The current value of the *TR Dequeue Pointer* for the endpoint should be written to its Endpoint Context.
- A Transfer Event shall be generated and:
    - The *TRB Pointer* field of the Transfer Event shall reference the Transfer TRB that the error occurred on.
    - The *TRB Transfer Length* field of the Transfer Event may indicate that the Transfer TRB had been partially completed.
    - The *Completion Code* field of the Transfer Event shall indicate *Stall Error*.
    - This Transfer Event shall be generated whether the *IOC* flag was set or not in the associated Transfer TRB.

The *HIT* or *HOT* flags may be set by the DbC hardware if *Data Buffer Error*, *Parameter Error*, *TRB Error*, *Vendor Defined Error*, or *Undefined Error* is detected, or by software.

The reception of a ClearFeature(ENDPOINT_HALT) request by the DbC shall clear the *HIT* or *HOT* flag for the respective endpoint, and shall clear any internal endpoint state, such that the address stored in the *TR Dequeue Pointer* field of the Endpoint Context shall point to the next TRB that shall be executed the next time the doorbell is rung, i.e. the DbC does not support Soft Retry.

The DbC shall not support the *Set Halt Feature* option. Note, section 9.4.5 in the USB3 spec defines the **Set Halt Feature** option, with the statement "The Halt feature may optionally be set with the SetFeature(ENDPOINT_HALT) request", however it does not explicitly define a device's response (i.e. ACK or STALL) to a SetFeature(ENDPOINT_HALT) request if a device chooses not to set Halt feature when it receives the request. It is highly recommended that the DbC respond with an ACK because this is what the USB device compliance tests expect when a SetFeature(ENDPOINT_HALT) request is issued to a devce, irrespective of whether a device supports the *Set Halt Feature* option or not.

Refer to Table 7-22 for more information on the *HOT* and *HIT* flags.

Note: The DbC is not required to advance the Dequeue Pointer of an endpoint to the next TD boundary when the HIT or HOT flag is asserted.

Note: The value of the Endpoint Context *TR Dequeue Pointer* field may not be equal to the value of the last Transfer Event *TRB Pointer* field when a halt condition occurs.

7.6.4.4 DbC-Configured Exit Behavior

There are several conditions which will cause the DbC to exit the DbC-Configured state (i.e. causing the DCCTRL.DCR field to make a ‘1’ to ‘0’ transition and DCCTRL.DRC to be set to ‘1’):

- Debug Host initiates a Warm or Hot USB Reset

- Debug Host issues a SetConfiguration(0) device request
- Timeout occurs in the DbC-Configured state
- USB cable is disconnected
- Debug Capability driver writes ‘0’ to DCPORTSC.PED
- Debug Capability driver writes ‘0’ to DCCTRL.DCE

Note: A port transition from the DbC-Configured state to the DbC-Error state shall also cause the DbC endpoints to transition to the Error state. And when the Debug Host issues reset the endpoints shall transition to the Stopped state.

When the DbC exits the DbC-Configured state or if the *HIT* or *HOT* flags are set to ‘1’, the following actions shall occur:

- If there is a valid Transfer TRB on the Transfer Ring, a Transfer Event shall be generated and:
    - The *TRB Pointer* field of the Transfer Event shall reference the Transfer TRB that the event occurred on.
    - The *TRB Transfer Length* field of the Transfer Event may indicate that the Transfer TRB had been partially completed.
    - The *Completion Code* field of the Transfer Event shall indicate USB Transaction Error.
    - This Transfer Event shall be generated whether the *IOC* flag was set or not in the associated Transfer TRB.
    - The DbC shall advance its TR Dequeue Pointer to reference the next TRB.
- The Endpoint Context shall be written with:
    - The *TR Dequeue Pointer* field set to the address of the next TRB that will be fetched.
    - The *Endpoint State* field reflecting the current endpoint state.

Software may detect the actions described above have occurred by reading the *DCCTRL.DRC* field as ‘1’ and the *Endpoint State* as Disabled. In response, software may read the *TR Dequeue Pointer* field in the Endpoint Context to determine where the DbC will restart the Transfer Ring, or update the *TR Dequeue Pointer* field to point to the next TRB that shall be executed after software clears *DCCTRL.DRC* and rings the doorbell.

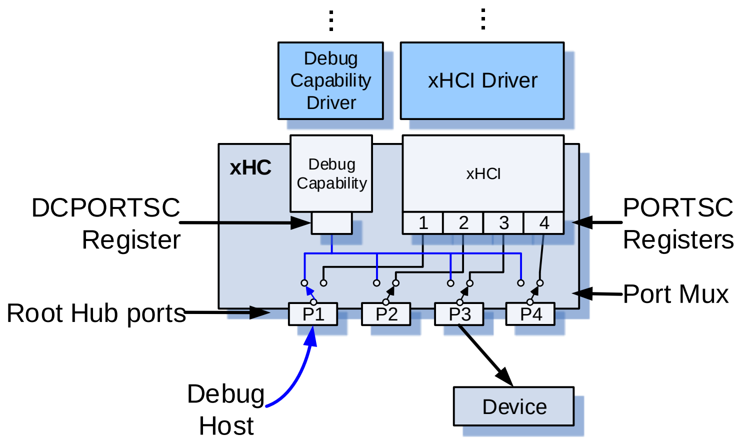
Note: The DbC is not required to advance the Dequeue Pointer of an endpoint to the next TD boundary when exiting the DbC-Configured state.

Normally while the DbC port is in the DbC-Configured state (DCCTRL.DCR = ‘1’) its endpoints are in the Running state. The exception is if the *HIT* or *HOT* flags are asserted, which shall cause a DbC endpoint to transition to the Halted state. And when the Debug Host issues ClearFeature(ENDPOINT_HALT) request, the respective endpoint shall then transition to the Stopped state. When software rings the doorbell, the endpoint shall transition to the Running state.

7.6.5 Port Routing and Control

Figure 7-7 provides a detailed view of the state of the Debug Capability Port Multiplexing mechanism after a Root Hub port (P1) is assigned to the Debug Capability.

Figure 7-7: Debug Port Multiplexing

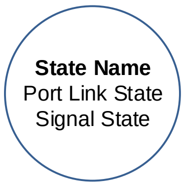


The xHCI Driver accesses the xHCI *Port Status and Control* (PORTSC) registers (5.4.8) and the Debug Capability Driver accesses the *Debug Capability Port Status and Control* (DCPORTSC) register (Debug). When the Root Hub port (P1 in Figure 7-7) is assigned to the Debug Capability, the associated PORTSC register (PORTSC 1 in Figure 7-7) shall mimic operations as if no device is attached it. Refer to section 4.19.1.2.4.3 for the states presented by PORTSC register to system software during this condition.The remaining PORTSC registers are still associated with their respective Root Hub ports and are fully operational through the xHCI.

After the Root Hub port is assigned to the DbC, the xHC shall begin emulating a USB Debug Class device, responding to enumeration related USB requests from the Debug Host, transitioning the Debug Device emulator through the standard USB Device States described in section 8.1 of the USB3 specification.

7.6.6 DbC Port State Machine

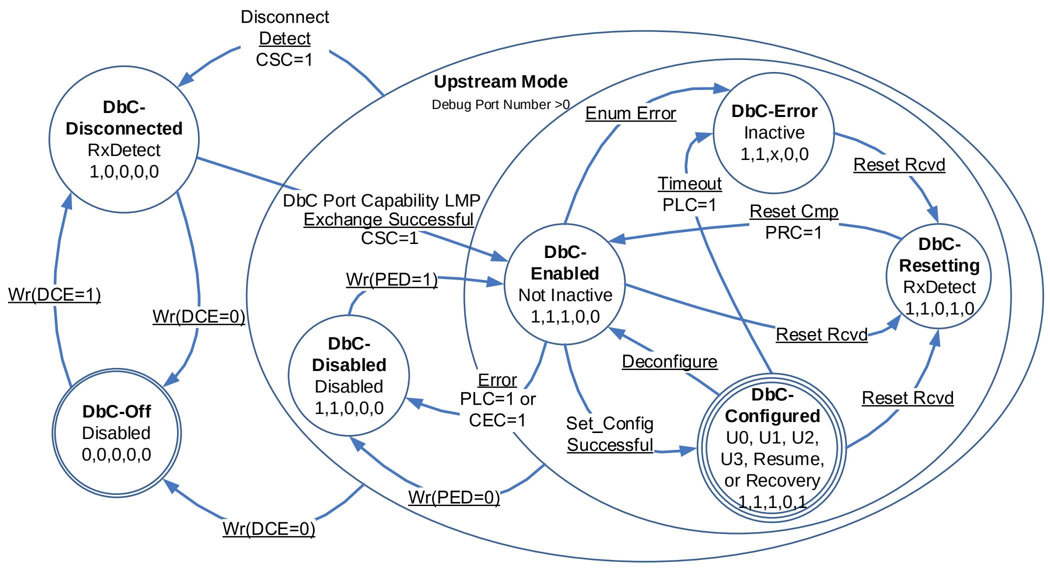
This section describes the DbC Port state machine. The following state machines utilize the following notation:



Where the **State Name** is an informative name defined by the xHCI specification, the *Port Link State* identifies the possible values for the DCPORTSC *PLS* field, and *Signal State* values are:

> DCCTRL *Debug Capability Enable* (DCE), DCPORTSC *Current Connect Status* (CCS), DCPORTSC *Port Enabled/Disabled* (PED), DCPORTSC *Port Reset* (PR), and DCCTRL *DbC Run* (DCR), respectively, e.g. 0,0,0,0,0 all signals are ‘0’.

Figure 7-8: DbC Port State Machine


Disconnect Detect CSC=1
Upstream Mode
Debug Port Number >0
DbC-Disconnected RxDetect 1,0,0,0,0
Enum Error
DbC-Error Inactive 1,1,x,0,0
Reset Rcvd
Timeout PLC=1
DbC Port Capability LMP Exchange Successful CSC=1
Reset Cmp PRC=1
DbC-Resetting RxDetect 1,1,0,1,0
Wr(PED=1)
DbC-Enabled Not Inactive 1,1,1,0,0
Wr(DCE=1)
Wr(DCE=0)
Reset Rcvd
DbC-Disabled Disabled 1,1,0,0,0
Deconfigure
Error PLC=1 or CEC=1
Reset Rcvd
DbC-Off Disabled 0,0,0,0,0
Set_Config Successful
DbC-Configured U0, U1, U2, U3, Resume, or Recovery 1,1,1,0,1
Wr(PED=0)
Wr(DCE=0)

Note that in all states except for **DbC-Off** and **DbC-Disconnected**, the Root Hub port is assigned to the Debug Capability (Debug Port Number > ‘0’) and in operating in a Upstream facing mode.

The *PLS* values cited in Figure 7-8 are not comprehensive. Refer to the respective state descriptions below for the more details on the specific PLS values that may be presented while in a state.

7.6.6.1 DbC-Off

This is the initial state after a Chip Hardware Reset or the assertion of *HCRST*.

In this DbC port state:

- The DbC Capability is off.
- All Root Hub ports act as normal downstream facing ports, i,e, only assert the Downstream *Direction* flag in the Port Capability LMPs that they generate and the Debug Capability Port Multiplexing mechanism will not switch the link to the DbC Port if a Downstream *Direction* flag is detected in a received Port Capability LMP.
- The *Debug Port Number* = ‘0’.
- The DbC Capability shall be in the *Attached* USB Device State.
- The ports’ LTSSM is not applicable.

A write to the DCCTRL register with *DCE* cleared to ‘0’ or a write to the USBCMD register with the *HCRST* flag set to ‘1’ shall transition from the DbC port from any state to the **DbC-Off** state (*Wr(DCE=0)*).

A write to the DCCTRL register with *DCE* set to ‘1’ shall transition from the DbC port to the **DbC-Disconnected** state (*Wr(DCE=1)*).

7.6.6.2 DbC-Disconnected

In this DbC port state:

- The DbC Capability shall be in the *Attached* USB Device State.
- The ports’ LTSSM state may be in the RxDetect, Polling, or U0 state.
- The *Debug Port Number* = ‘0’.

A transition of the *USB3 Root Hub Port Polling substate machine* (4.19.1.2.4.2) from the **CfgExg** state to the **DbC** state shall transition the DbC port to the **DbC-Enabled** state (*DbC Port Capability LMP Exchange Successful*). This transition shall set the *CSC* flag to ‘1’.

A Disconnect Detect in the any state, except **DbC-Off**, shall transition the DbC port to the **DbC-Disconnected** state (*Disconnect Detect*). This transition shall set the *CSC* flag to ‘1’.

7.6.6.3 DbC-Enabled

In this DbC port state:

- The Debug Host enumerates the DbC Capability, and the *USB Device State* of the DbC Capability attempts to advance from the *Powered* state, through the *Default* and *Address* states, to the *Configured* state. Refer to section 9.1 of the USB3 spec for more information on USB Device States.
- The ports’ LTSSM shall not be in the SS.Inactive or SS.Disabled states.
- The *Debug Port Number* > ‘0’.

If the *USB Device State* of the DbC Capability successfully advances to the *Configured* state, the DbC Port shall transition to the **DbC-Configured** state (*Set_Config Successful*).

If the *USB Device State* of the DbC Capability fails to enumerate successfully (i.e. the DbC USB Device State fails to advance to the *Configured* state), the DbC Port shall transition to the **DbC-Error** state (*Enum Error*). Note that this transition occurs only if an *internal* DbC resource or other issue caused an enumeration failure. Normally the Debug Host gives up if there is an *external* error (e.g. link, retry, etc.) that prevents the enumeration process from completing successfully, not vice versa. The DbC does not maintain any timers, retry counts, etc. related to external enumeration errors.

If any LTSSM Polling substate times out or if a tPortCongfigurationTimeout occurs, the DbC Port shall transition to the **DbC-Disabled** state (*Error*). An LTSSM Polling timeout shall set the *PLC* flag to '1' (PLC Condition: Training Error or Error). A tPortCongfigurationTimeout shall set the *CEC* flag to '1'.

If a Hot or Warm Reset is detected, the DbC Port shall transition to the **DbC-Resetting** state (*Reset Rcvd*).

7.6.6.4 DbC-Configured

In this DbC port state:

- *DCR* is asserted ('1').
- The *USB Device State* of the DbC Capability is the *Configured* state.
- The ports' LTSSM may be in the U0, U1, U2, U3, or Recovery states.

If the Debug Host deconfigures the device (i.e. issues a SET_CONFIGURATION(0) request), the DbC Port shall transition to the **DbC-Enabled** state (*Deconfigure*).

If the LTSSM exits the Recovery state after a timeout, the DbC Port shall transition to the **DbC-Error** state (*Timeout*). This transition shall set the PLC flag to '1' (PLC Condition: Error).

If a Hot or Warm Reset is detected, the DbC Port shall transition to the **DbC-Resetting** state (*Reset Rcvd*).

Note: While in this state the *PLC* flag shall be set to '1' if the DbC enters or exits the suspend state (PLC Condition: U0 -> U3 or U3 -> U0).

7.6.6.5 DbC-Resetting

In this DbC port state:

- The Debug Host is signaling a Hot or Warm reset.
- *PED* = '0' and *PR* = '1'.

- The *USB Device State* of the DbC Capability is the *Powered* state.
- The ports' LTSSM may be in the RxDetect, Recovery, Polling, U0, or Hot Reset state.

When the reset signaling is complete, the DbC Port shall transition to the **DbC-Enabled** state, and *PED* and *PRC* shall be asserted ('1') (*Reset Cmp*).

Note: *Reset Cmp* is true for a *Hot Reset* when the LTSSM *Exit from Hot Reset.Active* conditions described in section 7.5.12.3.2 of the USB3 spec are met.
Reset Cmp is true for a *Warm Reset* after a *Port Capability LMP Exchange* is successful.

7.6.6.6 DbC-Disabled

Software may place the DbC Port in this state to disconnect from the Debug Host but maintain ownership of the Root Hub Port (i.e. the *USB3 Root Hub Port Polling substate machine* remains in the *DbC* state).

In this DbC port state:

- The *USB Device State* of the DbC Capability is the *Attached* state.
- The ports' LTSSM shall be in the SS.Disabled state.

A write to the DCPORTSC register with *PED* cleared to '0' shall transition from the DbC port from the **DbC-Enabled**, **DbC-Configured**, **DbC-Resetting**, or **DbC-Error** state to the **DbC-Disabled** state (*Wr(PED=0)*).

A write to the DCPORTSC register with *PED* set to '1' shall transition the DbC port to the **DbC-Enabled** state (*Wr(PED=1)*).

7.6.6.7 DbC-Error

This state is entered due to the detection of an error condition in the DbC port **DbC-Enabled** or **Configured** states.

In this DbC port state:

- The *PED* flag shall maintain the value asserted by the previous state.
- The *USB Device State* of the DbC Capability shall maintain the value asserted by the previous state.
- The ports LTSSM shall be in the SS.Inactive state.

If a Hot or Warm Reset is detected, the DbC Port shall transition to the **DbC-Resetting** state (*Reset Rcvd*).

7.6.7 The USB Debug Device

A DbC is a standard USB device, in the sense that it supports a Default Control Endpoint, which responds to standard USB requests, e.g. SET_ADDRESS, GET_DESCRIPTOR, GET_CONFIGURATION, etc. Additionally, the DbC supports a single configuration with a single interface that contains a pair of bulk endpoints (one IN and one OUT). The xHC hardware provides the necessary logic to enumerate a DbC to a Debug Host and advance the Debug Device to the Configured state, where the two bulk endpoints are enabled. When the Debug Device is configured and the bulk endpoints are operational, the *DbC Run* bit in the DCCTRL register shall transition to '1'.

The Debug Host will expect the DbC to be ready to accept standard requests (GET_DESCRIPTOR, SET_ADDRESS, etc.) as soon as an attach is detected.

The USB descriptors presented by the Debug Device during the enumeration process are defined in section 7.6.10.

The protocol used to move debugger information between a Debug Host and a Debug Target is outside the scope of this specification.

7.6.7.1 Enumeration Mode

The transition of the Debug Capability Enable flag from '0' to '1' sets the Debug Capability into *Enumeration Mode*.

While in *Enumeration Mode*, debug capability logic services the standard USB enumeration related requests from the Debug Host (GET_DESCRIPTOR, SET_ADDRESS, SET_FEATURE, CLEAR_FEAURE, and SET_CONFIGURATION) though its Default Control Endpoint.

In Enumeration Mode, the IN and OUT Transfer Rings of the Debug Capability are disabled.

After the Debug Device software successfully completes a SET_CONFIGURATION request, the *DbC Run* bit in the DCCTRL register shall transition to '1'.

7.6.7.2 Run Mode

When the *DbC Run* bit is '1', the Debug Capability is in *Run Mode*.

While in *Run Mode*, Debug Capability software services Debug Capability IN and OUT data transfer requests from the Debug Host through the Data Endpoints of the Debug Capability. A Debug Device always declares a pair of Data endpoints, one bulk IN and one bulk OUT endpoint, which respond to TPs and DPs addressed to Endpoint Number 1.

In Run Mode, the IN and OUT Transfer Rings of the Debug Capability are dedicated to the OUT and IN Bulk endpoints of the Debug Device, respectively. Any IN TP or OUT DP targeted at a Data Endpoint of the Debug Device while it is in Run Mode, shall automatically be flow controlled, e.g. transmit a NRDY TP if the target Transfer Ring is empty.

Software rings the Debug Capability *Doorbell Register* with the *DB Target* field set to *Data EP 1 OUT Enqueue Pointer Update* to inform the xHC that data is available to transfer to the Debug Host. And sets the *DB Target* field set to *Data EP 1 IN Enqueue Pointer Update* to inform the xHC that buffers are available to receive data from the Debug Host.

7.6.7.2.1 Data Transfers

Software use *Normal TRBs* on the IN and OUT Transfer Rings to transfer data from/to the Debug Host. Software rings the Debug Capability Data IN or OUT doorbells to notify the xHC that work items are available on the respective Transfer Ring.

The operation of a Debug Capability Data endpoint is identical to a standard xHCI bulk endpoint, with the following exception: The Debug Capability Transfer Ring direction is the **opposite** of the TP/DP direction responded to by the Debug Capability. i.e. the Debug Capability IN Transfer Ring is used to receive data transferred by OUT DPs from the Debug Host, and the OUT Transfer Ring is used to send data transferred by Debug Host IN TPs.

If a DbC Bulk pipe had previously sent an NRDY, a doorbell ring shall cause the xHC to generate an ERDY. If an IN TP or OUT DP had not been received, the xHCI shall wait for the TP/DP transaction from the Debug Host. Software may use the TRB *IOC* flag to generate a *Transfer Event* on the Debug Event Ring when a Data TD completes.

Note: The Debug Capability software shall not set the *Immediate Data* (IDT) flag to ‘1’ in any TRB.

7.6.7.3 Event Generation

7.6.7.3.1 Data Transfers

Software shall use the TRB *IOC* flag to generate *Transfer Events* on the Debug Event Ring when a TD completes.

