Enhanced Host Controller Interface Specification for Universal Serial Bus

Date: March 12, 2002

Revision: 1.0

Revision History

Revision	Issue Date	Comments
0.8a	7/20/1999	Initial Revision
0.8b	8/10/1999	Added information in data structures to support split transactions, etc. Started adding information into operational model (Chapter 4).
0.9 rc1	12/21/1999	Significant additions to complete definition for register space, data structures and operational model.
0.9 rc2	1/3/2000	Updated based on 0.9rc1 feedback.
0.9 rc5	1/12/2000	Updated based on 0.9 rc2 feedback. Internal, incremental revisions account rcX jump.
0.9	1/13/2000	Editorial changes from 0.9 rc5.
0.91	7/6/2000	Update for yellow cover release. Also includes additions for debug port.
0.95 rc1	9/26/2000	Editorial updates. Deleted Chapter 5, Legacy Keyboard.
0.95	11/10/2000	Final editorial updates.
0.96 rc1	4/23/2001	Editorial clarifications; Fix for erratum on handling *CErr* during Interrupt split transactions and new features for managing frame-wrap FS/LS interrupt, Asynchronous Park-mode, FS/LS Rebalancing and new EHCI ownership semaphores.
0.96.rc2	5/30/2001	Fixed numerous typos, relaxed requirements on park mode, added CErr handling for FS/LS Setup transactions and numerous other clarifications.
0.96.rc3	6/12/2001	Added requirement to handling of full-speed isochronous-IN data streams (for compatibility).
0.96	6/20/2001	Final editorial edits, pagination, etc.
1.0 rc1	1/31/2002	Editorial changes accumulated from industry feedback.
1.0	3/13/2002	Editorial changes accumulated during Licensee Review Period.

NOTE TO SOFTWARE DEVELOPERS:

Revisions of the EHCI specification have introduced new features in the programming interface. Software must not attempt to use features defined in recent revisions of the specification on host controllers designed to older revisions. A summary of new features, per revision is provided below.

Revision	**Software-visible New Feature**
0.96	• Frame Span Traversal Nodes (FSTN), see Section 3.7.
	• Rebalance Lockout (I-Bit), see Sections 3.6, 4.12.2.5
	• Asynchronous Park Mode, see Sections 2.2.4, 2.3.1, and 4.10.3.2.
	• EHCI Extended Capabilities, see Section 5.
	• Legacy Support, see Section 5.1

Significant Contributors:

John S. Howard	Intel Corporation	Nobuo Furuya	NEC
Darren Abramson	Intel Corporation	James E. Guziak	Lucent
Michael N. Derr	Intel Corporation	Brian Leete	Intel Corporation
John Garney	Intel Corporation	Brad Hosler	Intel Corporation
Karthi Vadivelu	Intel Corporation	Chris Robinson	Microsoft
Ken Stufflebeam	Compaq		

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

Page intentionally left blank

Table of Contents

1. Introduction

The Enhanced Host Controller Interface (EHCI) specification describes the register-level interface for a Host Controller for the Universal Serial Bus (USB) Revision 2.0. 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 *Universal Serial Bus Specification*, Revision 2.0. In spite of due diligence, there may exist conflicts between this specification and the USB 2.0 Specification. The USB 2.0 Specification takes precedence on all issues of conflict.

Some key features of the EHCI specification are:

- **Full, Robust Support for all USB 2.0 Features.** This specification describes a host controller that correctly supports all compliant USB 2.0 Low-, Full-, and High-speed devices. This includes new USB 2.0 features such as split transactions for USB 2.0 Hubs and other extensions to the protocol such as the new PING protocol for high-speed OUT endpoints.
- **Low-risk support for Full- and Low-speed peripherals.** The EHCI specification provides support for all three device speeds on the root ports by integrating (using) existing hardware and software for USB 1.1 host controllers for support of Full- and Low-speed devices connected to the root ports. This allows the EHCI to focus on efficient support of high-speed operation, without having to accommodate any trade-offs in complexity or performance to support Full- and Low-speed devices.
- **System Power Management**. Current PC architectures are providing ubiquitous support for aggressive power management. 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.
- **Provides simple, robust solutions to USB 1.1 Host Controller Issues.** The EHCI host controller specification contains solutions to a myriad of issues, which have proven to be problematic for USB host controllers. Some of the issues resolved in the EHCI specification include: Memory thrashing, Memory access efficiency, and conflicts with cpu power management. The EHCI architecture provides both new specific features and optimizations to its architecture to solve the legacy issues.
- **Optimized for Best Memory Access Efficiency.** The EHCI utilizes a unique method to decrease the average number of memory accesses required to execute a USB transaction. Each schedule data structure is optimized to describe large client data buffers, thus minimizing the memory footprint and amount of memory overhead to traverse the schedule.
- **Minimized Hardware Complexity.** The EHCI provides a simple, asynchronous interface for software to provide the host controller with parameterized work items 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 any synchronization required. The interface supports a simple hardware scatter/gather method for all interface data structures.
- **Support for 32 and 64-bit Addressing.** Over the implementation lifetime of this specification, it is expected that EHCI controllers will be used increasingly in architectures that support more than 32-bits of addressable memory space. The EHCI includes optional interface extensions that support up to 64-bits of addressing.

This specification presents two chapters of pure structural definitions of the register space and schedule interface data structures. These definition chapters contain little or no operational requirements or usage models. The definition chapters are followed by a detailed description of the operational model requirements of the host controller, using the previously defined registers and schedule interface data structures. The following list summarizes the organization of this specification:

- Section 1.1 EHCI Product Compliance introduces the reader to the EHCI compliance program.
- Section 1.2 provides an overview of the architecture of the EHCI Host Controller.

- Chapter 2 Register Interface defines the register spaces of the EHCI Host Controller.
- Chapter 3 Data Structures defines the schedule data structures of the EHCI Host Controller.
- Chapter 4 Operational Model defines the details of the operational model for the EHCI Host Controller. It focuses on the operational requirements of the host controller hardware. It uses example behavioral abstractions to describe the operational requirements.
- Chapter 5 EHCI Extended Capabilities defines the EHCI-specific extended capability feature and also contains descriptions of each defined extended capability.
- Appendix A EHCI PCI Power Management Interface defines the details of the PCI Power Management Interface for the EHCI Host Controller.
- Appendix B EHCI 64-Bit Data Structures defines 64-bit versions of the data structures defined in Chapter 3.
- Appendix C Debug Port defines the interface to the optional debug port.
- Appendix D High Bandwidth Isochronous Rules enumerates the required behavior for the host controller execution of high-bandwidth isochronous transactions.

1.1 EHCI Product Compliance

Adopters and Contributors of the Enhanced Host Controller Interface Specification for USB have signed the Enhanced Host Controller Interface Specification for USB - Contributors 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 Enhanced Host Controller Interface Specification for USB. Adopters and Contributors can demonstrate compliance with the Specification through the testing program as defined by Intel.

1.2 Architectural Overview

A USB Host System is composed of a number of hardware and software layers. Figure 1-1 illustrates a conceptual block diagram of the building block layers in a host system that work in concert to support USB 2.0.

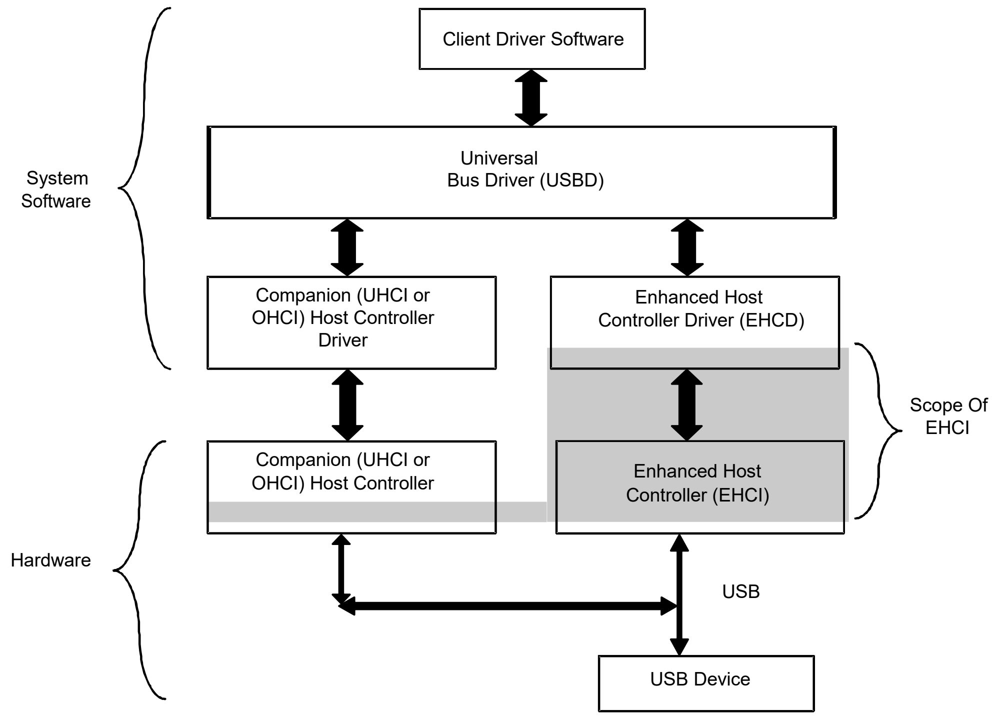

System Software
Client Driver Software
Universal Bus Driver (USBD)
Companion (UHCI or OHCI) Host Controller Driver
Enhanced Host Controller Driver (EHCD)
Scope Of EHCI
Companion (UHCI or OHCI) Host Controller
Enhanced Host Controller (EHCI)
Hardware
USB
USB Device

Figure 1-1. Universal Serial Bus, Revision 2.0 System Block Diagram

The component layers are:

- **Client Driver Software.** This software executes on the host PC corresponding to a particular USB device. Client 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 ofthe particular host controller driver for a particular operating system.
- **Host Controller Driver (xHCD).** xHCD provides the software layer between a specific Host Controller hardware and the USBD. The details of the host controller driver depend on the particular host controller hardware register interface definition.
- **Host Controller (xHC).** The host controller is the specific hardware implementation of the host controller. There is one host controller specification for the high-speed USB 2.0 functionality and two specifications for Full- and Low-speed host controllers. The *Universal Host Controller Interface* (UHCI) or *Open Host Controller Interface for USB* (OHCI) are the two industry standard USB 1.1 host controller interfaces.
- **USB Device.** This is a hardware device that performs a useful end-user function. Interactions with USB devices flow from the applications through the software and hardware layers to the USBdevices.

A USB 2.0 Host Controller includes one high-speed mode host controller and 0 or more USB 1.1 host controllers (see Figure 1-2). The high-speed host controller implements an EHCI interface. It is used for all high-speed communications to high-speed-mode devices connected to the root ports of the USB 2.0 host controller. This specification allows communications to Full- and Low-speed devices connected to the root ports of the USB 2.0 host controller to be provided by companion USB 1.1 host controllers. If an implementation does not include companion host controllers, the host controller must include a high-speed device permanently attached to each of the EHCI ports the implementation is planning to utilize. The EHCI controller cannot work with a Full- or Low-speed device.

This architecture allows the USB 2.0 host controller to provide USB functionality as long as there is at least USB 1.1 software support in the resident operating system. Full USB 2.0 functionality is delivered when both USB 1.1 and EHCI software is available in the operating system. The port transceiver routing logic is key to delivering this flexible operating environment. The state of the routing logic (see Figure 1-2) initially depends on whether software has configured the EHCI controller. Once the EHCD driver has configured the EHCI controller, it can specifically release the transceiver to the companion host controller port register if the attached device is not a high-speed device. When the operating system does not include support for the EHCI controller, the ports are default-routed to the companion host controllers and existing USB support for Full- and Low-speed devices remains.

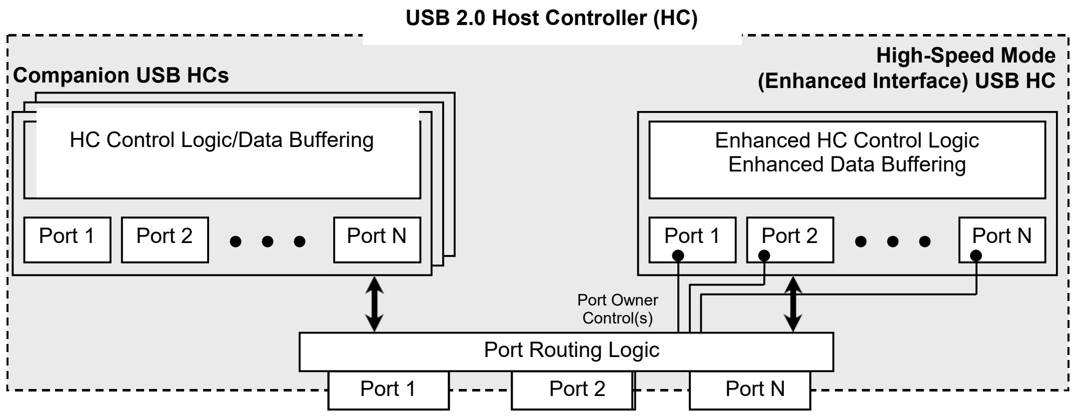

USB 2.0 Host Controller (HC)
Companion USB HCs
High-Speed Mode (Enhanced Interface) USB HC
HC Control Logic/Data Buffering
Enhanced HC Control Logic
Enhanced Data Buffering
Port 1
Port 2
Port N
Port Owner Control(s)
Port Routing Logic

Figure 1-2. USB 2.0 Host Controller

The Companion Host Controller (cHC) may be any USB 1.1 host controller (e.g. OHCI or UHCI). The companion host controllers always manage Full- and Low-speed USB devices connected to the root ports. The cHCs have no knowledge of the high-speed-mode host controller. They can possibly be integrated into a USB 2.0 host controller with no modification.

High-speed devices are always routed to and controlled by the EHCI host controller (eHC). When running and configured, the eHC is the default *owner* of all the root ports. The eHC and its driver initially detect all device attaches. It has additional control bits visible in each port register to manage the routing logic. For example: if the attached device is not a high-speed device, the eHC driver releases ownership of the port (and thus control of the device) to a companion host controller. For that port, enumeration starts over from the initial attach detect point and the device is enumerated under the cHC. Otherwise, the eHC retains ownership of the port and the device completes enumeration under the eHC.

This specification does not include descriptions for either the Universal Host Controller Interface (UHCI) or the Open Host Controller Interface for USB (OHCI for USB). This specification defines the register and schedule interfaces to an *Enhanced Host Controller Interface*.

1.2.1 Interface Architecture

The EHCI interface defines three interface spaces (see Figure 1-3):

- **PCI Configuration Space**. If the implementation includes PCI registers, they are used for system component enumeration and PCI power management.
- **Register Space.** Implementation-specific parameters and capabilities, plus operational control and status registers. This space, normally referred to as I/O space, must be implemented as memory-mapped I/O space.
- **Schedule Interface Space.** This is typically memory allocated and managed bythe eHC Driver for the periodic and asynchronous schedules.

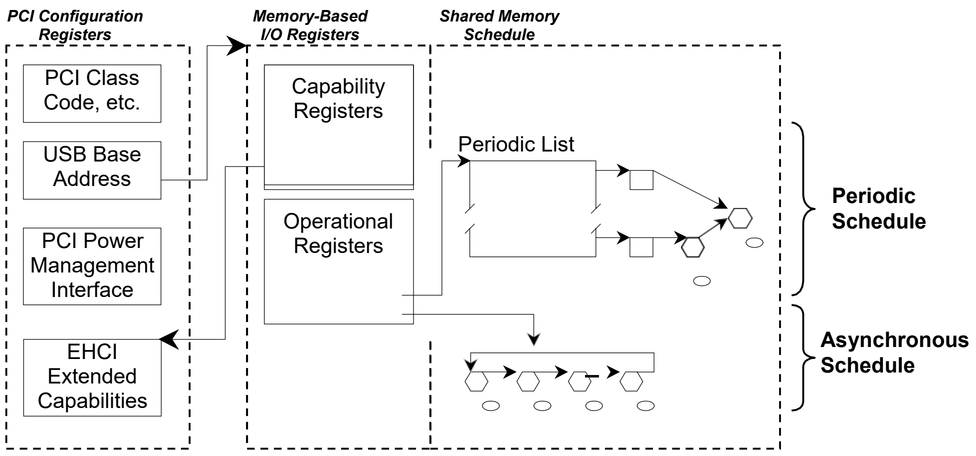


Figure 1-3. General Architecture of Enhanced Host Controller Interface

The EHCI provides support for two categories of transfer types: asynchronous and periodic. Periodic transfer types include both isochronous and interrupt. Asynchronous transfer types include control and bulk. Figure 1-3 illustrates that the EHCI schedule interface provides separate schedules for each category of transfer type.

The periodic schedule is based on a time-oriented frame list that represents a sliding *window* of time of host controller work items. All isochronous and interrupt transfers are serviced via the periodic schedule. The asynchronous schedule is a simple circular list of schedule work items that provides a round-robin service opportunity for all asynchronous transfers.

The EHCI host controller interface allows software to enable or disable each schedule. This allows system software to keep the USB alive with SOF traffic, but while both the schedules are disabled, the host controller will not access the schedule space. This keeps the host controller from accessing main memory in most implementations, and enables mobile systems to better manage CPU power (see Section 4.13).

1.2.2 EHCI Schedule Data Structures

The EHCI manages all interrupt, bulk and control transfer types using a simple buffer queuing data structure. The queuing data structure provides automatic, in-order streaming of data transfers. Software can asynchronously add data buffers to a queue and maintain streaming. USB-defined short packet semantics are fully supported on all processing boundary conditions without software intervention.

Split transactions for Full- and Low-speed non-isochronous endpoints are managed with the same data structures. The split transaction protocol is managed as a simple extension to the high-speed execution model.

High-speed and Full-speed isochronous transfers are managed using dedicated (and different) interface data structures. These data structures are optimized for the variability per data payload and time-oriented characteristics of the isochronous transfer type.

1.2.3 Root Hub Emulation

The host controller of a USB bus is required to implement the root hub. The operational register space contains port registers that contain the minimum hardware status and control needed to manage each port within the USB Specification. The host controller traverses the EHCI schedules and encounters work items that result in the host controller executing USB transactions. These transactions are broadcast through all enabled root ports to attached downstream USB devices.

The port registers provide system software with the control and status information required to manipulate the port in accordance with the USB Specification, Revision 2.0. 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.

2. Register Interface

The Enhanced USB Host Controller contains two sets of software accessible hardware registers—Memory-mapped Host Controller Registers and optional PCI configuration registers. Note that the PCI configuration registers are only needed for PCI devices that implement the Host Controller.

- **PCI Configuration Registers (For PCI devices).** In addition to the normal PCI header, power management, and device-specific registers, two registers are needed in the PCI Configuration space to support USB. The normal PCI header and device specific registers are beyond the scope of this document (The CLASSC register is shown in this document). Note that HCD does not interact with the PCI configuration space. This space is used only by the PCI enumerator to identify the USB Host Controller, and assign the appropriate system resources.
- **Memory-mapped USB Host Controller Registers.** This block of registers is memory-mapped into non-cacheable memory (see Figure 1-3). This memory space must begin on a DWord (32-bit) boundary. This register space is divided into two sections: a set of read-only capability registers and a set of read/write operational registers. Table 2-1, describes each register space.

  Note that host controllers are not required to support 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.

Table 2-1. Enhanced Interface Register Sets

Offset	Register Set	Explanation
0 to N-1	Capability Registers (Section 2.2)	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.
N to N+M-1	Operational Registers (Section 2.3)	The operational registers are used by system software to control and monitor the operational state of the host controller.

The reserved bits defined in this revision of the specification may be allocated in later revisions. Software should not assume reserved bits are always zero and should preserve these bits when writing to modifiable registers. The following notation is used to describe register access attributes:

RO **Read Only**. If a register is read only, writes have no effect.

WO **Write Only**. If a register is write only, reads return a zero for all bit positions.

R/W **Read/Write**. A register with this attribute can be read and written. Note that individual bits in some read/write registers may be read only.

R/WC Read/Write Clear. A register bit with this attribute can be read and written. However, a write of a 1 clears (sets to 0) the corresponding bit and a write of a 0 has no effect.

Registers in the auxiliary well are reset under different conditions than the registers in the core well. The auxiliary well, memory-space registers are initialized to their default values in the following cases:

- initial power-up of the auxiliary power well, or
- a value of 1b in *HCReset* (see Section 2.3.1)

The core well, memory-space registers are initialized to their default values in the following cases:

- assertion of the system (core-well) hardware reset, or
- a value of 1b in *HCReset*, or
- transition from the PCI Power Management D3hot state to the D0 state

PCI configuration-space registers implemented in the auxiliary power well are reset under different conditions than the registers in the core well. The auxiliary well, configuration-space registers are initialized to their default value in the following case:

- initial power-up of the auxiliary 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 Power Management D3hot state to the D0 state

Exceptions to these reset conditions will be defined in the associated register section.

2.1 PCI Configuration Registers (USB)

Table 2-2 lists the PCI configuration space register for an EHCI controller. The following subsections describe details about each set of registers defined.

Table 2-2. PCI Configuration Space Registers

Configuration Offset	Mnemonic	Register	Power Well	Register Access
00–08h	—	Register implementation as needed for specific PCI device	Core	—
09–0Bh	CLASSC	Class Code	Core	RO
0C–0Fh	—	Register implementation as needed for specific PCI device	Core	—
10–13h	USBBASE	Base Address to Memory-mapped Host Controller Register Space	Core	R/W
14–5Fh	—	Register implementation as needed for specific PCI device	Core	—
60h	SBRN	Serial Bus Release Number	Core	RO
61h	FLADJ	Frame Length Adjustment Register	Aux	R/W
62-63h	PORTWAKECAP	Port wake capabilities register (OPTIONAL)	Aux	R/W
64–FFh	—	Register implementation as needed for specific PCI device	Core	—
EECP+0h[1]	USBLEGSUP	USB Legacy Support EHCI Extended Capability Register	Aux	RO, R/W
EECP+4h	USBLEGCTLSTS	USB Legacy Support Control and Status Register	Aux	R/W, R/WC

[1] The EECP field is in the HCCPARAMS register, see Section 2.2.4.

2.1.1 PWRMGT — PCI Power Management Interface

EHCI Host Controller implementations are required to implement the PCI Power Management registers as defined in the *PCI Bus Power Management Interface Specification Revision 1.1*. Refer to Appendix A for the EHCI operational requirements for PCI Power Management.

2.1.2 CLASSC — CLASS CODE REGISTER

Address Offset: 09–0Bh
Default Value: 0C0320h
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.

Bit	Description
23:16	**Base Class Code (BASEC).** 0Ch= Serial Bus controller.
15:8	**Sub-Class Code (SCC)**. 03h=Universal Serial Bus Host Controller.
7:0	**Programming Interface (PI).** 20h=USB 2.0 Host Controller that conforms to this specification.

2.1.3 USBBASE — Register Space Base Address Register

Address Offset: 10–13h
Default Value: Implementation Dependent
Attribute: R/W
Size: 32 bits

This register contains the base address of the DWord-aligned memory-mapped host controller Registers. The number of writable bits in this register determines the actual size of the required memory space window. The minimum required is specified in this specification. Individual implementations may vary.

Bit	Description
31:8	**Base Address — R/W.** Corresponds to memory address signals [31:8].
7:3	**Reserved. — RO.** This bits are read only and hardwired to zero.
2:1	**Type. — RO.** This field has two valid values: **Value** **Meaning** 00b May only be mapped into 32-bit addressing space (Recommended). 10b May be mapped into 64-bit addressing space.
0	**Reserved — RO.** This bit is read only and hardwired to zero.

2.1.4 SBRN — Serial Bus Release Number Register

Address Offset: 60h
Default Value: See 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.

Bit	Description
7:0	**Serial Bus Specification Release Number.** All other combinations are reserved. **Bits[7:0]** **Release Number** 20h Release 2.0

2.1.5 Frame Length Adjustment Register (FLADJ)

Address Offset: 61h
Default Value: 20h
Attribute: R/W
Size: 8 bits

This register is in the auxiliary 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. Its initial programmed value is system dependent based on the accuracy of hardware USB clock and is initialized by system BIOS. This register should only be modified when the *HCHalted* bit in the USBSTS register is a one. Changing value of this register while the host controller is operating yields undefined results. It should not be reprogrammed by USB system software unless the default or BIOS programmed values are incorrect, or the system is restoring the register while returning from a suspended state.

Bit	Description
7:6	**Reserved.** These bits are reserved for future use and should read as zero.
5:0	**Frame Length Timing Value.** 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 micro-frame 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 (# High Speed 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)

2.1.6 Port Wake Capability Register (PORTWAKECAP)

Address Offset:	62h
Default Value:	Implementation Dependent
Attribute:	R/W
Size:	16 bits

This register is optional. When implemented this register is in the auxiliary power well. The intended use of this register is to establish a policy about which ports are to be used for wake events. Bit positions 1-15 in the mask correspond to a physical port implemented on the current EHCI controller. A one in a bit position indicates that a device connected below the port can be enabled as a wake-up device and the port may be enabled for disconnect/connect or over-current events as wake-up events. This is an information only mask register. The bits in this register DO NOT affect the actual operation of the EHCI host controller. The system-specific policy can be established by BIOS initializing this register to a system-specific value. System software uses the information in this register when enabling devices and ports for remote wake-up.

Bit	Description
15:0	**Port Wake Up Capability Mask.** Bit position zero of this register indicates whether the register is implemented. A one in bit position zero indicates that the register is implemented. Bit positions 1 through 15 correspond to a physical port implemented on this host controller. For example, bit position 1 corresponds to port 1, position 2 to port 2, etc.

2.1.7 USBLEGSUP — USB Legacy Support Extended Capability

Offset:	EECP + 00h
Default Value	Implementation Dependent
Attribute	RO, R/W
Size:	32 bits

This register is an EHCI extended capability register. It includes a specific function section and a pointer to the next EHCI extended capability. This register is used by pre-OS software (BIOS) and the operating system to coordinate ownership of the EHCI host controller. See Section 5.1 for details.

Table 2–3. USBLEGSUP — USB Legacy Support Extended Capability

Bit	Description
31:25	**Reserved.** These bits are reserved and must be set to zero.
24	**HC OS Owned Semaphore — R/W.** 0=Default. System software sets this bit to request ownership of the EHCI controller. Ownership is obtained when this bit reads as one and the *HC BIOS Owned Semaphore* bit reads as zero.
23:17	**Reserved.** These bits are reserved and must be set to zero.
16	**HC BIOS Owned Semaphore — R/W.** 0=Default. The BIOS sets this bit to establish ownership of the EHCI controller. System BIOS will set this bit to a zero in response to a request for ownership of the EHCI controller by system software.
15:8	**Next EHCI Extended Capability Pointer — RO.** This field points to the PCI configuration space offset of the next extended capability pointer. A value of 00h indicates the end of the extended capability list.
7:0	**Capability ID — RO.** This field identifies the extended capability. A value of 01h identifies the capability as Legacy Support. This extended capability requires one additional 32-bit register for control/status information, and this register is located at offset EECP+04h.

2.1.8 USBLEGCTLSTS — USB Legacy Support Control/Status

Offset:	EECP + 04h
Default Value	00000000h
Attribute	RO, R/W, R/WC
Size:	32 bits

Pre-OS (BIOS) software uses this register to enable SMIs for every EHCI/USB event it needs to track. Bits [21:16] of this register are simply shadow bit of USBSTS register [5:0].

Table 2–4. USBLEGCTLSTS — USB Legacy Support Control/Status

Bit	Description
31	**SMI on BAR** — **R/WC.** 0=Default. This bit is set to one whenever the **Base Address Register** (BAR) is written.
30	**SMI on PCI Command** — **R/WC.** 0=Default. This bit is set to one whenever the **PCI Command Register** is written.
29	**SMI on OS Ownership Change** — **R/WC.** 0=Default. This bit is set to one whenever the *HC OS Owned Semaphore* bit in the USBLEGSUP register transitions from 1 to a 0 or 0 to a 1
28:22	**Reserved**. These bits are reserved and should be zero.
21	**SMI on Async Advance** — **RO.** 0=Default. Shadow bit of the *Interrupt on Async Advance* bit in the USBSTS register see Section 2.3.2 for definition. To set this bit to a zero, system software must write a one to the *Interrupt on Async Advance* bit in the USBSTS register.
20	**SMI on Host System Error** — **RO.** 0=Default. Shadow bit of *Host System Error* bit in the USBSTS register, see Section 2.3.2 for definition and effects of the events associated with this bit being set to a one. To set this bit to a zero, system software must write a one to the *Host System Error* bit in the USBSTS register.
19	**SMI on Frame List Rollover** — **RO.** 0=Default. Shadow bit of *Frame List Rollover* bit in the USBSTS register see Section 2.3.2 for definition. To set this bit to a zero, system software must write a one to the *Frame List Rollover* bit in the USBSTS register.
18	**SMI on Port Change Detect** — **RO.** 0=Default. Shadow bit of *Port Change Detect* bit in the USBSTS register see Section 2.3.2 for definition. To set this bit to a zero, system software must write a one to the *Port Change Detect* bit in the USBSTS register.
17	**SMI on USB Error** — **RO.** 0=Default. Shadow bit of *USB Error Interrupt* (USBERRINT) bit in the USBSTS register see Section 2.3.2 for definition. To set this bit to a zero, system software must write a one to the *USB Error Interrupt* bit in the USBSTS register.
16	**SMI on USB Complete** — **RO.** 0=Default. Shadow bit of *USB Interrupt* (USBINT) bit in the USBSTS register see Section 2.3.2 for definition. To set this bit to a zero, system software must write a one to the *USB Interrupt* bit in the USBSTS register.
15	**SMI on BAR Enable** — **R/W.** 0=Default. When this bit is one and **SMI on BAR** is one, then the host controller will issue an SMI.
14	**SMI on PCI Command Enable** — **R/W.** 0=Default. When this bit is one and **SMI on PCI Command** is one, then the host controller will issue an SMI.
13	**SMI on OS Ownership Enable** — **R/W.** 0=Default. When this bit is a one AND the **OS Ownership Change** bit is one, the host controller will issue an SMI.
12:6	**Reserved**. These bits are reserved and should be zero.

Table 2–4. USBLEGCTLSTS — USB Legacy Support Control/Status (cont.)

Bit	Description
5	**SMI on Async Advance Enable — R/W.** 0=Default. When this bit is a one, and the *SMI on Async Advance* bit (above) in this register is a one, the host controller will issue an SMI immediately. [E]
4	**SMI on Host System Error Enable — R/W.** 0=Default. When this bit is a one, and the *SMI on Host System Error* bit (above) in this register is a one, the host controller will issue an SMI immediately. [E]
3	**SMI on Frame List Rollover Enable — R/W.** 0=Default. When this bit is a one, and the *SMI on Frame List Rollover* bit (above) in this register is a one, the host controller will issue an SMI immediately. [E]
2	**SMI on Port Change Enable — R/W.** 0=Default. When this bit is a one, and the *SMI on Port Change Detect* bit (above) in this register is a one, the host controller will issue an SMI immediately. [E]
1	**SMI on USB Error Enable — R/W.** 0=Default. When this bit is a one, and the *SMI on USB Error* bit (above) in this register is a one, the host controller will issue an SMI immediately. [E]
0	**USB SMI Enable — R/W.** 0=Default. When this bit is a one, and the *SMI on USB Complete* bit (above) in this register is a one, the host controller will issue an SMI immediately. [E]

Notes:

A. For all enable register bits, 1= Enabled, 0= Disabled
B. SMI – System Management Interrupt
C. BAR – Base Address Register
D. MSE – Memory Space Enable
E. SMI's are independent of the interrupt threshold value

2.2 Host Controller Capability Registers

These registers specify the limits, restrictions and capabilities of the host controller implementation.

Table 2-5. Enhanced Host Controller Capability Registers

Offset	Size	Mnemonic	Power Well	Register Name
00h	1	CAPLENGTH	Core	Capability Register Length
01h	1	Reserved	Core	N/A
02h	2	HCIVERSION	Core	Interface Version Number
04h	4	HCSPARAMS	Core	Structural Parameters
08h	4	HCCPARAMS	Core	Capability Parameters
0Ch	8	HCSP-PORTROUTE	Core	Companion Port Route Description

2.2.1 CAPLENGTH — Capability Registers Length

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.

2.2.2 HCIVERSION — Host Controller Interface Version Number

Address:	Base+ (02h)
Default Value:	0100h
Attribute	RO
Size:	16 bits

This is a two-byte register containing a BCD encoding of the EHCI revision number supported by this host controller. The most significant byte of this register represents a major revision and the least significant byte is the minor revision.

2.2.3 HCSPARAMS — Structural Parameters

Address:	Base+ (04h)
Default Value	Implementation Dependent
Attribute	RO
Size:	32 bits

This is a set of fields that are structural parameters: Number of downstream ports, etc.

Table 2-6.HCSPARAMS — Host Controller Structural Parameters

Bit	Description
31:24	**Reserved.** These bits are reserved and should be set to zero.
23:20	**Debug Port Number.** *Optional.* This register identifies which of the host controller ports is the debug port. The value is the port number (one-based) of the debug port. A non-zero value in this field indicates the presence of a debug port. The value in this register must not be greater than *N_PORTS* (see below).
19:17	**Reserved.** These bits are reserved and should be set to zero.
16	**Port Indicators (P_INDICATOR).** This bit indicates whether the ports support port indicator control. When this bit is a one, the port status and control registers include a read/writeable field for controlling the state of the port indicator. See Section 2.3.9 for definition of the port indicator control field.
15:12	**Number of Companion Controller (N_CC).** This field indicates the number of companion controllers associated with this USB 2.0 host controller. A zero in this field indicates there are no companion host controllers. Port-ownership hand-off is not supported. Only high-speed devices are supported on the host controller root ports. A value larger than zero in this field indicates there are companion USB 1.1 host controller(s). Port-ownership hand-offs are supported. High, Full- and Low-speed devices are supported on the host controller root ports.
11:8	**Number of Ports per Companion Controller (N_PCC).** This field indicates the number of ports supported per companion host controller. It is used to indicate the port routing configuration to system software. For example, if *N_PORTS* has a value of 6 and *N_CC* has a value of 2 then *N_PCC* could have a value of 3. The convention is that the first *N_PCC* ports are assumed to be routed to companion controller 1, the next *N_PCC* ports to companion controller 2, etc. In the previous example, the *N_PCC* could have been 4, where the first 4 are routed to companion controller 1 and the last two are routed to companion controller 2. The number in this field must be consistent with *N_PORTS* and *N_CC*.

Table 2-6.HCSPARAMS — Host Controller Structural Parameters (cont.)

Bit	Description
7	**Port Routing Rules.** This field indicates the method used by this implementation for how all ports are mapped to companion controllers. The value of this field has the following interpretation: **Value** **Meaning** 0 The first *N_PCC* ports are routed to the lowest numbered function companion host controller, the next *N_PCC* port are routed to the next lowest function companion controller, and so on. 1 The port routing is explicitly enumerated by the first *N_PORTS* elements of the *HCSP*-PORTROUTE array.
6:5	**Reserved.** These bits are reserved and should be set to zero.
4	**Port Power Control (PPC).** This field indicates whether the host controller implementation includes port power control. A one in this bit indicates the ports have port power switches. A zero in this bit indicates the port do not have port power switches. The value of this field affects the functionality of the *Port Power* field in each port status and control register (see Section 2.3.8).
3:0	**N_PORTS.** This field specifies the number of physical downstream ports implemented on this host controller. The value of this field determines how many port registers are addressable in the Operational Register Space (see Table 2-8). Valid values are in the range of 1H to FH. A zero in this field is undefined.

2.2.4 HCCPARAMS — Capability Parameters

Address:	Base+ (08h)
Default Value	Implementation Dependent
Attribute	RO
Size:	32 bits

Multiple Mode control (time-base bit functionality), addressing capability

Table 2-7. HCCPARAMS—Host Controller Capability Parameters

Bit	Description
31:16	**Reserved.** These bits are reserved and should be set to zero.
15:8	**EHCI Extended Capabilities Pointer (EECP).** Default = Implementation Dependent. This optional field indicates the existence of a capabilities list. A value of 00h indicates no extended capabilities are implemented. A non-zero value in this register indicates the offset in PCI configuration space of the first EHCI extended capability. The pointer value must be 40h or greater if implemented to maintain the consistency of the PCI header defined for this class of device.
7:4	**Isochronous Scheduling Threshold.** Default = implementation dependent. This field indicates, relative to the current position of the executing host controller, where software can reliably update the isochronous schedule. When bit [7] is zero, the value of the least significant 3 bits indicates the number of micro-frames a host controller can hold a set of isochronous data structures (one or more) before flushing the state. When bit [7] is a one, then host software assumes the host controller may cache an isochronous data structure for an entire frame. Refer to Section 4.7.2.1 for details on how software uses this information for scheduling isochronous transfers.
3	**Reserved.** This bit is reserved and should be set to zero.

Table 2-7. HCCPARAMS—Host Controller Capability Parameters (cont.)

Bit	Description
2	**Asynchronous Schedule Park Capability.** Default = Implementation dependent. If this bit is set to a one, then the host controller supports the park feature for high-speed queue heads in the Asynchronous Schedule. The feature can be disabled or enabled and set to a specific level by using the *Asynchronous Schedule Park Mode Enable* and *Asynchronous Schedule Park Mode Count* fields in the USBCMD register.
1	**Programmable Frame List Flag.** Default = Implementation dependent. If this bit is set to a zero, then system software must use a frame list length of 1024 elements with this host controller. The USBCMD register *Frame List Size* field is a read-only register and should be set to zero. If set to a one, then system software can specify and use a smaller frame list and configure the host controller via the USBCMD register *Frame List Size* field. The frame list must always be aligned on a 4K page boundary. This requirement ensures that the frame list is always physically contiguous.
0	**64-bit Addressing Capability[1].** This field documents the addressing range capability of this implementation. The value of this field determines whether software should use the data structures defined in Section 3 (32-bit) or those defined in Appendix B (64-bit). Values for this field have the following interpretation: 0b data structures using 32-bit address memory pointers 1b data structures using 64-bit address memory pointers

[1] This is not tightly coupled with the USBBASE address register mapping control. The *64-bit Addressing Capability* bit 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.

2.2.5 HCSP-PORTROUTE — Companion Port Route Description

Address:	Base+ (0Ch)
Default Value	Implementation Dependent
Attribute	RO
Size:	60 bits

This optional field is valid only if *Port Routing Rules* field in the HCSPARAMS register is set to a one.

The rules for organizing companion host controllers and an EHCI host controllers within PCI space are described in detail in Section 4.2. This field is used to allow a host controller implementation to explicitly described to which companion host controller each implemented port is mapped.

This field is a 15-element nibble array (each 4 bits is one array element). Each array location corresponds one-to-one with a physical port provided by the host controller (e.g. PORTROUTE[0] corresponds to the first PORTSC port, PORTROUTE[1] to the second PORTSC port, etc.). The value of each element indicates to which of the companion host controllers this port is routed. Only the first *N_PORTS* elements have valid information. A value of zero indicates that the port is routed to the lowest numbered function companion host controller. A value of one indicates that the port is routed to the next lowest numbered function companion host controller, and so on.

2.3 Host Controller Operational Registers

This section defines the enhanced host controller operational registers. These registers are located after the capabilities registers (see Section 2.1.7). The operational register base must be DWord aligned and is calculated by adding the value in the first capabilities register (CAPLENGTH, Section 2.2.1) to the base address of the enhanced host controller register address space. All registers are 32 bits in length. Software should read and write these registers using only DWord accesses.

These registers are divided into two sets. The first set at addresses 00h to 3Fh are implemented in the core power well. The second set at addresses 40h to the end of the implemented register space are implemented in the auxiliary power well.

Table 2-8. Host Controller Operational Registers

Offset	Mnemonic	Register Name	Power Well	Section
00h	USBCMD	USB Command	Core	2.3.1
04h	USBSTS	USB Status	Core	2.3.2
08h	USBINTR	USB Interrupt Enable	Core	2.3.3
0Ch	FRINDEX	USB Frame Index	Core	2.3.4
10h	CTRLDSSEGMENT	4G Segment Selector	Core	2.3.5
14h	PERIODICLISTBASE	Frame List Base Address	Core	2.3.6
18h	ASYNCLISTADDR	Next Asynchronous List Address	Core	2.3.7
1C-3Fh	Reserved		Core	
40h	CONFIGFLAG	Configured Flag Register	Aux	2.3.8
44h	PORTSC(1-N_PORTS)	Port Status/Control	Aux	2.3.9

2.3.1 USBCMD — USB Command Register

Address:	Operational Base+ (00h)
Default Value:	00080000h (00080B00h if *Asynchronous Schedule Park Capability is a one*)
Attribute	RO, R/W (field dependent), WO
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.

Table 2-9. USBCMD – USB Command Register Bit Definitions

Bit	Description
31:24	**Reserved.** These bits are reserved and should be set to zero.
23:16	**Interrupt Threshold Control** — **R/W.** Default 08h. This field is used by system software to select the maximum rate at which the host controller will issue interrupts. The only valid values are defined below. If software writes an invalid value to this register, the results are undefined. **Value** **Maximum Interrupt Interval** 00h Reserved 01h 1 micro-frame 02h 2 micro-frames 04h 4 micro-frames 08h 8 micro-frames (default, equates to 1 ms) 10h 16 micro-frames (2 ms) 20h 32 micro-frames (4 ms) 40h 64 micro-frames (8 ms) Refer to Section 4.15 for interrupts affected by this register. Any other value in this register yields undefined results. Software modifications to this bit while HCHalted bit is equal to zero results in undefined behavior.
15:12	**Reserved.** These bits are reserved and should be set to zero.
11	**Asynchronous Schedule Park Mode Enable (OPTIONAL)** — **RO or R/W.** If the *Asynchronous Park Capability* bit in the HCCPARAMS register is a one, then this bit defaults to a 1h and is R/W. Otherwise the bit must be a zero and is RO. Software uses this bit to enable or disable Park mode. When this bit is one, Park mode is enabled. When this bit is a zero, Park mode is disabled.
10	**Reserved.** This bit is reserved and should be set to zero.
9:8	**Asynchronous Schedule Park Mode Count (OPTIONAL)** — **RO or R/W.** If the *Asynchronous Park Capability* bit in the HCCPARAMS register is a one, then this field defaults to 3h and is R/W. Otherwise it defaults to zero and is RO. It contains a count of the number of successive transactions the host controller is allowed to execute from a high-speed queue head on the Asynchronous schedule before continuing traversal of the Asynchronous schedule. See Section 4.10.3.2 for full operational details. Valid values are 1h to 3h. Software must not write a zero to this bit when *Park Mode Enable* is a one as this will result in undefined behavior.

Table 2-9. USBCMD – USB Command Register Bit Definitions (cont.)

Bit	Description
7	**Light Host Controller Reset (OPTIONAL)** — **R/W.** This control bit is not required. If implemented, it allows the driver to reset the EHCI controller without affecting the state of the ports or the relationship to the companion host controllers. For example, the PORSTC registers should not be reset to their default values and the CF bit setting should not go to zero (retaining port ownership relationships). A host software read of this bit as zero indicates the Light Host Controller Reset has completed and it is safe for host software to re-initialize the host controller. A host software read of this bit as a one indicates the Light Host Controller Reset has not yet completed. If not implemented a read of this field will always return a zero.
6	**Interrupt on Async Advance Doorbell** — **R/W.** This bit is used as a doorbell by software to tell the host controller to issue an interrupt the next time it advances asynchronous schedule. Software must write a 1 to this bit to *ring* the doorbell. When the host controller has evicted all appropriate cached schedule state, it sets the *Interrupt on Async Advance* status bit in the USBSTS register. If the *Interrupt on Async Advance Enable* bit in the USBINTR register is a one then the host controller will assert an interrupt at the next interrupt threshold. See Section 4.8.2 for operational details. The host controller sets this bit to a zero after it has set the *Interrupt on Async Advance* status bit in the USBSTS register to a one. Software should not write a one to this bit when the asynchronous schedule is disabled. Doing so will yield undefined results.
5	**Asynchronous Schedule Enable** — **R/W.** Default 0b. This bit controls whether the host controller skips processing the Asynchronous Schedule. Values mean: 0b Do not process the Asynchronous Schedule 1b Use the ASYNCLISTADDR register to access the Asynchronous Schedule.
4	**Periodic Schedule Enable** — **R/W.** Default 0b. This bit controls whether the host controller skips processing the Periodic Schedule. Values mean: 0b Do not process the Periodic Schedule 1b Use the PERIODICLISTBASE register to access the Periodic Schedule.
3:2	**Frame List Size** — **(R/W or RO).** Default 00b. This field is R/W only if *Programmable Frame List Flag* in the HCCPARAMS registers is set to a one. This field specifies the size of the frame list. The size the frame list controls which bits in the Frame Index Register should be used for the Frame List Current index. Values mean: 00b 1024 elements (4096 bytes) Default value 01b 512 elements (2048 bytes) 10b 256 elements (1024 bytes) – for resource-constrained environments 11b **Reserved**

Table 2-9. USBCMD – USB Command Register Bit Definitions (cont.)

Bit	Description
1	**Host Controller Reset (HCRESET)** — **R/W.** This control bit is used by software to reset the host controller. The effects of this on Root Hub registers are similar to a Chip Hardware Reset. When software writes a one 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 USB is immediately terminated. A USB reset is not driven on 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. Port ownership reverts to the companion host controller(s), with the side effects described in Section 4.2. Software must reinitialize the host controller as described in Section 4.1 in order to return the host controller to an operational state. This bit is set to zero by the Host Controller when the reset process is complete. Software cannot terminate the reset process early by writing a zero to this register. Software should not set this bit to a one when the *HCHalted* bit in the USBSTS register is a zero. Attempting to reset an actively running host controller will result in undefined behavior.
0	**Run/Stop (RS)** —**R/W.** Default 0b. 1=Run. 0=Stop. When set to a 1, the Host Controller proceeds with execution of the schedule. The Host Controller continues execution as long as this bit is set to a 1. When this bit is set to 0, the Host Controller completes the current and any actively pipelined transactions on the USB and then halts. The Host Controller must halt within 16 micro-frames after software clears the Run bit. The HC Halted bit in the status register indicates when the Host Controller has finished its pending pipelined transactions and has entered the stopped state. Software must not write a one to this field unless the host controller is in the Halted state (i.e. *HCHalted* in the USBSTS register is a one). Doing so will yield undefined results.

2.3.2 USBSTS — USB Status Register

Address:	Operational Base + (04h)
Default Value:	00001000h
Attribute	RO, R/W, R/WC, (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. See Section 4.15 for additional information concerning USB interrupt conditions.

Table 2-10. USBSTS USB Status Register Bit Definitions

Bit	Description
31:16	**Reserved**. These bits are reserved and should be set to zero.
15	**Asynchronous Schedule Status** — **RO.** 0=Default. The bit reports the current real status of the Asynchronous Schedule. If this bit is a zero then the status of the Asynchronous Schedule is disabled. If this bit is a one then the status of the Asynchronous Schedule is enabled. The Host Controller is not required to *immediately* disable or enable the Asynchronous Schedule when software transitions the *Asynchronous Schedule Enable* bit in the USBCMD register. When this bit and the *Asynchronous Schedule Enable* bit are the same value, the Asynchronous Schedule is either enabled (1) or disabled (0).
14	**Periodic Schedule Status** — **RO.** 0=Default. The bit reports the current real status of the Periodic Schedule. If this bit is a zero then the status of the Periodic Schedule is disabled. If this bit is a one then the status of the Periodic Schedule is enabled. The Host Controller is not required to *immediately* disable or enable the Periodic Schedule when software transitions the *Periodic Schedule Enable* bit in the USBCMD register. When this bit and the *Periodic Schedule Enable* bit are the same value, the Periodic Schedule is either enabled (1) or disabled (0).
13	**Reclamation** — **RO.** 0=Default. This is a read-only status bit, which is used to detect an empty asynchronous schedule. The operational model of empty schedule detection is described in Section 4.8.3. The valid transitions for this bit are described in Section 4.8.6.
12	**HCHalted** — **RO.** 1=Default. This bit is a zero whenever the Run/Stop bit is a one. The Host Controller sets this bit to one after it has stopped executing as a result of the Run/Stop bit being set to 0, either by software or by the Host Controller hardware (e.g. internal error).
11:6	**Reserved**. These bits are reserved and should be set to zero.
5	**Interrupt on Async Advance** — **R/WC.** 0=Default. System software can force the host controller to issue an interrupt the next time the host controller advances the asynchronous schedule by writing a one to the *Interrupt on Async Advance Doorbell* bit in the USBCMD register. This status bit indicates the assertion of that interrupt source.
4	**Host System Error** — **R/WC.** The Host Controller sets this bit to 1 when a serious error occurs during a host system access involving the Host Controller 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 Host Controller clears the Run/Stop bit in the Command register to prevent further execution of the scheduled TDs.
3	**Frame List Rollover** — **R/WC.** The Host Controller sets this bit to a one when the *Frame List Index* (see Section 2.3.4) rolls over from its maximum value to zero. The exact value at which the rollover occurs depends on the frame list size. For example, if the frame list size (as programmed in the *Frame List Size* field of the USBCMD register) is 1024, the *Frame Index Register* rolls over every time FRINDEX[13] toggles. Similarly, if the size is 512, the Host Controller sets this bit to a one every time FRINDEX[12] toggles.

Table 2-10. USBSTS USB Status Register Bit Definitions (cont.)

Bit	Description
2	**Port Change Detect** — **R/WC.** The Host Controller sets this bit to a one when any port for which the *Port Owner* bit is set to zero (see Section 2.3.9) has a change bit transition from a zero to a one or a *Force Port Resume* bit transition from a zero to a one as a result of a J-K transition detected on a suspended port. This bit will also be set as a result of the *Connect Status Change* being set to a one after system software has relinquished ownership of a connected port by writing a one to a port's *Port Owner* bit (see Section 4.2.2). This bit is allowed to be maintained in the Auxiliary power well. Alternatively, it is also acceptable that on a D3 to D0 transition of the EHCI HC device, this bit is loaded with the OR of all of the PORTSC change bits (including: Force port resume, over-current change, enable/disable change and connect status change).
1	**USB Error Interrupt (USBERRINT)** — **R/WC.** The Host Controller sets this bit to 1 when completion of a USB transaction results in an error condition (e.g., error counter underflow). If the TD on which the error interrupt occurred also had its IOC bit set, both this bit and USBINT bit are set. See Section 4.15.1 for a list of the USB errors that will result in this bit being set to a one.
0	**USB Interrupt (USBINT)** — **R/WC.** The Host Controller sets this bit to 1 on the completion of a USB transaction, which results in the retirement of a Transfer Descriptor that had its IOC bit set. The Host Controller also sets this bit to 1 when a short packet is detected (actual number of bytes received was less than the expected number of bytes).

2.3.3 USBINTR — USB Interrupt Enable Register

Address:	Operational Base + (08h)
Default Value:	00000000h
Attributes	R/W
Size	32 bits

This register enables and disables reporting of the corresponding interrupt to the software. When a bit is set and the corresponding interrupt is active, an interrupt is generated to the host. Interrupt sources that are disabled in this register still appear in the USBSTS to allow the software to poll for events.

Each interrupt enable bit description indicates whether it is dependent on the interrupt threshold mechanism (see Section 4.15).