7.6.7.3.2 Debug Capability Status Changes

The Debug Capability automatically generates *Port Status Change Events* to report Debug Capability port state changes. Refer to section Event Generation for a discussion on Event Generation, and section Debug for more information on the individual Debug Capability status change flags.

7.6.7.4 Port Reset

Detection of Reset Signaling from the Debug Host by the Debug Device shall set the *Port Reset* (PR) flag to ‘1’ and clear the *DbC Run* bit, the DbC *Port Enabled/Disabled* (DCPORTSC:PED) bit, and the DbC *Device Address* field to ‘0’. When the Reset Signaling completes the *Port Reset* (PR) bit shall be cleared to

‘0’ and the DbC *Port Enabled/Disabled* bit shall be set to ‘1’ in the DBPORTSC register. When the DbC *Port Enabled/Disabled* bit transitions to ‘1’, the Debug Device shall be ready to receive standard USB requests and enumerate itself.

When the Debug Capability reports a port reset operation by the Debug Host to software, software is responsible for resetting the state of its USB Debug Device emulator.

When the port assigned to the Debug Capability is reset by the Debug Host (i.e. a transition to the DbC-Resetting state), the Debug Capability Transfer Rings shall be automatically disabled, and shall remain disabled until a SET_CONFIGURATION() request is received from the Debug Host. Any Debug Host generated TP or DP will not be responded to by the Debug Capability while the Transfer Rings are disabled and will time out. This action allows software to remove TDs that were pending before the port reset, reinitialize its internal Debug Device state, and cleanly restart Transfer Ring operation. The endpoints are re-enabled when the SET_CONFIGURATION() request is received from the Debug Host, after which the DbC bulk endpoints shall respond to any Debug Host generated TP or DP with an NRDY until software notifies the DbC that the respective Transfer Rings have been initialized by ringing their doorbells.

7.6.8 Debug Capability Structure

The xHCI Extended Capability List is used to provide a standard method for software to find and use the xHCI Debug Capability. Figure 7-9 illustrates the Debug Capability register layout, which consists of seven register sets; Capability, Doorbell, Event Ring Management, Control, Status, Port Management, and Endpoint Management.

Figure 7-9: Debug Capability Register Layout

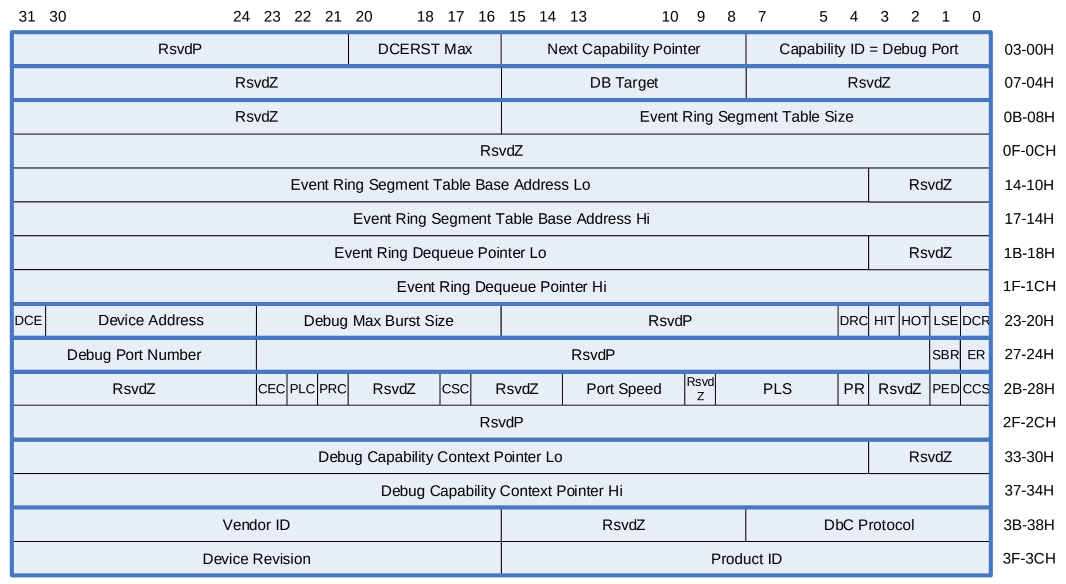

31 30 24 23 22 21 20 18 17 16 15 14 13 10 9 8 7 5 4 3 2 1 0
RsvdP | DCERST Max | Next Capability Pointer | Capability ID = Debug Port | 03-00H
RsvdZ | DB Target | RsvdZ | 07-04H
RsvdZ | Event Ring Segment Table Size | 0B-08H
RsvdZ | 0F-0CH
Event Ring Segment Table Base Address Lo | RsvdZ | 14-10H
Event Ring Segment Table Base Address Hi | 17-14H
Event Ring Dequeue Pointer Lo | RsvdZ | 1B-18H
Event Ring Dequeue Pointer Hi | 1F-1CH
DCE | Device Address | Debug Max Burst Size | RsvdP | DRC | HIT | HOT | LSE | DCR | 23-20H
Debug Port Number | RsvdP | SBR | ER | 27-24H
RsvdZ | CEC | PLC | PRC | RsvdZ | CSC | RsvdZ | Port Speed | RsvdZ | PLS | PR | RsvdZ | PED | CCS | 2B-28H
RsvdP | 2F-2CH
Debug Capability Context Pointer Lo | RsvdZ | 33-30H
Debug Capability Context Pointer Hi | 37-34H
Vendor ID | RsvdZ | DbC Protocol | 3B-38H
Device Revision | Product ID | 3F-3CH

Table 7-16: Debug Capability Structure

Register Name	Offset	Size (B)	Mnemonic	Section
Capability ID	0x00h	4	DCID	7.6.8.1
Doorbell	0x04h	4	DCDB	7.6.8.2
Event Ring Management				
Event Ring Segment Table Size	0x08h	4	DCERSTSZ	7.6.8.3.1
Event Ring Segment Table Base Address	0x10h	8	DCERSTBA	7.6.8.3.2
Event Ring Dequeue Pointer	0x18h	8	DCERDP	7.6.8.3.3
Control	0x20h	4	DCCTRL	7.6.8.4
Status	0x24h	4	DCST	7.6.8.5
Port Management				
Port Status and Control	0x28h	4	DCPORTSC	7.6.8.6
Endpoint Management				
Debug Capability Context Pointer	0x30h	8	DCCP	7.6.8.7
Device Descriptor Information				
Device Descriptor Info Register 1	0x38h	4	DCDDI1	7.6.8.8
Device Descriptor Info Register 2	0x3Ch	4	DCDDI2	7.6.8.9

7.6.8.1 Debug Capability ID Register (DCID)

Address:	Debug Capability Base + 0h
Default Value:	Refer to Table 7-17.
Attribute:	RO
Size:	32 bits

The Debug Capability *ID Register* links the USB Debug Capability into the xHCI list of Extended Capabilities and defines its basic capabilities.

Table 7-17: Offset 00h - Debug Capability Field Definitions (DCID)

Bits	Description
7:0	**Capability ID – RO.** Refer to Table 7-2 for the value that identifies that the function supports a Debug Device.
15:8	**Next Capability Pointer – RO.** Default = Implementation defined. This field indicates the location of the next capability with respect to the effective address of this capability. Refer to Table 7-1 for more information on this field.
20:16	**Debug Capability Event Ring Segment Table Max (DCERST Max) – RO.** Default = implementation dependent. Valid values are 0 – 15. This field determines the maximum value supported the *Debug Capability Event Ring Segment Table Base Size* registers (5.5.2.3.1), where: The maximum number of Event Ring Segment Table entries = $2^{\text{DCERST Max}}$. e.g. if DCERST Max = 7, then the *Debug Capability Event Ring Segment Table(s)* supports up to 128 entries, 15 then 32K entries, etc.
31:21	**RsvdP**.

7.6.8.2 Debug Capability Doorbell Register (DCDB)

Address:	Debug Capability Base + 04h
Default Value:	0000 0000
Attribute:	RW
Size:	32 bits

Table 7-18: Offset 04h - Debug Capability Field Definitions (DCDB)

Bits	Description
7:0	**RsvdP**.
15:8	**Doorbell Target (DB Target) – RW.** This field defines the target of the doorbell reference. The table below defines the Debug Capability notification that is generated by ringing the doorbell. Value Definition 0 Data EP 1 OUT Enqueue Pointer Update 1 Data EP 1 IN Enqueue Pointer Update 2:255 Reserved This field returns '0' when read and the value should be treated as undefined by software.
23:16	**RsvdP**.

7.6.8.3 Debug Capability Event Ring Registers

7.6.8.3.1 Debug Capability Event Ring Segment Table Size Reg (DCERSTSZ)

Address:	Debug Capability Base + 08h
Default Value:	0000 0000h
Attribute:	RW
Size:	32 bits

The Debug Capability *Event Ring Segment Table Size Register* defines the number of segments supported by the Debug Capability Event Ring Segment Table.

Table 7-19: Offset 08h - Debug Capability Bit Definitions (DCERSTSZ)

Bit	Description
15:0	**Event Ring Segment Table Size – RW.** Default = ‘0’. This field identifies the number of valid Event Ring Segment Table entries in the Event Ring Segment Table pointed to by the *Debug Capability Event Ring Segment Table Base Address* register. The maximum value supported by an xHC implementation for this register is defined by the *DCERST Max* field in the DCID register (7.6.8.1). Software shall initialize this register before setting the *Debug Capability Enable* field in the DCCTRL register to ‘1’.
31:16	**RsvdP**.

7.6.8.3.2 Debug Capability Event Ring Segment Table Base Address Register (DCERSTBA)

Address:	Debug Capability Base + 10h
Default Value:	0000 0000 0000 0000h
Attribute:	RW
Size:	64 bits

The *Debug Capability Event Ring Segment Table Base Address Register* identifies the start address of the Debug Capability Event Ring Segment Table.

Table 7-20: Offset 10h - Debug Capability Bit Definitions (DCERSTBA)

Bit	Description
3:0	**RsvdP**.

63:4	**Event Ring Segment Table Base Address Register – RW.** Default = '0'. This field defines the high order bits of the start address of the Debug Capability Event Ring Segment Table. Software shall initialize this register before setting the *Debug Capability Enable* field in the DCCTRL register to '1'.

7.6.8.3.3 Debug Capability Event Ring Dequeue Pointer Register (DCERDP)

Address:	Debug Capability Base + 18h
Default Value:	0000 0000 0000 0000h
Attribute:	RW
Size:	64 bits

The *Debug Capability Event Ring Dequeue Pointer Register* is written by software to define the Debug Capability Event Ring Dequeue Pointer location to the xHC. Software updates this pointer when it has finished the evaluation of an Event(s) on the Debug Capability Event Ring.

Table 7-21: Offset 18h - Debug Capability Bit Definitions (DCERDP)

Bit	Description
2:0	**Dequeue ERST Segment Index (DESI) - RW**. Default = '0'. This field may be used by the xHC to accelerate checking the Event Ring full condition. This field is written with the low order 3 bits of the offset of the ERST entry which defines the Event Ring segment that the Event Ring Dequeue Pointer resides in.
3	**RsvdP**.
63:4	**Dequeue Pointer - RW.** Default = '0'. This field defines the high order bits of the 64-bit address of the current Debug Capability Event Ring Dequeue Pointer. Software shall initialize this register before setting the *Debug Capability Enable* field in the DCCTRL register to '1'.

7.6.8.4 Debug Capability Control Register (DCCTRL)

Address:	Debug Capability Base + 20h
Default Value:	0000 0000.
Attribute:	RO, RW, RW1S, RW1C
Size:	32 bits

The *Debug Capability Control Register* is used to manage the Debug Capability.

Table 7-22: Offset 20h - Debug Capability Field Definitions (DCCTRL)

Bits	Description
0	**DbC Run (DCR) – RO.** Default = 0. When ‘0’, Debug Device is not in the Configured state. When ‘1’, Debug Device is in the Configured state and bulk Data pipe transactions are accepted by Debug Capability and routed to the IN and OUT Transfer Rings. A ‘0’ to ‘1’ transition of the *Port Reset* (DCPORTSC:PR) bit will clear this bit to ‘0’.
1	**Link Status Event Enable (LSE) - RW.** Default = ‘0’. Setting this bit to a ‘1’ enables the Debug Capability to generate Port Status Change Events due the *Port Link Status Change* bit transitioning from a ‘0’ to a ‘1’. Refer to section 4.19.2 for more information.
2	**Halt OUT TR (HOT) - RW1S**. Default = 0. While this bit is ‘1’ the Debug Capability shall generate STALL TPs for all IN TPs received for the OUT TR. The Debug Capability shall clear this bit when a ClearFeature(ENDPOINT_HALT) request is received for the endpoint. This field is valid *only* when the Debug Capability is in Run Mode (*DCR* = ‘1’). When not in Run Mode, this field shall return ‘0’ when read, and writes will have no effect. Refer to section 7.6.4.3.
3	**Halt IN TR (HIT) - RW1S**. Default = 0. While this bit is ‘1’ the Debug Capability shall generate STALL TPs for all OUT DPs received for the IN TR. The Debug Capability shall clear this bit when a ClearFeature(ENDPOINT_HALT) request is received for the endpoint. This field is valid *only* when the Debug Capability is in Run Mode (*DCR* = ‘1’). When not in Run Mode, this field shall return ‘0’ when read, and writes will have no effect. Refer to section 7.6.4.3.
4	**DbC Run Change (DRC) - RW1C**. Default = 0. This bit shall be set to '1' when DCR bit is cleared to '0', i.e. by any DbC Port State transition that exits the **DbC-Configured** state. While this bit is ‘1’ the *Debug Capability Doorbell Register* (DCDB) is disabled. Software shall clear this bit to re-enable the *DCDB*.
15:5	**RsvdP**.
23:16	**Debug Max Burst Size - RO.** Default = xHC Vendor defined. This field identifies the maximum burst size supported by the bulk endpoints of this DbC implementation.
30:24	**Device Address – RO.** Default = 0. This field reports the USB device address assigned to the Debug Device during the enumeration process. This field is valid when the *DbC Run* bit is ‘1’.
31	**Debug Capability Enable (DCE) – RW.** Default = 0. Setting this bit to a ‘1’ enables xHCI USB Debug Capability operation. This bit is a ‘0’ if the USB Debug Capability is disabled. Clearing this bit releases the Root Hub port assigned to the Debug Capability, and terminates any Debug Capability Transfer or Event Ring activity. Note that DCE may be cleared to ‘0’ by the assertion of a reset condition. Refer to the definition of SBR in Table 7-23 for more information on DbC reset conditions.

7.6.8.5 Debug Capability Status Register (DCST)

Address:	Debug Capability Base + 24h
Default Value:	0000 0000
Attribute:	RO
Size:	32 bits

The *Debug Capability Status Register* reports capability related status information to software.

Table 7-23: Offset 24h - Debug Capability Field Definitions (DCST)

Bits	Description
0	**Event Ring Not Empty (ER) – RO.** Default = ‘0’. When ‘1’, this field indicates that the Debug Capability Event Ring has a Transfer Event on it. It is automatically cleared to ‘0’ by the xHC when the Debug Capability Event Ring is empty, i.e. the Debug Capability Enqueue Pointer is equal to the *Debug Capability Event Ring Dequeue Pointer* register.
1	**DbC System Bus Reset (SBR) - RO**. When ‘1’, this field indicates that the assertion of Chip Hardware Reset, a System Bus (e.g. the assertion of PCI RST#), or a transition from the PCI PM D3hot state to the D0 state shall reset the DbC. When ‘0’, this field indicates that a Chip Hardware Reset or the assertion of Host Controller Reset (HCRST = '1') or Light Host Controller Reset (LHCRST = '1') shall reset the DbC. Resetting the DbC shall clear *DCE* to ‘0’.
23:2	**RsvdP**.
31:24	**Debug Port Number – RO.** Default = 0. This field provides the ID of the Root Hub port that the Debug Capability has been automatically attached to. The value is ‘0’ when the Debug Capability is not attached to a Root Hub port.

7.6.8.6 Debug Capability Port Status and Control Register (DCPORTSC)

Address:	Debug Capability Base + 28h
Default Value:	0000 0000 (field dependent)
Attribute:	RO, RW, RW1C (field dependent)
Size:	32 bits

The fields of the *Debug Capability PORTSC Register* are defined below and provide information about the state of the Root Hub port that is assigned to the Debug Capability. Note that the fields in this register function differently than those in a normal *Port Status and Control Register* (described in section 5.4.8) because the Root Hub port assigned to the Debug Capability is acting as an Upstream Facing Port, not a Downstream Facing Port.

Table 7-24: Offset 28h - Debug Capability Field Definitions (DCPORTSC)

Bits	Description
0	**Current Connect Status (CCS) – RO.** Default = ‘0’. ‘1’ = A Root Hub port is connected to a Debug Host and assigned to the Debug Capability. ‘0’ = No Debug Host is present. This value reflects the current state of the port, and may not correspond to the value reported by the *Connect Status Change* (CSC) field in the *Port Status Change Event* that was generated by a ‘0’ to ‘1’ transition of this bit. This flag is ‘0’ if *Debug Capability Enable* (DCE) is ‘0’.
1	**Port Enabled/Disabled (PED) – RW.** Default = ‘0’. ‘1’ = Enabled. ‘0’ = Disabled. This flag shall be set to '1' by a '0' to '1' transition of *CCS* or a '1' to '0' transition of the *PR*. When *PED* transitions from '1' to '0' due to the assertion of PR, the port's link shall transition to the Rx.Detect state. This flag may be used by software to enable or disable the operation of the Root Hub port assigned to the Debug Capability. The Debug Capability Root Hub port operation may be disabled by a fault condition (disconnect event or other fault condition, e.g. a LTSSM Polling substate timeout, tPortConfiguration timeout error, etc.), the assertion of DCPORTSC *PR*, or by software. 0 = Debug Capability Root Hub port is disabled. 1 = Debug Capability Root Hub port is enabled. When the port is disabled (*PED* = ‘0’) the port's link shall enter the SS.Disabled state and remain there until *PED* is reasserted ('1') or *DCE* is negated ('0'). Note that the Root Hub port is remains mapped to Debug Capability while *PED* = '0'. While *PED* = '0' the Debug Capability will appear to be disconnected to the Debug Host. This field is ‘0’ if *DCE* or CCS are ‘0’.
3:2	**RsvdZ**.
4	**Port Reset (PR) – RO.** Default = ‘0’. ‘1’ = Port is in Reset. ‘0’ = Port is not in Reset. This bit is set to ‘1’ when the bus reset sequence as defined in the USB Specification is detected on the Root Hub port assigned to the Debug capability. It is cleared when the bus reset sequence is completed by the Debug Host, and the DbC shall transition to the USB Default state. A ‘0’ to ‘1’ transition of this bit shall clear DCPORTSC *PED* (‘0’). This field is ‘0’ if *DCE* or CCS are ‘0’.

8:5	**Port Link State (PLS) – RO.** Default = undefined. This field reflects its current link state. This field is only relevant when a Debug Host is attached (*Debug Port Number* > '0'). **Value Meaning** 0 Link is in the **U0** State 1 Link is in the **U1** State 2 Link is in the **U2** State 3 Link is in the **U3** State (Device Suspended) 4 Link is in the **Disabled** State 5 Link is in the **RxDetect** State 6 Link is in the **Inactive** State 7 Link is in the **Polling** State 8 Link is in the **Recovery** State 9 Link is in the **Hot Reset** State 15:10 Reserved Note: Transitions between different states are not reflected until the transition is complete.
9	**RsvdZ**.
13:10	**Port Speed (Port Speed) – RO.** Default = '0'. This field identifies the speed of the port. This field is only relevant when a Debug Host is attached (*CCS* = '1') in all other cases this field shall indicate *Undefined Speed*. Value Meaning 0 Undefined Speed 1-15 *Protocol Speed ID* (PSI), refer to section 7.2.1 for the definition of PSIs. Note: The Debug Capability does not support LS, FS, or HS operation.
16:14	**RsvdZ**.
17	**Connect Status Change (CSC) – RW1C.** Default = '0'. '1' = Change in Current Connect Status. '0' = No change. Indicates a change has occurred in the port's *Current Connect Status*. The xHC sets this bit to '1' for all changes to the Debug Device connect status, even if system software has not cleared an existing DbC Connect Status Change. For example, the insertion status changes twice before system software has cleared the changed condition, hardware will be "setting" an already-set bit (i.e., the bit will remain '1'). Software shall clear this bit by writing a '1' to it. This field is '0' if *DCE* is '0'.
20:18	**RsvdZ**.
21	**Port Reset Change (PRC) – RW1C.** Default = '0'. This bit is set when reset processing on this port is complete (i.e. a '1' to '0' transition of *PR*). '0' = No change. '1' = Reset complete.Software shall clear this bit by writing a '1' to it. This field is '0' if *DCE* is '0'.