Table 2-11. USBINTR - USB Interrupt Enable Register

Bit	Interrupt Source	Description
31:6	**Reserved**.	These bits are reserved and should be zero.
5	**Interrupt on Async Advance Enable**	When this bit is a one, and the *Interrupt on Async Advance* bit in the USBSTS register is a one, the host controller will issue an interrupt at the next interrupt threshold. The interrupt is acknowledged by software clearing the *Interrupt on Async Advance* bit.
4	**Host System Error Enable**	When this bit is a one, and the *Host System Error Status* bit in the USBSTS register is a one, the host controller will issue an interrupt. The interrupt is acknowledged by software clearing the *Host System Error* bit.

Table 2-11. USBINTR - USB Interrupt Enable Register (cont.)

Bit	Description	
3	**Frame List Rollover Enable.**	When this bit is a one, and the *Frame List Rollover* bit in the USBSTS register is a one, the host controller will issue an interrupt. The interrupt is acknowledged by software clearing the *Frame List Rollover* bit.
2	**Port Change Interrupt Enable.**	When this bit is a one, and the Port Change Detect bit in the USBSTS register is a one, the host controller will issue an interrupt. The interrupt is acknowledged by software clearing the *Port Change Detect* bit.
1	**USB Error Interrupt Enable.**	When this bit is a one, and the USBERRINT bit in the USBSTS register is a one, the host controller will issue an interrupt at the next interrupt threshold. The interrupt is acknowledged by software clearing the *USBERRINT* bit.
0	**USB Interrupt Enable.**	When this bit is a one, and the USBINT bit in the USBSTS register is a one, the host controller will issue an interrupt at the next interrupt threshold. The interrupt is acknowledged by software clearing the *USBINT* bit.

Note: for all enable register bits, 1= Enabled, 0= Disabled

2.3.4 FRINDEX — Frame Index Register

Address:	Operational Base + (0Ch)
Default Value:	00000000h
Attribute:	R/W (Writes must be DWord Writes)
Size	32 bits

This register is used by the host controller to index into the periodic frame list. The register updates every 125 microseconds (once each micro-frame). Bits [N:3] are used to select a particular entry in the Periodic Frame List during periodic schedule execution. The number of bits used for the index depends on the size of the frame list as set by system software in the *Frame List Size* field in the USBCMD register (see Table 2-9).

This register must be written as a DWord. Byte writes produce undefined results. This register cannot be written unless the Host Controller is in the Halted state as indicated by the *HCHalted* bit (USBSTS register Section 2.3.2). A write to this register while the Run/Stop bit is set to a one (USBCMD register, Section 2.3.1) produces undefined results. Writes to this register also affect the SOF value. See Section 4.5 for details.

Table 2-12. FRINDEX — Frame Index Register

Bit	Description
31:14	**Reserved.**
13:0	**Frame Index.** The value in this register increments at the end of each time frame (e.g. micro-frame). Bits [N:3] are used for the Frame List current index. This means that each location of the frame list is accessed 8 times (frames or micro-frames) before moving to the next index. The following illustrates values of *N* based on the value of the *Frame List Size* field in the USBCMD register.

USBCMD[Frame List Size]	Number Elements	N
00b	(1024)	12
01b	(512)	11
10b	(256)	10
11b	Reserved	

The SOF frame number value for the bus SOF token is derived or alternatively managed from this register. Please refer to Section 4.5 for a detailed explanation of the SOF value management requirements on the host controller. The value of FRINDEX must be 125 µsec (1 micro-frame) ahead of the SOF token value. The SOF value may be implemented as an 11-bit shadow register. For this discussion, this shadow register is 11 bits and is named SOFV. SOFV updates every 8 micro-frames. (1 millisecond). An example implementation to achieve this behavior is to increment SOFV each time the FRINDEX[2:0] increments from a zero to a one.

Software must use the value of FRINDEX to derive the current micro-frame number, both for high-speed isochronous scheduling purposes and to provide the *get micro-frame number* function required for client drivers. Therefore, the value of FRINDEX and the value of SOFV must be kept consistent if chip is reset or software writes to FRINDEX. Writes to FRINDEX must also *write-through* FRINDEX[13:3] to SOFV[10:0]. In order to keep the update as simple as possible, software should never write a FRINDEX value where the three least significant bits are 111b or 000b. Please refer to Section 4.5.

2.3.5 CTRLDSSEGMENT — Control Data Structure Segment Register

Address:	Operational Base + (10h)
Default Value:	00000000h
Attribute:	R/W (Writes must be DWord Writes)
Size:	32 bits

This 32-bit register corresponds to the most significant address bits [63:32] for all EHCI data structures. If the *64-bit Addressing Capability* field in HCCPARAMS is a zero, then this register is not used. Software cannot write to it and a read from this register will return zeros.

If the *64-bit Addressing Capability* field in HCCPARAMS is a one, then this register is used with the link pointers to construct 64-bit addresses to EHCI control data structures. This register is concatenated with the link pointer from either the *PERIODICLISTBASE*, *ASYNCLISTADDR*, or any control data structure link field to construct a 64-bit address.

This register allows the host software to locate all control data structures within the same 4 Gigabyte memory segment.

2.3.6 PERIODICLISTBASE — Periodic Frame List Base Address Register

Address:	Operational Base + (14h)
Default Value:	Undefined
Attribute:	R/W (Writes must be DWord Writes)
Size:	32 bits

This 32-bit register contains the beginning address of the Periodic Frame List in the system memory. If the host controller is in 64-bit mode (as indicated by a one in the *64-bit Addressing Capability* field in the

HCCSPARAMS register), then the most significant 32 bits of every control data structure address comes from the *CTRLDSSEGMENT* register (see Section 2.3.5). System software loads this register prior to starting the schedule execution by the Host Controller (see 4.1). The memory structure referenced by this physical memory pointer is assumed to be 4-Kbyte aligned. The contents of this register are combined with the Frame Index Register (FRINDEX) to enable the Host Controller to step through the Periodic Frame List in sequence.

Table 2-13. PERIODICLISTBASE — Periodic Frame List Base Address Register

Bit	Description
31:12	**Base Address (Low).** These bits correspond to memory address signals [31:12], respectively.
11:0	**Reserved.** Must be written as 0s. During runtime, the values of these bits are undefined.

2.3.7 ASYNCLISTADDR — Current Asynchronous List Address Register

Address:	Operational Base + (18h)
Default Value:	Undefined
Attribute:	Read/Write (Writes must be DWord Writes)
Size:	32 bits

This 32-bit register contains the address of the next asynchronous queue head to be executed. If the host controller is in 64-bit mode (as indicated by a one in *64-bit Addressing Capability* field in the HCCPARAMS register), then the most significant 32 bits of every control data structure address comes from the *CTRLDSSEGMENT* register (See Section 2.3.5). Bits [4:0] of this register cannot be modified by system software and will always return a zero when read. The memory structure referenced by this physical memory pointer is assumed to be 32-byte (cache line) aligned.

Table 2-14. ASYNCLISTADDR — Current Asynchronous List Address Register

Bit	Description
31:5	**Link Pointer Low (LPL).** These bits correspond to memory address signals [31:5], respectively. This field may only reference a Queue Head (QH), see Section 3.6.
4:0	**Reserved.** These bits are reserved and their value has no effect on operation.

2.3.8 CONFIGFLAG — Configure Flag Register

Address:	Operational Base+ (40h)
Default Value:	00000000h
Attribute	R/W
Size	32 bits

This register is in the auxiliary power well. It is only reset by hardware when the auxiliary power is initially applied or in response to a host controller reset.

Table 2-15. CONFIGFLAG — Configure Flag Register Bit Definitions

Bit	Description
31:1	**Reserved.** These bits are reserved and should be set to zero.
0	**Configure Flag (CF) —R/W.** Default 0b. Host software sets this bit as the last action in its process of configuring the Host Controller (see Section 4.1). This bit controls the default port-routing control logic. Bit values and side-effects are listed below. See Section 4.2 For operational details. 0b Port routing control logic default-routes each port to an implementation dependent classic host controller. 1b Port routing control logic default-routes all ports to this host controller.

2.3.9 PORTSC — Port Status and Control Register

Address:	Operational Base + (44h + (4*Port Number-1)) where: *Port Number* is 1, 2, 3, … N_PORTS
Default:	00002000h (w/*PPC* set to one); 00003000h (w/*PPC* set to a zero)
Attribute:	RO, R/W, R/WC (field dependent)
Size	32 bits

A host controller must implement one or more port registers. The number of port registers implemented by a particular instantiation of a host controller is documented in the HCSPARAMs register (Section 2.2.3). 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 auxiliary power well. It is only reset by hardware when the auxiliary power is initially applied or in response to a host controller reset. The initial conditions of a port are:

- No device connected,
- Port disabled

If the port has port power control, software cannot change the state of the port until after it applies power to the port by setting port power to a 1. Software must not attempt to change the state of the port until after power is stable on the port. The host is required to have power stable to the port within 20 milliseconds of the zero to one transition.

Note1: When a device is attached, the port state transitions to the connected state and system software will process this as with any status change notification. Refer to Section 4.3 for operational requirements for how change events interact with port suspend mode.

Note2: If a port is being used as the Debug Port, then the port may report device connected and enabled when the Configured Flag is a zero.

Table 2-16. PORTSC — Port Status and Control

Bit	Description
31:23	**Reserved.** These bits are reserved for future use and should return a value of zero when read.
22	**Wake on Over-current Enable (WKOC_E) — R/W.** Default = 0b. Writing this bit to a one enables the port to be sensitive to over-current conditions as wake-up events. See Section 4.3 for effects of this bit on resume event behavior. Refer to Section 4.3.1 for operational model. This field is zero if *Port Power* is zero.
21	**Wake on Disconnect Enable (WKDSCNNT_E) — R/W.** Default = 0b. Writing this bit to a one enables the port to be sensitive to device disconnects as wake-up events. See Section 4.3 for effects of this bit on resume event behavior. Refer to Section 4.3.1 for operational model. This field is zero if *Port Power* is zero.
20	**Wake on Connect Enable (WKCNNT_E) — R/W.** Default = 0b. Writing this bit to a one enables the port to be sensitive to device connects as wake-up events. See Section 4.3 for effects of this bit on resume event behavior. Refer to Section 4.3.1 for operational model. This field is zero if *Port Power* is zero.

Table 2-16. PORTSC — Port Status and Control (cont.)

Bit	Description
19:16	**Port Test Control—R/W.** Default = 0000b. When this field is zero, 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. The encoding of the test mode bits are (0110b - 1111b are reserved): **Bits** — **Test Mode** 0000b — Test mode not enabled 0001b — Test J_STATE 0010b — Test K_STATE 0011b — Test SE0_NAK 0100b — Test Packet 0101b — Test FORCE_ENABLE Refer to Section 4.14 for the operational model for using these test modes and the USB Specification Revision 2.0, Chapter 7 for details on each test mode.
15:14	**Port Indicator Control.** Default = 00b. Writing to these bits has no effect if the *P_INDICATOR* bit in the HCSPARAMS register is a zero. If *P_INDICATOR* bit is a one, then the bit encodings are: **Bit Value** — **Meaning** 00b — Port indicators are off 01b — Amber 10b — Green 11b — Undefined Refer to the USB Specification Revision 2.0 for a description on how these bits are to be used. This field is zero if *Port Power* is zero.
13	**Port Owner—R/W** Default = 1b. This bit unconditionally goes to a 0b when the *Configured* bit in the CONFIGFLAG register makes a 0b to 1b transition. This bit unconditionally goes to 1b whenever the *Configured* bit is zero. System software uses this field to release ownership of the port to a selected host controller (in the event that the attached device is not a high-speed device). Software writes a one to this bit when the attached device is not a high-speed device. A one in this bit means that a companion host controller owns and controls the port. See Section 4.2 for operational details.
12	**Port Power (PP)—R/W** or **RO.** The function of this bit depends on the value of the *Port Power Control* (*PPC*) field in the HCSPARAMS register. The behavior is as follows: **PPC** — **PP** — **Operation** 0b — 1b — RO—Host controller does not have port power control switches. Each port is hard-wired to power. 1b — 1b/0b — R/W—Host controller has port power control switches. This bit represents the current setting of the switch (0 = off, 1 = on). When power is not available on a port (i.e. *PP* equals a 0), the port is non-functional and will not report attaches, detaches, etc. When an over-current condition is detected on a powered port and *PPC* is a one, the *PP* bit in each affected port may be transitioned by the host controller from a 1 to 0 (removing power from the port).

Table 2-16. PORTSC — Port Status and Control (cont.)

Bit	Description
11:10	**Line Status—RO.** These bits reflect the current logical levels of the D+ (bit 11) and D- (bit 10) signal lines. These bits are used for detection of low-speed USB devices prior to the port reset and enable sequence. This field is valid only when the port enable bit is zero and the current connect status bit is set to a one. The encoding of the bits are: **Bits[11:10]** / **USB State** / **Interpretation** 00b / SE0 / Not Low-speed device, perform EHCI reset 10b / J-state / Not Low-speed device, perform EHCI reset 01b / K-state / Low-speed device, release ownership of port 11b / Undefined / Not Low-speed device, perform EHCI reset. This value of this field is undefined if *Port Power* is zero.
9	**Reserved.** This bit is reserved for future use, and should return a value of zero when read.
8	**Port Reset—R/W**. 1=Port is in Reset. 0=Port is not in Reset. Default = 0. When software writes a one to this bit (from a zero), the bus reset sequence as defined in the USB Specification Revision 2.0 is started. Software writes a zero to this bit to terminate the bus reset sequence. Software must keep this bit at a one long enough to ensure the reset sequence, as specified in the USB Specification Revision 2.0, completes. Note: when software writes this bit to a one, it must also write a zero to the *Port Enable* bit. Note that when software writes a zero to this bit there may be a delay before the bit status changes to a zero. The bit status will not read as a zero until after the reset has completed. If the port is in high-speed mode after reset is complete, the host controller will automatically enable this port (e.g. set the *Port Enable* bit to a one). A host controller must terminate the reset and stabilize the state of the port within 2 milliseconds of software transitioning this bit from a one to a zero. For example: if the port detects that the attached device is high-speed during reset, then the host controller must have the port in the enabled state within 2ms of software writing this bit to a zero. The *HCHalted* bit in the USBSTS register should be a zero before software attempts to use this bit. The host controller may hold Port Reset asserted to a one when the *HCHalted* bit is a one. This field is zero if *Port Power* is zero.
7	**Suspend—R/W.** 1=Port in suspend state. 0=Port not in suspend state. Default = 0. Port Enabled Bit and Suspend bit of this register define the port states as follows: **Bits [Port Enabled, Suspend]** / **Port State** 0X / Disable 10 / Enable 11 / Suspend When in suspend state, downstream propagation of data is blocked on this port, except for port reset. The blocking occurs at the end of the current transaction, if a transaction was in progress when this bit was written to 1. In the suspend state, the port is sensitive to resume detection. Note that the bit status does not change until the port is suspended and that there may be a delay in suspending a port if there is a transaction currently in progress on the USB. A write of zero to this bit is ignored by the host controller. The host controller will unconditionally set this bit to a zero when: • Software sets the *Force Port Resume* bit to a zero (from a one). • Software sets the *Port Reset* bit to a one (from a zero). If host software sets this bit to a one when the port is not enabled (i.e. Port enabled bit is a zero) the results are undefined. This field is zero if *Port Power* is zero.

Table 2-16. PORTSC — Port Status and Control (cont.)

Bit	Description
6	**Force Port Resume — R/W.** 1= Resume detected/driven on port. 0=No resume (K-state) detected/driven on port. Default = 0. This functionality defined for manipulating this bit depends on the value of the *Suspend* bit. For example, if the port is not suspended (*Suspend* and *Enabled* bits are a one) and software transitions this bit to a one, then the effects on the bus are undefined. Software sets this bit to a 1 to drive resume signaling. The Host Controller sets this bit to a 1 if a J-to-K transition is detected while the port is in the Suspend state. When this bit transitions to a one because a J-to-K transition is detected, the *Port Change Detect* bit in the USBSTS register is also set to a one. If software sets this bit to a one, the host controller must not set the *Port Change Detect* bit. Note that when the EHCI controller owns the port, the resume sequence follows the defined sequence documented in the USB Specification Revision 2.0. The resume signaling (Full-speed 'K') is driven on the port as long as this bit remains a one. Software must appropriately time the Resume and set this bit to a zero when the appropriate amount of time has elapsed. Writing a zero (from one) causes the port to return to high-speed mode (forcing the bus below the port into a high-speed idle). This bit will remain a one until the port has switched to the high-speed idle. The host controller must complete this transition within 2 milliseconds of software setting this bit to a zero. This field is zero if *Port Power* is zero.
5	**Over-current Change—R/WC.** Default = 0. 1=This bit gets set to a one when there is a change to Over-current Active. Software clears this bit by writing a one to this bit position.
4	**Over-current Active—RO.** Default = 0. 1=This port currently has an over-current condition. 0=This port does not have an over-current condition. This bit will automatically transition from a one to a zero when the over current condition is removed.
3	**Port Enable/Disable Change—R/WC.** 1=Port enabled/disabled status has changed. 0=No change. Default = 0. For the root hub, this bit gets set to a one only when a port is disabled due to the appropriate conditions existing at the EOF2 point (See Chapter 11 of the USB Specification for the definition of a Port Error). Software clears this bit by writing a 1 to it. This field is zero if *Port Power* is zero.
2	**Port Enabled/Disabled—R/W.** 1=Enable. 0=Disable. Default = 0. Ports can only be enabled by the host controller as a part of the reset and enable. Software cannot enable a port by writing a one to this field. The host controller will only set this bit to a one when the reset sequence determines that the attached device is a high-speed device. Ports can be disabled by either a fault condition (disconnect event or other fault condition) or by host software. 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 and bus events. See Section 4.2 for full details on port reset and enable. When the port is disabled (0b) downstream propagation of data is blocked on this port, except for reset. This field is zero if *Port Power* is zero.
1	**Connect Status Change—R/WC.** 1=Change in Current Connect Status. 0=No change. Default = 0. Indicates a change has occurred in the port's Current Connect Status. The host controller sets this bit for all changes to the port device connect status, 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, hub hardware will be "setting" an already-set bit (i.e., the bit will remain set). Software sets this bit to 0 by writing a 1 to it. This field is zero if *Port Power* is zero.

Table 2-16. PORTSC — Port Status and Control (cont.)

Bit	Description
0	**Current Connect Status—RO.** 1=Device is present on port. 0=No device is present. Default = 0. This value reflects the current state of the port, and may not correspond directly to the event that caused the Connect Status Change bit (Bit 1) to be set. This field is zero if *Port Power* is zero.

3. Data Structures

This section defines the interface data structures used to communicate control, status and data between HCD (software) and the Enhanced Host Controller (hardware). The data structure definitions in this chapter support a 32-bit memory buffer address space. Appendix B illustrates 64-bit versions of the interface data structures. The interface consists of a Periodic Schedule, Periodic Frame List, Asynchronous Schedule, Isochronous Transaction Descriptors, Split-transaction Isochronous Transfer Descriptors, Queue Heads and Queue Element Transfer Descriptors.

The periodic frame list is the root of all periodic (isochronous and interrupt transfer type) support for the host controller interface. The asynchronous list is the root for all the bulk and control transfer type support. Isochronous data streams are managed using Isochronous Transaction Descriptors (iTDs are described in Section 3.3). Isochronous split-transaction data streams are managed with Split-transaction Isochronous Transfer Descriptors (siTDs are described in Section 3.4). All Interrupt, Control and Bulk data streams are managed via queue heads (Section 3.6) and Queue Element Transfer Descriptors (qTDs described in Section 3.5).[1] These data structures 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.

Note that software must ensure that no interface data structure reachable by the EHCI host controller spans a 4K page boundary.

The data structures defined in this chapter are (from the host controller's perspective) a mix of read-only and read/writeable fields. The host controller must preserve the read-only fields on all data structure writes.

3.1 Periodic Frame List

This schedule is for all periodic transfers (isochronous and interrupt). The periodic schedule is referenced from the operational registers space using the *PERIODICLISTBASE* address register and the *FRINDEX* register. The periodic schedule is based on an array of pointers called the Periodic Frame List. The *PERIODICLISTBASE* address register is combined with the *FRINDEX* register to produce a memory pointer into the frame list. The Periodic Frame List implements a *sliding window* of work over time.

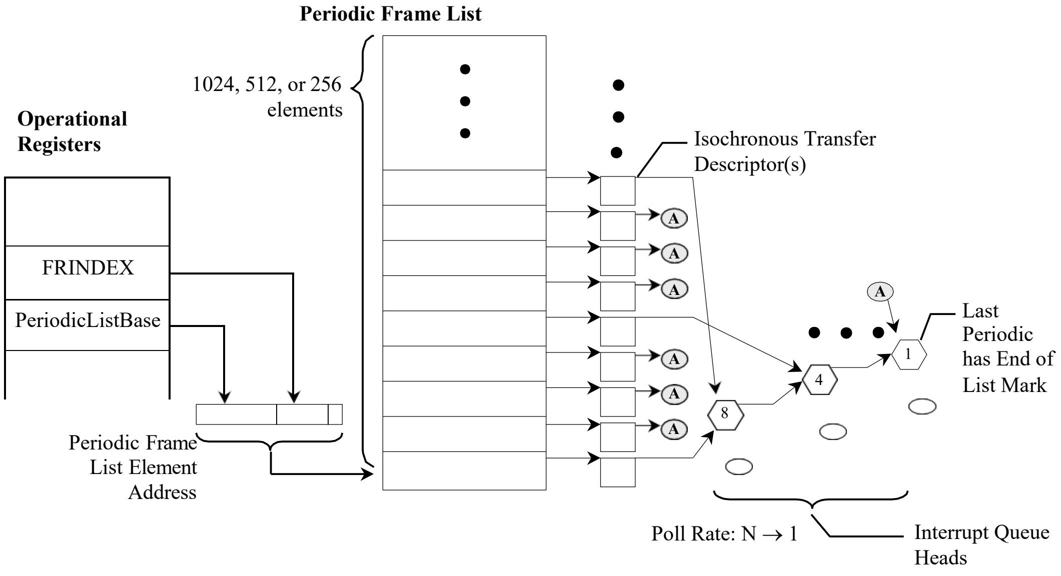

Periodic Frame List
1024, 512, or 256 elements
Operational Registers
FRINDEX
PeriodicListBase
Periodic Frame List Element Address
Isochronous Transfer Descriptor(s)
A
8
4
1
Last Periodic has End of List Mark
Poll Rate: N → 1
Interrupt Queue Heads

Figure 3-1. Periodic Schedule Organization

[1] Split transaction Interrupt, Bulk and Control are also managed using queue heads and queue element transfer descriptors.

The periodic frame list is a 4K-page aligned array of Frame List Link pointers. The length of the frame list may be programmable. The programmability of the periodic frame list is exported to system software via the HCCPARAMS register. If non-programmable, the length is 1024 elements. If programmable, the length can be selected by system software as one of 256, 512, or 1024 elements. An implementation must support all three sizes. Programming the size (i.e. the number of elements) is accomplished by system software writing the appropriate value into *Frame List Size* field in the USBCMD register.

Frame List Link pointers direct the host controller to the first work item in the frame's periodic schedule for the current micro-frame. The link pointers are aligned on DWORD boundaries within the Frame List.

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	
Frame List Link Pointer	0	Typ	T	03-00H

Figure 3-2. Format of Frame List Element Pointer

Frame List Link pointers always reference memory objects that are 32-byte aligned. The referenced object may be an isochronous transfer descriptor for high-speed devices, a split-transaction isochronous transfer descriptor (for full-speed isochronous endpoints), or a queue head (used to support high-, full- and low-speed interrupt). System software should not place non-periodic schedule items into the periodic schedule. The least significant bits in a frame list pointer are used to key the host controller as to the type of object the pointer is referencing.

The least significant bit is the *T-Bit* (bit 0). When this bit is set to a one, the host controller will never use the value of the frame list pointer as a physical memory pointer. The *Typ* field is used to indicate the exact type of data structure being referenced by this pointer. The value encodings are:

Table 3-1. Typ Field Value Definitions

Value	Meaning
00b	Isochronous Transfer Descriptor (iTD, see Section 3.3)
01b	Queue Head (QH, see Section 3.6)
10b	Split Transaction Isochronous Transfer Descriptor (siTD, see Section 3.4).
11b	Frame Span Traversal Node (FSTN, see Section 3.7)

Refer to Section 4.4 for host controller operational details with the frame list.

3.2 Asynchronous List Queue Head Pointer

The Asynchronous Transfer List (based at the ASYNCLISTADDR register), is where all the control and bulk transfers are managed. Host controllers use this list only when it reaches the end of the periodic list, the periodic list is disabled, or the periodic list is empty.

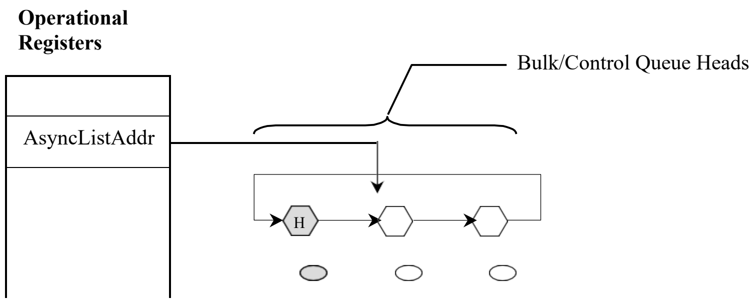


Figure 3-3. Asynchronous Schedule Organization

The Asynchronous list is a simple circular list of queue heads. The ASYNCLISTADDR register is simply a pointer to the *next* queue head. This implements a pure round-robin service for all queue heads linked into the asynchronous list.

3.3 Isochronous (High-Speed) Transfer Descriptor (iTD)

The format of an isochronous transfer descriptor is illustrated in Figure 3-4. This structure is used only for high-speed isochronous endpoints. All other transfer types should use queue structures. Isochronous TDs must be aligned on a 32-byte boundary.