22	**Port Link Status Change (PLC) = RW1C.** Default = '0'. This flag is set to '1' due to the following *PLS* transitions: **Transition Condition** U0 -> U3 Suspend signaling detected from Debug Host U3 -> U0 Resume complete Polling -> Disabled Training Error Ux or Recovery -> Inactive Error Software shall clear this bit by writing a '1' to it. This field is '0' if *DCE* is '0'.
23	**Port Config Error Change (CEC) – RW1C**. Default = '0'. This flag indicates that the port failed to configure its link partner. 0 = No change. 1 = Port Config Error detected. Software shall clear this bit by writing a '1' to it.
31:24	**RsvdZ**.

Note: If the Debug Capability Event Ring is full, the xHC will be unable to generated Port Status Change Events due to transitions in the Change bits. In this case, a Change bit will remain set until cleared by software.

7.6.8.7 Debug Capability Context Pointer Register (DCCP)

Address:	Debug Capability Base + 30h
Default Value:	0000 0000 0000 0000
Attribute:	RW
Size:	64 bits

The *Debug Capability Context Pointer Register* identifies the start address of the array of data structures that are used to manage the Debug Capability Transfer Rings.

Table 7-25: Offset 30h - Debug Capability Context Pointer Field Definitions (DCCP)

Bits	Description
3:0	**RsvdP**.
63:4	**Debug Capability Context Pointer Register – RW.** Default = '0'. This field defines the high order bits of the start address of the Debug Capability Context data structure (refer to section 7.6.9) associated with the Debug Capability. Software shall initialize this register before setting the *Debug Capability Enable* bit in the *Debug Capability Control Register* to '1'.

7.6.8.8 Debug Capability Device Descriptor Info Register 1 (DCDDI1)

Address:	Debug Capability Base + 38h
Default Value:	0000 0000
Attribute:	RW
Size:	32 bits

The *Debug Capability Device Descriptor Register 1* identifies the Device Protocol and Vendor ID values that shall be reported by DbC in its Device Descriptor when it is enumerated by a Debug Host. Refer to section 9.6.1, Table 9-8 in the USB3 spec.

This register shall be initialized before enabling the DbC (DCE = ‘1’).

Table 7-26: Offset 38h - Debug Capability Device Descriptor Info Field Definitions (DCDDI1)

Bits	Description
7:0	**DbC Protocol – RW**. This field is presented by the Debug Device in the USB Interface Descriptor *bInterfaceProtocol* field. **Value Function** 0 Debug Target vendor defined. 1 GNU Remote Debug Command Set supported. 2-255 Reserved.
15:8	**RsvdZ**.
31:16	**Vendor ID – RW**. This field is presented by the Debug Device in the USB Device Descriptor *idVendor* field.

7.6.8.9 Debug Capability Device Descriptor Info Register 2 (DCDDI2)

Address:	Debug Capability Base + 3Ch
Default Value:	0000 0000
Attribute:	RW
Size:	32 bits

The *Debug Capability Device Descriptor Register 2* identifies the Device Revision and Product ID values that shall be reported by DbC in its Device Descriptor when it is enumerated by a Debug Host. Refer to section 9.6.1, Table 9-8 in the USB3 spec.

This register shall be initialized before enabling the DbC (DCE = ‘1’).

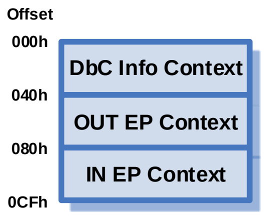
Table 7-27: Offset 3Ch - Debug Capability Device Descriptor Info Field Definitions (DCDDI2)

Bits	Description
15:0	**Product ID – RW**. This field is presented by the Debug Device in the USB Device Descriptor *idProduct* field.
31:16	**Device Revision – RW**. This field is presented by the Debug Device in the USB Device Descriptor *bcdDevice* field.

7.6.9 Data Structures

The *Debug Capability Context Pointer Register* (DCCP) references the *Debug Capability Context*, which is a data structure that contains a *Debug Capability Info Context* (DbC Info) data structure followed by 2 *Endpoint Context* data structures. The *Endpoint Context* entry at offset 40h defines the *Endpoint Context* for the OUT Transfer Ring, and the entry at offset 80h defines the *Endpoint Context* for the IN Transfer Ring. The Transfer Rings referenced by the Endpoint Contexts are Bulk endpoints as described in section Endpoint Contexts and Transfer Rings.

Figure 7-10: Debug Capability Context Data Structure



Note: Figure 7-10 illustrates the Debug Capability Context, which includes 64 byte *DbC Info* and *Endpoint Contexts*. The *Context Size* (CSZ) field in the HCCPARAMS1 register does *not* apply to DbC related contexts. All DbC data structure consume 64 bytes. Refer to section 6.2.3 for more information on the Endpoint Context data structure.

The *Debug Capability Event Ring Registers* work identically to the normal Event Ring Registers described in section 4.9.4. i.e. the *Debug Capability Event Ring Segment Table Base Address Register* references an *Event Ring Segment Table* data structure as described in section 6.5.

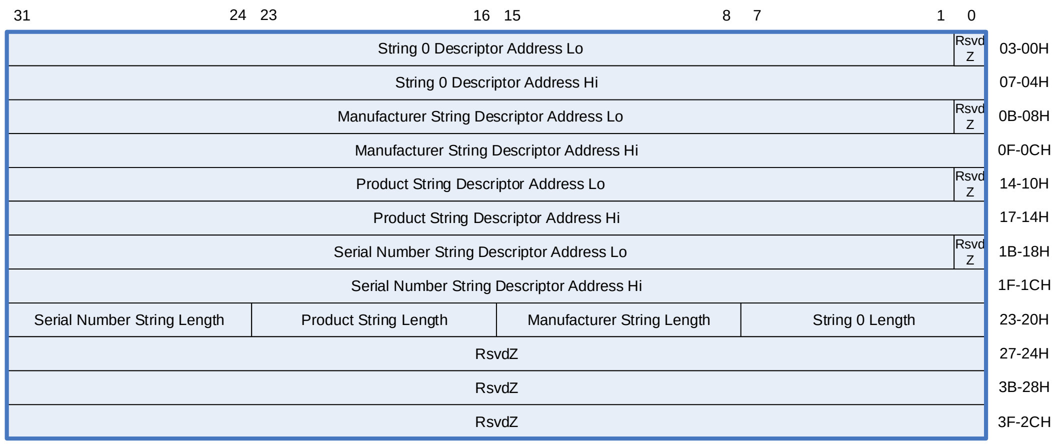
Normally if the *Debug Capability Enable* (DCE) bit in the *Debug Capability Control Register* (DCCTRL) is '1', the xHC maintains ownership of the data structures, except while an endpoint is in the Stopped state where the ownership of the Transfer Ring is relinquished by the xHC, allowing software to add, delete, or modify any TD on the ring.

7.6.9.1 Debug Capability Info Context (DbCIC)

The 64 byte Debug Device Info Context data structure defines parameters that are presented by the Debug Device when it is enumerated.

Note: Software sets the values in the DbCIC to reflect the specific debugging environment that it supports, e.g. if software supported the *GDB Remote Debug* protocol, then the *Manufacturer String* may = “Linux”, the *Product String* may = “Remote GDB”. If a vendor does not have a USB-IF assigned Vendor ID, then they could use the development reserved *Vendor ID* = FFFFh. The *Device Revision* field would reflect the revision of remote debug protocol, etc.

Figure 7-11: Debug Capability Info Context Data Structure (DbCIC)


31
24
23
16
15
8
7
1
0
String 0 Descriptor Address Lo
RsvdZ
03-00H
String 0 Descriptor Address Hi
07-04H
Manufacturer String Descriptor Address Lo
RsvdZ
0B-08H
Manufacturer String Descriptor Address Hi
0F-0CH
Product String Descriptor Address Lo
RsvdZ
14-10H
Product String Descriptor Address Hi
17-14H
Serial Number String Descriptor Address Lo
RsvdZ
1B-18H
Serial Number String Descriptor Address Hi
1F-1CH
Serial Number String Length
Product String Length
Manufacturer String Length
String 0 Length
23-20H
RsvdZ
27-24H
RsvdZ
3B-28H
RsvdZ
3F-2CH

The String referenced by this field shall be returned when the Debug Device receives a GET_DESCRIPTOR(STRING, 0) request.

Table 7-28: Offset 00h - Debug Capability Info Context Field Definitions (DbCIC)

Bits	Description
0	**RsvdZ**.
63:1	**String 0 Descriptor Address**. This field represents the high order bits of the 64-bit pointer to a USB String Descriptor that contains which specifies the Languages Supported by the DbC.

The String referenced by this field shall be returned when the Debug Device receives a GET_DESCRIPTOR(STRING, 1) request.

Table 7-29: Offset 08h - Debug Capability Info Context Field Definitions (DbCIC)

Bits	Description
0	**RsvdZ**.
63:1	**Manufacturer String Descriptor Address**. This field represents the high order bits of the 64-bit pointer to a USB String Descriptor that contains which describes the manufacturer.

The String referenced by this field shall be returned when the Debug Device receives a GET_DESCRIPTOR(STRING, 2) request.

Table 7-30: Offset 10h - Debug Capability Info Context Field Definitions (DbCIC)

Bits	Description
0	**RsvdZ**.
63:1	**Product String Descriptor Address**. This field represents the high order bits of the 64-bit pointer to a USB String Descriptor that contains which describes the product.

The String referenced by this field shall be returned when the Debug Device receives a GET_DESCRIPTOR(STRING, 3) request.

Table 7-31: Offset 18h - Debug Capability Info Context Field Definitions (DbCIC)

Bits	Description
0	**RsvdZ**.
63:1	**Serial Number String Descriptor Address**. This field represents the high order bits of the 64-bit pointer to a USB String Descriptor that contains which describes the device's serial number.

Note: If a string is not defined for a specific attribute (Manufacture, Product, or Serial Number), software shall point the respective String Length to '0' and the String Descriptor Address field shall be ignored by the xHC.

Table 7-32: Offset 20h - Debug Capability Info Context Field Definitions (DbCIC)

Bits	Description
7:0	**String 0 Length**. The size of String 0 in bytes.
15:8	**Manufacturer String Length**. The size of Manufacturer String in bytes.
23:16	**Product String Length**. The size of Product String in bytes.
31:24	**Serial Number String Length**. The size of Serial Number String in bytes.

7.6.9.2 Debug Capability Endpoint Context

The Debug Device utilizes the Endpoint Context data structure defined in section 6.2.3 with following exceptions:

- The DbC does not support Streams, so the *MaxPStreams*, *LSA*, and *HID* fields are reserved and shall be set to '0'.
- The DbC endpoints are bulk, so the *Interval*, *Mult*, and *Max ESIT Payload* fields are reserved and shall be set to '0'.
- Figure 6-3 illustrates a 32 byte *Endpoint Context* data structure. When used by the DbC it is always a 64 byte data structure, where bytes (14-1Fh) are dedicated for exclusive use by the DbC and shall be treated by system software as Reserved and Opaque (RsvdO).

7.6.10 USB Descriptors for Debug Class Device

This section defines the USB descriptors that shall be returned by a USB Debug Device when it receives GET_DESCRIPTOR requests.

The Debug Device is built using one interface which declares 2 Bulk endpoints, an IN and an OUT. Refer to section 8 of the USB3 specification for more information on the following descriptor types.

7.6.10.1 Device Descriptor

This section defines the USB Device Descriptor that shall be returned by a USB Debug Device when it receives a GET_DESCRIPTOR(DEVICE) request.

Table 7-33: DbC Device Descriptor

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	12h
bDescriptorType	1	1	Device Descriptor Type (assigned by USB)	01h
bcdUSB	2	2	USB 3.0 Specification	0300h
bDeviceClass	4	1	Class code (Defined in the Interface descriptor).	00h[122]
bDeviceSubClass	5	1	Subclass code (Defined in the Interface descriptor).	00h
bDeviceProtocol	6	1	Protocol code (Defined in the Interface descriptor).	00h
bMaxPacketSize0	7	1	Maximum packet size for endpoint zero.	09h
idVendor	8	2	Vendor ID (assigned by USB).	DCDDI1 Vendor ID[123]
idProduct	10	2	Product ID.	DCDDI2 Product ID[124]
bcdDevice	12	2	Device release number	DCDDI2 Device Revision[124]
iManufacturer	14	1	Index of String descriptor describing manufacturer. xHCI vendor defined.	01h
iProduct	15	1	Index of String descriptor describing the product.	02h

[122]The DbC declares its Class Code, Subclass Code, and Protocol values in the Interface Descriptor to enable implementation in a composite device refer to section 7.6.10.3.

[123]Refer to section 7.6.8.8, Table 7-26.

[124]Refer to section 7.6.8.9, Table 7-27.

iSerialNumber	16	1	Index of String descriptor describing the device's serial number. xHCI vendor defined.	03h
bNumConfigurations	17	1	Number of possible configurations.	01h

7.6.10.2 Configuration Descriptor

The USB Configuration Descriptor declared by a USB Debug Device.

Table 7-34: DbC Configuration Descriptor

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	09h
bDescriptorType	1	1	Configuration Descriptor Type (assigned by USB)	02h
wTotalLength	2	2	Total length of data returned for this configuration. Includes the combined length of all returned descriptors (configuration, interface, and endpoint) returned for this configuration.	002Ch
bNumInterfaces	4	1	Number of interfaces supported by this configuration.	01h
bConfigurationValue	5	1	Value to use as an argument to Set Configuration to select this configuration.	01h
iConfiguration	6	1	Index of string descriptor describing this configuration. (None defined)	00h
bmAttributes	7	1	Configuration characteristics Bit Function 7 Reserved (set to one) 6 Self Powered 5 Remote Wakeup 4-0 Reserved (reset to 0)	C0h

bMaxPower	8	1	Maximum power consumption of USB device from bus in this specific configuration when the device is fully operational.	xHCI vendor defined

7.6.10.3 Interface Descriptor

The USB Interface Descriptor declared by a USB Debug Device.

Table 7-35: DbC Interface Descriptor

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	09h
bDescriptorType	1	1	Interface Descriptor Type (assigned by USB)	04h
bInterfaceNumber	2	1	Number of interfaces.	00h
bAlternateSetting	3	1	Value used to select alternate setting for the interface identified in the prior field.	00h
bNumEndpoints	4	1	Number of endpoints used by this interface (excluding endpoint zero).	02h
bInterfaceClass	5	1	Class code.	DCh[125]
bInterfaceSubClass	6	1	Subclass code.	02h[126]
bInterfaceProtocol	7	1	Protocol code.	DCDDI1 DbC Protocol field[127]

[125]"Diagnostic Device" class, assigned by USB-IF.

[126]"Debug Device" SubClass, assigned by USB-IF.

[127]Refer to section 7.6.8.8, Table 7-26.

iInterface	8	1	Index of string descriptor describing this interface.	00h

7.6.10.4 Endpoint Descriptor 1 (Bulk OUT)

The USB Endpoint Descriptor declared for the Bulk OUT endpoint by a USB Debug Device.

Table 7-36: DbC Endpoint Descriptor 1 OUT

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	07h
bDescriptorType	1	1	Endpoint Descriptor Type (assigned by USB)	05h
bEndpointAddress	2	1	The address of the endpoint on the USB device described by this descriptor.	01h
bmAttributes	3	1	This field describes the endpoint's attributes when it is configured using the bConfigurationValue. Transfer Type = Bulk, Direction = OUT.	02h
wMaxPacketSize	4	2	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. Size = 1KB.	0400h
bInterval	6	1	Interval for polling endpoint for data transfers	00h

7.6.10.5 SuperSpeed Endpoint Companion Descriptor 1 (Bulk OUT)

The USB SuperSpeed Endpoint Companion Descriptor declared for the Bulk OUT endpoint by a USB Debug Device.

Table 7-37: DbC SuperSpeed Endpoint Companion Descriptor 1 OUT

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	06h
bDescriptorType	1	1	SuperSpeed Endpoint Companion Descriptor Type (assigned by USB)	30h
bMaxBurst	2	1	The maximum number of packets the endpoint can send or receive as part of a burst. Valid values are from 0 to 15.	DCCTRL Debug Max Burst Size[128]
bmAttributes	3	1	This field describes the endpoint's SuperSpeed attributes when it is configured using the bConfigurationValue. Mult = 0, MaxStreams = 0.	0
wBytesPerInterval	4	2	The total number of bytes this endpoint will transfer every service interval. This field is only valid for periodic endpoints.	0000h

7.6.10.6 Endpoint Descriptor 2 (Bulk IN)

The USB Endpoint Descriptor declared for the Bulk IN endpoint by a USB Debug Device.

Table 7-38: DbC Endpoint Descriptor 2 IN

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	07h
bDescriptorType	1	1	Endpoint Descriptor Type (assigned by USB)	05h

[128]Refer to section 7.6.8.4, Table 7-22.

bEndpointAddress	2	1	The address of the endpoint on the USB device described by this descriptor.	81h
bmAttributes	3	1	This field describes the endpoint's attributes when it is configured using the bConfigurationValue. Transfer Type = Bulk, Direction = IN.	02h
wMaxPacketSize	4	2	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. Size = 1KB.	0400h
bInterval	6	1	Interval for polling endpoint for data transfers	00h

7.6.10.7 SuperSpeed Endpoint Companion Descriptor 2 (Bulk IN)

The SuperSpeed Endpoint Companion Descriptor declared for the Bulk IN endpoint by a USB Debug Device.

Table 7-39: DbC SuperSpeed Endpoint Companion Descriptor 2 IN

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	06h
bDescriptorType	1	1	SuperSpeed Endpoint Companion Descriptor Type (assigned by USB)	30h
bMaxBurst	2	1	The maximum number of packets the endpoint can send or receive as part of a burst. Valid values are from 0 to 15.	DCCTRL Debug Max Burst Size[129]
bmAttributes	3	1	This field describes the endpoint's SuperSpeed attributes when it is configured using the bConfigurationValue. Mult = 0, MaxStreams = 0.	0

[129]Refer to section 7.6.8.4, Table 7-22.

wBytesPerInterval	4	2	The total number of bytes this endpoint will transfer every service interval. This field is only valid for periodic endpoints.	0000h

7.6.10.8 Binary Object Store (BOS) Descriptor

This section defines the BOS descriptor and Device Capability Descriptors that shall be returned by a USB Debug Device when it receives a GET_DESCRIPTOR(BOS) request.

Table 7-40: BOS Descriptor

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	5h
bDescriptorType	1	1	BOS Descriptor Type (assigned by USB)	0Fh
wTotalLength	2	2	Length of this descriptor and all of its sub descriptors.	0Fh
bNumDeviceCaps	4	1	The number of separate device capability descriptors in the BOS.	01h

Table 7-41: BOS SS Device Capability Descriptor

Part	Offset (Byte)	Size (Bytes)	Description	Value
bLength	0	1	Numeric expression specifying the size of this descriptor in bytes.	0Ah
bDescriptorType	1	1	Device Capability Descriptor Type (assigned by USB)	10h
bDeviceCapabilityType	2	1	Capability Type: SUPERSPEED_USB.	03h
bmAttributes	3	1	Not LTM Capable.	00h
wSpeedsSupported	4	2	This device only supports operation at 5 Gbs.	0008h

bFunctionalitySupported	6	1	All functionality available only at 5Gbs.	03h
bU1DevExitLat	7	1	U1 Device Exit Latency. Worst case latency to transition from U1 to U0.	xHC Vendor Defined
wU2DevExitLat	8	2	U2 Device Exit Latency. Worst case latency to transition from U2 to U0.	xHC Vendor Defined

7.6.10.9 String Descriptors

Refer to the String Descriptor section (9.6.8) in the USB3 spec.

Note: Only a single LANGID definition is supported by the DbC in String 0.

7.7 xHCI I/O Virtualization (xHCI-IOV) Capability

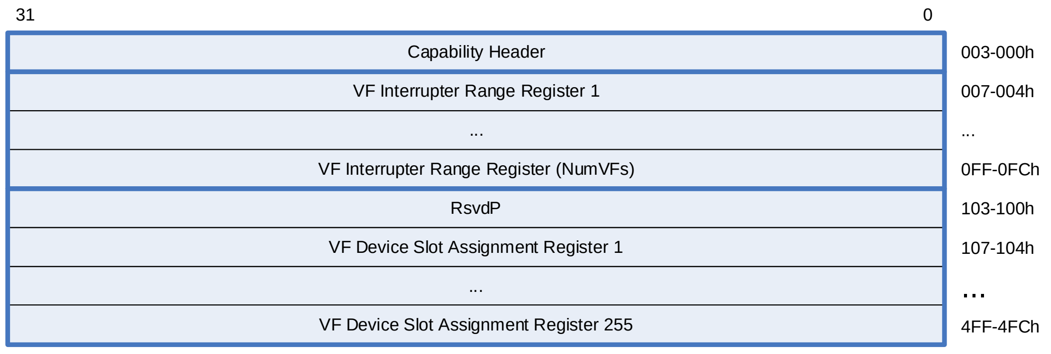
The *xHCI-IOV Extended Capability Structure* defines required parameters for managing xHC instances in a virtualized environment. The *xHCI-IOV Extended Capability Structure* is an optional normative capability defined for the xHCI. The registers defined by the xHCI-IOV capability complement those defined by the PCIe SR-IOV *Extended Capability Structure*. Both capability structures shall be defined if the xHC supports virtualization. Refer to section 8.2.1 for more information on the PCIe SR-IOV Extended Capability.

The *xHCI-IOV Extended Capability Structure* consists of two arrays of registers: the *VF Interrupter Range* and the *VM Device Slot Assignment*.

This capability is chained through the xHCI Extended Capabilities Pointer (xECP) field and resides in MMIO space.

An xHC implementation shall provide one **VF Interrupter Range Register** for each **Virtual Function** (VF) (as defined by the SR-IOV Extended Capabilities structure *TotalVFs* field). Each VF Interrupter Range Register defines **Interrupter Base Offset** and **Interrupter Count** fields. These fields allow the **Virtual Machine Manager** (VMM) to assign a specific subset of the available Interrupters to a VF. After hardware reset all VF Interrupter Range Registers = ‘0’, i.e. no Interrupters are owned by VFs.

Figure 7-12: xHCI-IOV Capability Structure


31
0
Capability Header
003-000h
VF Interrupter Range Register 1
007-004h
...
...
VF Interrupter Range Register (NumVFs)
0FF-0FCh
RsvdP
103-100h
VF Device Slot Assignment Register 1
107-104h
...
...
VF Device Slot Assignment Register 255
4FF-4FCh

Note: No VF Interrupter Range Register 0 is defined. VM Interrupter Range Register 0 would logically reference **Physical Function 0** (PF0), however PF0 provides the pool from which all Interrupters are allocated.

Note: The xHCI limits the maximum number of VFs supported to 63, i.e. the SR-IOV Extended Capabilities structure *TotalVFs* field shall be <= 63 for xHCI implementations.

For example, Logically the PF0 Interrupter Base Offset and Interrupter Count are initialized to '0' and *MaxIntrs*, respectively. As Interrupters are allocated to VFs, the number of Interrupters available to PF0 are reduced accordingly. At any time, the number of Interrupters available to PF0 is equal to the *MaxIntrs* – SUM(Interrupter Count 1-1024).

Interrupter 0 shall not be assigned to a VF.

An xHC implementation shall provide MaxSlots **VF Device Slot Assignment Registers**. Each VF Device Slot Assignment Register defines a **Slot Emulated** and **Device Slot n VF** field. The *VM Device Slot Assignment Registers* shall be used by the VMM to assign a Device Slot to a VF. The *Device Slot n VF* field contains the VF ID of the PF or VF that owns the Device Slot. After hardware reset all Device Slots are assigned the Physical Function 0 (*Device Slot n VF* = 0). The *Slot Emulated* field identifies whether a Device Slot is being emulated by the VMM for a VM or direct-assigned to a VM. Refer to section 8.1.1 for more information on device emulation.

IMPLEMENTATION NOTE

Page Size Management

This version of the xHCI spec only allows an implementation to support a single page size, as reported by the PAGESIZE register. The page size affects the following registers:

- The *RTSOFF* and *DBOFF* registers - If virtualization is enabled, then the PF and VF Capability/Operational, Runtime, and Doorbell register sets are each required to reside on separate memory pages, so that the VF Capability/Operational register sets may be trapped and emulated by a VMM. The boundaries between the register sets shall depend on the page size and affect the size of the required PF/VF MMIO space. If virtualization is not supported (i.e. no SR-IOV or xHCI-IOV Capabilities are defined), the xHCI register sets may be packed on a single page.

- PCI Configuration Space *BAR0* register - If virtualization is supported, the page size affects the size of the MMIO space declared by the *BAR0* register. Refer to section 6.2.5 of the PCI spec for information on the operation of the BAR0 register.
- *SR-IOV Supported Page Size* register - This register shall indicate support for single page size, and its size shall be identical the page size defined in the *Page Size* (PAGESIZE) register. Refer to section 3.3.12 in the SR-IOV spec.
- *SR-IOV Page Size* register - This register shall be written by software with a value identical the page size defined in the Operational *Page Size* (PAGESIZE) register. Refer to section 3.3.13 in the SR-IOV spec.

7.7.1 Capability Header

Offset:	xECP + 00h
Default Value:	Implementation Dependent
Attribute:	RO
Size:	32 bits

This register is an xHCI Extended Capability register. It includes a specific function section and a pointer to the next xHCI Extended Capability. This register is used by a VMM to configure and manage the xHC virtual functions.

Figure 7-13: xHCI-IOV Capability Header

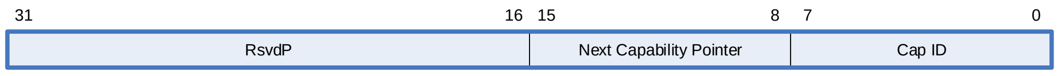


Table 7-42: xHCI_IOV Capability Header Field Definitions

Bits	Description
7:0	**Capability ID – RO.** Refer to Table 7-2 for the value that identifies the capability as xHCI I/O Virtualization.
15:8	**Next Capability Pointer – RO.** This field indicates the location of the next capability with respect to the effective address of this capability. Refer to Table 7-1 for more information on this field.

31:16	**RsvdP**.

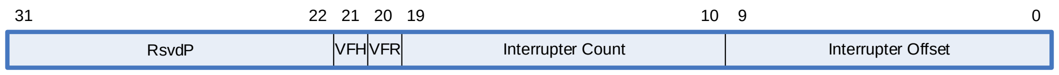
7.7.2 VF Interrupter Range Registers

Offset:	xECP + (4 * VF ID)
	where: VF ID is 1, 2, 3, … *TotalVFs*
Default Value:	0000 0000h
Attribute:	RW
Size:	32 bits

One *VF Interrupter Range Register* exists for each VF supported by the xHC. The number of VFs supported by an xHC implementation is defined by the *TotalVFs* field in the SR-IOV Extended Capability Structure. These registers are addressed by the VF ID. They are used by a VMM to assign physical Interrupters to VFs.

After hardware reset, all Interrupters are assigned to PF0. The VMM shall use the *Interrupter Count* and *Interrupter Offset* fields of this register to allocate the PF0 Interrupters to the associated VF.

Figure 7-14: VF Interrupter Range Register



For a particular VF:

The *Interrupter Count* field establishes the number of Interrupters that shall be mapped to the VF. The value of the *Interrupter Count* field shall be identical to the value of the *MaxIntrs* field in the emulated HCSPARAMS1 register presented by the VMM to a VM.

The *Interrupter Offset* field defines the physical to **virtual Interrupter mapping**. The value of the *Interrupter Offset* field shall be used by the xHC to map the set of PF0 Interrupter Registers from *Interrupter Offset* to *Interrupter Offset + Interrupter Count – 1,* to VF Interrupters *0* to *Interrupter Count – 1*.

The xHC uses these register values to translate and filter VM references to the Interrupter Registers. For example, if the xHC supports 16 interrupters and 3 VFs. VF Interrupter Range registers 4-63 would be invalid. If the *Interrupter Count* fields for VF Interrupter Range Registers 1-3 were set to 4 and the *Interrupter Offset* fields were 4, 8, and 12, respectively. Then PF0 would own Interrupters 0-3, VF 1 Interrupters 4-7, VF 2 Interrupters 8-11, and VF 3

Interrupters 12-15. The VMM would be required to present *MaxIntrs* = 4 in the HCSPARAMS1 registers that it emulates to each VM.

Software uses these registers to manage the state and Interrupter resources of a VF. Software shall not modify the *Interrupter Count* and *Interrupter Offset* fields if *VFH* = '0'.

Table 7-43: VM Interrupter Range Register Field Definitions

Bit	Description
9:0	**Interrupter Offset (IRROFF) – RW.** Default = '0'. This field specifies the index of the starting PF0 Interrupter allocated to the VF. Valid values set by software are '0' to *MaxIntrs*-1. Writing a value of '0' "unmaps" the Interrupters from a VF back to PF0.
19:10	**Interrupter Count (IRRCNT) – RW.** Default = '0'. This field identifies the number of PF0 Interrupters allocated to the VF. Valid values set by software are '1' to *MaxIntrs*-1.
20	**VF Run (VFR) – RW**. Default = '0'. '1' = Run. '0' = Stop. When set to '1', the Host Controller places endpoints associated with this VF on its Pipe Schedule. When this bit is cleared to '0', the xHC completes the current and any actively pipelined transactions on the USB associated with this VF, then removes all endpoints associated with this VF from its Pipe Schedule. The Host Controller shall halt VF endpoints within 16 microframes after software clears the *VFR* bit. The *VF Halted* bit indicates when the xHC has finished its pending pipelined transactions and has entered the stopped state for this VF. Software shall not write a '1' to this field unless the VF is in the Halted state (i.e. *VF Halted* is a '1'). Doing so will yield undefined results.
21	**VF Halted (VFH) – RO**. Default = '1'. This bit is a '0' whenever the *VF Run* bit is a '1'. The xHC sets this bit to '1' after it has stopped executing as a result of the *VF Run* bit being cleared to '0', either by software or by the xHC hardware (e.g. internal error).
31:22	**RsvdP**.

Note: Interrupter 0 is always owned by PF0.

7.7.3 VF Device Slot Assignment Registers

Offset:	xECP + (04h + (4 * *TotalVFs*)) + (4 * Slot ID))
	where: Slot ID is 1, 2, 3, … MaxSlots
Default Value:	0000 0000h
Attribute:	RW
Size:	32 bits

These registers are used by the VMM to assign a Doorbell Register (i.e. Device Slot) to a VF. All device slots are assigned to the PF0 (0) after reset.

Figure 7-15: VF Device Slot Assignment Register

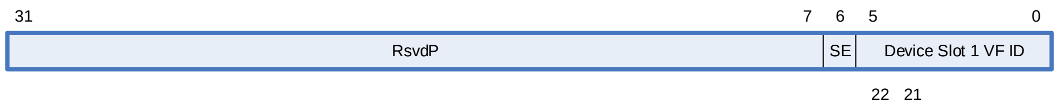


Table 7-44: VF Device Slot Assignment Register Field Definitions

Bit	Description
5:0	**Device Slot VF ID (DSAVFID) – RW.** Default = ‘0’ (all slots are assigned to PF0). This field specifies the ID of VM that the respective Device Slot is allocated. Valid values set by software are ‘0’ to NumVFs (defined in the SR-IOV Capability). A value of ‘0’ reassigns this Device Slot to PF0.
6	**Slot Emulated (DSASE) – RW.** Default = ‘0’. This field specifies if the Device Slot is emulated or direct-assigned. A value of ‘1’ shall cause the host controller to generate a *Doorbell Event* to the PF0 Primary Event Ring when the doorbell is rung. A value of ‘0’ shall cause the host controller to process the DB Target code when the doorbell is rung.
31:7	**RsvdP**.

Note: The *USB Device Address* for the slot shall be cleared to ‘0’ by the xHC when this register is written.

Note: The VMM shall issue a *Slot Enable Command* to obtain an emulated (DSASE = ‘1’) Device Slot to assign to a VF.

7.8 xHCI Local Memory Capability

An xHCI implementation may define this optional normative xHCI Extended Capability to provide RAM for debug port execution prior to initializing system memory.

Figure 7-16: xHCI Local Memory Capability

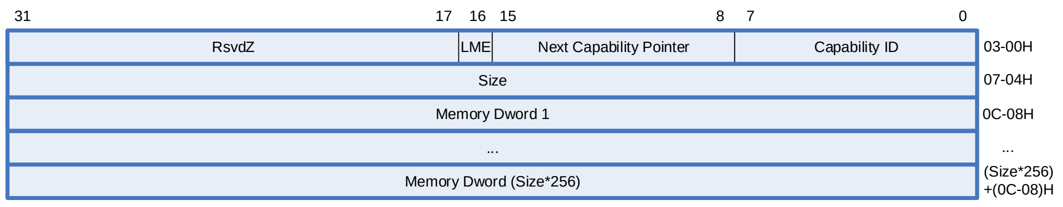


Table 7-45: Offset 00h - xHCI Local Memory Capability Field Definitions

Bits	Description
7:0	**Capability ID – RO.** Refer to Table 7-2 for the value that identifies the capability as Local Memory Protocol.
15:8	**Next Capability Pointer – RO.** This field indicates the location of the next capability with respect to the effective address of this capability. Refer to Table 7-1 for more information on this field.
16	**Local Memory Enable (LME) - RW**. Default = ‘0’. Setting this bit to a ‘1’ enables the Local Memory Capability. Clearing this bit to a ‘0’ disables the Local Memory Capability.
31:17	**RsvdZ**.

Table 7-46: Offset 04h - xHCI Local Capability Field Definitions

Bits	Description
31:0	**Size – RO.** This field identifies the size of the Local Memory space exposed by this capability in 1KB blocks.

Table 7-47: Offset 08h - xHCI Local Capability Field Definitions

Bits	Description
(Size*256) 31:0	**Local Memory – RW.** This field is a byte addressable array of read/write memory locations that is exposed by the *xHCI Local Memory Capability*.

Note: The xHCI Debug Capability requires that the data structures necessary to manage it (Debug Capability Data Structure, Transfer Rings, Event Ring, etc.) are set up in read/write memory. This is problematic if attempting to debug the code that initializes the system memory controller, and system memory is not available. This capability allows the xHC to temporarily map a portion of its internal SRAM in to MMIO space for use by the debugger prior to system memory being available.

8 Virtualization

Virtualization allows multiple **Operating System Instances** (OSI) to concurrently run within a platform. The default interface (i.e. virtualization is disabled) presented by the xHC to the host system is a single **Physical Function** (PF or PF0) or eXtensible Host Controller Interface (e.g. Figure 3-3). When the xHC virtualization capabilities are turned on, multiple **Virtual Functions** (VF) are enabled. To minimize hardware requirements, the physical interface presented by an xHC VF is a subset of that presented by the PF and the virtualization software shall emulate portions of the VF interface to fill the gaps.

Only the PF shall present xHC virtualization capabilities, i.e. SR-IOV and xHCI-IOV Capability Structures. All VFs appear as non-virtualization capable xHC instances.

Note that the xHC virtualization capabilities discussed in this document rely heavily on the virtualization concepts and mechanisms defined in the PCIe Single Root – I/O Virtualization (SR-IOV) specification.

This specification assumes three principal classes of software are supported under the virtual machine architecture:

- **Virtual Machine Manager (VMM):** The VMM acts as a host and has full control of the processor(s) and other platform hardware. The VMM presents guest software (refer to the Virtual Machine (VM) description below) with an abstraction of a virtual processor and allows it to execute directly on a logical processor. There is only one instance of a VMM in a virtualized environment, and it is able to retain selective control of platform resources: processor resources, physical memory, interrupt management, I/O, etc. A VMM may own a physical resource and provide services to share that resource across multiple VMs. Or it may *Direct-Assign* a physical resource exclusively to a VM.
- **Virtual Machine (VM):** Each Virtual Machine (VM) is a guest software environment that supports a stack consisting of operating system (OS) and application software. Each VM operates independently of other VMs and uses the same interface to processor(s), memory, storage, graphics, and I/O provided by a physical platform. The VM software stack (or OSI) may act as if it were running on a platform with no VMM. Software executing in a VM shall operate with reduced privilege so that the VMM can retain control of platform resources.
- **Hypervisor:** The hypervisor is a transport mechanism, which provides a communication path between VMs and the VMM. Features of the hypervisor allow it to trap VM requests for platform resources and forward those requests to the VMM.

Some virtualization environments combine the VMM and Hypervisor functionality into a single entity.

To reduce hardware requirements, the xHC architecture depends on the VMM to emulate the PCI Configuration Space, the xHCI Capability and Operational Registers, and several other features of a Virtual Function (VF).

To minimize the hardware requirements associated with a VF the xHCI architecture partitions its registers in to “low touch” and “high touch”. *Low touch* registers are referenced infrequently, i.e. only at initialization time or when a USB device is enumerated. *High touch* registers are referenced regularly during the normal operation of the xHC.

Low touch registers can be trapped and emulated by the VMM because the performance impact of VMM intervention is minimal. The xHCI Capability Registers and Operational Registers are considered to be Low Touch registers. The Capability Registers are generally only referenced at initialization time, and the Operational Registers are referenced infrequently during runtime, i.e. during initialization or when a USB device is attached or detached.

The high touch registers are the Interrupt and Event Ring management registers, and the Doorbell registers. The Interrupt and Event Ring registers reside in the *Runtime Register Space*. The Runtime and Doorbell Registers are physically presented by the xHC to each VF.

The xHCI is designed such that the interface presented by a combination of xHC hardware and VMM hardware emulation to a VM may be indistinguishable from the interface that the VM would see through the PF if it exclusively owned the xHC. This is accomplished through VMM emulation of the Capability and Operation registers, and xHC hardware support for filtering VF access to the physical Doorbell and Runtime register sets. The result allows a VMM to handle the emulation of the xHCI registers associated with device enumeration and other non-time critical xHCI operations, and the xHC to present hardware registers to a VM for the time critical USB device control and data transfer management.

The xHCI defines independent base addresses in MMIO space for the Runtime and Doorbell Registers so that they can be positioned on page boundaries to allow easy mapping to a VM.

Additionally, the xHCI supports the ability for the VMM to emulate a USB device to a VM. In cases were the resources of single USB device needs to be shared across multiple VMs, the VMM may own the physical device and emulate the operation of that device to multiple VMs. For example, the VMM would own the Device Slot assigned to a USB keyboard, and create emulated versions of that keyboard for each of the VM. The VMM will manage switching the keystroke stream to the VM that currently has user focus. The USB device emulation support of the xHCI also allows the VMM to emulate external USB hubs to VMs, the importance of which will be discussed below.

8.1 Operation

For the VMM to provide xHCI functionality to a VM, it shall present an xHC VF in the VM's address space. To enable the xHC virtualization capabilities the VMM shall perform the following basic steps:

- Create the VFs by enabling and configuring the PCIe Single Root – IO Virtualization (SR-IOV) capability.
- Assign xHC resources to a VF (Interrupters and Device Slots) by enabling and configuring the xHCI – IO Virtualization (xHCI-IOV) capability.
- Allocate PCI Configuration Space and Memory Mapped I/O (MMIO) Space in the VMs address space for the VF.
- Establish Hypervisor traps for VM references to the emulated VF registers.

These steps allow the combination of xHC hardware and VMM register-level emulation to present a fully functional xHC to a VM, without requiring hardware support for every feature of a VF. They also allow the VMM to act as an intermediary, managing the shared xHC resources across many VMs.

8.1.1 Resource Assignment

To minimize VMM overhead, Device Slots and Interrupters may be "direct-assigned" to Virtual Functions.

The VMM shall always own PF0. And only PF0 shall present the SR-IOV and xHCI-IOV Extended Capabilities Structures.

8.1.1.1 MMIO Space

The PCI Configuration space *BAR0* and *BAR1* fields contain a 64 bit address that points to the base of the xHC PF0 MMIO space. This pointer will be referred to as ***PBAR0***.

The SR-IOV *VF Enable* field shall be set to '1' to enable xHC virtualization support.

The SR-IOV *TotalVFs* field identifies the maximum number of VFs that can be associated with the PF.

The SR-IOV *NumVFs* field identifies the number of VFs that shall be visible in the MMIO space after both *NumVFs* is set to a valid value and *VF Enable* is set to '1'. Valid values for *NumVFs* are 1 to *TotalVFs*, SR-IOV *VF BAR0* and *VF BAR1* fields contain a 64 bit address that points to the base of the xHC VF MMIO space. This pointer will be referred to as ***VFBAR0***. These fields behave as normal PCI BARs, as described in the PCI specification section 6.2.5. They can be sized by writing all 1's and reading back the contents of the BARs as described in the PCI Specification, complying with the low order bits that define the BAR type fields.

The size decoded by VFBAR0 is referred to as ***VFBAR0.Size***. The amount of address space decoded by VFBAR0 shall be an integral multiple of SR-IOV *System Page Size* field. *VFBAR0* determines the alignment requirement and size (VFBAR0.Size) for a single VF. The total MMIO space consumed by the xHC is VFBAR0.Size * NumVFs. The MMIO space associated with each VF begins on a page boundary as defined by the *System Page Size* field of the SR-IOV Extended Capability structure.

i.e. if *VFBAR0.size* = 16KB and NumVFs = 4, then the MMIO space allocated to all VFs is 64KB (16K * 4) bytes.

PF0 MMIO Register locations:

- Capability Registers reside at PBAR0.
- Operational Registers reside at PBAR0 + CAPLENGTH.
- Runtime Registers reside at PBAR0 + RTSOFF.
- Doorbell Register Array resides at PBAR0 + DBOFF.

VF n MMIO Register locations, where n = 1 to NumVFs:

- Capability Registers reside at VFBAR0 + (VFBAR0.Size * (n-1)).
- Operational Registers reside at VFBAR0 + (VFBAR0.Size * (n-1)) + CAPLENGTH.
- Runtime Registers reside at VFBAR0 + (VFBAR0.Size * (n-1)) + RTSOFF.
- Doorbell Register Array resides at VFBAR0 + (VFBAR0.Size * (n-1)) + DBOFF.