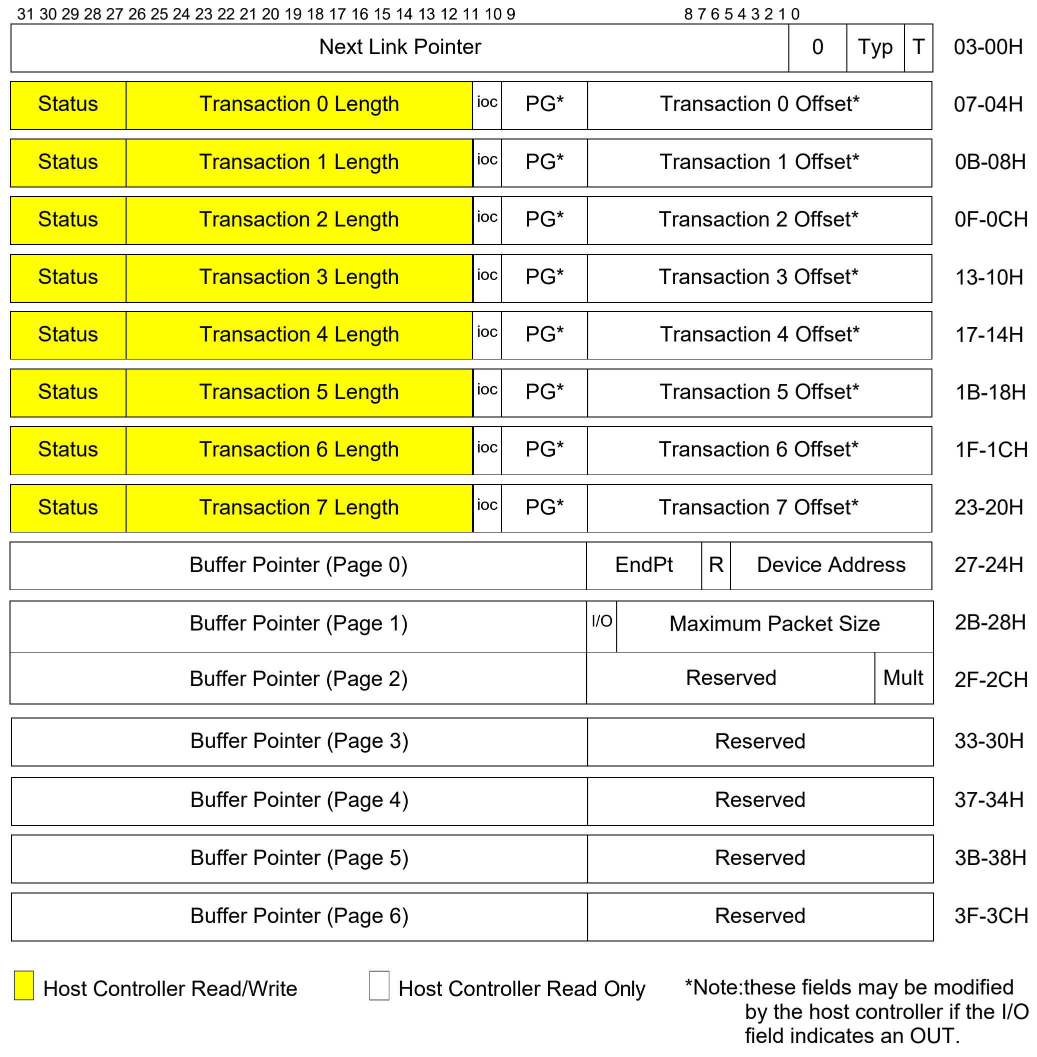

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Next Link Pointer | 0 | Typ | T | 03-00H
Status | Transaction 0 Length | ioc | PG* | Transaction 0 Offset* | 07-04H
Status | Transaction 1 Length | ioc | PG* | Transaction 1 Offset* | 0B-08H
Status | Transaction 2 Length | ioc | PG* | Transaction 2 Offset* | 0F-0CH
Status | Transaction 3 Length | ioc | PG* | Transaction 3 Offset* | 13-10H
Status | Transaction 4 Length | ioc | PG* | Transaction 4 Offset* | 17-14H
Status | Transaction 5 Length | ioc | PG* | Transaction 5 Offset* | 1B-18H
Status | Transaction 6 Length | ioc | PG* | Transaction 6 Offset* | 1F-1CH
Status | Transaction 7 Length | ioc | PG* | Transaction 7 Offset* | 23-20H
Buffer Pointer (Page 0) | EndPt | R | Device Address | 27-24H
Buffer Pointer (Page 1) | I/O | Maximum Packet Size | 2B-28H
Buffer Pointer (Page 2) | Reserved | Mult | 2F-2CH
Buffer Pointer (Page 3) | Reserved | 33-30H
Buffer Pointer (Page 4) | Reserved | 37-34H
Buffer Pointer (Page 5) | Reserved | 3B-38H
Buffer Pointer (Page 6) | Reserved | 3F-3CH
🟨 Host Controller Read/Write
☐ Host Controller Read Only
*Note:these fields may be modified by the host controller if the I/O field indicates an OUT.

Figure 3-4. Isochronous Transaction Descriptor (iTD)

3.3.1 Next Link Pointer

The first DWord of an iTD is a pointer to the next schedule data structure.

Table 3-2. Next Schedule Element Pointer

Bit	Description
31:5	**Link Pointer (LP).** These bits correspond to memory address signals [31:5], respectively. This field points to another Isochronous Transaction Descriptor (iTD/siTD), Queue Head (QH) or FSTN.
4:3	**Reserved.** These bits are reserved and their value have no effect on operation. Software should initialize this field to zero.

Table 3-2. Next Schedule Element Pointer (cont.)

Bit	Description
2:1	**QH/(s)iTD/FSTN Select (Typ).** This field indicates to the Host Controller whether the item referenced is a iTD, siTD or a QH. This allows the Host Controller to perform the proper type of processing on the item after it is fetched. Value encodings are: **Value** **Meaning** 00b iTD (isochronous transfer descriptor) 01b QH (queue head) 10b siTD (split transaction isochronous transfer descriptor) 11b FSTN (Frame Span Traversal Node)
0	**Terminate (T).** 1= Link Pointer field is not valid. 0= Link Pointer field is valid.

Refer to Section 4.7 for the host controller operational model for iTDs.

3.3.2 iTD Transaction Status and Control List

Dwords 1 through 8 are eight slots of transaction control and status. Each slot has the format illustrated in Figure 3-4. Each transaction description includes:

- Status results field
- Transaction length (bytes to send for OUT transactions and bytes received for IN transactions).
- Buffer offset. The *PG* and *Transaction X Offset* fields are used with the buffer pointer list to construct the starting buffer address for the transaction.

The host controller uses the information in each transaction description plus the endpoint information contained in the first three dwords of the Buffer Page Pointer list, to execute a transaction on the USB.

Table 3-3. iTD Transaction Status and Control

Bit	Description
31:28	**Status.** This field records the status of the transaction executed by the host controller for this slot. This field is a bit vector with the following encoding: Bit Definition 31 **Active.** Set to 1 by software to enable the execution of an isochronous transaction by the Host Controller. When the transaction associated with this descriptor is completed, the Host Controller sets this bit to 0 indicating that a transaction for this element should not be executed when it is next encountered in the schedule. 30 **Data Buffer Error.** Set to a 1 by the Host Controller during status update to indicate 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 4.15.1.1.2 defines the requirements of the host controller when an underrun error occurs. If an overrun condition occurs, no action is necessary. 29 **Babble Detected.** Set to a 1 by the Host Controller during status update when a “babble” is detected during the transaction generated by this descriptor. 28 **Transaction Error (XactErr).** Set to a one by the Host Controller during status update in the case where the host did not receive a valid response from the device (Timeout, CRC, Bad PID, etc.). This bit may only be set for isochronous IN transactions.

Table 3-3. iTD Transaction Status and Control (cont.)

Bit	Description
27:16	**Transaction X Length.** For an OUT, this field is the number of data bytes the host controller will send during the transaction. The host controller is not required to update this field to reflect the actual number of bytes transferred during the transfer. For an IN, the initial value of the field is the number of bytes the host expects the endpoint to deliver. During the status update, the host controller writes back the number of bytes successfully received. The value in this register is the actual byte count (e.g. 0 → zero length data, 1 → one byte, 2 → two bytes, etc.). The maximum value this field may contain is 0xC00 (3072). Refer to Section 4.7 for a description of the operational requirements for packet sizes larger than 0x400 (1024).
15	**Interrupt On Complete (IOC).** If this bit is set to a one, it specifies that when this transaction completes, the Host Controller should issue an interrupt at the next interrupt threshold.
14:12	**Page Select (PG).** These bits are set by software to indicate which of the buffer page pointers the offset field in this slot should be concatenated to produce the starting memory address for this transaction. The valid range of values for this field is 0 to 6.
11:0	**Transaction X Offset.** This field is a value that is an offset, expressed in bytes, from the beginning of a buffer. This field is concatenated onto the buffer page pointer indicated in the adjacent *PG* field to produce the starting buffer address for this transaction.

3.3.3 iTD Buffer Page Pointer List (Plus)

Dwords 9-15 of an isochronous transaction descriptor are nominally, page pointers (4K aligned) to the data buffer for this transfer descriptor. This data structure requires the associated data buffer to be contiguous (relative to virtual memory), but allows the physical memory pages to be non-contiguous. Seven page pointers are provided to support the expression of 8 isochronous transfers. The seven pointers allow for 3 (transactions) * 1024 (maximum packet size) * 8 (transaction records) (24576 bytes) to be moved with this data structure, regardless of the alignment offset of the first page.

Since each pointer is a 4K aligned page pointer, the least significant 12 bits in several of the page pointers are used for other purposes, as defined in Table 3-4 and Table 3-5.

Table 3-4. iTD Buffer Pointer Page 0 (Plus)

Bit	Description
31:12	**Buffer Pointer (Page 0).** This is a 4K aligned pointer to physical memory. Corresponds to memory address bits [31:12].
11:8	**Endpoint Number (Endpt).** This 4-bit field selects the particular endpoint number on the device serving as the data source or sink.
7	**Reserved.** This bit is reserved for future use and should be initialized by software to zero.
6:0	**Device Address.** This field selects the specific device serving as the data source or sink.

Table 3-5. iTD Buffer Pointer Page 1 (Plus)

Bit	Description
31:12	**Buffer Pointer (Page 1).** This is a 4K aligned pointer to physical memory. Corresponds to memory address bits [31:12].
11	**Direction (I/O).** 0 = OUT; 1 = IN. This field encodes whether the high-speed transaction should use an IN or OUT PID.
10:0	**Maximum Packet Size.** This directly corresponds to the maximum packet size of the associated endpoint (*wMaxPacketSize*). This field is used for high-bandwidth endpoints where more than one transaction is issued per transaction description (.e.g. per micro-frame). This field is used with the *Multi* field to support high-bandwidth pipes. This field is also used for all IN transfers to detect packet babble. See Section 4.7.1 for operational model. Software should not set a value larger than 1024 (400h). Any value larger yields undefined results.

Table 3-6. iTD Buffer Pointer Page 2 (Plus)

Bit	Description
31:12	**Buffer Pointer.** This is a 4K aligned pointer to physical memory. Corresponds to memory address bits [31:12].
11:2	**Reserved.** This bit is reserved for future use and should be set to zero.
1:0	**Multi.** This field is used to indicate to the host controller the number of transactions that should be executed per transaction description (e.g. per micro-frame). The valid values are: **Value** **Meaning** 00b Reserved. A zero in this field yields undefined results. 01b One transaction to be issued for this endpoint per micro-frame 10b Two transactions to be issued for this endpoint per micro-frame 11b Three transactions to be issued for this endpoint per micro-frame

Table 3-7. iTD Buffer Pointer Page 3-6

Bit	Description
31:12	**Buffer Pointer.** This is a 4K aligned pointer to physical memory. Corresponds to memory address bits [31:12].
11:0	**Reserved.** These bits are reserved for future use and should be set to zero.

3.4 Split Transaction Isochronous Transfer Descriptor (siTD)

All Full-speed isochronous transfers through Transaction Translators are managed using the siTD data structure. This data structure satisfies the operational requirements for managing the split transaction protocol. See Section 4.12.3 for operational behavior for siTDs.

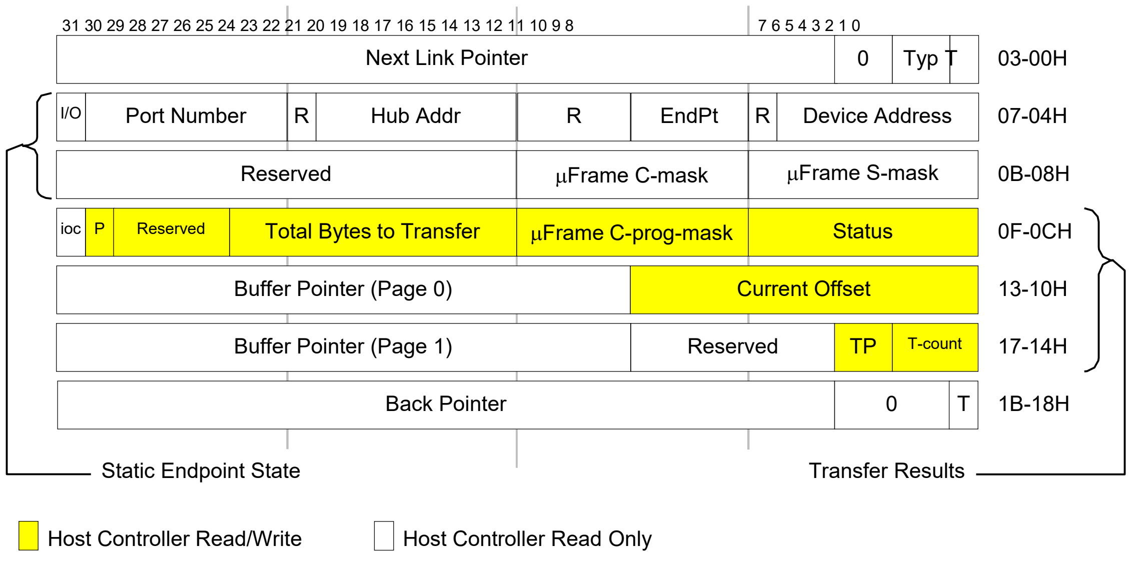


Figure 3-5. Split-transaction Isochronous Transaction Descriptor (siTD)

3.4.1 Next Link Pointer

DWord 0 of an siTD is a pointer to the next schedule data structure.

Table 3-8. Next Link Pointer

Bit	Description
31:5	**Next Link Pointer (LP).** This field contains the address of the next data object to be processed in the periodic list and corresponds to memory address signals [31:5], respectively.
4:3	**Reserved.** These bits must be written as 0s.
2:1	**QH/(s)iTD/FSTN Select (Typ).** This field indicates to the Host Controller whether the item referenced is a iTD/siTD or a QH. This allows the Host Controller to perform the proper type of processing on the item after it is fetched. Value encodings are: **Value** **Meaning** 00b iTD (isochronous transfer descriptor) 01b QH (queue head) 10b siTD (split transaction isochronous transfer descriptor) 11b FSTN (Frame Span Traversal Node)
0	**Terminate (T).** 1=Link Pointer field is not valid. 0=Link Pointer is valid.

3.4.2 siTD Endpoint Capabilities/Characteristics

Dwords 1 and 2 specify static information about the full-speed endpoint, the addressing of the parent transaction translator and micro-frame scheduling control.

Table 3-9. Endpoint and Transaction Translator Characteristics

Bit	Description
31	**Direction (I/O).** 0 = OUT; 1 = IN. This field encodes whether the full-speed transaction should be an IN or OUT.
30:24	**Port Number.** This field is the port number of the recipient transaction translator.

Table 3-9. Endpoint and Transaction Translator Characteristics (cont.)

Bit	Description
23	**Reserved.** This bit is reserved and should be set to zero.
22:16	**Hub Address.** This field holds the device address of the transaction translators' hub.
15:12	**Reserved.** This field is reserved and should be set to zero.
11:8	**Endpoint Number (Endpt).** This 4-bit field selects the particular endpoint number on the device serving as the data source or sink.
7	**Reserved.** This bit is reserved for future use. It should be set to zero.
6:0	**Device Address.** This field selects the specific device serving as the data source or sink.

Table 3-10. Micro-frame Schedule Control

Bit	Description
31:16	**Reserved.** This field is reserved for future use. It should be set to zero.
15:8	**Split Completion Mask (μFrame C-Mask).** This field (along with the *Active* and *SplitX-state* fields in the *Status* byte) are used to determine during which micro-frames the host controller should execute complete-split transactions. When the criteria for using this field are met, an all zeros value in this field has undefined behavior. The host controller uses the value of the three low-order bits of the FRINDEX register to index into this bit field. If the FRINDEX register value indexes to a position where the *μFrame C-Mask* field is a one, then this siTD is a candidate for transaction execution. There may be more than one bit in this mask set.
7:0	**Split Start Mask (μFrame S-mask).** This field (along with the *Active* and *SplitX-state* fields in the *Status* byte) are used to determine during which micro-frames the host controller should execute start-split transactions. The host controller uses the value of the three low-order bits of the FRINDEX register to index into this bit field. If the FRINDEX register value indexes to a position where the *μFrame S-mask* field is a one, then this siTD is a candidate for transaction execution. See Section 4.12.3.3.1 For a comprehensive list of criteria that must be met before host controller will execute a transaction. An all zeros value in this field, in combination with existing in the periodic frame list has undefined results.

3.4.3 siTD Transfer State

Dwords 3-6 are used to manage the state of the transfer.

Table 3-11. siTD Transfer Status and Control

Bit	Description
31	**Interrupt On Complete (ioc).** 0 = Do not interrupt when transaction is complete. 1 = Do interrupt when transaction is complete. When the host controller determines that the split transaction has completed it will assert a hardware interrupt at the next interrupt threshold.
30	**Page Select (P).** Used to indicate which data page pointer should be concatenated with the *CurrentOffset* field to construct a data buffer pointer (0 selects *Page 0* pointer and 1 selects *Page 1*). The host controller is not required to write this field back when the siTD is retired (*Active* bit transitioned from a one to a zero).
29:26	**Reserved.** This field is reserved for future use and should be set to zero.
25:16	**Total Bytes To Transfer.** This field is initialized by software to the total number of bytes expected in this transfer. Maximum value is 1023 (3FFh)

Table 3-11. siTD Transfer Status and Control (cont.)

Bit	Description
15:8	**µFrame Complete-split Progress Mask (C-prog-Mask).** This field is used by the host controller to record which split-completes have been executed. See Section 4.12.3.3.2 for behavioral requirements.
7:0	**Status.** This field records the status of the transaction executed by the host controller for this slot. This field is a bit vector with the following encoding: __Bit__ __Definition__ 7 **Active.** Set to 1 by software to enable the execution of an isochronous split transaction by the Host Controller. Refer to Section 4.12.3.3. 6 **ERR.** Set to a 1 by the Host Controller when an ERR response is received from the transaction translator. 5 **Data Buffer Error.** Set to a 1 by the Host Controller during status update to indicate 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). In the case of an underrun, the Host Controller will transmit an incorrect CRC (thus invalidating the data at the endpoint). If an overrun condition occurs, no action is necessary. 4 **Babble Detected.** Set to a 1 by the Host Controller during status update when a "babble" is detected during the transaction generated by this descriptor. 3 **Transaction Error (XactErr).** Set to a 1 by the Host Controller during status update in the case where the host did not receive a valid response from the device (Timeout, CRC, Bad PID, etc.). This bit will only be set for IN transactions. 2 **Missed Micro-Frame.** The host controller detected that a host-induced hold-off caused the host controller to miss a required complete-split transaction. 1 **Split Transaction State (SplitXstate).** See Section 4.12.3.3 for operational details. The bit encodings are: **Value** **Meaning** 0b **Do Start Split**. This value directs the host controller to issue a Start split transaction to the endpoint when a match is encountered in the S-mask. 1b **Do Complete Split**. This value directs the host controller to issue a Complete split transaction to the endpoint when a match is encountered in the C-mask. 0 **Reserved.** This bit is reserved for future use and should be set to zero.

3.4.4 siTD Buffer Pointer List (plus)

Dwords 4 and 5 are the data buffer page pointers for the transfer. This structure supports one physical page cross. The most significant 20 bits of each Dword in this section are the 4K (page) aligned buffer pointers. The least significant 12 bits of each Dword are used as additional transfer state.

Table 3-12. Buffer Page Pointer List (plus)

Bit	Description
31:12	**Buffer Pointer List.** Bits [31:12] of Dwords 4 and 5 are 4K paged aligned, physical memory addresses. These bits correspond to physical address bits [31:12] respectively. The lower 12 bits in each pointer are defined and used as specified below. The field *P* (see Table 3-11) specifies the *current* active pointer.

Table 3-12. Buffer Page Pointer List (plus) (cont.)

Bit	Description
11:0	**Page 0:** **Current Offset.** The 12 least significant bits of the Page 0 pointer is the current byte offset for the current page pointer (as selected with the page indicator bit (*P* field)). The host controller is not required to write this field back when the siTD is retired (*Active* bit transitioned from a one to a zero). The least significant bits of Page 1 pointer is split into three sub-fields: **Page 1:** **Bits Description** 11:5 Reserved. 4:3 **Transaction position (TP).** This field is used with T-count to determine whether to send *all*, *first*, *middle*, or *last* with each outbound transaction payload. System software must initialize this field with the appropriate starting value. The host controller must correctly manage this state during the lifetime of the transfer. The bit encodings are: **Value Meaning** 00b **All.** The entire full-speed transaction data payload is in this transaction (i.e. less than or equal to 188 bytes). 01b **Begin.** This is the first data payload for a full-speed transaction that is greater than 188 bytes. 10b **Mid.** This the *middle* payload for a full-speed OUT transaction that is larger than 188 bytes. 11b **End.** This is the *last* payload for a full-speed OUT transaction that was larger than 188 bytes. 2:0 **Transaction count (T-Count).** Software initializes this field with the number of OUT start-splits this transfer requires. Any value larger than 6 is undefined.

3.4.5 siTD Back Link Pointer

Dword 6 of an siTD is simply another schedule link pointer. This pointer is always zero, or references an siTD. This pointer cannot reference any other schedule data structure. See Section 4.12.3.3.2.1 for details on the operational model.

Table 3-13. siTD Back Link Pointer

Bit	Description
31:5	**siTD Back Pointer.** This field is a physical memory pointer to an siTD. See Section 4.12.3.3.2.1 for operational details.
4:1	**Reserved.** This field is reserved for future use. It should be set to zero.
0	**Terminate (T).** 1 = siTD Back Pointer field is not valid. 0 = siTD Back Pointer field is valid. Refer to Section 4.12.3.3.2.1 for use.

3.5 Queue Element Transfer Descriptor (qTD)

This data structure is only used with a queue head (see Section 3.6). This data structure is used for one or more USB transactions. See Section 4.10 for a complete description of the behavioral model. This data structure is used to transfer up to 20480 (5*4096) bytes. The structure contains two structure pointers used for queue advancement, a Dword of transfer state and a five-element array of data buffer pointers. This structure is 32 bytes (or one 32-byte cache line). This data structure must be physically contiguous.

The buffer associated with this transfer must be virtually contiguous. The buffer may start on any byte boundary. A separate buffer pointer list element must be used for each physical page in the buffer, regardless of whether the buffer is physically contiguous.

Host controller updates (host controller writes) to stand-alone qTDs only occur during transfer retirement (see Section 4.10.4). References in the following bit field definitions of updates to the 'qTD' are to the qTD portion of a queue head (see Figure 3-7).

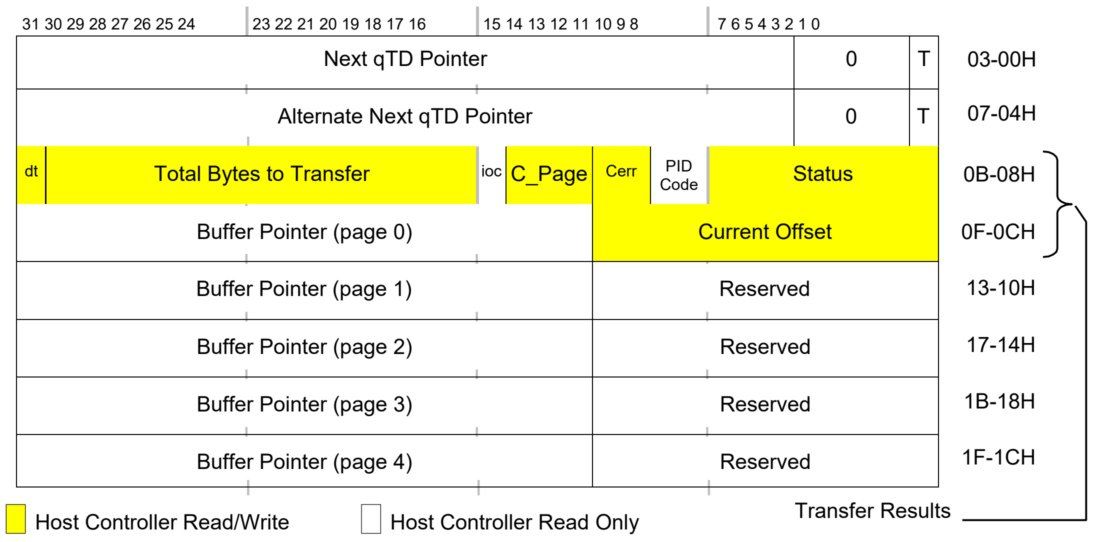


Figure 3-6. Queue Element Transfer Descriptor Block Diagram

Queue Element Transfer Descriptors must be aligned on 32-byte boundaries.

3.5.1 Next qTD Pointer

The first DWord of an element transfer descriptor is a pointer to another transfer element descriptor.