Figure 8-1: VF MMIO Space

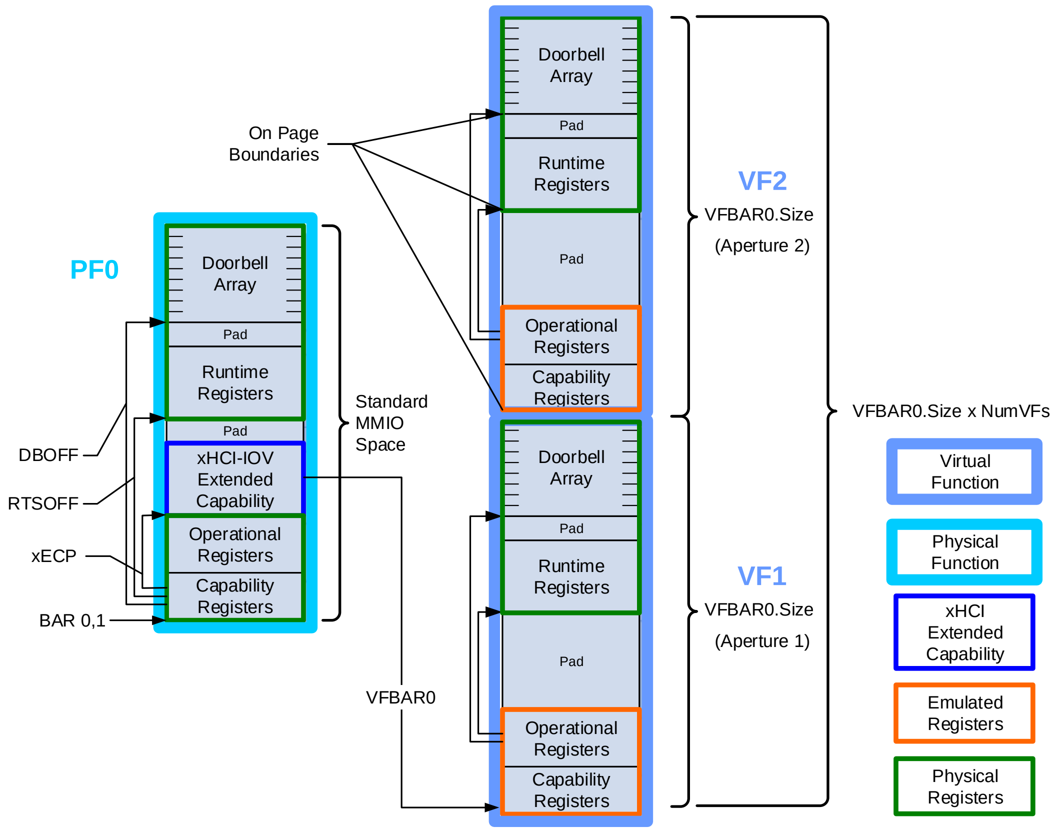


Figure 8-1 illustrates an xHC implementation that supports two VFs. Note that the MMIO address space allocated for VFs is a contiguous array. Each VFBAR0.Size space may also be referred to as an “aperture”.

Note: The SR-IOV *VF MSE* field shall be set to ‘1’ for the xHC to respond to VF MMIO memory space accesses.

8.1.1.2 Device Slots

The *VF Device Slot Assignment Registers* allow the VMM to map specified Doorbell Registers out of its (PF0) Doorbell Array and into a VFs Doorbell Array.

Virtualization is not enabled (default Doorbell Register addressing):

n = Slot ID, valid values = 1 to MaxSlots

Address of Doorbell *n* = PBAR0 + DBOFF + (n * 4)

If Virtualization is enabled:

x = *VF Device Slot Assignment Register:Device Slot VF ID*, valid values = 0 to *NumVFs* n = *VF Device Slot Assignment Register* index, valid values = 1 to *MaxSlots*

If x = 0:

Address of Doorbell n = PBAR0 + DBOFF + (n * 4)

If x > 0:

Address of Doorbell n = VFBAR0 + (VFBAR0.Size * (x-1)) + DBOFF + (n * 4)

Note: All Doorbell addresses are physical addresses.

When a Device Slot *n* is remapped from PF0 MMIO space to a VF's MMIO space, the associated Doorbell Register shall be inaccessible by the VMM through the PF0 Doorbell Array. Device Slot *n* shall be accessible to the VM through the Doorbell Register *n* of the VF assigned to the VM.

8.1.1.3 Interrupters

The *VF Interrupter Range Registers* (section 7.7.2) allow the VMM to map specified Interrupters out of its (PF0) Runtime Register space and into a VFs Runtime Register space. The Primary Interrupter Register Set (0) is always assigned to PF0. Only secondary Interrupter Register Sets (1 to *MaxIntrs*-1) may be assigned to a VF. Assignment of an Interrupter Register Set to a VF is exclusive.

Virtualization is not enabled (default Interrupter Register Set addressing):

n = Physical Interrupter Register Set ID (0 to *MaxIntrs*-1)

Interrupter Register Set n shall be located at physical address:

PBAR0 + RTSOFF + (n * 32), where 32 is the size of the Interrupter Register Set.

If Virtualization is enabled:

IRROFF = *VF Device Interrupter Range Register:Interrupter Offset*, valid values = 1 to *MaxIntrs*-1

IRRCNT = *VF Device Interrupter Range Register:Interrupter Count*, valid values = 1 to *MaxIntrs*-1

IRRINDX = *VF Device Interrupter Range Register* index, valid values = 0 to TotalVFs

n_p = Physical Interrupter Register Set ID, valid values = 0 to *MaxIntrs*

n_v = VM Interrupter Register Set ID, valid values = 0 to IRRCNT-1

Interrupter Register Set n_p + IRROFF shall be located at physical address:

VFBAR0 + (VFBAR0.Size * (IRRINDX -1)) + RTSOFF + (n_v * 32)

The sum of IRRCNT values for all *VF Device Interrupter Range Registers* shall not exceed *MaxIntrs* -1.

Note: Interrupter Register Sets are mapped exclusively. i.e. If virtualization is enabled and Interrupter Register Set n_p is remapped via a *VF Interrupter Range Register*, then Interrupter Register Set n_p is no longer accessible at PBAR0 + RTSOFF + (n_p * 32).

Note: The Event Ring of physical Interrupter Register Set 0 shall receive all non-Transfer Events generated by the xHC. And until reassigned by the VMM or a VM, the Event Ring of physical Interrupter Register Set 0 shall also receive all Transfer Events generated by the xHC.

Note: A minimum of one Interrupter Register Set shall be implemented per supported VF. System software is responsible for mapping the Interrupter Register Sets to VFs when VFs are enabled.

Note: Only secondary Interrupter Register Sets may be assigned to VFs, therefore only Transfer Events may be redirected to an Interrupter owned by a VF, including its Interrupter Register Set 0. All other Event types presented on the VFs' ("Primary") Interrupter Register Set 0 Event Ring are generated by the VMM through Force Event Commands.

Note: All Events generated by a Force Event Command are automatically directed to Interrupter Register Set 0 Event Ring of the VF specified in the Force Event Command. e.g. if the *Interrupter Offset* field for *VF Interrupter Range Register 1* = 4, then the Event Ring of Interrupter 4 shall receive the Event TRBs pointed to by all Force Event Commands targeted at VF 1.

If more than one Interrupter Register Set is available to a VF, a VM can direct the Transfer Events of selected device slots to the alternate Interrupters (1-n), using the *Interrupter Target* field in Transfer TRBs. The xHC shall translate the *Interrupter Target* field of TRBs associated with Device Slots owned by a VF with the following formula:

Physical Interrupter Register Set index = VF *Interrupter Target* + IRROFF

8.1.2 Device Enumeration and Handoff

The enumeration of a USB device in virtualized environment is a four step process: The VMM, 1) enumerates a device when it detects an attach event, 2) determines the VM that the device will be assigned to, 3) emulates an attach event of the same device to the VM, and 4) the VM enumerates the device following the steps described in section 4.3.

By default, all Device Slots are assigned to PF0, hence they are all owned by the VMM. Since the VMM owns PF0, it also has access to the physical Root Hub ports of the xHC. When a device is attached on a Root Hub Port, the VMM also follows the steps described in section 4.3, up to the point of configuring the device. The VMM only needs to retrieve enough information from a USB device to determine how it should be managed. That is, whether the device is to be owed by the VMM and emulated to VMs, direct-assigned to a VM, or simply owned and used by the VMM itself.

In the latter case, the VMM will configure the device and manage it like any other USB device in a non-virtualized environment.

In the direct-assigned case the VMM, which is emulating the PORTSC registers and Command Ring of the VM, shall emulate an attach event for the device to the VM, then map the Device Slot that it used to enumerate the device to the VF owned by the VM.

If the device is to be emulated to VMs, then the VMM should load a “master” driver that is capable of sharing the resources of the device across multiple VMs, and for each VF that the device will be shared with, emulate an attach event for the device to the VM, establish an emulated Device Slot, and map that slot to the VF owned by the respective VM. Subsequent work items generated by VFs will be processed by VMM’s master driver for the device and forwarded to the physical USB device owned by the VMM.

Note: Undefined behavior may occur if the VMM does not ensure that no more than one VM has a USB device in the Default state.

8.1.2.1 Root Hub Attach Emulation

The device enumeration process of non-virtualized environments is described in section 4.3. Much of that process also applies in virtualized environments. The VMM owns the physical Root Hub so when a device is attached; it is the entity that receives the notification. When a device is attached the VMM should decide which VM to allocate it to. The device allocation policies are outside the scope of this specification, however the VMM will be required to retrieve the Device Descriptor and possibly Configuration Descriptors from the device to determine the target VM. The VMM hub driver will follow the steps described in section 4.3 up to but not including, configuring the device (step 8).

Once the target VM has been identified, the following steps should be performed to pass the device to the VM:

1. The VMM generates a *Port Status Change Event* to the VM.
    a. Issue a Force Event Command on its Command Ring. The Force Event Command points to a Port Status Change Event TRB, and identifies the VM whose Event Ring will receive the TRB.
2. Upon reception of the Port Status Change Event TRB, the VM will begin initiating the steps described in section 4.3. The first step requires the VM to reset the device.
    a. Reset a USB2 device by setting the *Port Reset* (PR) bit to ‘1’ in the PORTSC register that was indicated by the *Port Status Change Event*. Not necessary for USB3 devices because they are implicitly reset.
3. The VMM traps the VM’s reference to its PORTSC register.
    a. When the VMM detects the *PR* bit set in the VM reference to the

emulated PORTSC register it will assert the *PR* bit in the physical PORTSC register.
Note that the VMM may filter VM references to physical PORTSC registers, e.g. in a case where the VM is attempting to reset a Root Hub Port attached to a hub, as some of the devices attached to that hub are owned by other VMs.

4. After the appropriate timeout the VM will obtain a Device Slot for the “newly” attached device.
    a. It does this by placing an *Enable Slot Command* on its Command Ring, and writing the Host Controller (VM Device Slot 0) *Doorbell* register with a *DB Target* code of *Host Controller Command*.
5. The VM reference to the Doorbell register generates a *Doorbell Event* to the VMM.
    a. The VMM parses the Doorbell Event and determines that the Command Ring has been modified.
    b. The VMM retrieves the Command TRB from the VM’s Command Ring, updating the VM Command TRB Status field and advancing the Ring Indices appropriately.
6. The VMM examines the retrieved Command TRB, decoding the *Enable Slot Command*, and processes it for the VM.
    a. The VMM uses the appropriate VM Slot Assignment Register to map the Device Slot that it used to enumerate the device to the VF owned by the VM.
    b. Releases any data structures that it was using to manage the device.
    c. Generates a Command Completion Event to the VM by issuing a Force Event Command. The Force Event Command points to a Command Completion Event TRB. The Command Response field of the Command Completion TRB will include the ID of the Device Slot that the VMM had assigned to the VM.
7. Upon reception of the Command Completion Event, the VM will proceed to initialize its Device Context data structures, Device Context Base Address Array, etc., finally issuing an Address Device Command to enable the control endpoint of the device.
8. When the VMM examines VM’s Command Ring it finds the Address Device Command and processes it for the VM. The Address Device Command informs the xHC that the Device Context data structures associated with the Device Slot have changed.
    a. The VMM forwards the Address Device Command to the xHC by placing the identical command on the PF0 Command Ring.
    b. Then returns the PF0 Command Completion Event to the VM using a

Force Event Command.

9. After receiving the Command Completion Event, the VM will then issue several requests directly to the device's control endpoint, reading Device and Configuration Descriptors to determine the configuration that it wants to select.

10. When a decision has been made, the VM shall issue a Configure Endpoint Command to enable the endpoints defined by the target configuration.

11. Again, the VMM which is trapping VM Command Ring operations simply forwards the Configure Endpoint Command to the xHC on the PF0 Command Ring and returns the returned *Command Completion Event* to the VM using a *Force Event Command*. This operation also informs the xHC that the Endpoint Context data structures associated with the Device Slot have changed.

From this point on, unless the device is detached or the VM attempts to power manage or reconfigure the device, the VMM is not involved. The direct-assignment feature of the xHCI allows the VM to communicate directly with the xHC hardware interface and the device.

8.1.2.2 External Hub Attach Emulation

All external hubs shall be owned and managed by the VMM, which enables the VMM to manage the overall USB bus topology.

A VMM implementation may choose whether or not it exposes external hubs to a VM. For instance, a VMM could present a "flat" topology to a VM, where a VM never sees an attach event for a hub and the number of Root Hub Ports that the VMM declares for the emulated xHC instance is equal to the *Number of Device Slots* (i.e. MaxSlots = MaxPorts). In this case the VM will power manage a device by manipulating the PORTSC registers. The VMM would have to translate the VM PORTSC register references into Root Hub or external hub port registers. Note that a VMM shall provide "flattened" devices with a means of asserting the correct values for their Slot Context *Route String*, *MTT*, *TT Port Number*, and *TT Hub Slot ID* fields (e.g. reflect the physical topology). This mechanism is outside the scope of this specification. The advantage of this approach is that Device Slots are not consumed by emulating external hubs to VMs.

If the VMM does present external hubs to a VM, then the physical hub shall be assigned to the VMM and the VMM shall present an emulated instance of the hub to VMs. As described above, when a device is attached the VMM shall evaluate it and selectively assign it to a VM, however in this case the VMM will emulate an attach event on the VM's emulated external hub instance, rather than generating a Port Status Change Event on the VMs Event Ring.

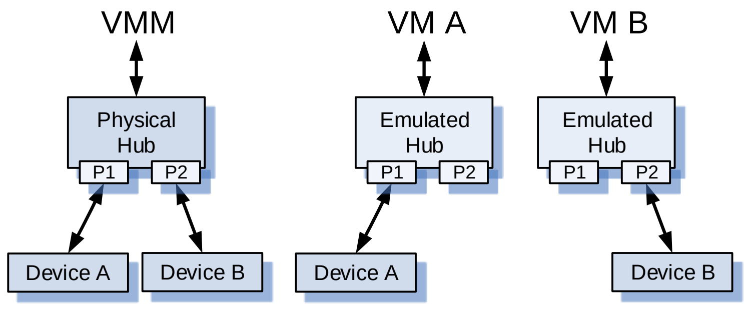
The VMM uses an additional feature of the xHCI to emulate external hubs to VMs. An external hub is enumerated to a VM by the VMM the same way that any other USB device is (as described above). To emulate a hub (or device) to a VM, the VMM utilizes the *Doorbell Event TRB*. To enable Doorbell Events the VMM shall set the *Slot Emulated* (SE) flag in the *VM Slot Assignment Register* when it assigned the Device Slot to the VM. If the *Slot Emulated* flag is ‘1’, the xHC shall not process the DB Target field when the VM rings the doorbell associated with an emulated slot, but shall generate a Doorbell Event to Event Ring 0, which is owned by the VMM. The *Slot ID*, *VM ID*, and *DB Reason* fields of the Doorbell Event TRB will indicate the source VM and value of the DB Target written to the Doorbell register.

The VMM shall manage all Transfer Rings associated with an emulated device, retrieving information from them when the doorbell is rung, and emulating their operation. The VMM shall use the *Force Event Command* to generate *Transfer Events* to the VM. The device interface presented to a VM by the xHC/VMM emulation shall be indistinguishable from the interface presented by the xHC for the equivalent direct-assigned device.

Note, that to eliminate VMM involvement for direct-assigned devices, all Event Rings are managed by xHC hardware. Transfer Events for direct-assigned and emulated Device Slots are placed on an Event Ring. To ensure Event Ring consistency, the xHCI provides the *Force Event Command* for a VMM to insert a *Transfer Event* generated for an emulated slot on the same Event Ring that is used by the xHC hardware for Transfer Events generated by direct-assigned slots.

The VMM is also responsible for hiding a USB device assigned to one VM from another. Consider a case where Device A is attached to Port 1 of a physical hub and Device B is attached to Port 2 of the same hub, however the devices are assigned to VMs A and B, respectively. Figure 8-1 illustrates the views of the USB topology seen by the VMM and each of the VMs. Each VM sees an emulated instance of the physical hub. But the VMM will have generated an attach event for Device A on Port 1 to VM A, and an attach event for Device B on Port 2 to VM B. As far as VM A is concerned, Port 2 of its emulated hub has no device attached, and VM B thinks that Port 1 has no device attached. The Devices themselves are direct-assigned to the respective VMs.

Figure 8-2: Emulated Hub Device Attachment Example



If VM B decides to place the Device B into suspend mode, it will generate the appropriate requests to its emulated hub. Since as far as VM B is concerned there are no other devices attached to the hub, it will attempt to propagate the power state up the topology by placing the hub in suspend mode as well. The VMM shall filter these requests to ensure that Device A remains operational for VM A. Since the VMM owns the physical external hub, it determines whether the hub will be placed in the suspend state or not. The VMM can fake a response back to VM B for the emulated hub, allowing the VM to think that it has placed the emulated hub in the suspend state.

8.2 SR-IOV Extended Capability

This section defines how the PCIe Single Root-I/O Virtualization (SR-IOV) capability is interpreted in an xHC implementation.

The SR-IOV capability structure is used to discover and configure a Physical Function's (PF) virtualization capabilities. These virtualization capabilities include the number of Virtual Functions (VF) the PCIe Device will associate with a PF and the type of BAR mechanism supported by those VFs.

When VFs are enabled, the PF MMIO space pointed to by a BAR is replicated for each VF. The replication of the PF MMIO space is in the form of an array of **MMIO Apertures**. The base of the VF Aperture array is pointed to by a VF BAR in the SR-IOV capability. The size of an MMIO Aperture is defined by the standard BAR sizing mechanism. The number of MMIO Apertures is defined by the *NumVFs* field in the SR-IOV capability structure. The **Aperture ID** is the index of a specific MMIO Aperture in the array. Valid *Aperture ID* values are 1 to *NumVFs*.

The VMM emulates a PF-like Configuration Space to each VM. The SR-IOV specification defines the mapping between the PCI defined *Configuration Space Header* and the SR-IOV defined *PF/VF Configuration Space Headers* (SR-IOV spec, section 3.4). The SR-IOV specification requires that a subset of the fields in the PFs Configuration Space Header be replicated in the VF Configuration Space Headers by xHC hardware. The xHCI VF Configuration Space is used by the VMM to manage VFs and not accessed by VMs. Refer to the SR-IOV specification for details.