Table 3-14. qTD Next Element Transfer Pointer (DWord 0)

Bit	Description
31:5	**Next Transfer Element Pointer.** This field contains the physical memory address of the next qTD to be processed. The field corresponds to memory address signals [31:5], respectively.
4:1	**Reserved.** These bits are reserved and their value has no effect on operation.
0	**Terminate (T).** 1= pointer is invalid. 0=Pointer is valid (points to a valid Transfer Element Descriptor). This bit indicates to the Host Controller that there are no more valid entries in the queue.

3.5.2 Alternate Next qTD Pointer

The second DWord of a queue element transfer descriptor is used to support hardware-only advance of the data stream to the next client buffer on short packet. To be more explicit the host controller will always use this pointer when the current qTD is retired due to short packet.

Table 3-15. qTD Alternate Next Element Transfer Pointer (DWord 1)

Bit	Description
31:5	**Alternate Next Transfer Element Pointer.** This field contains the physical memory address of the next qTD to be processed in the event that the current qTD execution encounters a short packet (for an IN transaction). The field corresponds to memory address signals [31:5], respectively.
4:1	**Reserved.** These bits are reserved and their value has no effect on operation.
0	**Terminate (T).** 1= pointer is invalid. 0=Pointer is valid (points to a valid Transfer Element Descriptor). This bit indicates to the Host Controller that there are no more valid entries in the queue.

3.5.3 qTD Token

The third DWORD of a queue element transfer descriptor contains most of the information the host controller requires to execute a USB transaction (the remaining endpoint-addressing information is specified in the queue head).

Note: the field descriptions forward reference fields defined in the queue head (Section 3.6). Where necessary, these forward references are preceded with a *QH.* notation.

Table 3-16. qTD Token (DWord 2)

Bit	Description
31	**Data Toggle.** This is the data toggle sequence bit. The use of this bit depends on the setting of the *Data Toggle Control* bit in the queue head. See Section 4.10 for the operational model of data toggle.
30:16	**Total Bytes to Transfer.** This field specifies the total number of bytes to be moved with this transfer descriptor. This field is decremented by the number of bytes actually moved during the transaction, only on the successful completion of the transaction. The maximum value software may store in this field is 5 * 4K (5000H). This is the maximum number of bytes 5 page pointers can access. Please refer to Section 4.10.3 for a complete description of the host controller requirements when this field decrements to zero. If the value of this field is zero when the host controller fetches this transfer descriptor (and the active bit is set), the host controller executes a zero-length transaction and retires the transfer descriptor. It is not a requirement for OUT transfers that *Total Bytes To Transfer* be an even multiple of QHD.Maximum Packet Length. If software builds such a transfer descriptor for an OUT transfer, the last transaction will always be less than QHD.Maximum Packet Length.
15	**Interrupt On Complete (IOC).** If this bit is set to a one, it specifies that when this qTD is completed, the Host Controller should issue an interrupt at the next interrupt threshold.
14:12	**Current Page (C_Page).** This field is used as an index into the qTD buffer pointer list (Section 4.10.6). Valid values are in the range 0H to 4H. The host controller is not required to write this field back when the qTD is retired (see Section 4.10.4).

Table 3-16. qTD Token (DWord 2) (cont.)

Bit	Description
11:10	**Error Counter (CERR).** This field is a 2-bit down counter that keeps track of the number of consecutive Errors detected while executing this qTD. If this field is programmed with a non zero value during setup, the Host Controller decrements the count and writes it back to the qTD if the transaction fails. If the counter counts from one to zero, the Host Controller marks the qTD inactive, sets the *Halted* bit to a one and error status bit for the error that caused *CERR* to decrement to zero. An interrupt will be generated if the *USB Error Interrupt Enable* bit in the USBINTR register is set to a one. If HCD programs this field to zero during setup, the Host Controller will not count errors for this qTD and there will be no limit on the retries of this qTD. Note that write-backs of intermediate execution state are to the queue head overlay area, not the qTD. **Error** / **Decrement Counter** / **Error** / **Decrement Counter** Transaction Error / Yes / Data Buffer Error / No[3] Stalled / No[1] / Babble Detected / No[1] No Error / No[2] [1] Detection of Babble or Stall automatically halts the queue head. Thus, count is not decremented. [2] If the EPS field indicates a HS device or the queue head is in the Asynchronous Schedule (and *PIDCode* indicates an IN or OUT) and a bus transaction completes and the host controller does not detect a transaction error, then the host controller should reset *CERR* to extend the total number of errors for this transaction. For example, *CERR* should be reset with maximum value (3) on each successful completion of a transaction. The host controller must never reset this field if the value at the start of the transaction is 00b. See Sections 4.12.2.4.1and 4.12.2.4.2 for CERR adjustment rules when the EPS field indicates a FS or LS device and the queue head is in the Periodic Schedule. See Section 4.12.1.2 for CERR adjustment rules when the EPS field indicates a FS or LS device, the queue head is in the Asynchronous schedule and the *PIDCode* indicates a SETUP. [3] Data buffer errors are host problems. They don't count against the device's retries. **Note:** Software must not program CErr to a value of zero when the EPS field is programmed with a value indicating a Full- or Low-speed device. This combination could result in undefined behavior.
9:8	**PID Code.** This field is an encoding of the token which should be used for transactions associated with this transfer descriptor. Encodings are: 00b OUT Token generates token (E1H) 01b IN Token generates token (69H) 10b SETUP Token generates token (2DH) (undefined if endpoint is an Interrupt transfer type, e.g. μ*Frame S-mask* field in the queue head is non-zero.) 11b Reserved
7:0	**Status.** This field is used by the Host Controller to communicate individual command execution states back to HCD. This field contains the status of the last transaction performed on this qTD. The bit encodings are: **Bit** **Status Field Description** 7 **Active.** Set to 1 by software to enable the execution of transactions by the Host Controller. Refer to Section 4.10.3 for operational details about when the host controller transitions this bit from a one to a zero.

Table 3-16. qTD Token (DWord 2) (cont.)

Bit	Description
7:0 (cont.)	**Bit** **Status Field Description** 6 **Halted.** Set to a 1 by the Host Controller during status updates to indicate that a serious error has occurred at the device/endpoint addressed by this qTD. This can be caused by babble, the error counter counting down to zero, or reception of the STALL handshake from the device during a transaction. Any time that a transaction results in the Halted bit being set to a one, the Active bit is also set to 0. 5 **Data Buffer Error.** Set to a 1 by the Host Controller during status update to indicate 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 4.15.1.1.2 defines the requirements of the host controller when an underrun error occurs. If a overrun condition occurs, the Host Controller will force a timeout condition on the USB, invalidating the transaction at the source. If the host controller sets this bit to a one, then it remains a one for the duration of the transfer. 4 **Babble Detected.** Set to a 1 by the Host Controller during status update when a "babble" is detected during the transaction. In addition to setting this bit, the Host Controller also sets the *Halted* bit to a 1. Since "babble" is considered a fatal error for the transfer, setting the Halted bit to a 1 insures that no more transactions occur as a result of this descriptor. 3 **Transaction Error (XactErr).** Set to a one by the Host Controller during status update in the case where the host did not receive a valid response from the device (Timeout, CRC, Bad PID, etc.). Refer to Section 4.15.1.1 for summary of the conditions that affect this bit. If the host controller sets this bit to a one, then it remains a one for the duration of the transfer. 2 **Missed Micro-Frame.** This bit is ignored unless the *QH.EPS* field indicates a full- or low-speed endpoint and the queue head is in the periodic list. This bit is set when the host controller detected that a host-induced hold-off caused the host controller to miss a required complete-split transaction. If the host controller sets this bit to a one, then it remains a one for the duration of the transfer. 1 **Split Transaction State (SplitXstate).** This bit is ignored by the host controller unless the *QH.EPS* field indicates a full- or low-speed endpoint. When a Full-or Low-speed device, the host controller uses this bit to track the state of the split-transaction. The functional requirements of the host controller for managing this state bit and the split transaction protocol depends on whether the endpoint is in the periodic or asynchronous schedule. See Section 4.12 for operational details. The bit encodings are: **Value** **Meaning** 0b **Do Start Split**. This value directs the host controller to issue a Start split transaction to the endpoint. 1b **Do Complete Split**. This value directs the host controller to issue a Complete split transaction to the endpoint. 0 **Ping State (P)/ERR.** If the *QH.EPS* field indicates a High-speed device and the *PID_Code* indicates an OUT endpoint, then this is the state bit for the Ping protocol. See Section 4.11 for operational details. The bit encodings are: **Value** **Meaning** 0b **Do OUT**. This value directs the host controller to issue an OUT PID to the endpoint. 1b **Do Ping**. This value directs the host controller to issue a PING PID to the endpoint. If the *QH.EPS* field does not indicate a High-speed device, then this field is used as an error indicator bit. It is set to a one by the host controller whenever a periodic split-transaction receives an ERR handshake.

3.5.4 qTD Buffer Page Pointer List

The last five DWords of a queue element transfer descriptor is an array of physical memory address pointers. These pointers reference the individual pages of a data buffer. Refer to Section 4.10.6 for operational requirements.

System software initializes *Current Offset* field to the starting offset into the current page, where current page is selected via the value in the *C_Page* field.

Table 3-17. qTD Buffer Pointer(s) (DWords 3-7)

Bit	Description
31:12	**Buffer Pointer List.** Each element in the list is a 4K page aligned, physical memory address. The lower 12 bits in each pointer are reserved (except for the first one) as each memory pointer must reference the start of a 4K page. The field *C_Page* (see Table 3-16) specifies the *current* active pointer. When the transfer element descriptor is fetched, the starting buffer address is selected using *C_Page* (similar to an array index to select an array element). If a transaction spans a 4K buffer boundary, the host controller must detect the page-span boundary in the data stream, increment *C_Page* and advance to the next buffer pointer in the list, and conclude the transaction via the new buffer pointer. Refer to 4.10.6 for a complete description of the host controller operational requirements.
11:0	**Current Offset (Reserved).** This field is reserved in all pointers except the first one (e.g. Page 0). The host controller should ignore all reserved bits. For the page 0 current offset interpretation, this field is the byte offset into the current page (as selected by *C_Page*). The host controller is not required to write this field back when the qTD is retired (see Section 4.10.4). Software should ensure the Reserved fields are initialized to zeros.

3.6 Queue Head

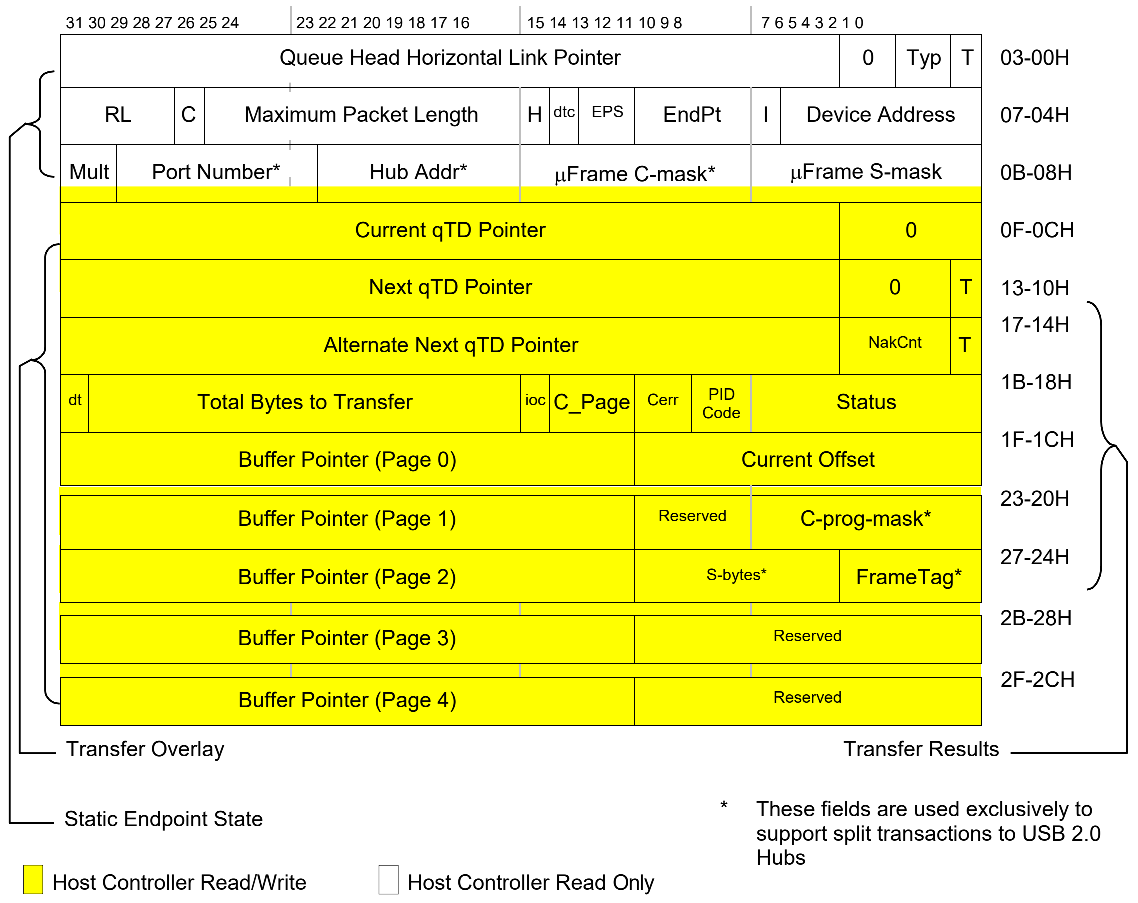


Figure 3-7. Queue Head Structure Layout

3.6.1 Queue Head Horizontal Link Pointer

The first DWord of a Queue Head contains a link pointer to the next data object to be processed after any required processing in this queue has been completed, as well as the control bits defined below.

This pointer may reference a queue head (Section 3.6) or one of the isochronous transfer descriptors ((Sections 3.3 and 3.4). It must not reference a queue element transfer descriptor (Section 3.5).

Table 3-18. Queue Head DWord 0

Bit	Description
31:5	**Queue Head Horizontal Link Pointer (QHLP).** This field contains the address of the next data object to be processed in the horizontal list and corresponds to memory address signals [31:5], respectively.
4:3	**Reserved.** These bits must be written as 0s.

Table 3-18. Queue Head DWord 0 (cont.)

Bit	Description
2:1	**QH/(s)iTD/FSTN Select (Typ).** This field indicates to the hardware whether the item referenced by the link pointer is a iTD, siTD or a QH. This allows the Host Controller to perform the proper type of processing on the item after it is fetched. Value encodings are: **Value** **Meaning** 00b iTD (isochronous transfer descriptor) 01b QH (queue head) 10b siTD (split transaction isochronous transfer descriptor) 11b FSTN (Frame Span Traversal Node)
0	**Terminate (T).** 1=Last QH (pointer is invalid). 0=Pointer is valid. If the queue head is in the context of the periodic list, a one bit in this field indicates to the host controller that this is the end of the periodic list. This bit is ignored by the host controller when the queue head is in the Asynchronous schedule. Software must ensure that queue heads reachable by the host controller always have valid horizontal link pointers. See Section 4.8.2

3.6.2 Endpoint Capabilities/Characteristics

The second and third Dwords of a Queue Head specifies static information about the endpoint. This information does not change over the lifetime of the endpoint. There are three types of information in this region:

- **Endpoint Characteristics.** These are the USB endpoint characteristics including addressing, maximum packet size, and endpoint speed.
- **Endpoint Capabilities.** These are adjustable parameters of the endpoint. They effect how the endpoint data stream is managed by the host controller. See Section 4.10.
- **Split Transaction Characteristics.** This data structure is used to manage full- and low-speed data streams for bulk, control, and interrupt via split transactions to USB 2.0 Hub Transaction Translator. There are additional fields used for addressing the hub and scheduling the protocol transactions (for periodic). See Section 4.12.

The host controller must not modify the bits in this region.

Table 3-19. Endpoint Characteristics: Queue Head DWord 1

Bit	Description
31:28	**Nak Count Reload (RL).** This field contains a value, which is used by the host controller to reload the Nak Counter field.
27	**Control Endpoint Flag (C).** If the *QH.EPS* field indicates the endpoint is not a high-speed device, and the endpoint is an control endpoint, then software must set this bit to a one. Otherwise it should always set this bit to a zero.
26:16	**Maximum Packet Length.** This directly corresponds to the maximum packet size of the associated endpoint (*wMaxPacketSize*). The maximum value this field may contain is 0x400 (1024).
15	**Head of Reclamation List Flag (H).** This bit is set by System Software to mark a queue head as being the head of the reclamation list. See Section 4.8 for operational model.
14	**Data Toggle Control (DTC).** This bit specifies where the host controller should get the initial data toggle on an overlay transition. 0b Ignore DT bit from incoming qTD. Host controller preserves DT bit in the queue head. 1b Initial data toggle comes from incoming qTD DT bit. Host controller replaces DT bit in the queue head from the DT bit in the qTD.

Table 3-19. Endpoint Characteristics: Queue Head DWord 1 (cont.)

Bit	Description
13:12	**Endpoint Speed (EPS).** This is the speed of the associated endpoint. Bit combinations are: Value Meaning 00b Full-Speed (12Mbs) 01b Low-Speed (1.5Mbs) 10b High-Speed (480 Mb/s) 11b Reserved This field must not be modified by the host controller.
11:8	**Endpoint Number (Endpt).** This 4-bit field selects the particular endpoint number on the device serving as the data source or sink.
7	**Inactivate on Next Transaction (I).** This bit is used by system software to request that the host controller set the Active bit to zero. See Section 4.12.2.5 for full operational details. This field is only valid when the queue head is in the Periodic Schedule and the *EPS* field indicates a Full or Low-speed endpoint. Setting this bit to a one when the queue head is in the Asynchronous Schedule or the *EPS* field indicates a high-speed device yields undefined results.
6:0	**Device Address.** This field selects the specific device serving as the data source or sink.

Table 3-20. Endpoint Capabilities: Queue Head DWord 2

Bit	Description
31:30	**High-Bandwidth Pipe Multiplier (Mult).** This field is a multiplier used to key the host controller as the number of successive packets the host controller may submit to the endpoint in the current execution. The host controller makes the simplifying assumption that software properly initializes this field (regardless of location of queue head in the schedules or other run time parameters). See Section 4.10.3 for correct software operational model. The valid values are: **Value** **Meaning** 00b Reserved. A zero in this field yields undefined results. 01b One transaction to be issued for this endpoint per micro-frame 10b Two transactions to be issued for this endpoint per micro-frame 11b Three transactions to be issued for this endpoint per micro-frame
29:23	**Port Number.** This field is ignored by the host controller unless the *EPS* field indicates a full- or low-speed device. The value is the port number identifier on the USB 2.0 Hub (for hub at device address *Hub Addr* below), below which the full- or low-speed device associated with this endpoint is attached. This information is used in the split-transaction protocol. See Section 4.12.
22:16	**Hub Addr.** This field is ignored by the host controller unless the *EPS* field indicates a full- or low-speed device. The value is the USB device address of the USB 2.0 Hub below which the full- or low-speed device associated with this endpoint is attached. This field is used in the split-transaction protocol. See Section 4.12.

Table 3-20. Endpoint Capabilities: Queue Head DWord 2 (cont.)

Bit	Description
15:8	**Split Completion Mask (μFrame C-Mask).** This field is ignored by the host controller unless the *EPS* field indicates this device is a low- or full-speed device and this queue head is in the periodic list. This field (along with the *Active* and *SplitX-state* fields) is used to determine during which micro-frames the host controller should execute a complete-split transaction. When the criteria for using this field are met, a zero value in this field has undefined behavior. This field is used by the host controller to match against the three low-order bits of the FRINDEX register. If the FRINDEX register bits decode to a position where the μ*Frame C-Mask* field is a one, then this queue head is a candidate for transaction execution. See Section 4.10.3 for a comprehensive list of criteria that must be met before host controller will execute a transaction. There may be more than one bit in this mask set. See Section 4.12.2.1.
7:0	**Interrupt Schedule Mask (μFrame S-mask).** This field is used for all endpoint speeds. Software should set this field to a zero when the queue head is on the asynchronous schedule. A non-zero value in this field indicates an interrupt endpoint. The host controller uses the value of the three low-order bits of the FRINDEX register as an index into a bit position in this bit vector. If the μ*Frame S-mask* field has a one at the indexed bit position then this queue head is a candidate for transaction execution. See Section 4.10.3 for a comprehensive list of criteria that must be met before host controller will execute a transaction. If the *EPS* field indicates the endpoint is a high-speed endpoint, then the transaction executed is determined by the *PID_Code* field contained in the execution area. This field is also used to support split transaction types: Interrupt (IN/OUT). See Section 4.12.2. This condition is true when this field is non-zero and the *EPS* field indicates this is either a full- or low-speed device. A zero value in this field, in combination with existing in the periodic frame list has undefined results.

3.6.3 Transfer Overlay

The nine DWords in this area represent a *transaction working space* for the host controller. The general operational model is that the host controller can detect whether the overlay area contains a description of an active transfer. If it does not contain an active transfer, then it follows the *Queue Head Horizontal Link Pointer* to the next queue head. The host controller will never follow the *Next Transfer Queue Element or Alternate Queue Element* pointers unless it is actively attempting to advance the queue (see Section 4.10). For the duration of the transfer, the host controller keeps the incremental status of the transfer in the overlay area. When the transfer is complete, the results are written back to the original queue element. The complete operational model of how this area is used by the host controller is described in Section 4.10.

The DWord 3 of a Queue Head contains a pointer to the source qTD currently associated with the overlay. The host controller uses this pointer to write back the overlay area into the source qTD after the transfer is complete.

Table 3-21. Current qTD Link Pointer

Bit	Description
31:5	**Current Element Transaction Descriptor Link Pointer.** This field contains the address of the current transaction being processed in this queue and corresponds to memory address signals [31:5], respectively.
4:0	**Reserved (R).** These bits are ignored by the host controller when using the value as an address to write data. The actual value may vary depending on the usage.

The DWords 4-11 of a queue head are the transaction overlay area. This area has the same base structure as a Queue Element Transfer Descriptor, defined in Section 3.5. The queue head utilizes the reserved fields of the page pointers defined in Figure 3-7 to implement tracking the state of split transactions.

This area is characterized as an *overlay* because when the queue is advanced to the next queue element, the source queue element is *merged* onto this area. This area serves an execution cache for the transfer. Section 4.10 describes which fields in the queue head are over-written during queue advancement.

Table 3-22. Host-Controller Rules for Bits in Overlay (DWords 5, 6, 8 and 9)

Dword	Bit	Description
5	4:1	**Nak Counter (NakCnt)—RW.** This field is a counter the host controller decrements whenever a transaction for the endpoint associated with this queue head results in a Nak or Nyet response. This counter is reloaded from *RL* before a transaction is executed during the first pass of the reclamation list (relative to an Asynchronous List Restart condition). See Section 4.9. It is also loaded from RL during an overlay. See Section 4.10.2.
6	31	**Data Toggle.** The *Data Toggle Control* controls whether the host controller preserves this bit when an overlay operation is performed.
6	15	**Interrupt On Complete (IOC).** The IOC control bit is always inherited from the source qTD when the overlay operation is performed.
6	11:10	**Error Counter (C_ERR).** This two-bit field is copied from the qTD during the overlay and written back during queue advancement. Refer to Table 3-16 for details.
6	0	**Ping State (P)/ERR.** If the *EPS* field indicates a high-speed endpoint, then this field should be preserved during the overlay operation.
8	7:0	**Split-transaction Complete-split Progress (C-prog-mask).** This field is initialized to zero during any overlay. This field is used to track the progress of an interrupt split-transaction. See Section 4.12.2 for details of the operational model.
9	4:0	**Split-transaction Frame Tag (Frame Tag).** This field is initialized to zero during any overlay. This field is used to track the progress of an interrupt split-transaction. See Section 4.12.2 for details of the operational model.
9	11:5	**S-bytes.** Software must ensure that the *S-bytes* field in a *qTD* are zero before activating the *qTD*. This field is used to keep track of the number of bytes sent or received during a IN or OUT split transaction. Refer to Section 4.12.2 for details of the operational model.

3.7 Periodic Frame Span Traversal Node (FSTN)

This data structure is to be used only for managing Full- and Low-speed transactions that span a Host-frame boundary. See Section 4.12.2.2 for full operational details. Software must not use an FSTN in the Asynchronous Schedule. An FSTN in the Asynchronous schedule results in undefined behavior. Software must not use the FSTN feature with a host controller whose HCIVERSION register indicates a revision implementation below 0096h. FSTNs are not defined for implementations before 0.96 and their use will yield undefined results.

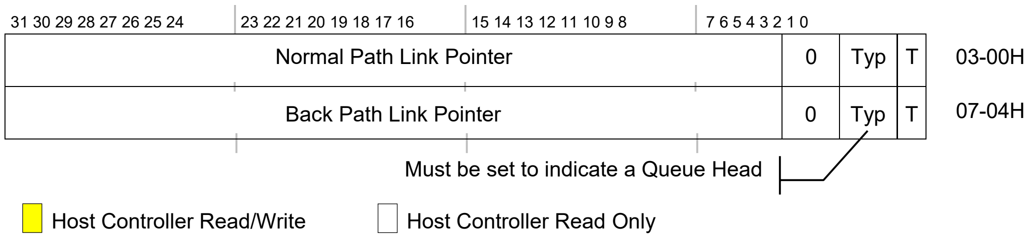


Figure 3-8. Frame Span Traversal Node Structure Layout

3.7.1 FSTN Normal Path Pointer

The first DWord of an FSTN contains a link pointer to the next schedule object. This object can be of any valid periodic schedule data type.

Bit	Description
31:5	**Normal Path Link Pointer (NPLP).** This field contains the address of the next data object to be processed in the periodic list and corresponds to memory address signals [31:5], respectively.
4:3	**Reserved.** These bits must be written as 0s.
2:1	**QH/(s)iTD/FSTN Select (Typ).** This field indicates to the Host Controller whether the item referenced is a iTD/siTD, a QH or an FSTN. This allows the Host Controller to perform the proper type of processing on the item after it is fetched. Value encodings are: **Value** **Meaning** 00b iTD (isochronous transfer descriptor) 01b QH (queue head) 10b siTD (split transaction isochronous transfer descriptor) 11b FSTN (Frame Span Traversal Node)
0	**Terminate (T).** 1=Link Pointer field is not valid. 0=Link Pointer is valid.

3.7.2 FSTN Back Path Link Pointer

The second DWord of an FTSN node contains a link pointer to a queue head. If the *T-bit* in this pointer is a zero, then this FSTN is a *Save-Place* indicator. Its *Typ* field must be set by software to indicate the target data structure is a queue head. If the *T-bit* in this pointer is set to a one, then this FSTN is the *Restore* indicator. When the *T-bit* is a one, the host controller ignores the *Typ* field.

Bit	Description
31:5	**Back Path Link Pointer (BPLP).** This field contains the address of a Queue Head. This field corresponds to memory address signals [31:5], respectively.
4:3	**Reserved.** These bits must be written as 0s.
2:1	**Typ.** Software must ensure this field is set to indicate the target data structure is a Queue Head. Any other value in this field yields undefined results.
0	**Terminate (T).** 1=Link Pointer field is not valid (i.e. the host controller must not use bits [31:5] (in combination with the CTRLDSSEGMENT register if applicable) as a valid memory address). This value also indicates that this FSTN is a *Restore* indicator. 0=Link Pointer is valid (i.e. the host controller may use bits [31:5] (in combination with the CTRLDSSEGMENT register if applicable) as a valid memory address). This value also indicates that this FSTN is a *Save-Place* indicator.

4. Operational Model

The general operational model is for the enhanced interface host controller hardware and enhanced interface host controller driver (generally referred to as system software). Each significant operational feature of the EHCI host controller is discussed in a separate section. Each section presents the operational model requirements for the host controller hardware. Where appropriate, recommended system software operational models for features are also presented.

4.1 Host Controller Initialization

When the system boots, the host controller is enumerated, assigned a base address for the register space and BIOS sets the FLADJ register to a system-specific value. After initial power-on or *HCReset* (hardware or via *HCReset* bit in the USBCMD register), all of the operational registers will be at their default values, as illustrated in Table 4–1. After a hardware reset, only the operational registers not contained in the Auxiliary power well will be at their default values.

Table 4–1. Default Values of Operational Register Space

Operational Register	Default Value (after Reset)
USBCMD	00080000h (00080B00h if *Asynchronous Schedule Park Capability is a one*)
USBSTS	00001000h
USBINTR	00000000h
FRINDEX	00000000h
CTRLDSSEGMENT	00000000h
PERIODICLISTBASE	Undefined
ASYNCLISTADDR	Undefined
CONFIGFLAG	00000000h
PORTSC	00002000h (w/*PPC* set to one); 00003000h (w/*PPC* set to a zero)

In order to initialize the host controller, software should perform the following steps

- Program the CTRLDSSEGMENT register with 4-Gigabyte segment where all of the interface data structures are allocated.
- Write the appropriate value to the USBINTR register to enable the appropriate interrupts.
- Write the base address of the Periodic Frame List to the PERIODICLIST BASE register. If there are no work items in the periodic schedule, all elements of the Periodic Frame List should have their *T-Bits* set to a one.
- Write the USBCMD register to set the desired interrupt threshold, frame list size (if applicable) and turn the host controller *ON* via setting the *Run/Stop* bit.
- Write a 1 to CONFIGFLAG register to route all ports to the EHCI controller (see Section 4.2).

At this point, the host controller is up and running and the port registers will begin reporting device connects, etc. System software can enumerate a port through the reset process (where the port is in the enabled state). At this point, the port is active with SOFs occurring down the enabled port enabled High-speed ports, but the schedules have not yet been enabled. The EHCI Host controller will not transmit SOFs to enabled Full- or Low-speed ports.

In order to communicate with devices via the asynchronous schedule, system software must write the ASYNDLISTADDR register with the address of a control or bulk queue head. Software must then enable the asynchronous schedule by writing a one to the *Asynchronous Schedule Enable* bit in the USBCMD register. In order to communicate with devices via the periodic schedule, system software must enable the periodic schedule by writing a one to the *Periodic Schedule Enable* bit in the USBCMD register. Note that the schedules can be turned on before the first port is reset (and enabled).

Any time the USBCMD register is written, system software must ensure the appropriate bits are preserved, depending on the intended operation.

4.2 Port Routing and Control

A USB 2.0 Host controller (as described in Section 1.1) is comprised of one high-speed host controller, which implements the EHCI programming interface and 0 to N USB 1.1 companion host controllers. Companion host controllers (cHCs) may be implementations of either Universal or Open host controller specifications. This configuration is used to deliver the required full USB 2.0-defined port capability; e.g. Low-, Full-, and High-speed capability for every port. Figure 4-1 illustrates a simple block diagram of the port routing logic and its relationship to the high-speed and companion host controllers within a USB 2.0 host controller.

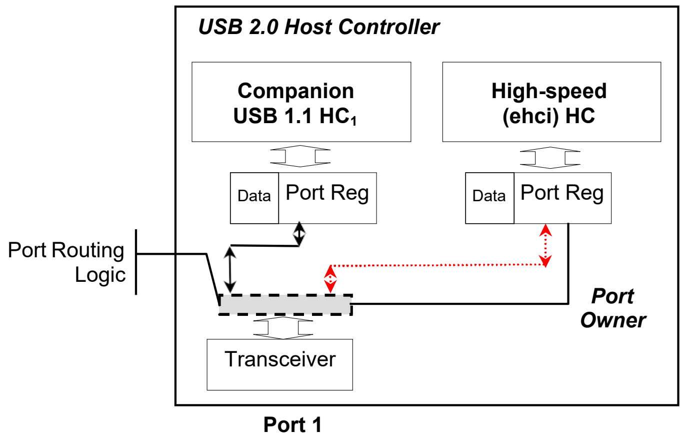


Figure 4-1. Example USB 2.0 Host Controller Port Routing Block Diagram

There exists one transceiver per physical port and each host controller module has its own port status and control registers. The EHCI controller has port status and control registers for every port. Each companion host controller has only the port control and status registers it is required to operate. Each transceiver can be controlled by either the EHCI host controller or one companion host controller. Routing logic lies between the transceiver and the port status and control registers.[2] The port routing logic is controlled from signals originating in the EHCI host controller. The EHCI host controller has a *global* routing policy control field and per-port *ownership* control fields. The *Configured Flag* (*CF*) bit (defined in Section 2.3.8) is the global routing policy control. At power-on or reset, the default routing policy is to the companion controllers (if they exist). If the system does not include a driver for the EHCI host controller and the host controller includes Companion Controllers, then the ports will still work in Full- and Low-speed mode (assuming the system includes a driver for the companion controllers). In general, when the EHCI owns the ports, the companion host controllers' port registers do not see a connect indication from the transceiver. Similarly, when a companion host controller owns a port, the EHCI controller's port registers do not see a connect indication from the transceiver. The details on the rules for the port routing logic are described in the following sections.

The USB 2.0 host controller must be implemented as a multi-function PCI device if the implementation includes companion controllers. The companion host controllers' function numbers must be less than the EHCI host controller function number. The EHCI host controller must be a larger function number with respect to the companion host controllers associated with this EHCI host controller. If a PCI device

[2] The routing logic should not be implemented in the 480 MHz clock domain of the transceiver.

implementation contains only an EHCI controller (i.e. no companion controllers or other PCI functions), then the EHCI host controller must be function zero, in accordance with the PCI Specification.

The *N_CC* field in the Structural Parameter register (*HCSPARAMS*) indicates whether the controller implementation includes companion host controllers. When *N_CC* has a non-zero value there exists companion host controllers. If N_CC has a value of zero, then the host controller implementation does not include companion host controllers. If the host controller root ports are exposed to attachment of full- or low-speed devices, the ports will always fail the high-speed chirp during reset and the ports will not be enabled. System software can notify the user of the illegal condition. This type of implementation requires a USB 2.0 hub be connected to a root port to provide full and low-speed device connectivity.

System software uses information in the host controller capability registers to determine how the ports are routed to the companion host controllers. See Sections 2.2.5 and 2.2.3.[3]

4.2.1 Port Routing Control via EHCI *Configured* (*CF*) Bit

Each port in the USB 2.0 host controller can be routed either to a single companion host controller or to the EHCI host controller. The port routing logic is controlled by two mechanisms in the EHCI HC: a host controller global flag and per-port control. The *Configured Flag* (*CF*) bit (defined in Section 2.3.8), is used to globally set the policy of the routing logic. Each port register has a *Port Owner* control bit which allows the EHCI Driver to explicitly control the routing of individual ports. Whenever the *CF bit* transitions from a zero to a one (this transition is only available under program control) the port routing unconditionally routes all of the port registers to the EHCI HC (all *Port Owner* bits go to zero). While the *CF-bit* is a one, the EHCI Driver can control individual ports' routing via the *Port Owner* control bit. Likewise, whenever the *CF bit* transitions from a one to a zero (as a result of Aux power application, *HCRESET*, or software writing a zero to *CF-bit*), the port routing unconditionally routes all of the port registers to the appropriate companion HC. The default value for the EHCI HC's *CF bit* (after Aux power application or *HCRESET*) is zero. Table 4–2 summarizes the default routing for all the ports, based on the value of the EHCI HC's *CF bit*.

The *view* of the port depends on the current owner. A Universal or Open companion host controller will see port register bits consistent with the appropriate specification. Port bit definitions that are required for EHCI host controllers are not visible to companion host controllers.

Table 4–2. Default Port Routing Depending on EHCI HC CF Bit

HS CF Bit	Default Port Ownership	Explanation
0B	Companion HCs	The companion host controllers own the ports and only Full- and Low-speed devices are supported in the system. The exact port assignments are implementation dependent. The ports behave only as Full- and Low-speed ports in this configuration.
1B	EHCI HC	The EHCI host controller has default ownership over all of the ports. The routing logic inhibits device connect events from reaching the companion HCs' port status and control registers when the port owner is the EHCI HC. The EHCI HC has access to the additional port status and control bits defined in this specification (see Section 2.3.9). The EHCI HC can temporarily release control of the port to a companion HC by setting the *PortOwner* bit in the PORTSC register to a one.

[3] If an implementation includes more than one set of companion and EHCI host controllers, they are organized as groups of companion host controllers with intermixed EHCI controllers.

4.2.2 Port Routing Control via *PortOwner* and Disconnect Event

Manipulating the port routing via the *CF-bit* is an extreme process and not intended to be used during normal operation. The normal mode of port ownership transferal is on the granularity of individual ports using the *Port Owner* bit in the EHCI HC's PORTSC register (for hand-offs from EHCI to companion host controllers). Individual port ownership is returned to the EHCI controller when the port registers a device disconnect. When the disconnect is detected, the port routing logic immediately returns the port ownership to the EHCI controller. The companion host controller port register detects the device disconnect and operates normally.

Under normal operating conditions (assuming all HC drivers loaded and operational and the EHCI *CF-bit* is set to a one), the typical port enumeration sequence proceeds as illustrated below:

- Initial condition is that EHCI is port owner. A device is connected causing the port to detect a connect, set the port connect change bit and issue a port-change interrupt (if enabled).
- EHCI Driver identifies the port with the new connect change bit asserted and sends a change report to the hub driver. Hub driver issues a GetPortStatus() request and identifies the connect change. It then issues a request to clear the connect change, followed by a request to reset and enable the port.
- When the EHCI Driver receives the request to reset and enable the port, it first checks the value reported by the *LineStatus* bits in the PORTSC register. If they indicate the attached device is a full-speed device (e.g. D+ is asserted), then the EHCI Driver sets the *PortReset* control bit to a one (and sets the *PortEnable* bit to a zero) which begins the reset-process. Software times the duration of the reset, then terminates reset signaling by writing a zero to the port reset bit. The reset process is actually complete when software reads a zero in the *PortReset* bit. The EHCI Driver checks the *PortEnable* bit in the PORTSC register. If set to a one, the connected device is a high-speed device and EHCI Driver (root hub emulator) issues a change report to the hub driver and the hub driver continues to enumerate the attached device.
- At the time the EHCI Driver receives the port reset and enable request the *LineStatus* bits might indicate a low-speed device. Additionally, when the port reset process is complete, the *PortEnable* field may indicate that a full-speed device is attached. In either case the EHCI driver sets the *PortOwner* bit in the PORTSC register to a one to release port ownership to a companion host controller.
- When the EHCI Driver sets *PortOwner* bit to a one, the port routing logic makes the connection state of the transceiver available to the companion host controller port register and removes the connection state from the EHCI HC port. The EHCI PORTSC register observes and reports a disconnect event via the disconnect change bit. The EHCI Driver detects the connection status change (either by polling or by port change interrupt) and then sends a change report to the hub driver. When the hub driver requests that port-state, the EHCI Driver responds with a reset complete change set to a one, a connect change set to a one and a connect status set to a zero. This information is derived directly from the EHCI port register. This will allow the hub driver to assume the device was disconnected during reset. It will acknowledge the change bits and wait for the next change event. While the EHCI controller does not own the port, it simply remains in a state where the port reports no device connected.

  The device-connect evaluation circuitry of the companion HC activates and detects the device, the companion Driver detects the connection and enumerates the port.

When a port is routed to a companion HC, it remains under the control of the companion HC until the device is disconnected from the root port (ignoring for now the scenario where EHCI's *CF-bit* transitions from a 1b to a 0b). When a disconnect occurs, the disconnect event is detected by both the companion HC port control and the EHCI port ownership control. On the event, the port ownership is returned immediately to the EHCI controller. The companion HC stack detects the disconnect and acknowledges as it would in an ordinary standalone implementation. Subsequent connects will be detected by the EHCI port register and the process will repeat.

4.2.3 Example Port Routing State Machine

Figure 4-2 illustrates an example of how the port ownership should be managed. The following sections describe the entry conditions to each state.

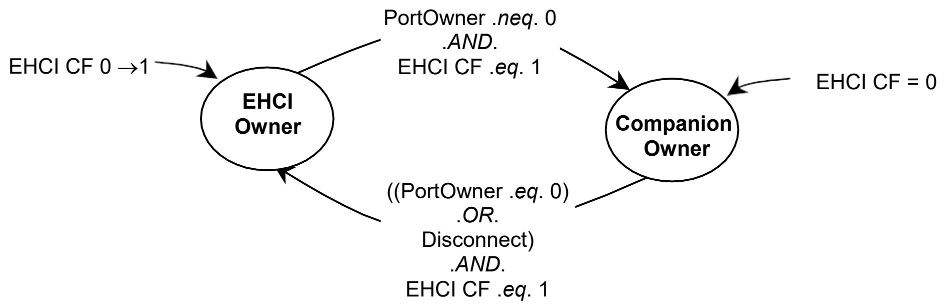


Figure 4-2. Port Owner Handoff State Machine

4.2.3.1 EHCI HC Owner

Entry to this state occurs whenever one of the following events occur:

- When the EHCI HC's *Configure Flag* (*CF*) bit in the CONFIGFLAG register transitions from a zero to a one. This signals the fact that the system has a host controller driver for the EHCI HC and that all ports in the USB 2.0 host controller must default route to the EHCI controller.
- When the port is owned by a companion HC and the device is disconnected from the port. The EHCI port routing control logic is notified of the the disconnect, and returns port routing to the EHCI controller. The connection state of the companion HC goes immediately to the disconnected state (with appropriate side effect to connect change, enable and enable change). The companion HC driver will acknowledge the disconnect by setting the connect status change bit to a zero. This allows the companion HC's driver to interact with the port completely through the disconnect process.
- When system software writes a zero to the *PortOwner* bit in the PORTSC register. This allows software to take ownership of a port from a companion host controller. When this occurs, the routing logic to the companion HC effectively signals a disconnect to the companion HC's port status and control register.

4.2.3.2 Companion HC Owner

Entry to this state occurs whenever one of the following events occur:

- When the *Port Owner* field transitions from a zero to a one.
- When the HS-mode HC's *Configure Flag* (*CF*) is equal to zero.

On entry to this state, the routing logic allows the companion HC port register to detect a device connect. Normal port enumeration proceeds.

4.2.4 Port Power

The *Port Power Control* (*PPC*) bit in the HCSPARAMS register indicates whether the USB 2.0 host controller has port power control (See Section 2.2.3). When this bit is a zero, then the host controller does not support software control of port power switches. When in this configuration, the port power is always available and the companion host controllers must implement functionality consistent with port power always on.

When the *PPC* bit is a one, then the host controller implementation includes port power switches. Each available switch has an output enable, which is referred to in this discussion as *PortPowerOutputEnable*

(*PPE*). *PPE* is controlled based on the state of the combination bits *PPC* bit, EHCI *Configured* (*CF*)-bit and individual *Port Power* (*PP*) bits. Table 4–3 illustrates the summary behavioral model.

Table 4–3. Port Power Enable Control Rules

CF	CHC[2] (PP)	EHC[3] (PP)	Owner	PPE[1]	Description
0	0	X	CHC	0	When the EHCI controller has not been configured, the port is owned by the companion host controller. When the companion HC's port power select is off, then the port power is off.
0	1	X	CHC	1	Similar to previous entry. When the companion HC's port power select is on, then the port power is on.
1	0	0	CHC	0	Port owner has port power turned off, the power to port is off.
1	0	0	EHC	0	Port owner has port power turned off, the power to port is off.
1	0	1	EHC	1	Port owner has port power on, so power to port is on.
1	0	1	CHC	1	If either HC has port power turned on, the power to the port is on.
1	1	0	EHC	1	If either HC has port power turned on, the power to the port is on.
1	1	0	CHC	1	Port owner has port power on, so power to port is on.
1	1	1	CHC	1	Port owner has port power on, so power to port is on.
1	1	1	EHC	1	Port owner has port power on, so power to port is on.

[1]PPE (Port Power Enable). This bit actually turns on the port power switch (if one exists).

[2]CHC (Companion Host Controller).

[3]EHC (EHCI Host Controller).

4.2.5 Port Reporting Over-Current

Host controllers are by definition power providers on USB. Whether the ports are considered high- or low-powered is a platform implementation issue. Each EHCI PORTSC register has an over-current status and over-current change bit. The functionality of these bits is specified in the USB Specification Revision 2.0.

The over current detection and limiting logic usually resides outside the host controller logic. This logic may be associated with one or more ports. When this logic detects an over-current condition it is made available to both the companion and EHCI ports. The effect of an over-current status on a companion host controller port is beyond the scope of this document. The over-current condition effects the following bits in the PORTSC register on the EHCI port:

- *Over-current Active* bits are set to a one. When the over-current condition goes away, the *Over-current Active* bit will transition from a one to a zero.
- *Over-current Change* bits are set to a one. On every transition of the *Over-current Active* bit the host controller will set the *Over-current Change* bit to a one. Software sets the *Over-current Change* bit to a zero by writing a one to this bit.
- *Port Enabled/Disabled* bit is set to a zero. When this change bit gets set to a one, then the *Port Change Detect* bit in the USBSTS register is set to a one.

- *Port Power (PP)* bits may optionally be set to a zero. There is no requirement in USB that a power provider shut off power in an over current condition. It is sufficient to *limit* the current and leave power applied.

When the *Over-current Change* bit transitions from a zero to a one, the host controller also sets the *Port Change Detect* bit in the USBSTS register to a one. In addition, if the *Port Change Interrupt Enable* bit in the USBINTR register is a one, then the host controller will issue an interrupt to the system. Refer to Table 4–4 for summary behavior for over-current detection when the host controller is halted (suspended from a device component point of view).

4.3 Suspend/Resume

The EHCI host controller provides an equivalent suspend and resume model as that defined for individual ports in a USB 2.0 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 wake-up events are:

- Remote-wakeup enabled device asserts resume signaling. In similar kind to USB 2.0 Hubs, EHCI controllers must 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* bit in the USBCMD register to a zero. The EHCI module can then be placed into a lower device state via the PCI power management interface (See Appendix A).

When a wake event occurs the system will resume operation and system software will eventually set the *Run/Stop* bit to a one and resume the suspended ports. Software must not set the *Run/Stop* bit to a one 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* bit in the USBCMD register can be set to a one. There are also minimum system software delays defined in the PCI Power Management Specification. Refer to this specification for more information.

4.3.1 Port Suspend/Resume

System software places individual ports into suspend mode by writing a one into the appropriate PORTSC *Suspend* bit. Software must only set the *Suspend* bit when the port is in the enabled state (*Port Enabled* bit is a one) and the EHCI is the port owner (*Port Owner* bit is a zero).

The host controller may evaluate the *Suspend* bit immediately or wait until a micro-frame 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 micro-frames of activity on the port until the host controller evaluates the *Suspend* bit. The host controller must evaluate the *Suspend* bit at least every frame boundary.

System software can initiate a resume on a selectively suspended port by writing a one to the *Force Port Resume* bit. Software should not attempt to resume a port unless the port reports that it is in the suspended state (see Section 2.3.9). If system software sets *Force Port Resume* bit to a one when the port is not in the suspended state, the resulting behavior is undefined. In order to assure proper USB device operation, software must wait for at least 10 milliseconds after a port indicates that it is suspended (*Suspend* bit is a one) before initiating a port resume via the *Force Port Resume* bit. When *Force Port Resume* bit is a one, the host controller sends resume signaling down the port. System software times the duration of the resume (nominally 20 milliseconds) then sets the *Force Port Resume* bit to a zero. When the host controller receives the write to transition *Force Port Resume* to zero, it completes the resume sequence as defined in the USB specification, and sets both the *Force Port Resume* and *Suspend* bits to zero. Software-initiated port resumes do not affect the *Port Change Detect* bit in the USBSTS register nor do they cause an interrupt if the *Port Change Interrupt Enable* bit in the USBINTR register is a one.

An external USB event may also initiate a resume. The wake events are defined above. When a wake event occurs on a suspended port, the resume signaling is detected by the port and the resume is reflected downstream within 100 µsec. The port's *Force Port Resume* bit is set to a one and the *Port Change Detect* bit in the USBSTS register is set to a one. If the *Port Change Interrupt Enable* bit in the USBINTR register is a one the host controller will issue a hardware interrupt.

System software observes the resume event on the port, delays a port resume time (nominally 20 msec), then terminates the resume sequence by writing zero to the *Force Port Resume* bit in the port. The host controller receives the write of zero to *Force Port Resume*, terminates the resume sequence and sets *Force Port Resume* and *Suspend* port bits to zero. Software can determine that the port is enabled (not suspended) by sampling the PORTSC register and observing that the *Suspend* and *Force Port Resume* bits are zero.

Software must ensure that the host controller is running (i.e. *HCHalted* bit in the USBSTS register is a zero), before terminating a resume by writing a zero to a port's *Force Port Resume* bit. If *HCHalted* is a one when *Force Port Resume* is set to a zero, then SOFs will not occur down the enabled port and the device will return to suspend mode in a maximum of 10 milliseconds.

Table 4–4 summarizes the wake-up events. Whenever a resume event is detected, the *Port Change Detect* bit in the USBSTS register is set to a one. If the *Port Change Interrupt Enable* bit is a one in the USBINTR register, the host controller will also generate an interrupt on the resume event. Software acknowledges the resume event interrupt by clearing the *Port Change Detect* status bit in the USBSTS register.

Table 4–4. Behavior During Wake-up Events

Port Status and Signaling Type	Signaled Port Response	Device State	
		D0	not D0
Port disabled, resume K-State received	No Effect	N/A	N/A
Port suspended, Resume K-State received	Resume reflected downstream on signaled port. Force Port Resume status bit in PORTSC register is set to a one. Port Change Detect bit in USBSTS register set to a one.	[1], [2]	[2]
Port is enabled, disabled or suspended, and the port's WKDSCNNT_E bit is a one. A disconnect is detected.	Depending in the initial port state, the PORTSC Connect and Enable status bits are set to zero, and the Connect Change status bit is set to a one. Port Change Detect bit in the USBSTS register is set to a one.	[1], [2]	[2]
Port is enabled, disabled or suspended, and the port's WKDSCNNT_E bit is a zero. A disconnect is detected.	Depending on the initial port state, the PORTSC Connect and Enable status bits are set to zero, and the Connect Change status bit is set to a one. Port Change Detect bit in the USBSTS register is set to a one.	[1], [3]	[3]
Port is not connected and the port's WKCNNT_E bit is a one. A connect is detected.	PORTSC Connect Status and Connect Status Change bits are set to a one. Port Change Detect bit in the USBSTS register is set to a one.	[1], [2]	[2]
Port is not connected and the port's WKCNNT_E bit is a zero. A connect is detected.	PORTSC Connect Status and Connect Status Change bits are set to a one. Port Change Detect bit in the USBSTS register is set to a one.	[1], [3]	[3]

Table 4–4. Behavior During Wake-up Events (cont.)

Port Status and Signaling Type	Signaled Port Response	Device State	
		D0	Not D0
Port is connected and the port's WKOC_E bit is a one. An over-current condition occurs.	PORTSC Over-current Active, Over-current Change bits are set to a one. If Port Enable/Disable bit is a one, it is set to a zero. Port Change Detect bit in the USBSTS register is set to a one	[1], [2]	[2]
Port is connected and the port's WKOC_E bit is a zero. An over-current condition occurs.	PORTSC Over-current Active, Over-current Change bits are set to a one. If Port Enable/Disable bit is a one, it is set to a zero. Port Change Detect bit in the USBSTS register is set to a one.	[1], [3]	[3]

[1] Hardware interrupt issued if Port Change Interrupt Enable bit in the USBINTR register is a one.
[2] PME# asserted if enabled (Note: PME Status must always be set to a one).
[3] PME# not asserted.

4.4 Schedule Traversal Rules

The host controller executes transactions for devices using a simple, shared-memory schedule. The schedule is comprised of a few data structures, organized into two distinct lists. The data structures are designed to provide the maximum flexibility required by USB, minimize memory traffic and hardware / software complexity.