Figure 8-3: xHCI BAR Space Example

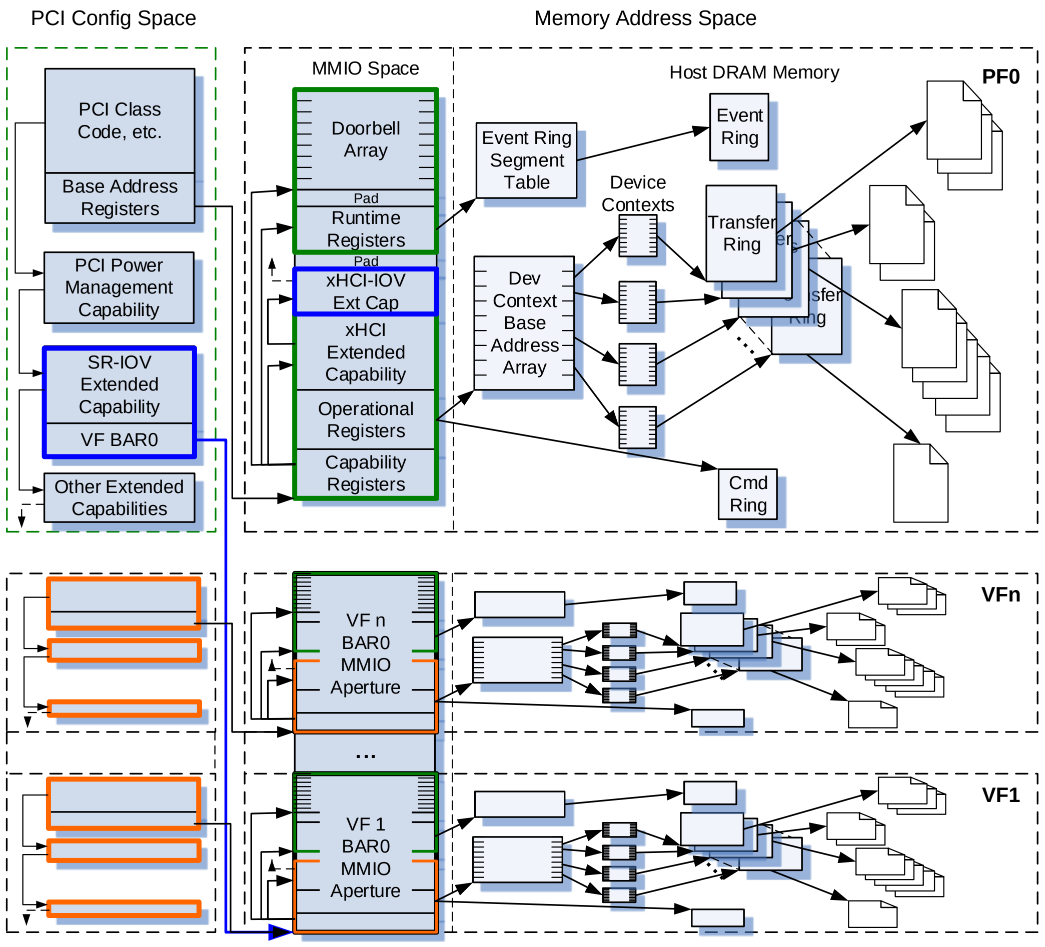


Figure 8-3 illustrates the VF MMIO Aperture configuration for the xHC. To minimize the hardware requirements for virtualization, many of the xHC MMIO registers are emulated by the VMM. The SR-IOV and xHCI-IOV Extended Capability structures (blue bordered) exists only for PF0. The SR-IOV capability defines the starting memory space address of VF1 MMIO Aperture. The xHCI-IOV Extended Capability defines the xHC registers needed to manage the individual Virtual Functions. The (orange bordered) xHCI Capability Registers, Operational Registers, and Extended Capabilities presented by a VF are emulated by the VMM. The (green bordered) xHCI Extended Runtime Registers and Doorbell arrays are physical registers presented by a VF. The (orange bordered) PCI Configuration Space as seen by the VMs is emulated by the VMM.

The physical VF register spaces (Operational, Runtime, etc.) reside on *System Page Size* boundaries. The details of their mapping are described below.

8.2.1 SR-IOV Extended Capability Structure

The xHC PF and each VF requires a unique **Requester Identifier** (RID) to distinguish its respective DMA activity. The **First VF Offset** and **VF Stride** fields in the SR-IOV Capability Structure shall define the xHC RID to PF0/VFn assignment. Refer to the SR-IOV spec for the definition and use of RIDs and all other SR-IOV Capability Structure fields.

xHCI support for *VF Migration* is outside the scope of this specification, and left to definition by specific implementations.

Note: The *PCI Express Capability Structure* is required by the SR-IOV capability.

8.2.2 xHCI-IOV Extended Capability Structure

The *xHCI-IOV Extended Capability Structure* defines required parameters for managing xHC instances in a virtualized environment. The *xHCI-IOV Extended Capability Structure* is an optional normative capability defined for the xHCI. Refer to section 7.7 for detailed information on the xHCI-IOV Extended Capability Structure.

8.3 Doorbell Registers and Virtualization

This section describes how an xHC implementation shall interpret Doorbell Register References when virtualization is enabled. The VM Device Slot Assignment Register *Device Slot VF ID* field allows a Device Slot to be assigned to a VF. If a Device Slot is assigned to a VF, then the *Slot Emulated* flag determines whether the xHC interprets references to a Device Slot's Doorbell Register as direct-assigned or emulated.

A **Valid VF Doorbell Register Reference** is defined as a *Doorbell Register* reference through an *MMIO Aperture*, where the *Aperture ID* is equal to the value of the *Device Slot VF ID* field for the referenced Device Slot (n).

The xHC shall respond to *Valid VF Doorbell Register References* through MMIO Apertures.

The xHC shall not respond to *Doorbell Register* references through MMIO Apertures, if the value of a VM Device Slot Assignment Register *Device Slot VF ID* field is equal to '0' or if the value is greater than *NumVFs*.

The *Doorbell Register* of any Device Slot not assigned to a VF by the VM Device Slot Assignment Register *Device Slot VF ID* field, shall be accessible by through the PF0 Doorbell Array.

8.3.1 Direct-Assigned Device Slot

System software rings the Doorbell Register of a Device Slot to indicate to the xHC that it has changed the slot's Device Context or the added work items to a Transfer Ring.

If for a *Valid VF Doorbell Register Reference* the *Slot Emulated* flag equals '0', then the xHC shall process the Doorbell Register reference normally. i.e. process the Doorbell Register *DB Target* field.

8.3.2 Emulated Device Slot

If for a *Valid VF Doorbell Register Reference* the *Slot Emulated* flag equals '1', then the xHC shall not process the Doorbell Register *DB Target* field, but capture the value of the field and pass it to the VMM through Event Ring 0 in the *DB Reason* field of a *Doorbell Event*.

8.4 Interrupter Mapping

If virtualization is supported, then the following requirements shall be met:

- The *Max Interrupters* (*MaxIntrs*) field shall be equal to or greater than TotalVFs + 1.
- The VM Interrupter Range Registers shall be implemented.

A minimum of one Interrupter shall be assigned to each VF. The VMM may allocate remaining Interrupters to VFs as desired by presenting the appropriate values in the Interrupter Range Registers, and the emulated Structural Parameters 2 (HCSPARAMS2) register *MaxIntrs* field.

If Interrupter Mapping is provided to a VF, the VMM shall emulate the **Interrupter Mapping Enable** bit in the Configure (CONFIG) register (section 5.4.7) to enable or disable it. If Interrupter Mapping is disabled for VF, the VMM shall set the *Interrupter Count* field to '1'. If the *Interrupter Count* field is set to '1', the xHC shall ignore the Transfer TRB *Interrupter Target* field and all Transfer Events for the VF are targeted at the Interrupter identified by the *Interrupter Offset* field.Refer to section 6.4.1 for more information on the *Interrupter Target* field.

Interrupter Mapping may be used to facilitate distribution of interrupts across cores in a multi-core platform.

8.5 Register Space Emulation

The VMM traps and emulates all xHCI Capability and Operational registers for all VFs.

The *VF Run* (VFR) and *VF Halted* (VFH) bits in the VM Interrupter Range Register provide the VMM with ability to manage the state of each VF. These bits provide for a VF, what the *Run/Stop* (R/S) and *HCHalted* (HCH) bits provide for the xHC as a whole. When the VMM detects a VM manipulating the *Run/Stop* (R/S) bit in their emulated USBCMD register, it shall reflect that state in the VFR bit for the respective VF.

The VMM shall monitor the associated *VFH* bit and reflect its status in *HCHalted* (HCH) bit of the emulated USBSTS register.

Appendix A xHCI PCI Power Management Interface

An advanced power management capabilities interface compliant with PCI Bus Power Management Interface Specification (PCI PM) is incorporated into the xHCI. This interface allows the xHCI to be placed in various power management states offering a variety of power savings for a host system.

Table A-1 highlights the xHCI support for power management states and features supported for each of the power management states. An xHC implementation may internally gate-off USB clocks and suspend the USB transceivers (low power consumption mode) to provide these power savings. The methods utilized by each xHC vendor to achieve the required behavior, is implementation specific. The xHC will assert PME# and retain chip context in accordance with the rules defined in the *PCI PM Specification* and this specification.

The controller software driver shall place all enabled downstream USB ports of the xHC in the USB suspended state before exiting the D0 state. This is to ensure all downstream devices are in an inactive, low-power mode.

Table A-1: xHCI Support for Power Management States

PCI Power Management State	State Required/ Optional by Spec	Comments
D0	Required	Fully awake backwards compatible state. All logic in full power mode.
D1	Optional	USB Sleep state with xHC bus master capabilities disabled. All USB ports in suspended state. All logic in low latency power savings mode because of low latency returning to D0 state.
D2	Optional	USB Sleep state with xHC bus master capabilities disabled. All USB ports in suspended state.
D3hot	Required	Deep USB Sleep state with xHC bus master capabilities disabled. All USB ports in suspended state.
D3cold	Required	Fully asleep backwards compatible state. All downstream devices are either suspended or disconnected based on the implementation's capability to supply downstream port power within the power budget.

A.1 PCI Power Management Register Interface

xHC implementations follow the PCI Power Management register interface specified in the PCI PM Specification. Specific requirements and clarifications for xHCI implementations are:

- The host controller should be capable of asserting PME# when in any supported device state. However, if the host controller supports systems in which the PME# assertion from D3cold is not possible (i.e. insufficient or non-existent Aux Power), then the "PME_Support" bit for D3cold (bit 15 of the PCI PM *PMC* Register) shall be modifiable. Motherboard-down devices may use a software (BIOS) scheme for modifying the value reported in this read-only bit, while other devices may use a pin-strapping to determine the value that is reported.
- The PCI PM *PMC.Aux_Current* field or *Data* register value reported by the xHC should represent the maximum current that the host controller device will consume. It shall not include power consumed by devices connected to the downstream USB ports. Note that if the host controller has been configured to not generate PME# from D3cold, then the *PMC.Aux_Current* field or *Data* register (D3 Power Consumed, D3 Power Dissipated) shall report "000".

All other registers and field should follow the PCI PM specification.

A.1.1 Power State Transitions

The xHC enters the D0 power state from the D3cold power state when Vcc is applied and a hardware or software reset occurs. A software reset shall not affect the PCI power management registers. The hardware reset may be either a PCI reset input or an optional power-on reset input.

Power management software transitions the xHC through D0, D1, D2, and D3hot power states via xHC-owned PCI Power Management register accesses. Additional power management policy may be implemented to switch or continuously apply an Aux Power well voltage supply (e.g. PCIe *3.3Vaux* power), to the xHC when Vcc (i.e. the Core Power well voltage supply) is removed. While in this power state, referred to as D3cold, the xHC exhibits identical behavior as the D3hot power state (except that configuration space accesses are not supported) and no additional xHC hardware is required to distinguish between the D3hot and D3cold states.

Per the PCI PM specification, the xHC function asserts an internal reset during the D3hot to D0 transition. The host controller shall retain all relevant wake context when transitioning from D3hot to D0 in order for system software to process a wake request. In PCI configuration space, this means that the *PMCSR.PME_Status* and *PMCSR.PME_En* bits shall be maintained. Additionally, the *PMC.PME_Support(D3cold)* bit shall be maintained.

Additionally, the xHC shall retain function-specific context that meets any of the following criteria:

1. BIOS-configured registers that are programmed during system initialization
2. Context needed to avoid USB re-enumeration
3. Context needed for properly generating wake events
4. Status bits for software to determine the source of a wake event

Specifically, the following xHC registers shall not be reset during the D3hot to D0 transition and shall be maintained in the Aux Power well (refer to section Power State Definitions):

- USB Legacy Support Registers
- Port Status and Control Registers

Note that all of the registers described above are only reset upon initial Aux Power-up or software reset. Software should specifically clear any of these bits during subsequent initialization sequences, if desired. The memory-space bits may also be cleared using the *Host Controller Reset* (HCRST) mechanism in the USB Command Register.

A.1.2 Power State Definitions

This section defines the xHC behavior per power state when programmed using *PMCSR.PowerState*. Power management software may use alternate register mechanisms to place the xHC in similar states. The xHC shall support the D0, D3hot, and D3cold power states and is recommended that the D1, D2 power states are also supported.

Any wakeup events as specified in Table A-2 will set *PMCSR.PME_Status* when the xHC is programmed with *PMCSR.PowerState* set to D0, and a PCI PME# wake-up shall be signaled if enabled via *PMCSR.PME_En*. It is possible for one interrupt event, which is also a wakeup event to cause the xHC, to signal both a PCI interrupt and a PME# to the host. Power management software shall either be designed to handle this condition or to mask the PME# signal when the xHC is in D0.

Software shall place each downstream USB port with power enabled into the Suspend or Disabled state before it attempts to move the xHC out of the D0 power state.

All xHC contexts are retained in all power states except D3cold. For D3cold, the same context that is described in the previous section relative to the D3hot-to-D0 internal reset shall be retained.

The functional and wake-up characteristics for the xHC power states are summarized in Table A-2.

Table A-2: xHCI Power State Summary

Power State	Functional Characteristics	Wake-up Characteristics (Associated Enables shall be Set)
D0	Fully functional xHC device state. Unmasked interrupts are fully functional.	Resume Detected on suspended port. Connect or Disconnect detected on port. Over Current detected on port.
D1	xHC shall preserve PCI configuration. xHC shall preserve USB configuration. Hardware masks functional interrupts. All ports are disabled or suspended.	Resume Detected on suspended port. Connect or Disconnect detected on port. Over Current detected on port.
D2	xHC shall preserve PCI configuration. xHC shall preserve USB configuration. Hardware masks functional interrupts. All ports are disabled or suspended.	Resume Detected on suspended port. Connect or Disconnect detected on port. Over Current detected on port.
D3hot	xHC shall preserve PCI configuration. xHC shall preserve USB configuration. Hardware masks functional interrupts. All ports are disabled or suspended.	Resume Detected on suspended port. Connect or Disconnect detected on port. Over Current detected on port.
D3cold	PME Context in PCI Configuration space is preserved. Wake Context in xHC Memory Space is preserved. All ports are disabled or suspended.	Resume Detected on suspended port. Connect or Disconnect detected on port. Over Current detected on port.

Note: Software is responsible for placing root hub ports associated with devices that have been enabled for Remote Wakeup into the suspend before transitioning to a non-D0 state.

A.2 PCI PME# Signal

The PCI PME# signal shall be implemented as an open drain, active low signal that is driven low by the xHC to request a change in its current power management state. PME# has additional electrical requirements over and above standard open drain signals that allow it to be shared between devices that are powered off and those which are powered on. Refer to the PCI PM specification for more details.

Appendix B High Bandwidth Isochronous Rules

B.1 High-speed

High-speed High Bandwidth isochronous streams utilize addition PIDs in the USB2 protocol. The tables in this appendix completely enumerate all of the required responses an xHC shall make in the execution of a high-bandwidth isochronous data stream.

Each table is organized with the following fields:

- **Inputs:** lists the inputs or initial conditions for the behavioral data point. The input values are:
  - **Burst:** this is the value of the Max Burst Size field in an instantiation of an Endpoint Context. This is a constant value for the lifetime of the Endpoint Context. It serves as the initial value for Cnt (see below). This field is set based on USB framework parameters provided by the device. It is not set relative to buffer size, etc.
  - **Cnt:** this is the transaction iterator. It is the current value of an internal transaction counter that for an OUT, is initially loaded with the contents of Burst. For an IN, Cnt is initially set from the first bus transaction's PID response (see below).
  - **Remaining Buffer:** the amount of buffer remaining is indicated by the current value of the Transaction X Length field in the current transaction record. The initial value of this field is set by software to indicate the amount of buffering available for this transaction record. It is adjusted by the xHC as transactions are executed and data is moved.
- **Response:** lists the response from the device (PID code and data size) and the effects on the Transfer Event Completion Code field and transaction iterator (Cnt).
  - PID/(data size): indicates the host stimulus, data PID or other response from the device.
  - Maxpacket = value of Endpoint Context *Max Packet Size* field.
- **Result:** list the effects of the response on the bits in the Status field and the iterator.
  - Advance = Advance Dequeue Pointer to the next TD. Refer to section 4.10.1 for more information on advancement rules.
  - Babble = The assertion of a Babble Detected Error. Refer to section 4.10.2.4.
  - BufErr = The assertion of a Data Buffer Error. Refer to section 4.10.2.5.

- XactEr = The assertion of a USB Transaction Error for the TRB associated with the error. Refer to section 4.10.2.3.

Each row in each table illustrates the required xHC behavior for all of the inputs/response combinations for a HS high-bandwidth isochronous transaction. There are two tables in this appendix. The first enumerates the required behavior for OUT transactions and the second enumerates the required behavior for IN transactions.

Table B-1: HS High-Bandwidth Behavior for OUT Transactions

Inputs			Response	Results	Explanation
Burst	Cnt	Remaining Buffer	PID (data size)		
1 2 3	1	≥ Maxpacket	PID → DATA0(Maxpacket) PID → DATA1(Maxpacket) PID → DATA2(Maxpacket)	Advance	Normal completion (for micro-frame) of 1, 2 or 3 high bandwidth transaction; send Maxpacket bytes.[130]
1 2 3	1	< Maxpacket	PID → DATA0(Xfer Length) PID → DATA1(Xfer Length) PID → DATA2(Xfer Length)	Advance	Normal completion (for frame) of 1, 2, or 3 high-bandwidth transaction; send as many bytes as are available in the buffer.
2,3	2	> Maxpacket	PID → MDATA(Maxpacket)	No Advance	Intermediate transaction in high-bandwidth sequence; send Maxpacket bytes with an MDATA PID.
2 3	2	≤ Maxpacket	PID → DATA0(Xfer Length) PID → DATA1(Xfer Length)	Advance	Software did not have Burst*Maxpacket bytes to send for this transaction (microframe).
3	3	> Maxpacket	PID → MDATA(Maxpacket)	No Advance	Intermediate transaction in high-bandwidth sequence; send Maxpacket bytes with an MDATA PID.

[130]Note that the ≥ Maxpacket where the > applies is just to account for the case where software has incorrectly programmed Burst or *Max Packet Size*.

3	3	≤ Maxpacket	PID → DATA0(Xfer Length)	Advance	Software did not have Burst*Maxpacket bytes to send for this transaction (microframe).
3,2,1	>1	≥ Maxpacket	PID → MDATA(buffer error)	Advance BufErr	xHC experienced a buffer error before being able to deliver all of the data. It shall not execute any further requests on this endpoint.

Any time there is a buffer error (in this case a buffer under-run), the host controller will abandon the remaining portions of a high-bandwidth transaction. For example, if the current PID was an MDATA, and there was a buffer error on getting the data from main memory to the HC in a timely fashion, then the host controller will set the Buffer Error status bit to a '1' and immediately clear the Active status bit to '0'. This will cause the host controller to effectively skip the remaining bus transactions (if there was any pending, based on the value of Cnt).

The xHC's requirements for managing a high-bandwidth IN bus transaction sequence are described using a state machine model. The model is summarized in the state-transition table Table B-2. This is only an example state machine whose intent is to define the operational requirements of the host controller.

The intent of this section is to clearly define the appropriate data PID sequences for a high bandwidth isochronous data stream and set a priority on detection and reporting of errors that are detectable during a high-bandwidth transaction sequence.

The premise of the high-bandwidth PID tracking state machine is that the sequence of DATA PIDs for the current microframe is determined by the device's response to the first IN of the microframe. Based on PID response, the host controller sets an internal count variable (Cnt) that is used to drive the state machine through the remaining phases (states) of the high-bandwidth transaction sequence.

Each microframe, the machine is initialized to the Start state. In this state, the value of the internal counter is a don't care (X). The host controller issues the initial IN, and then sets the internal counter (Cnt) to the value number (Y) of the data PID received. For example, if the PID response is DATA2, then Cnt is loaded with the value '2'. When the PID is a DATA1 or DATA2, then two additional checks are performed. If neither of these checks fail, then the host controller transitions to the Next state.

1. The size of the data payload shall be equal to maximum packet length (Maxpacket), and

2. The host controller shall check that the starting PID response is in the range configured for this endpoint, as specified in Mult. If the PID value number (Y) is less than the value of Burst, then the received data PID is in the appropriate range. For example, if Burst is 2 and the device returns a DATA1, then Y=1 is less than Burst so the received PID is acceptable.

When the PID received in the Start state is DATA0, then the high-bandwidth transaction is complete for this microframe and the host controller shall set the Active to Inactive. A valid DATA0 PID is allowed to have a data payload size less than or equal to Maxpacket. If a babble error is detected, then the host controller will additionally set the Babble bit to a '1'.

Table B-2: HS High-Bandwidth Behavior for IN Transactions

Current State		Endpoint Response			Results	Next State	Explanation
Cnt		PID[Y]					
Start	X	PID← DATA[2,1]	Y < Burst	= Maxpacket		Cnt = [2,1]	Acceptable PID response. If no babble error, then go to Next state.
				< Maxpacket	Advance, XactErr	Done	Data payload shall be equal to maximum packet size.
				> Maxpacket	Advance, Babble	Done	Data payloads larger than maximum packet size are a babble condition.
			Y ≥ Burst	Don't care	Advance, XactErr	Done	Starting DATA PID is larger than allowed for this endpoint.
		PID← DATA0		≤ Maxpacket	Advance	Done	Acceptable PID response. If no babble error, then go to Next state.
				> Maxpacket	Advance, Babble	Done	Data payloads larger than maximum packet size are a babble condition.
Next	2	PID← DATA2		Don't care	Advance, XactEr	Done	Endpoint responded twice with DATA2 PID.

		PID← DATA1	= Maxpacket		Cnt = 1	Acceptable PID response. If no babble error, then go to Next state.
			< Maxpacket	Advance, XactErr	Done	Data payload shall be equal to maximum packet size.
			> Maxpacket	Advance, Babble	Done	Data payloads larger than maximum packet size are a babble condition.
		PID← DATA0	Don't care	Advance, XactErr	Done	Device went from DATA2 to DATA0; invalid transition.
	1	PID← DATA[2,1]	Don't care	Advance, XactErr	Done	Endpoint repeated a DATA2 or DATA1 PID.
		PID← DATA0	≤ Maxpacket	Advance	Done	Acceptable PID response. If no babble error, transaction sequence completed normally.
			> Maxpacket	Advance, Babble	Done	Data payloads larger than maximum packet size are a babble condition.

In the **Next** state, the xHC issues an IN token and checks the value number (Y) of the PID response against the value of the internal counter (Cnt). If the value number (Y) is equal to (Cnt – 1), then the PID response is correct and the host controller sets the internal counter (Cnt) to the value number of the data PID received.

When the received PID response is acceptable and is a DATA1, then the xHC shall also check that the size of the data payload is equal to the configured maximum packet length (Maxpacket). If the length check passes, the PID check has passed and the xHC does a final babble check. If no babble error, the xHC remains in the Next state and executes another bus transaction. If there was an error, the xHC flags the error and advances to the next TD. If the length check fails, the xHC generates a Transaction Error (XactErr) for the TD. If the babble check fails, the xHC shall generate a Babble Error (Babble) for the TD.

When the received PID response is acceptable and is a DATA0, then the high-bandwidth transaction is complete for this microframe and the xHC shall advance to the next TD and wait for the next Interval. The data payload is allowed to be less than or equal to the configured maximum packet size. If a babble error is detected, then the xHC shall generate a Babble Error (Babble) for the TD.

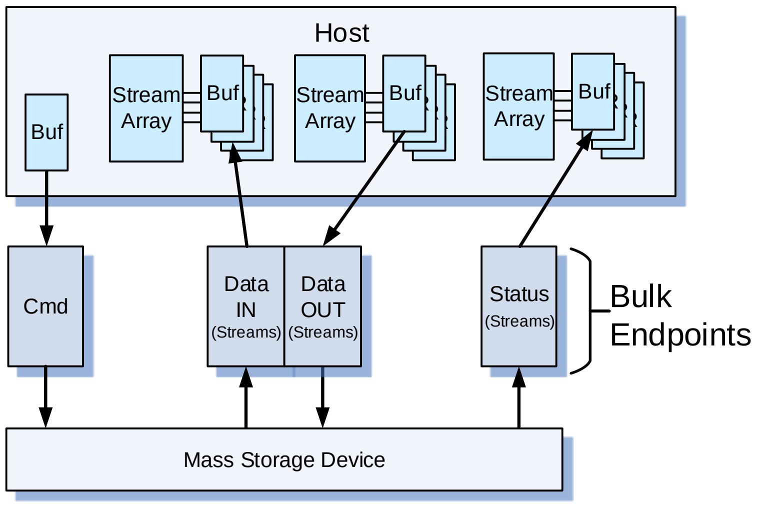
Any time the individual transaction completes in a Timeout, the xHC shall Advance to the next TD and generate a Transaction Error (XactErr) for the TD.

Note that this state machine is for illustrative purposes. Implementations may optimize appropriately to avoid arithmetic operations where possible, as long as the resultant behavior is correct.

Appendix C Stream Usage Models

The Stream Protocol may be used by USB disk drives to provide Command Queuing and First-party DMA (FPDMA) support through the xHCI. By tying a disk command with a particular Stream ID, the data associated with the command may be directed by the device to specific buffers in host memory.

Figure C-1: Mass Storage Stream Usage Model



USB Mass Storage devices utilize a three phase command execution sequence; Command, Data, And Status. Figure C-1 illustrates an example where 4 USB pipes are employed to support read and write commands to the disk; a Command OUT (Cmd) pipe, Data IN and OUT pipes, and a Status IN pipe. All are Bulk pipes, however the Data pipes also support Streams.

Consider a disk read: Before posting a disk command to the Cmd pipe, system software would first post a buffer to the Status pipe to receive the completion status for the command, and set up a Stream to receive the data associated with the command. Once both the Data and Status were set up for the command, software would post the Command to the Cmd pipe.

To post the Status buffer, software simply adds a TD to the Status IN Transfer Ring.

To set up the Stream associated with the Read Data transfer, software would select an available *Stream ID*, initialize a Transfer Ring to point to the host memory that will receive the read data, load a pointer to the Transfer Ring into the *TR Dequeue Pointer* field of the *Stream Context* in the *Stream Array* associated with the selected Stream ID, and ring the doorbell for the Data IN Endpoint. Note that the selected *Stream ID* is written to the Doorbell register when software rings the Data IN doorbell, however it is not necessary for basic Stream Protocol operation.

To post the Command, software adds a TD to the Cmd OUT Transfer Ring. The data portion of the Command packet will include the Stream ID allocated for the Command.

When the Device returns the Read Data for the Command, it uses the Stream ID provided by the Command to set the *Current Stream* in the xHC for the pipe, then moves the Read Data. The xHC uses the Current Stream to select a Stream Context in the Stream Array. The Transfer Ring referenced by the Stream Context will be used to move the Read Data into host memory.

When the Data transfer is complete, the Device sends the completion Status up the waiting Status pipe. After software receives the completion Status for the command it can free the associated Stream ID for reuse by another disk command.

Disk **Command Queuing** allows software to queue multiple Commands to a drive and the drive to decide on their order of execution. Due to the physical geometry of the disk or other internal parameters, the disk reorders Commands to minimize latency and maximize throughput. The ability for the drive to complete commands out of order is critical for Command Queuing to work. Because the disk can control Stream selection in the xHC and a different Stream ID is associated with each Command, the disk may set the *Current Stream* in xHC as function of the Command that it is currently completing.

FPDMA is enabled by the fact that separate data buffers may be assigned to each Stream. This allows the disk, as the “First Party”, to direct the data associated with a particular Command to specific buffers in host memory as a function of the Stream ID.

Streams may also be used for **Core Targeting**. Core Targeting is the ability to direct the interrupt associated with a transfer (or Command) to a specific core in a multi-core system. The fact that separate Transfer Rings may be specific for each Stream and that the Transfer Event for a TRB in a Transfer Ring can be directed at any Interrupter via the Interrupter Target field allows the device to direct completions at specific cores as function of the *Current Stream* that it selects.

Appendix D Port to Connector Mapping

This section describes an ACPI method that allows a platform to communicate to the operating system, certain USB host controller capabilities that are not provided for through the xHCI specification (e.g. If implemented, software may examine these characteristics at boot time in order to gain knowledge about the platform USB topology, mapping of xHC root hub ports to platform connectors, etc. This method is also applicable to topologies that include USB hubs that are integrated with the xHC silicon or implemented as discrete components on the motherboard.

This method utilizes the ACPI USB Port Capabilities (_UPC, refer to section 9.14 in the ACPI spec) and Physical Device Location (_PLD, refer to section 6.1.6 in the ACPI spec) objects.

Note: The _UPC declarations for LS/FS/HS and Enhanced SS ports that are grouped to form a USB3 compatible connector. A "group" is defined by two or more ports that declare _PLDs with identical **Panel**, **Vertical Position**, **Horizontal Position**, **Shape**, **Group Orientation**, **Group Position** and **Group Token** parameter values.

D.1 Example

The following is an example of the ACPI objects defined for an xHC that implements a High-speed and SuperSpeed Bus Instance, that are associated with USB2 and USB3 Protocol Root Hub Ports, respectively. The xHC also supports an integrated High-speed hub to provide Low- and Full-speed functionality. The External Ports defined by the xHC implementation provide either a USB2 data bus (i.e. a D+/D- signal pair) or an Enhanced SuperSpeed data bus (e.g. SSRx+/SSRx- and SSTx+/SSTx- signal pairs).

Where:

- The motherboard presents 5 user visible connectors C1 – C5.
  - Motherboard connectors C1 and C2 support USB2 (LS/FS/HS) devices.
  - Motherboard connectors C3, C4 and C5 support USB3 (LS/FS/HS/SS) devices.
- The xHC implements a High-speed Bus Instance associated with USB2 Protocol Root Hub ports, e.g. HCP1 and HCP2 are High-speed only, i.e. they provide no Low- or Full-speed support.
- The xHC presents 7 External Ports (P1 – P7).
  - External Port 1 (P1) is HS only and is not visible or connectable.
  - External Ports 2 – 5 (P2 – P5) support LS/FS/HS devices.
    - P2 is attached to motherboard USB2 connector C1.
    - P3 is attached to motherboard USB2 connector C2.
    - P4 is attached to the USB 2.0 logical hub of the Embedded USB3 Hub on the

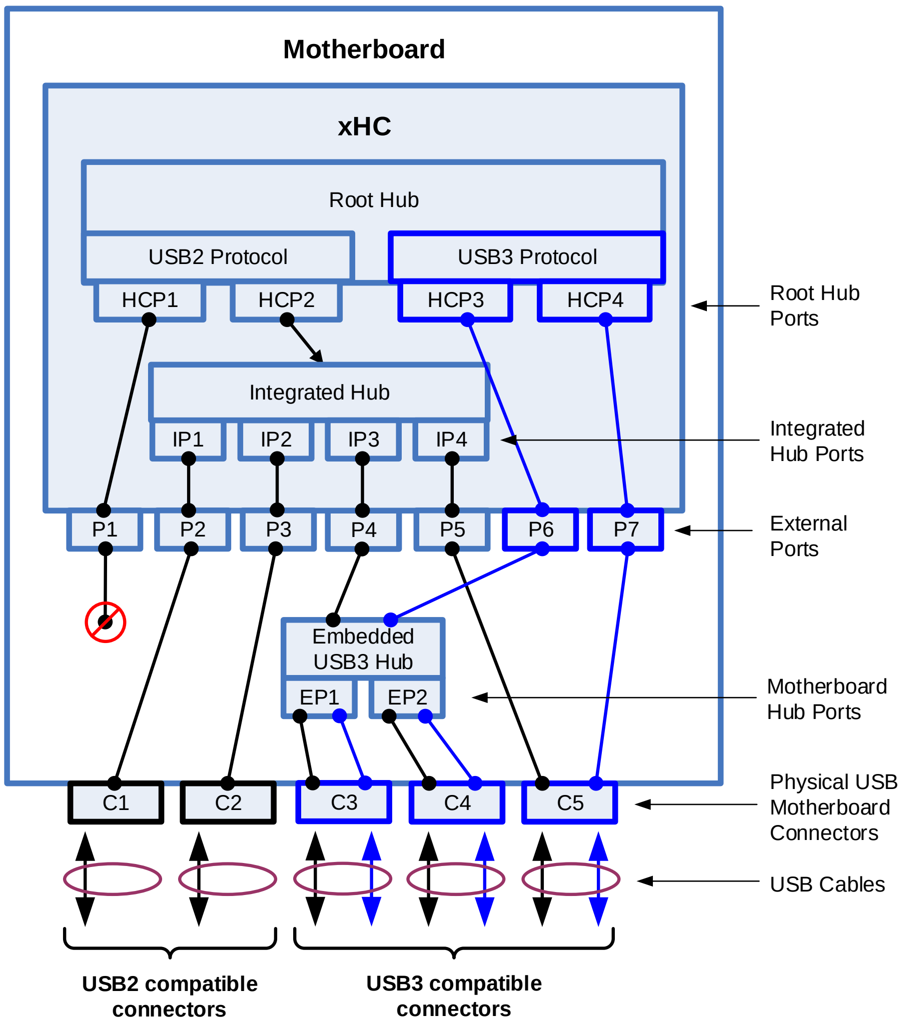
motherboard. The USB 2.0 logical hub supports the LS/FS/HS connections for 2 ports (EP1 – EP2).

        - The USB 2.0 connections of motherboard hub ports EP1 and EP2 are attached to motherboard connectors C3 and C4 respectively, providing the LS/FS/HS support for the USB3 connectors.

    - P5 is attached to motherboard connector C5, providing the LS/FS/HS support to the motherboard USB3 connector C5.
    - External Port 6 (P6) is attached to the SuperSpeed logical hub of the Embedded USB3 Hub on the motherboard. The SuperSpeed logical hub supports the SS connections of 2 ports (EP1 – EP2).
        - The SuperSpeed connections of motherboard hub ports EP1 and EP2 are attached to motherboard connectors C3 and C4 respectively, providing the SS support for the USB3 connectors.
  - External Port 7 (P7) is attached to motherboard connectors C5, providing the SS support for the USB3 connector.

- The xHC implements 4 internal HS Root Hub ports (HCP1 – HCP4), 2 High-speed and 2 SuperSpeed.
  - Internal Port 1 (HCP1) maps directly to External Port 1 (P1).
  - Internal Port 2 (HCP2) is attached to a HS Integrated Hub. The Integrated Hub supports 4 ports (IP1 – IP4).
    - Ports 1 to 4 (IP1-IP4) of the Integrated Hub attach to External Ports 2 to 5 (P2-P5), respectively.
  - Internal Ports 3 and 4 (HCP3, HCP4) attach to External Ports 6 and 7 (P6, P7), respectively.
- All connectors are located on the back panel and assigned to the same Group.
- Connectors C1 and C2 are USB2 compatible and their color is not specified. Connectors C3 to C5 are USB3 compatible and their color is specified.
- External Ports P1 - P5 present a USB2 data bus (i.e. a D+/D- signal pair). External Ports P6 and P7 present a SuperSpeed data bus (i.e. SSRx+/SSRx- and SSTx+/SSTx- signal pairs).

Figure D-1: Root Hub Port to USB Connector Mapping Example



D.1.1 ACPI Code Example

Note: In the Intel ASL (iASL) compiler an ACPI Buffer takes a list of bytes (not Dwords). In the following example Dwords were used to permit a more compact description, where the general notation is 0x*mm*xxxx*ll*, *mm* = the Most significant byte and *ll* = the least significant byte of the Dword.

```
Scope( \_SB ) {
    ...
    Device( PCI0 ) {
        ...
        // Host controller ( xHCI )
        Device( USB0 ) {
```

```
// PCI device#/Function# for this HC. Encoded as specified in the ACPI
// specification
Name( _ADR, 0xyyyyzzzz )
// Root hub device for this HC #1.
Device( RHUB ) {
    Name( _ADR, 0x00000000 ) // must be zero for USB root hub
    // Root Hub port 1 ( HCP1 )
    Device( HCP1 ) {          // USB0.RHUB.HCP1
        Name( _ADR, 0x00000001 )
        // USB port configuration object. This object returns the system
        // specific USB port configuration information for port number 1
        Name( _UPC, Package() {
            0x01,                  // Port is connectable but not visible
            0xFF,                  // Connector type (N/A for non-visible ports)
            0x00000000, // Reserved 0 – must be zero
            0x00000000} )          // Reserved 1 – must be zero
    } // Device( HCP1 )
    // Root Hub port 2 ( HCP2 )
    Device( HCP2 ) {          // USB0.RHUB.HCP2
        Name( _ADR, 0x00000002 )
        Name( _UPC, Package() {
            0xFF,                  // Port is connectable
            0x00,                  // Connector type – (N/A for non-visible ports)
            0x00000000, // Reserved 0 – must be zero
            0x00000000} )          // Reserved 1 – must be zero
            // provide internal connection point info
            Name( _PLD, Buffer( 0x10) {
                0x00000081,        // Revision 1, Ignore color
                                   // Color (ignored), width and height not
                0x00000000,        // required as this is not a user visble
                                   // connector
                0x00808000,        // Not user visible, Group Token = 1,
                                   // Group Position 1 (This is the group for all
                                   // internal connections. Each connection should
                                   // have a unique position in this group)
                0x00000000} )      // Ignored for not visible connectors
            // Integrated hub port 1 ( IP1 )
            Device( IP1 ) { // USB0.RHUB.HCP2.IP1
                // Address object for the port. Because the port is
                // implemented on integrated hub port #1, this value must be 1
                Name( _ADR, 0x00000001 )
                Name( _UPC, Package() {
                    0xFF,          // Port is connectable
                    0x00,                      // Connector type – Type 'A'
                    0x00000000,                // Reserved 0 – must be zero
                    0x00000000} )              // Reserved 1 – must be zero
                // provide physical connector location info
                Name( _PLD, Buffer( 0x10) {
                    0x00000081,    // Revision 1, Ignore color
                                   // Color (ignored), width and height not
                    0x00000000,    // required as this is a standard USB 'A' type
                                   // connector
                0x00800c69,// User visible, Back panel, Center, left,
                            // shape = vert. rect, Group Token = 0,
                            // Group Position 1 (i.e. Connector C1)
                0x00000003} )// ejectable, requires OPSM eject assistance
        } // Device( IP1 )
        // Integrated Hub port 2 ( IP2 )
        Device( IP2 ) {// USB0.RHUB.HCP2.IP2
            // Address object for the port. Because the port is
            // implemented on integrated hub port #2, this value must be 2
            Name( _ADR, 0x00000002 )
            Name( _UPC, Package() {
                0xFF,                   // Port is connectable
                0x00,                   // Connector type – Type 'A'
                0x00000000,  // Reserved 0 – must be zero
                0x00000000} )           // Reserved 1 – must be zero
            // provide physical connector location info
            Name( _PLD, Buffer( 0x10) {
                0x00000081,  // Revision 1, Ignore color
                             // Color (ignored), width and height not
                0x00000000,  // required as this is a standard USB 'A' type
                             // connector
                0x01000c69,  // User visible, Back panel, Center, Left,
                             // Shape = vert. rect, Group Token = 0,
                             // Group Position 2 (i.e. Connector C2)
                0x00000003} ) // ejectable, requires OPSM eject assistance
        } // Device( IP2 )
        // Integrated Hub port 3 ( IP3 )
        Device( IP3 ) {// USB0.RHUB.HCP2.IP3
            // Address object for the port. Because the port is implemented
            // on integrated hub port #3, this value must be 3
            Name( _ADR, 0x00000003 )// Must match the _UPC declaration for
                                    // USB0.RHUB.HCP3 as this port shares
                                    // the same connection point.
            Name( _UPC, Package() {
                0xFF,// Port is connectable
                0x00,        // Connector type – (N/A for non-visible ports)
                0x00000000,  // Reserved 0 – must be zero
                0x00000000} )// Reserved 1 – must be zero
                // provide internal connection point info
            Name( _PLD, Buffer( 0x10) {
                0x00000081,  // Revision 1, Ignore color
                             // Color (ignored), width and height not
                0x00000000,  // required as this is not a user visble
                             // connector
                0x01008000,  // Not user visible, Group Token = 1,
                             // Group Position 2
                0x00000000} )// Ignored for not visible connectors
                // Motherboard Embedded Hub 2.0 Logical Hub port 1 ( EP1 )
                Device( EP1 ) { // USB0.RHUB.HCP2.IP3.EP1
                    Name( _ADR, 0x00000001 )
                    // Must match the _UPC declaration for
                    // USB0.RHUB.HCP3.EP1 as this port provides
            // the LS/FS/HS connection for C3
            Name( _UPC, Package() {
                0xFF,  // Port is connectable
                0x03, // Connector type – USB 3 Type 'A'
                0x00000000,         // Reserved 0 – must be zero
                0x00000000} )       // Reserved 1 – must be zero
            // provide physical connector location info
            Name( _PLD, Buffer( 0x10) {
                0x0072C601,         // Revision 1, Color valid
                                    // Color (0072C6h), width and height
                0x00000000,         // not required as this is a standard
                                    // USB 'A' type connector
                0x01800c69,         // User visible, Back panel, Center,
                                    // Left, shape = vert.
                                    // rect, Group Token = 0,
                                    // Group Position 3
                                    //(i.e. Connector C3)
                0x00000003} )       // ejectable, requires OPSM eject
                                    // assistance
        } // Device(EP1)
        // Motherboard Embedded Hub 2.0 Logical Hub port 2 ( EP2 )
        Device( EP2 ) { //USB0.RHUB.HCP2.IP3.EP2
            Name( _ADR, 0x00000002 )
            // Must match the _UPC declaration for
            // USB0.RHUB.HCP3.EP2 as this port provides
            // the LS/FS/HS connection for C4
            Name( _UPC, Package() {
                0xFF,  // Port is connectable
                0x03, // Connector type – USB 3 Type 'A'
                0x00000000,         // Reserved 0 – must be zero
                0x00000000} )       // Reserved 1 – must be zero
            // provide physical connector location info
            Name( _PLD, Buffer( 0x10) {
                0x0072C601,         // Revision 1, Color valid
                                    // Color (0072C6h), width and height
                0x00000000,         // not required as this is a standard
                                    // USB 'A' type connector
                0x02000c69,         // User visible, Back panel, Center,
                                    // Left, Shape = vert.
                                    // rect, Group Token = 0,
                                    // Group Position 4 (i.e. Connector C4)
                0x00000003} )       // ejectable, requires OPSM eject
                                    // assistance
        } // Device( EP2 )
    } // Device( IP3 )

    // Integrated hub port 4 ( IP4 )
    Device( IP4 ) { // USB0.RHUB.HCP2.IP4
        Name(_ADR, 0x00000004)
        // Must match the _UPC declaration for USB0.RHUB.HCP4 as
        // this port provides the LS/FS/HS connection for C5
        Name( _UPC, Package() {
                0xFF,                   // Port is connectable
                0x03,                   // Connector type – USB 3 Type 'A'
                0x00000000,             // Reserved 0 – must be zero
                0x00000000} )           // Reserved 1 – must be zero
            // provide physical connector location info
            Name( _PLD, Buffer(0x10) {
                0x0072C601, // Revision 1, Color valid
                          // Color (0072C6h), width and height not
                0x00000000, // required as this is a standard USB
                          // 'A' type connector
                0x02800c69, // User visible, Back panel, Center, Left,
                          // Shape = vert. rectangle, Group Token = 0,
                          // Group Position 5 (i.e. Connector C5)
                0x00000003} )// ejectable, requires OPSM eject
                           // assistance
        } // Device( IP4 )
    } // Device( HCP2 )

    // Root Hub port 3 ( HCP3 )
    Device( HCP3 ) {
        Name( _ADR, 0x00000003 )
        // Must match the _UPC declaration for USB0.RHUB.HPC2.IP3 as
        // this port shares the connection point
        Name( _UPC, Package() {
            0xFF,// Port is connectable
            0x00,          // Connector type – (N/A for non-visible ports)
            0x00000000,    // Reserved 0 – must be zero
            0x00000000} )// Reserved 1 – must be zero
        // Internal connection points require a _PLD that identifies
        // the shared connection point info
        Name( _PLD, Buffer( 0x10) {
            0x00000081,    // Revision 1, Ignore color
                           // Color (ignored), width and height not
            0x00000000,    // required as this is not a user visible
                           // connector
            0x01008000,    // Not user visible, Group Token = 1,
                           // Group Position 2
            0x00000000} )// Ignored for not visible connectors
        // Motherboard Embedded Hub SS Logical Hub port 1 ( EP1 )
        Device( EP1 ) {// USB0.RHUB.HCP3.EP1
            Name( _ADR, 0x00000001 )
            // Must match the _UPC declaration for
            // USB0.RHUB.HCP2.IHUB.IP3.EHUB.EP1 as this port
            // provides the SS connection for C3
            Name( _UPC, Package() {
                0xFF,                   // Port is connectable
                0x03,                   // Connector type – USB 3 Type 'A'
                0x00000000,             // Reserved 0 – must be zero
                0x00000000} )           // Reserved 1 – must be zero
            // provide physical connector location info
            Name( _PLD, Buffer( 0x10) {
                0x0072C601,             // Revision 1, Color valid
```

```
                                   // Color (0072C6h), width and height
            0x00000000,            // not required as this is a standard
                                   // USB 'A' type connector
            0x01800c69,            // User visible, Back panel, Center,
                                   // Left, shape = vert.
                                   // rect, Group Token = 0,
                                   // Group Position 3
                                   //(i.e. Connector C3)
            0x00000003} )          // ejectable, requires OPSM eject
                                   // assistance
} // Device(EP1)
// Motherboard Embedded Hub SS Logical Hub port 2 ( EP2 )
Device( EP2 ) {// USB0.RHUB.HCP3.EP2
        Name( _ADR, 0x00000002 )
        // Must match the _UPC declaration for
        // USB0.RHUB.HCP2.IHUB.IP3.EP2 as this port
        // provides the SS connection for C4
        Name( _UPC, Package() {
            0xFF,                  // Port is connectable
            0x03,                  // Connector type – USB 3 Type 'A'
            0x00000000,            // Reserved 0 – must be zero
            0x00000000} )          // Reserved 1 – must be zero
        // provide physical connector location info
        Name( _PLD, Buffer( 0x10) {
            0x0072C601,            // Revision 1, Color valid
                                   // Color (0072C6h), width and height
            0x00000000,            // not required as this is a standard
                                   // USB 'A' type connector
            0x02000c69,            // User visible, Back panel, Center,
                                   // Left, Shape = vert.
                                   // rect, Group Token = 0,
                                   // Group Position 4 (i.e. Connector C4)
            0x00000003} )          // ejectable, requires OPSM eject
                                   // assistance
        } // Device( EP2 )
} // Device( HCP3 )
// Root Hub port 4 ( HCP4 )
Device( HCP4 ) { // USB0.RHUB.HCP4
        Name( _ADR, 0x00000004 )
        // Must match the _UPC declaration for USB0.RHUB.HCP2.IP4
        // as this port provides the SS connection for C5
        Name( _UPC, Package() {
            0xFF,                  // Port is connectable
            0x03,                  // Connector type – USB 3 Type 'A'
            0x00000000,            // Reserved 0 – must be zero
            0x00000000} )          // Reserved 1 – must be zero
        // provide physical connector location info
        Name( _PLD, Buffer( 0x10) {
            0x0072C601,            // Revision 1, Color valid
                                   // Color (0072C6h), width and height
            0x00000000,            // not required as this is a standard
                                   // USB 'A' type connector
```

```
                                    0x02800c69,            // User visible, Back panel, Center,
                                                           // Left,
                                                           // Shape = vert. rect, Group Token = 0,
                                                           // Group Position 5 (i.e. Connector C5)
                                    0x00000003} )          // ejectable, requires OPSM eject
                                                           // assistance
                              } // Device( HCP4 )
                        } // Device( RHUB )
                        ...
                  } // Device( USB0 )
                  //
                  // Define other control methods, etc
                  ...
            } // Device( PCIO )
            ...
      } // Scope( \_SB )
```