System software maintains two schedules for the host controller: a periodic schedule and an asynchronous schedule. The root of the periodic schedule is the *PERIODICLISTBASE* register (see Section 2.3.6). The *PERIODICLISTBASE* register is the physical memory base address of the periodic frame list. The periodic frame list is an array of physical memory pointers. The objects referenced from the frame list must be valid schedule data structures as defined in Section 3. In each micro-frame, if the periodic schedule is enabled (see Section 4.6) then the host controller must execute from the periodic schedule before executing from the asynchronous schedule. It will only execute from the asynchronous schedule after it encounters the end of the periodic schedule. The host controller traverses the periodic schedule by constructing an array offset reference from the *PERIODICLISTBASE* and the *FRINDEX* registers (see Figure 4-3). It fetches the element and begins traversing the graph of linked schedule data structures.

The end of the periodic schedule is identified by a *next* link pointer of a schedule data structure having its *T-bit* set to a one. When the host controller encounters a *T-Bit* set to a one during a horizontal traversal of the periodic list, it interprets this as an End-Of-Periodic-List mark. This causes the host controller to cease working on the periodic schedule and transitions immediately to traversing the asynchronous schedule. Once this transition is made, the host controller executes from the asynchronous schedule until the end of the micro-frame.

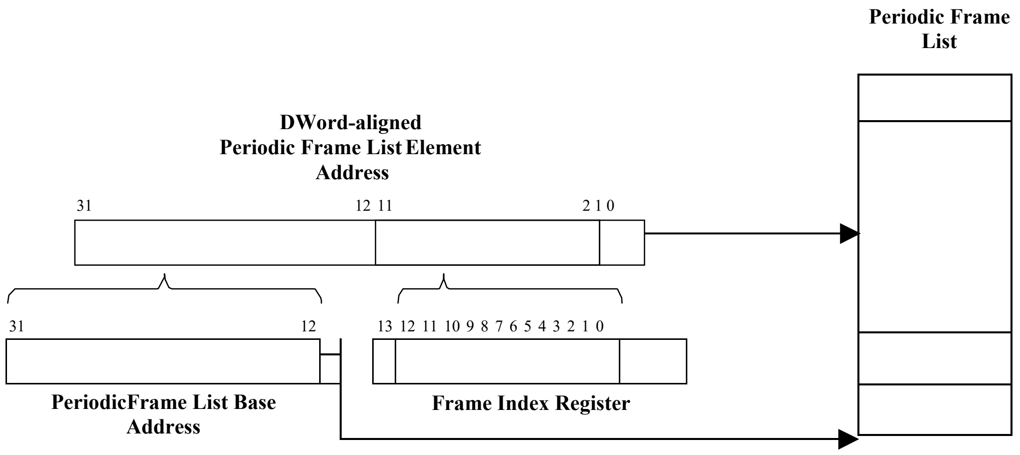


Figure 4-3. Derivation of Pointer into Frame List Array

When the host controller determines that it is time to execute from the asynchronous list, it uses the operational register *ASYNCLISTADDR* to access the asynchronous schedule, see Figure 4-4.

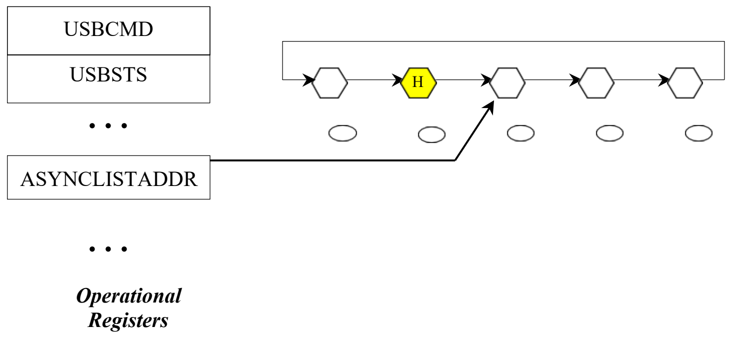


Figure 4-4. General Format of Asynchronous Schedule List

The *ASYNCLISTADDR* register contains a physical memory pointer to the *next* queue head. When the host controller makes a transition to executing the asynchronous schedule, it begins by reading the queue head referenced by the *ASYNCLISTADDR* register. Software must set queue head *horizontal* pointer *T-bits* to a zero for queue heads in the asynchronous schedule. See Section 4.8 for complete operational details.

4.4.1 Example - Preserving Micro-Frame Integrity

One of the requirements of a USB host controller is to maintain *Frame Integrity*. This means that the HC must preserve the micro-frame boundaries. For example: SOF packets must be generated on time (within the specified allowable jitter), and High-speed EOF1,2 thresholds must be enforced. The end of micro-frame timing points EOF1 and EOF2 are clearly defined in the USB Specification Revision 2.0.

One implication of this responsibility is that the HC must ensure that it does not start transactions that will not be completed before the end of the micro-frame. More precisely, no transactions should be started by the host controller, which cannot be completed in their entirety before the EOF1 point. In order to enforce this rule, the host controller must check each transaction before it starts to ensure that it will complete before the end of the micro-frame.

So, what exactly needs to be involved in this check? Fundamentally, the transaction data payload, plus bit stuffing, plus transaction overhead must be taken into consideration. It is possible to be extremely accurate on how much time the next transaction will take. Take OUTs for an example. The host controller must fetch all of the OUT data from memory in order to send it onto the USB bus. A host controller implementation could pre-fetch *all* of the OUT data, and pre-compute the actual number of bits in the token and data packets. In addition, the system knows the depth of the target endpoint, so it could closely estimate turnaround time for handshake. In addition, the host controller knows the size of a handshake packet. Pre-computing effects of bit stuffing and summing up the other overhead numbers could allow the host controller to know exactly whether there was enough bus time, before EOF1 to complete the OUT transaction. To accomplish this particular approach takes an inordinate amount of time and hardware complexity.

The alternative is to make a reasonable guess whether the next transaction can be started. An example approximation algorithm is described below. This example algorithm relies on the EHCI policy that periodic transactions are scheduled first in the micro-frame. It is a reasonable assumption that software will never over-commit the micro-frame to periodic transactions greater than the specification allowable 80%. In the available remaining 20% bandwidth, the host controller has some ability (in this example) to decide whether or not to execute a transaction. The result of this algorithm is that sometimes, under some circumstances a transaction will not be executed that *could* have been executed. However, under all circumstances, a transaction will never be started unless there is enough time in the frame to complete the transaction.

4.4.1.1 Transaction Fit - A Best-Fit Approximation Algorithm

A curve is calculated which represents the *latest* start time for every packet size, at which software will schedule the start of a periodic transaction. This curve is the 80% bandwidth curve. Another curve is calculated which is the absolute, latest permitted start time for every packet size. This curve represents the absolute latest time, that a transaction of each packet size can be started and completed, in the micro-frame. A plot of these two curves is illustrated in Figure 4-5. The plot Y-axis represents the number of byte-times left in a frame.

The space between the *80%* and the *Last Start* plots is bandwidth reclamation area. In this algorithm the host controller may skip transactions during this time if it is prudent.

The Best-Fit Approximation method plots a function (*f*(*x*)) between the 80% and *Last Start* curves. The function *f*(*x*) adds a constant to every transaction's maximum packet size and the result compared with the number of bytes left in the frame. The constant represents an approximation of the effects of bit stuffing and protocol overhead. The host controller starts transactions whose results land *above* the function curve. The host controller will not start transactions whose results land below the function curve.

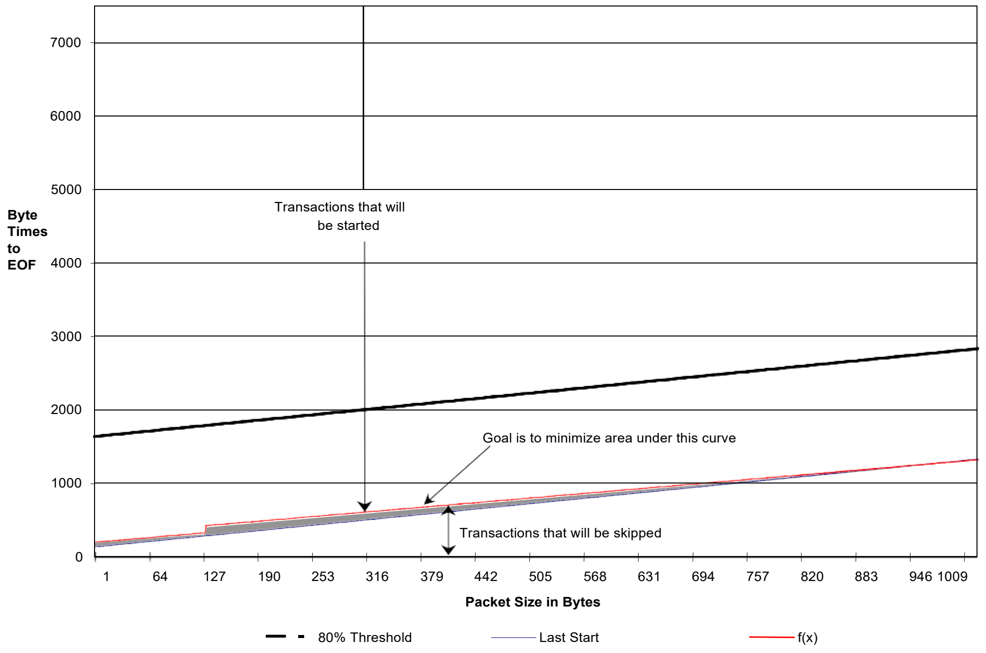


Figure 4-5. Best Fit Approximation

The *LastStart* line was calculated in this example to assume the absolute worst-case bus overhead per transaction. The particular transaction used was a start-split, zero-length OUT transaction with a handshake. A summary of the component parts are listed in Table 4–5. The component times were derived from the protocol timings defined in the USB Specification Revision 2.0.

Table 4–5. Example Worse-case Transaction Timing Components

Component	Bit time	Byte Times	Explanation
Split Token	76	9.5	Split token as defined in USB core specification. Includes sync, token, eop, etc.
Host 2 Host IPG	88	11	Number of bit times required between consecutive host packets
Token	67	8.375	Token as defined in USB core specification. Includes sync, token, eop, etc.
Host 2 Host IPG	88	11	Same as above
Data Packet (0 data bytes)	66.7	8.34	Zero-length data packet. Includes sync, PID, crc16, eop, etc.
Turnaround time	721	90.125	Time for packet initiator (Host) to see the beginning of a response to a transmitted packet.
Handshake packet	48	6	Handshake packet as defined in USB core specification. Includes sync, PID, eop, etc.
		144	**Total**

The exact details of the function (f(x)) are up to the particular implementation. However, it should be obvious that the goal is to minimize the area under the curve between the approximation function and the *Last Start* curve, without dipping below the *LastStart* line, while at the same time keeping the check as simple as possible for hardware implementation. The f(x) in Figure 4-5 was constructed using the following pseudo-code test on each transaction size data point. This algorithm assumes that the host controller keeps track of the remaining bits in the frame.

```
Alorithm CheckTransactionWillFit (MaximumPacketSize, HC_BytesLeftInFrame)
Begin
   Local Temp   = MaximumPacketSize + 192
   Local rvalue = TRUE

   If MaximumPacketSize >= 128 then
      Temp += 128
   End If

   If Temp > HC_BytesLeftInFrame then
      Rvalue = FALSE
   End If
   Return rvalue
End
```

This algorithm takes two inputs, the current maximum packet size of the transaction and a hardware counter of the number of bytes left in the current micro-frame. It unconditionally adds a simple constant of 192 to the maximum packet size to account for a first-order effect of transaction overhead and bit stuffing. If the transaction size is greater than or equal to 128 bytes, then an additional constant of 128 is added to the running sum to account for the additional worst-case bit stuffing of payloads larger than 128. An inflection point was inserted at 128 because the *f*(*x*) plot was getting *close* to the *LastStart* line.

4.5 Periodic Schedule Frame Boundaries vs Bus Frame Boundaries

The USB Specification Revision 2.0 requires that the frame boundaries (SOF frame number changes) of the high-speed bus and the full- and low-speed bus(s) below USB 2.0 Hubs be strictly aligned. Super-imposed on this requirement is that USB 2.0 Hubs manage full- and low-speed transactions via a micro-frame pipeline (see start- (SS) and complete- (CS) splits illustrated in Figure 4-6). A simple, direct projection of the frame boundary model into the host controller interface schedule architecture creates tension (complexity for both hardware and software) between the frame boundaries and the scheduling mechanisms required to service the full- and low-speed transaction translator periodic pipelines.

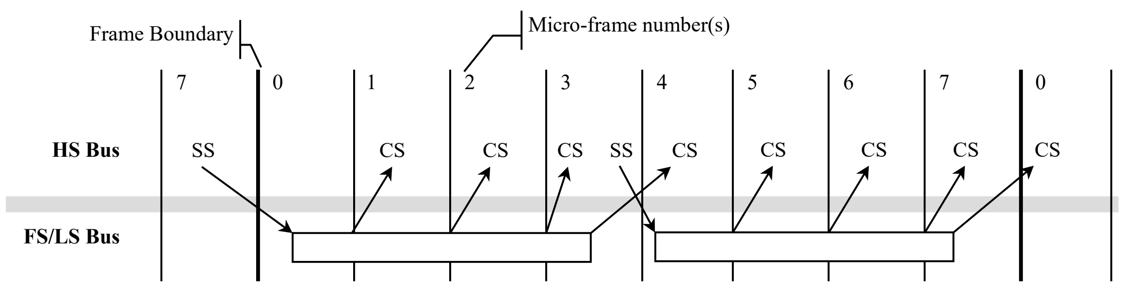


Figure 4-6. Frame Boundary Relationship between HS bus and FS/LS Bus

The simple projection, as Figure 4-6 illustrates, introduces frame-boundary wrap conditions for scheduling on both the beginning and end of a frame. In order to reduce the complexity for hardware and software, the host controller is required to implement a one micro-frame phase shift for its view of frame boundaries. The phase shift eliminates the beginning of frame and frame-wrap scheduling boundary conditions.

The implementation of this phase shift requires that the host controller use one register value for accessing the periodic frame list and another value for the frame number value included in the SOF token. These two values are separate, but tightly coupled. The periodic frame list is accessed via the Frame List Index Register (FRINDEX) documented in Section 2.3.4 and initially illustrated in Section 4.4. Bits FRINDEX[2:0], represent the micro-frame number. The SOF value is coupled to the value of FRINDEX[13:3]. Both FRINDEX[13:3] and the SOF value are incremented based on FRINDEX[2:0]. It is required that the SOF value be delayed from the FRINDEX value by one micro-frame. The one micro-frame delay yields host controller periodic schedule and bus frame boundary relationship as illustrated in Figure 4-7. This adjustment allows software to trivially schedule the periodic start and complete-split transactions for full- and low-speed periodic endpoints, using the natural alignment of the periodic schedule interface. The reasons for selecting this phase-shift are beyond the scope of this specification.

Figure 4-7 illustrates how periodic schedule data structures relate to schedule frame boundaries and bus frame boundaries. To aid the presentation, two terms are defined. The host controller's view of the 1-millisecond boundaries is called *H-Frames*. The high-speed bus's view of the 1-millisecond boundaries is called *B-Frames*.

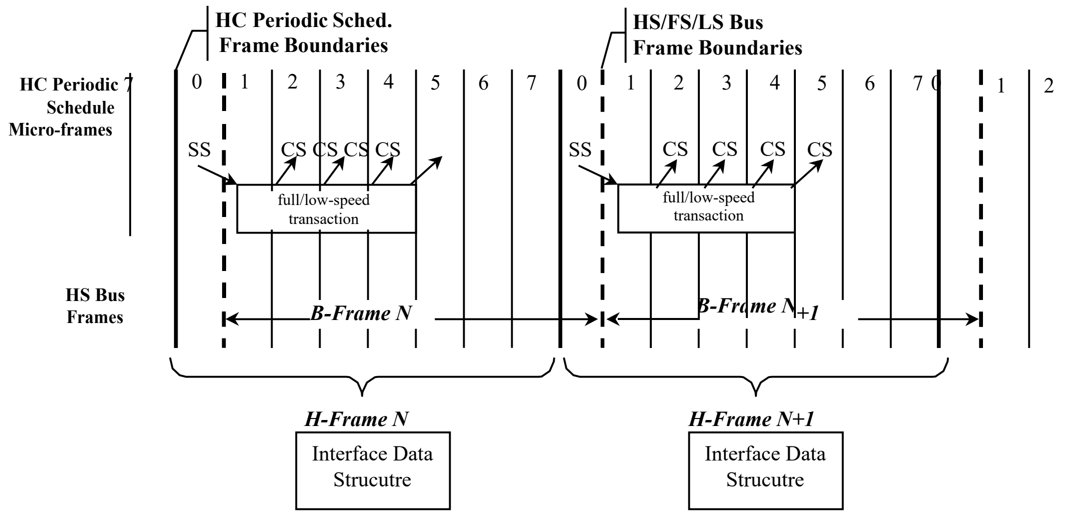

HC Periodic Sched. Frame Boundaries
HS/FS/LS Bus Frame Boundaries
HC Periodic Schedule Micro-frames
HS Bus Frames
SS
CS CS CS CS
SS
CS CS CS CS
full/low-speed transaction
full/low-speed transaction
B-Frame N
B-Frame N+1
H-Frame N
H-Frame N+1
Interface Data Strucutre
Interface Data Strucutre

Figure 4-7. Relationship of Periodic Schedule Frame Boundaries to Bus Frame Boundaries

H-Frame boundaries for the host controller correspond to increments of FRINDEX[13:3]. Micro-frame numbers for the *H-Frame* are tracked by FRINDEX[2:0]. *B-Frame* boundaries are visible on the high-speed bus via changes in the SOF token's frame number. Micro-frame numbers on the high-speed bus are only derived from the SOF token's frame number (i.e. the high-speed bus will see eight SOFs with the same frame number value). *H-Frames* and *B-Frames* have the fixed relationship (i.e. *B-Frames* lag *H-Frames* by one micro-frame time) illustrated in Figure 4-7.

The host controller's periodic schedule is naturally aligned to *H-Frames*. Software schedules transactions for full- and low-speed periodic endpoints relative the *H-Frames*. The result is these transactions execute on the high-speed bus at exactly the right time for the USB 2.0 Hub periodic pipeline.

As described in Section 2.3.4, the SOF Value can be implemented as a shadow register (in this example, called SOFV), which lags the FRINDEX register bits [13:3] by one micro-frame count. Table 4–6 illustrates the required relationship between the value of FRINDEX and the value of SOFV. This lag behavior can be accomplished by incrementing FRINDEX[13:3] based on carry-out on the 7 to 0 increment of FRINDEX[2:0] and incrementing SOFV based on the transition of 0 to 1 of FRINDEX[2:0].

Software is allowed to write to FRINDEX. Section 2.3.4 provides the requirements that software should adhere when writing a new value in FRINDEX.

Table 4–6. Operation of FRINDEX and SOFV (SOF Value Register)

Current			Next		
FRINDEX[F]	**SOFV**	**FRINDEX[μF]**	**FRINDEX[F]**	**SOFV**	**FRINDEX[μF]**
N	N	111b	N+1	N	000b
N+1	N	000b	N+1	N+1	001b
N+1	N+1	001b	N+1	N+1	010b
N+1	N+1	010b	N+1	N+1	011b
N+1	N+1	011b	N+1	N+1	100b
N+1	N+1	100b	N+1	N+1	101b
N+1	N+1	101b	N+1	N+1	110b
N+1	N+1	110b	N+1	N+1	111b

Where [F] = [13:3]; [μF] = [2:0]

4.6 Periodic Schedule

The periodic schedule traversal is enabled or disabled via the *Periodic Schedule Enable* bit in the USBCMD register. If the *Periodic Schedule Enable* bit is set to a zero, then the host controller simply does not try to access the periodic frame list via the *PERIODICLISTBASE* register. Likewise, when the *Periodic Schedule Enable* bit is a one, then the host controller does use the *PERIODICLISTBASE* register to traverse the periodic schedule. The host controller will not react to modifications to the *Periodic Schedule Enable* immediately. In order to eliminate conflicts with split transactions, the host controller evaluates the *Periodic Schedule Enable* bit only when FRINDEX[2:0] is zero. System software must not disable the periodic schedule if the schedule contains an active split transaction work item that spans the 000b micro-frame. These work items must be removed from the schedule before the *Periodic Schedule Enable* bit is written to a zero.

The *Periodic Schedule Status* bit in the USBSTS register indicates status of the periodic schedule. System software enables (or disables) the periodic schedule by writing a one (or zero) to the *Periodic Schedule Enable* bit in the USBCMD register. Software then can poll the *Periodic Schedule Status* bit to determine when the periodic schedule has made the desired transition. Software must not modify the *Periodic Schedule Enable* bit unless the value of the *Periodic Schedule Enable* bit equals that of the *Periodic Schedule Status* bit.

The periodic schedule is used to manage all isochronous and interrupt transfer streams. The base of the periodic schedule is the periodic frame list. Software links schedule data structures to the periodic frame list to produce a graph of scheduled data structures. The graph represents an appropriate sequence of transactions on the USB. Figure 4-8 illustrates isochronous transfers (using iTDs and siTDs) with a period of one are linked directly to the periodic frame list. Interrupt transfers (are managed with queue heads)and isochronous streams with periods other than one are linked following the period-one iTD/siTDs. Interrupt queue heads are linked into the frame list ordered by poll rate. Longer poll rates are linked first (e.g. closest to the periodic frame list), followed by shorter poll rates, with queue heads with a poll rate of one, on the very end.

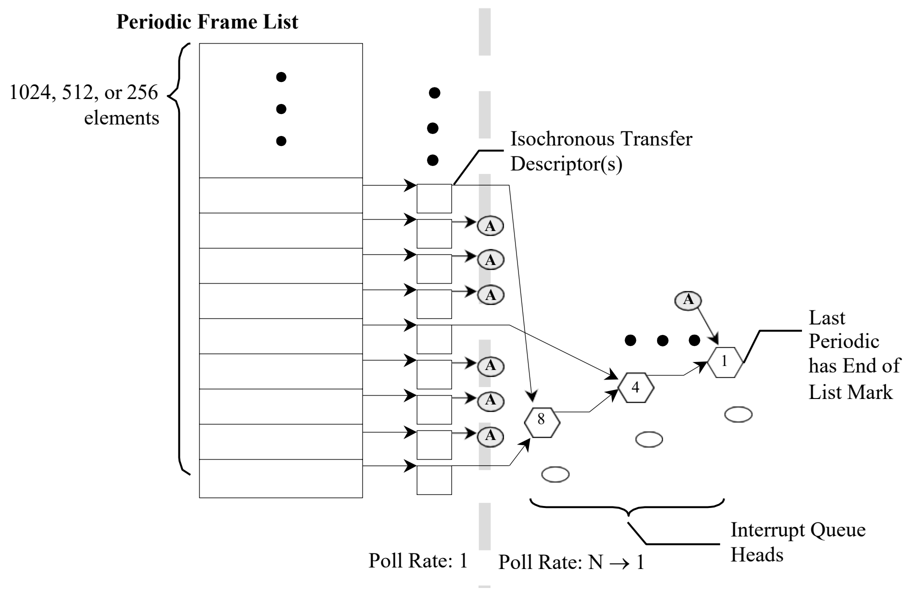

Periodic Frame List
1024, 512, or 256 elements
Isochronous Transfer Descriptor(s)
A
Last Periodic has End of List Mark
8
4
1
Interrupt Queue Heads
Poll Rate: 1
Poll Rate: N → 1

Figure 4-8. Example Periodic Schedule

4.7 Managing Isochronous Transfers Using iTDs

The structure of an iTD is presented in 3.3. There are four distinct sections to an iTD:

- The first field is the *Next Link Pointer*. This field is for schedule linkage purposes only;
- Transaction description array. This area is an eight-element array. Each element represents control and status information for one micro-frame's worth of transactions for a single high-speed isochronous endpoint.
- The buffer page pointer array is a 7-element array of physical memory pointers to data buffers. These are 4K aligned pointers to physical memory.
- Endpoint capabilities. This area utilizes the unused low-order 12 bits of the buffer page pointer array. The fields in this area are used across all transactions executed for this iTD, including endpoint addressing, transfer direction, maximum packet size and high-bandwidth multiplier.

4.7.1 Host Controller Operational Model for iTDs

The host controller uses FRINDEX register bits [12:3] to index into the periodic frame list. This means that the host controller visits each frame list element eight consecutive times before incrementing to the next periodic frame list element. Each iTD contains eight transaction descriptions, which map directly to FRINDEX register bits [2:0]. Each iTD can span 8 micro-frames worth of transactions.

When the host controller fetches an iTD, it uses FRINDEX register bits [2:0] to index into the transaction description array. If the *active* bit in the *Status* field of the indexed transaction description is set to zero, the host controller ignores the iTD and follows the *Next* pointer to the next schedule data structure.

When the indexed *active* bit is a one the host controller continues to parse the iTD. It stores the indexed transaction description and the general endpoint information (device address, endpoint number, maximum packet size, etc.). It also uses the *Page Select (PG)* field to index the *buffer pointer* array, storing the selected *buffer pointer* and the next sequential *buffer pointer*. For example, if *PG* field is a 0, then the host controller will store Page 0 and Page 1.

The host controller constructs a physical data buffer address by concatenating the current buffer pointer (as selected using the current transaction description's *PG* field) and the transaction description's *Transaction Offset* field. The host controller uses the endpoint addressing information and *I/O-bit* to execute a transaction to the appropriate endpoint. When the transaction is complete, the host controller clears the active bit and writes back any additional status information to the *Status* field in the currently selected transaction description.