Note: The USB spec recommends that USB 3.0 specific connectors are identified with a standardized blue color (Pantone 300C). In this example Pantone 300C is mapped to the RGB value of 0(R), 114(G), 198(B) (0072C6h).

Appendix E State Machine Notation

State diagrams should not be taken as a required implementation, but to specify the required behavior.

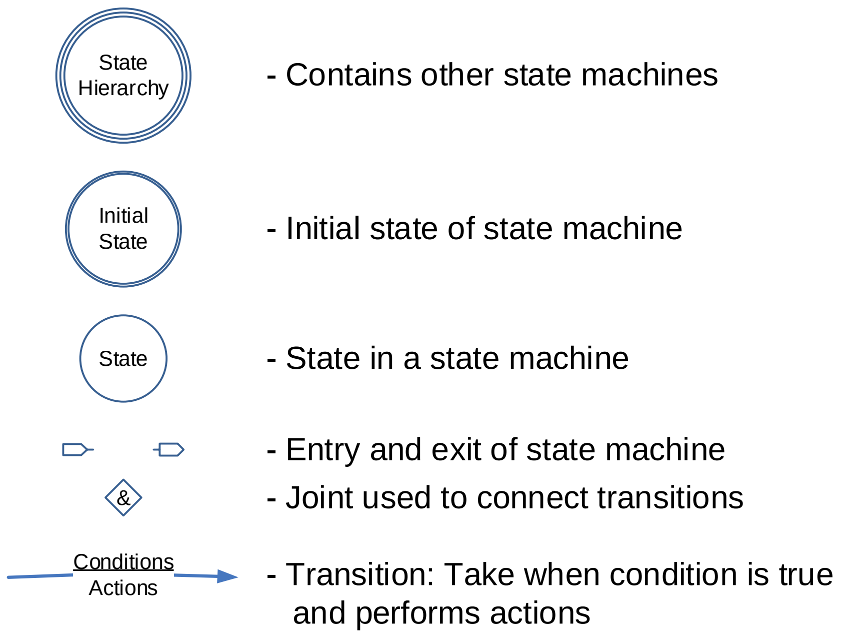
Figure 8-25 shows the legend for the state machine diagrams. A circle with a three line border indicates a reference to another (hierarchical) state machine. A circle with a two line border indicates an initial state. A circle with a single line border is a simple state.

The Entry and Exit symbols are used by lower lever state machines to indicate an entry from, or an exit to, a higher level state machine.

A diamond (joint) is used to join several transitions to a common point. A joint allows a single input transition with multiple output transitions or multiple input transitions and a single output transition. All conditions on the transitions of a path involving a joint must be true for the path to be taken. A path is simply a sequence of transitions involving one or more joints.

A transition is labeled with a block with a line in the middle separating the (upper) *Conditions* and the (lower) *Actions*. If no line is displayed the transition label is a *Condition*. The Condition is required to be true to take the transition. The Actions are performed if the transition is taken. The syntax for actions and conditions is VHDL. A circle includes a state name in bold and optionally additional state information, e.g. one or more actions that are performed upon entry to the state, signal states, etc.

Figure E-1: Legend for State Machines


State Hierarchy
- Contains other state machines
Initial State
- Initial state of state machine
State
- State in a state machine
- Entry and exit of state machine
&
- Joint used to connect transitions
Conditions
Actions
- Transition: Take when condition is true and performs actions

Note: The xHCI state machines describe the exit conditions from a state, and entry conditions to a state. Only conditions specifically described as an entry or exit condition shall result in a state transition.

Appendix F SS Bus Access Constraints

The following tables calculate the transaction limits for transfers on a downstream link, with the assumption that the upstream link is idle.

Refer to 7.2.1.2.3 in the USB3 spec for the overhead (32 symbols) associated with a SS DP (DPH + DPP).

Refer to 7.2.2.1 in the USB3 spec for the symbol overhead associated with a SS Link Command. Two Link Commands (an GOOD_n and an L_CRD) are transmitted on the downstream link for every header (TP or DPH) received on the upstream link.

Refer to 7.2.1.1.1 in the USB3 spec for the overhead (20 bytes) associated with each SS TP (Header Packet).

Table Labels

Protocol Overhead

> The downstream link overhead in bytes. The components of overhead are described by the cell to the right.

TD Transfer Size

> TD Transfer Size in bytes.

Max Bandwidth

> The maximum achievable bandwidth given the TD Transfer Size in KBytes/second.

% Microframe Bandwidth per TD

> The percentage of microframe bandwidth consumed by a single TD.

Max TDs

> The maximum number of TD Transfer Size TDs than may be scheduled per microframe.

Bytes Remaining

> The remaining byte times in a microframe after transferring one TD.

Bytes/Microframe Useful Data

> TD Transfer Size * Max TDs

F.1 Bulk Transfer Bus Access Constraints

Refer to section 5.8.4 of the USB2 spec for a general overview of USB bulk transfer access constraints, and for the Full-speed and High-speed Transaction Limits.

The bus frequency and microframe timing limit the maximum number of SuperSpeed bulk DPs within a microframe for any USB3 system to less than 905 one-byte data payloads. Table F-1 lists information about different-sized SuperSpeed bulk transactions and the maximum number of transactions possible in a microframe, for the downstream link of a bulk OUT pipe while the upstream link is saturated with bulk IN traffic.

The Protocol Overhead is calculated for the downstream link as follows: For each DP moved for a TD in the OUT direction (32B), there is one ACK TP for the DP in the IN direction, which requires 1 LGOOD_n and 1 L_CRD Link Command (8B each) to be transmitted in the OUT direction for a total of 48 bytes.

Table F-1: SuperSpeed Bulk OUT Transaction Limits

Protocol Overhead (48B)		1 DP, 2 Link Commands			
TD Transfer Size	Max Bandwidth (KBytes/second)	% Microframe Bandwidth per TD	Max TDs	Bytes Remaining	Bytes/Microframe Useful Data
1	10200	1	1275	25	1275
2	20000	1	1250	0	2500
4	38432	1	1201	48	4804
8	71424	1	1116	4	8928
16	124928	1	976	36	15616
32	199936	1	781	20	24992
64	285696	1	558	4	35712
128	363520	1	355	20	45440
256	419840	1	205	180	52480
512	454656	1	111	340	56832

1024	475136	2	58	324	59392
2048	475136	4	29	324	59392
4096	458752	7	14	2468	57344
8192	458752	14	7	2468	57344
16384	393216	28	3	11044	49152
32768	262144	55	1	28196	32768
59392	475136	100	1	324	59392

xHC implementations are free to determine how the individual bus transactions for specific bulk transfers are moved over the bus within and across microframes. An endpoint could see all bus transactions for a bulk transfer within the same microframe or spread across several microframes. An xHC, for various implementation reasons, may not be able to provide the above maximum number of transactions per (micro)frame.

Note: For a given TD Transfer Size, simultaneous bulk IN and OUT transfers would incur an additional 36 bytes of Protocol Overhead per OUT TD, i.e. 1 for the IN DP's ACK TP (20B) and 2 Link Commands for the IN DP (8B each).

F.2 Interrupt Transfer Bus Access Constraints

SuperSpeed endpoints can be allocated at most 90% of a microframe for periodic transfers. The bus frequency and microframe timing limit the maximum number of SuperSpeed interrupt DPs within a microframe for any USB3 system to less than 1025 one-byte data payloads. Table F-2 lists information about different-sized SuperSpeed interrupt transactions and the maximum number of transactions possible in a microframe.

The Protocol Overhead is calculated identically to bulk transfers.

No more than 3 Max Packet Size DPs (3KB or 3072B) may be scheduled for a single interrupt endpoint within a single microframe, i.e. the minimum ESIT. Interrupt TDs that exceed 3KB shall transfer over multiple ESITs at up to 3KB per ESIT.

Table F-2: SuperSpeed Interrupt Transaction Limits

Protocol Overhead (48B)		1 DP, 2 Link Commands			
TD Transfer Size	Max Bandwidth (KBytes/second)	% Microframe Bandwidth per TD	Max TDs	Bytes Remaining	Bytes/Microframe Useful Data
1	10200	1	1275	25	1275
2	20000	1	1250	0	2500
4	38432	1	1201	48	4804
8	71424	1	1116	4	8928
16	124928	1	976	36	15616
32	199936	1	781	20	24992
64	285696	1	558	4	35712
128	363520	1	355	20	45440
256	419840	1	205	180	52480
512	454656	1	111	340	56832
1024	475136	2	58	324	59392
2048	475136	4	29	324	59392
3072	466944	6	19	1396	58368

Note: For a given TD Transfer Size, simultaneous interrupt IN and OUT transfers would incur an additional 36 bytes of Protocol Overhead on the downstream link per OUT TD, i.e. 1 for the IN DP's ACK TP (20B) and 2 Link Commands for the IN DP (8B each).

F.3 Isochronous Transfer Bus Access Constraints

SuperSpeed endpoints can be allocated at most 90% of a microframe for periodic transfers. The bus frequency and microframe timing limit the maximum number of SuperSpeed Isoch DPs within a microframe for any USB3 system to less than 1025 one-byte data payloads. Table F-3 lists information about different-sized SuperSpeed isochronous transactions and the maximum number of transactions possible in a microframe.

For only Isoch OUT transfers the downstream Protocol Overhead is that associated with the transmission of a single DP (32B).

No more that 48 Max Packet Size DPs (48KB or 49152B) may be scheduled for a single Isoch endpoint within a single microframe, i.e. the minimum ESIT. If an Isoch *TD Transfer Size* exceeds the *Max ESIT Payload* or the *Maximum Allowed ESIT Payload* (48KB), then a *Bandwidth Overrun Error* shall be generated.

Table F-3: SuperSpeed Isoch Transaction Limits

Protocol Overhead (32B)		1 DP, 1 Link Command			
TD Transfer Size	Max Bandwidth (KBytes/second)	% Microframe Bandwidth per TD	Max TDs	Bytes Remaining	Bytes/ Microframe Useful Data
1	15144	1	1893	31	1893
2	29408	1	1838	8	3676
4	55552	1	1736	4	6944
8	99968	1	1562	20	12496
16	166656	1	1302	4	20832
32	249856	1	976	36	31232
64	333312	1	651	4	41664
128	399360	1	390	100	49920
256	444416	1	217	4	55552
512	466944	1	114	484	58368
1024	483328	2	59	196	60416
2048	475136	4	29	1252	59392
4096	458752	7	14	3364	57344
8192	458752	14	7	3364	57344
16384	393216	28	3	11812	49152

32768	262144	55	1	28708	32768
49152	393216	82	1	11812	49152

Note: For a given TD Transfer Size, simultaneous isoch IN and OUT transfers would incur an additional 16 bytes of Protocol Overhead on the downstream link per OUT TD, i.e. 2 Link Commands for the IN DP (8B each).

Appendix G 0.96 Exceptions

This appendix defines the significant differences between 0.96 and 1.0 implementations. See following exceptions:

G.1 Skip Link TRB IOC flag

Section 4.10.1.1 of the 0.96 release was silent on the handling Link TRBs while advancing to the next TD after the detection of a Short Packet. Some 0.96 implementations may not generate an event if it encounters a Link TRB with its IOC flag set while advancing to the next TD.

G.2 Force Stopped Event Optional

Forced Stopped Event support was optional for 0.96 implementations, but is required in 1.0. Refer to section 4.6.9. In 0.96 implementations bit 8 of the HCCPARAMS1 register was defined as follows:

Table G-1: Forced Stopped Event (FSE) Option Flag

Bit	Description
8	**Force Stopped Event (FSE)**. This flag indicates whether the host controller implementation generates a Stopped Transfer Event when a Transfer Ring stops between TDs. A '1' in this bit indicates that Forced Stopped Events are supported. A '0' in this bit indicates that Forced Stopped Events are not supported. Refer to Section 4.6.9 for more information on the use of this flag.

G.3 Secondary Bandwidth Domain Reporting Optional

Secondary Bandwidth Domain Reporting support was optional for 0.96 implementations, but is required in 1.0. Refer to section 4.16.2. In 0.96 implementations bit 9 of the HCCPARAMS1 register was defined as follows:

Table G-2: Secondary Bandwidth Domain Reporting (SBD) Option Flag

Bits	Description
9	**Secondary Bandwidth Domain Reporting (SBD)**. This flag indicates whether the host controller implementation is capable of reporting Secondary Bandwidth Domain information. A ‘1’ in this bit indicates that Secondary Bandwidth Domain reporting is supported. A ‘0’ in this bit indicates that Secondary Bandwidth Domain reporting is not supported. Refer to Section 4.16.2 for more information on the use of this flag.

G.4 USB2 L1 Capability Optional

L1 Capability support was optional for 0.96 implementations. Refer to section 4.23.5.1.1. In 0.96 implementations bit 16 at Dword offset 08h of the *xHCI Supported Protocol Capability* was defined as follows:

Table G-3: L1 Capability (L1C) Option Flag

Bits	Description
16	**L1 Capability (L1C) - RO**. Default = Implementation dependent. If this bit is set to ‘1’ the xHC supports the USB2 Link Power Management L1 (Sleep) state and the associated USB2 protocol fields as defined in the PORTSC and USB2 PORTPMSC registers are valid, specifically USB2 protocol functionality of the *PLS* and *PLC* fields in the PORTSC register, and the fields of the USB2 PORTPMSC register. Note that software is prohibited from using the *PLS* field initiate a transition to an L1 state or using the USB2 PORTPMSC fields unless this bit is set to ‘1’.

Appendix H Release 1.1 Notes

H.1 Required 1.0 Capabilities/Features

The following capabilities/features that were optional for xHCI 1.0 implementations are now required in xHCI 1.1 implementations.

H.1.1 Hardware LMP Capability

The *Hardware LMP Capability* (HLE = ‘1’) and *BESL LMP Capability* (BLC = ‘1’), refer to section 4.23.5.1.1.

H.1.2 Contiguous Frame ID Capability

The *Contiguous Frame ID Capability* (CFC = ‘1’), refer to section 4.11.2.5.

H.1.3 Stopped EDTLA Capability

The *Stopped EDTLA Capability* (SEC = '1'), refer to section 4.12.

H.1.4 U3 Entry Capability

The *U3 Entry Capability* (U3C = '1') refer to section 4.15.1.

H.1.5 Stopped - Short Packet Capability

The *Stopped - Short Packet Capability* (SPC = '1'), refer to section 4.6.9.

H.1.6 Force Save Context Capability

The *Force Save Context Capability* (FSC = '1'), refer to section 5.3.9.

H.1.7 Compliance Transition Capability

The *Compliance Transition Capability* (CTC = '1'), refer to section 4.19.1.2.4.1.

H.1.8 Configuration Information Capability

The *Configuration Information Capability* (CIC = '1'), refer to section 6.2.5.1. xHC 1.1 compliant drivers shall always set CIE = ‘1’ and provide extended Configuration Information.

H.2 New 1.1 Features

The following capabilities/features are new features required in xHCI 1.1 implementations.

H.2.1 Ring Underrun/Overrun Transfer Event Handling

Ring Underrun and *Ring Overrun* Transfer Events shall set the *TRB Pointer* field to the address of the invalid TRB, refer to section 4.11.3.1.

§ §