The data buffer associated with the iTD must be virtually contiguous memory. Seven page pointers are provided to support eight high-bandwidth transactions regardless of the starting packet's offset alignment into the first page. A starting buffer pointer (physical memory address) is constructed by concatenating the page pointer (example: page 0 pointer) selected by the active transaction descriptions' *PG* (example value: 00B) field with the transaction offset field. As the transaction moves data, the host controller must detect when an increment of the current buffer pointer will cross a page boundary. When this occurs the host controller simply replaces the current buffer pointer's page portion with the next page pointer (example: page 1 pointer) and continues to move data. The size of each bus transaction is determined by the value in the *Maximum Packet Size* field.

An iTD supports high-bandwidth pipes via the *Mult* (multiplier) field. When the *Mult* field is 1, 2, or 3, the host controller executes the specified number of *Maximum Packet* sized bus transactions for the endpoint in the current micro-frame. In other words, the *Mult* field represents a transaction count for the endpoint in the current micro-frame. If the *Mult* field is zero, the operation of the host controller is undefined. The transfer description is used to service all transactions indicated by the *Mult* field.

For OUT transfers, the value of the *Transaction X Length* field represents the total bytes to be sent during the micro-frame. The *Mult* field must be set by software to be consistent with *Transaction X Length* and *Maximum Packet Sixe.* The host controller will send the bytes in *Maximum Packet Size*'d portions. After each transaction, the host controller decrements it's local copy of *Transaction X Length* by *Maximum Packet Size*. The number of bytes the host controller sends is always *Maximum Packet Size* or *Transaction X Length*, whichever is less. The host controller advances the transfer state in the transfer description, updates the appropriate record in the iTD and moves to the next schedule data structure. The maximum sized transaction supported is 3 x 1024 bytes.

For IN transfers, the host controller issues *Mult* transactions. It is assumed that software has properly initialized the iTD to accommodate all of the possible data. During each IN transaction, the host controller must use *Maximum Packet Size* to detect packet babble errors. The host controller keeps the sum of bytes received in the *Transaction X Length* field. After all transactions for the endpoint have completed for the micro-frame, *Transaction X Length* contains the total bytes received. If the final value of *Transaction X Length* is less than the value of *Maximum Packet Size*, then less data than was allowed for was received from the associated endpoint. This *short packet* condition does not set the *USBINT* bit in the USBSTS register to a one. The host controller will not detect this condition. If the device sends more than *Transaction X Length* or *Maximum Packet Size* bytes (whichever is less), then the host controller will set the *Babble Detected* bit to a one and set the *Active* bit to a zero. Note, that the host controller is not required to update the iTD field *Transaction X Length* in this error scenario.

If the *Mult* field is greater than one, then the host controller will automatically execute the value of *Mult* transactions. The host controller will not execute all *Mult* transactions if:

- The endpoint is an OUT and *Transaction X Length* goes to zero before all the *Mult* transactions 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 *Mult* transactions have been executed.

The end of micro-frame may occur before all of the transaction opportunities have been executed. When this happens, the transfer state of the transfer description is advanced to reflect the progress that was made, the result written back to the iTD and the host controller proceeds to processing the next micro-frame. Refer to

Appendix D for a table summary of the host controller required behavior for all the high-bandwidth transaction cases.

4.7.2 Software Operational Model for iTDs

A client buffer request to an isochronous endpoint may span 1 to N micro-frames. When N is larger than one, system software may have to use multiple iTDs to read or write data with the buffer (if N is larger than eight, it must use more than one iTD).

Figure 4-9 illustrates the simple model of how a client buffer is mapped by system software to the periodic schedule (i.e. the periodic frame list and a set of iTDs). On the right is the client description of its request. The description includes a buffer base address plus additional annotations to identify which portions of the buffer should be used with each bus transaction. In the middle is the iTD data structures used by the system software to service the client request. Each iTD can be initialized to service up to 24 transactions, organized into eight groups of up to three transactions each. Each group maps to one micro-frame's worth of transactions. The EHCI controller does not provide per-transaction results within a micro-frame. It treats the per-micro-frame transactions as a single logical transfer. On the left is the host controller's frame list. System software establishes references from the appropriate locations in the frame list to each of the appropriate iTDs.

If the buffer is large, then system software can use a small set of iTDs to service the entire buffer. System software can activate the transaction description records (contained in each iTD) in any pattern required for the particular data stream.

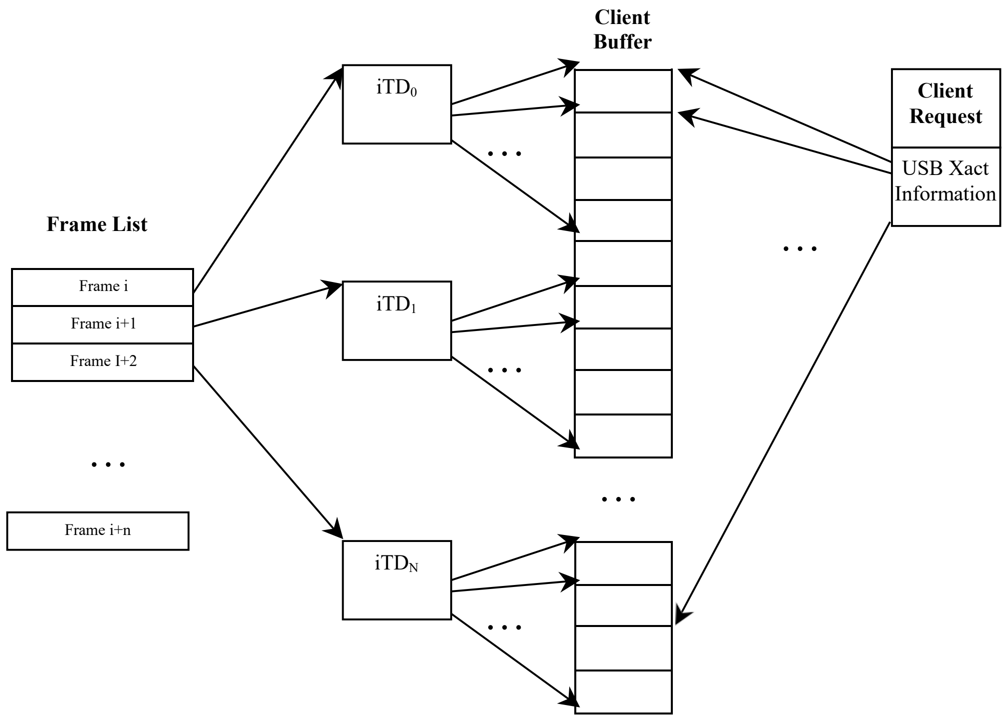


Figure 4-9. Example Association of iTDs to Client Request Buffer

As noted above, the client request includes a pointer to the base of the buffer and offsets into the buffer to annotate which buffer sections are to be used on each bus transaction that occurs on this endpoint. System software must initialize each transaction description in an iTD to ensure it uses the correct portion of the client buffer. For example, for each transaction description, the *PG* field is set to index the correct physical buffer page pointer and the *Transaction Offset* field is set relative to the correct buffer pointer page (e.g. the same one referenced by the *PG* field). When the host controller executes a transaction it selects a transaction description record based on FRINDEX[2:0]. It then uses the current *Page Buffer Pointer* (as selected by the

PG field) and concatenates to the *transaction offset* field. The result is a starting buffer address for the transaction. As the host controller moves data for the transaction, it must watch for a page wrap condition and properly advance to the next available *Page Buffer Pointer*.

System software must not use the Page 6 buffer pointer in a transaction description where the length of the transfer will wrap a page boundary. Doing so will yield undefined behavior. The host controller hardware is not required to 'alias' the page selector to page zero.

USB 2.0 isochronous endpoints can specify a period greater than one. Software can achieve the appropriate scheduling by linking iTDs into the appropriate frames (relative to the frame list) and by setting appropriate transaction description elements active bits to a one.

4.7.2.1 Periodic Scheduling Threshold

The *Isochronous Scheduling Threshold* field in the HCCPARAMS capability register is an indicator to system software as to how the host controller pre-fetches and effectively caches schedule data structures. It is used by system software when adding isochronous work items to the periodic schedule. The value of this field indicates to system software the minimum distance it can update isochronous data (relative to the current location of the host controller execution in the periodic list) and still have the host controller process them.

The iTD and siTD data structures each describe 8 micro-frames worth of transactions. The host controller is allowed to cache one (or more) of these data structures in order to reduce memory traffic. There are three basic caching models that account for the fact the isochronous data structures span 8 micro-frames. The three caching models are: no caching, micro-frame caching and frame caching.

When software is adding new isochronous transactions to the schedule, it always performs a read of the FRINDEX register to determine the current frame and micro-frame the host controller is currently executing. Of course, there is no information about where in the micro-frame the host controller is, so a constant uncertainty-factor of one micro-frame has to be assumed. Combining the knowledge of where the host controller is executing with the knowledge of the caching model allows the definition of simple algorithms for how closely software can reliably work to the executing host controller.

No caching is indicated with a value of zero in the *Isochronous Scheduling Threshold* field. The host controller may pre-fetch data structures during a periodic schedule traversal (per micro-frame) but will always dump any accumulated schedule state at the end of the micro-frame. At the appropriate time relative to the beginning of every micro-frame, the host controller always begins schedule traversal from the frame list. Software can use the value of the FRINDEX register (plus the constant 1 uncertainty-factor) to determine the approximate position of the executing host controller. When no caching is selected, software can add an isochronous transaction as near as 2 micro-frames in front of the current executing position of the host controller.

Frame caching is indicated with a non-zero value in bit [7] of the *Isochronous Scheduling Threshold* field. In the frame-caching model, system software assumes that the host controller caches one (or more) isochronous data structures for an entire frame (8 micro-frames). Software uses the value of the FRINDEX register (plus the constant 1 uncertainty) to determine the *current* micro-frame/frame (assume modulo 8 arithmetic in adding the constant 1 to the micro-frame number). For any current frame N, if the current micro-frame is 0 to 6, then software can safely add isochronous transactions to Frame N + 1. If the current micro-frame is 7, then software can add isochronous transactions to Frame N + 2.

Micro-frame caching is indicated with a non-zero value in the least-significant 3 bits of the *Isochronous Scheduling Threshold* field. System software assumes the host controller caches one or more periodic data structures for the number of micro-frames indicated in the *Isochronous Scheduling Threshold* field. For example, if the count value were 2, then the host controller keeps a *window* of 2 micro-frames worth of state (current micro-frame, plus the next) on-chip. On each micro-frame boundary, the host controller releases the current micro-frame state and begins accumulating the next micro-frame state.

4.8 Asynchronous Schedule

The Asynchronous schedule traversal is enabled or disabled via the *Asynchronous Schedule Enable* bit in the USBCMD register. If the *Asynchronous Schedule Enable* bit is set to a zero, then the host controller simply does not try to access the asynchronous schedule via the *ASYNCLISTADDR* register. Likewise, when the *Asynchronous Schedule Enable* bit is a one, then the host controller does use the *ASYNCLISTADDR* register to traverse the asynchronous schedule. Modifications to the *Asynchronous Schedule Enable* bit are not necessarily immediate. Rather the new value of the bit will only be taken into consideration the next time the host controller needs to use the value of the *ASYNCLISTADDR* register to get the next queue head.

The *Asynchronous Schedule Status* bit in the USBSTS register indicates status of the asynchronous schedule. System software enables (or disables) the asynchronous schedule by writing a one (or zero) to the *Asynchronous Schedule Enable* bit in the USBCMD register. Software then can poll the *Asynchronous Schedule Status* bit to determine when the asynchronous schedule has made the desired transition. Software must not modify the *Asynchronous Schedule Enable* bit unless the value of the *Asynchronous Schedule Enable* bit equals that of the *Asynchronous Schedule Status* bit.

The asynchronous schedule is used to manage all Control and Bulk transfers. Control and Bulk transfers are managed using queue head data structures. The asynchronous schedule is based at the *ASYNCLISTADDR* register. The default value of the *ASYNCLISTADDR* register after reset is undefined and the schedule is disabled when the *Asynchronous Schedule Enable* bit is a zero.

Software may only write this register with defined results when the schedule is disabled .e.g. *Asynchronous Schedule Enable* bit in the USBCMD and the *Asynchronous Schedule Status* bit in the USBSTS register are zero. System software enables execution from the asynchronous schedule by writing a valid memory address (of a queue head) into this register. Then software enables the asychronous schedule by setting the *Asynchronous Schedule Enable* bit is set to one. The asynchronous schedule is actually enabled when the *Asynchronous Schedule Status* bit is a one.

When the host controller begins servicing the asynchronous schedule, it begins by using the value of the *ASYNCLISTADDR* register. It reads the first referenced data structure and begins executing transactions and traversing the linked list as appropriate. When the host controller "completes" processing the asynchronous schedule, it retains the value of the last accessed queue head's horizontal pointer in the *ASYNCLISTADDR* register. Next time the asynchronous schedule is accessed, this is the first data structure that will be serviced. This provides round-robin fairness for processing the asynchronous schedule.

A host controller "completes" processing the asynchronous schedule when one of the following events occur:

- The end of a micro-frame occurs.
- The host controller detects an empty list condition (i.e. see Section 4.8.3)
- The schedule has been disabled via the *Asynchronous Schedule Enable* bit in the USBCMD register.

The queue heads in the asynchronous list are linked into a simple circular list as shown in Figure 4-4. Queue head data structures are the only valid data structures that may be linked into the asynchronous schedule. An isochronous transfer descriptor (iTD or siTD) in the asynchronous schedule yields undefined results.

The maximum packet size field in a queue head is sized to accommodate the use of this data structure for all non-isochronous transfer types. The USB Specification, Revision 2.0 specifies the maximum packet sizes for all transfer types and transfer speeds. System software should always parameterize the queue head data structures according to the core specification requirements.

4.8.1 Adding Queue Heads to Asynchronous Schedule

This is a software requirement section. There are two independent events for adding queue heads to the asynchronous schedule. The first is the initial activation of the asynchronous list. The second is inserting a new queue head into an activated asynchronous list.

Activation of the list is simple. System software writes the physical memory address of a queue head into the *ASYNCLISTADDR* register, then enables the list by setting the *Asynchronous Schedule Enable* bit in the USBCMD register to a one.

When inserting a queue head into an active list, software must ensure that the schedule is always coherent from the host controllers' point of view. This means that the system software must ensure that all queue head pointer fields are valid. For example qTD pointers have *T-Bit*s set to a one or reference valid qTDs and the *Horizontal Pointer* references a valid queue head data structure. The following algorithm represents the functional requirements:

```
InsertQueueHead (pQHeadCurrent, pQueueHeadNew)
    --
    -- Requirement: all inputs must be properly initialized.
    --
    -- pQHeadCurrent is a pointer to a queue head that is already in the active list
    -- pQHeadNew is a pointer to the queue head to be added
    --
    -- This algorithm links a new queue head into a existing list
    --
    pQueueHeadNew.HorizontalPointer      = pQueueHeadCurrent.HorizontalPointer
    pQueueHeadCurrent.HorizontalPointer = physicalAddressOf(pQueueHeadNew)
End InsertQueueHead
```

4.8.2 Removing Queue Heads from Asynchronous Schedule

This is a software requirement section. There are two independent events for removing queue heads from the asynchronous schedule. The first is shutting down (deactivating) the asynchronous list. The second is extracting a single queue head from an activated list.

Software deactivates the asynchronous schedule by setting the *Asynchronous Schedule Enable* bit in the USBCMD register to a zero. Software can determine when the list is idle when the *Asynchronous Schedule Status* bit in the USBSTS register is a zero.

The normal mode of operation is that software removes queue heads from the asynchronous schedule without shutting it down. Software must not remove an active queue head from the schedule. Software should first deactivate all active qTDs, wait for the queue head to go inactive, then remove the queue head from the asynchronous list. Software removes a queue head from the asynchronous list via the following algorithm. As illustrated, the unlinking is quite easy. Software merely must ensure all of the link pointers reachable by the host controller are kept consistent.

```
UnlinkQueueHead (pQHeadPrevious, pQueueHeadToUnlink, pQHeadNext)
    --
    -- Requirement: all inputs must be properly initialized.
    --
    -- pQHeadPrevious is a pointer to a queue head that references the
    -- queue head to remove
    -- pQHeadToUnlink is a pointer to the queue head to be removed
    -- pQheadNext is a pointer to a queue head still in the schedule. Software
    -- provides this pointer with the following strict rules:
    --    if the host software is one queue head, then pQHeadNext must be the
    --    same as pQueueheadToUnlink.HorizontalPointer. If the host software is
    --    unlinking a consecutive series of queue heads, pQHeadNext must be
    --    set by software to the queue head remaining in the schedule.
    -- This algorithm unlinks a queue head from a circular list
    --
    pQueueHeadPrevious.HorizontalPointer   = pQueueHeadToUnlink.HorizontalPointer
    pQueueHeadToUnlink.HorizontalPointer   = pQHeadNext
End UnlinkQueueHead
```

If software removes the queue head with the *H-bit* set to a one, it must select another queue head still linked into the schedule and set its *H-bit* to a one. This should be completed before removing the queue head. The requirement is that software keep one queue head in the asynchronous schedule, with its *H-bit* set to a one.

At the point software has removed one or more queue heads from the asynchronous schedule, it is unknown whether the host controller has a cached pointer to them. Similarly, it is unknown how long the host controller might retain the cached information, as it is implementation dependent and may be affected by the actual dynamics of the schedule load. Therefore, once software has removed a queue head from the asynchronous list, it must retain the coherency of the queue head (link pointers, etc.). It cannot disturb the removed queue heads until it knows that the host controller does not have a local copy of a pointer to any of the removed data structures.

The method software uses to determine when it is safe to modify a removed queue head is to handshake with the host controller. The handshake mechanism allows software to remove items from the asynchronous schedule, then execute a simple, lightweight handshake that is used by software as a key that it can free (or reuse) the memory associated the data structures it has removed from the asynchronous schedule.

The handshake is implemented with three bits in the host controller. The first bit is a command bit (*Interrupt on Async Advance Doorbell* bit in the USBCMD register) that allows software to inform the host controller that something has been removed from its asynchronous schedule. The second bit is a status bit (*Interrupt on Async Advance* bit in the USBSTS register) that the host controller sets after it has released all on-chip state that may potentially reference one of the data structures just removed. When the host controller sets this status bit to a one, it also sets the command bit to a zero. The third bit is an interrupt enable (*Interrupt on Async Advance* bit in the USBINTR register) that is matched with the status bit. If the status bit is a one and the interrupt enable bit is a one, then the host controller will assert a hardware interrupt.

Figure 4-10 illustrates a general example. In this example, consecutive queue heads (B and C) are unlinked from the schedule using the algorithm above. Before the unlink operation, the host controller has a copy of queue head A. The unlink algorithm requires that as software unlinks each queue head, the unlinked queue head is loaded with the address of a queue head that will remain in the asynchronous schedule.

When the host controller observes that doorbell bit being set to a one, it makes a note of the local reachable schedule information. In this example, the local reachable schedule information includes both queue heads (A & B). It is sufficient that the host controller can set the status bit (and clear the doorbell bit) as soon as it has traversed beyond current reachable schedule information (i.e. traversed beyond queue head (B) in this example).

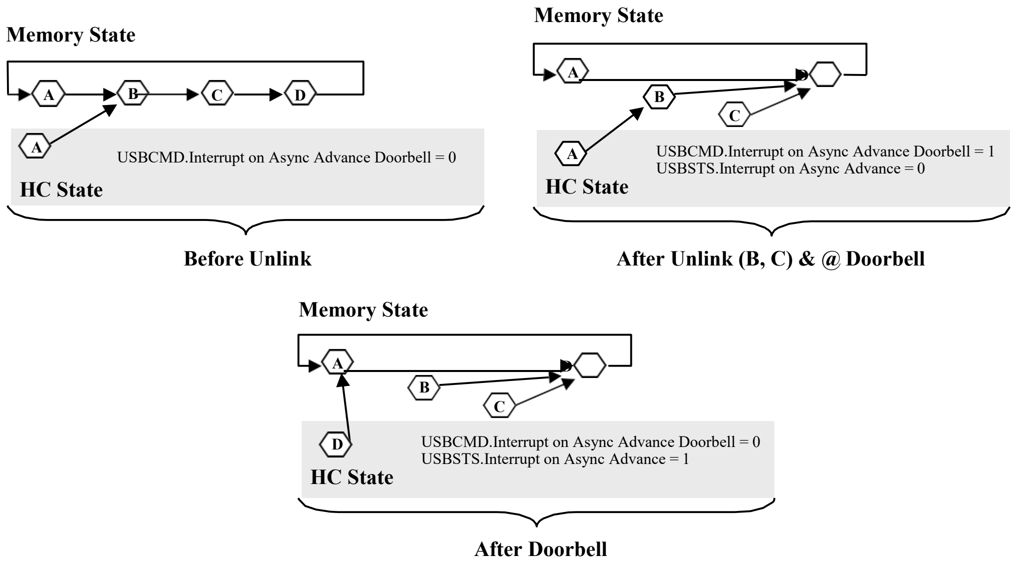

Memory State
A
B
C
D
A
USBCMD.Interrupt on Async Advance Doorbell = 0
HC State
Before Unlink
Memory State
A
B
C
A
USBCMD.Interrupt on Async Advance Doorbell = 1
USBSTS.Interrupt on Async Advance = 0
HC State
After Unlink (B, C) & @ Doorbell
Memory State
A
B
C
D
USBCMD.Interrupt on Async Advance Doorbell = 0
USBSTS.Interrupt on Async Advance = 1
HC State
After Doorbell

Figure 4-10. Generic Queue Head Unlink Scenario

Alternatively, a host controller implementation is allowed to traverse the entire asynchronous schedule list (e.g. observed the head of the queue (twice)) before setting the *Advance on Async* status bit to a one.

Software may re-use the memory associated with the removed queue heads after it observes the *Interrupt on Async Advance* status bit is set to a one, following assertion of the doorbell. Software should acknowledge the *Interrupt on Async Advance* status as indicated in the USBSTS register, before using the doorbell handshake again.

4.8.3 Empty Asynchronous Schedule Detection

The Enhanced Host Controller Interface uses two bits to detect when the asynchronous schedule is empty. The queue head data structure (see Figure 3-7) defines an *H-bit* in the queue head, which allows software to mark a queue head as being the *head* of the reclaim list. The Enhanced Host Controller Interface also keeps a 1-bit flag in the USBSTS register (*Reclamation*) that is set to a zero when the Enhanced Interface Host Controller observes a queue head with the H-bit set to a one. The reclamation flag in the status register is set to one when any USB transaction from the asynchronous schedule is executed (or whenever the asynchronous schedule starts, see Section 4.8.5).

If the Enhanced Host Controller Interface ever encounters an *H-bit* of one and a *Reclamation* bit of zero, the EHCI controller simply stops traversal of the asynchronous schedule.

An example illustrating the H-bit in a schedule is illustrated in Figure 4-11.

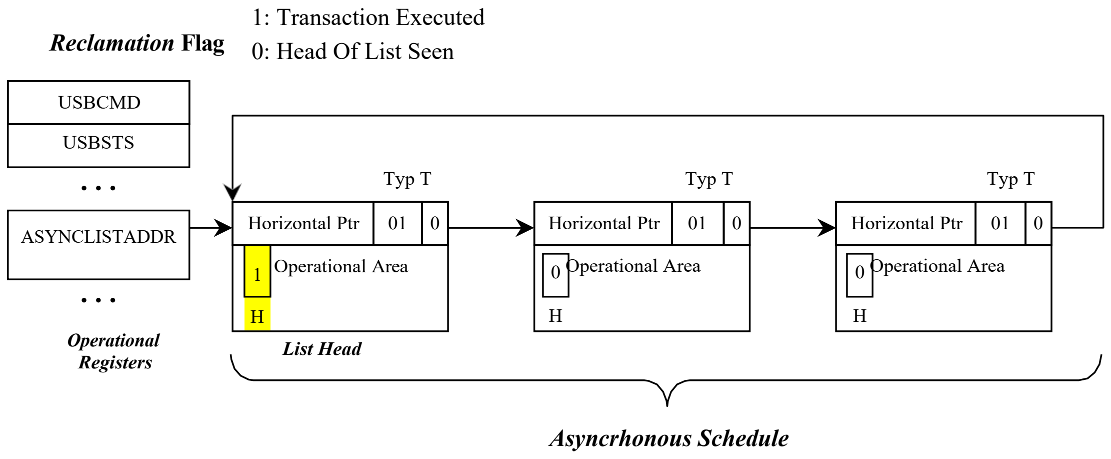


Figure 4-11. Asynchronous Schedule List w/Annotation to Mark Head of List

Software must ensure there is at most one queue head with the *H-bit* set to a one, and that it is always coherent with respect to the schedule.

4.8.4 Restarting Asynchronous Schedule Before EOF

There are many situations where the host controller will detect an empty list *long* before the end of the micro-frame. It is important to remember that under many circumstances the schedule traversal has stopped due to Nak/Nyet responses from all endpoints.

An example of particular interest is when a start-split for a bulk endpoint occurs early in the micro-frame. Given the EHCI simple traversal rules, the complete-split for that transaction may Nak/Nyet out very quickly. If it is the only item in the schedule, then the host controller will cease traversal of the Asynchronous schedule very early in the micro-frame. In order to provide reasonable service to this endpoint, the host controller should issue the complete-split before the end of the current micro-frame, instead of waiting until the next micro-frame.

When the reason for host controller idling asynchronous schedule traversal is because of empty list detection, it is mandatory the host controller implement a 'waking' method to resume traversal of the asynchronous schedule. An example method is described below.

4.8.4.1 Example Method for Restarting Asynchronous Schedule Traversal

The reason for idling the host controller when the list is empty is to keep the host controller from unnecessarily occupying too much memory bandwidth. The question is: *how long should the hostcontroller stay idle before restarting*?

The answer in this example is based on deriving a manifest constant, which is the amount of time the host controller will stay idle before restarting traversal. In this example, the manifest constant is called *AsyncSchedSleepTime*, and has a value of 10µsec. The value is derived based on the analysis in Section 4.8.4.2. The traversal algorithm is simple:

- Traverse the Asynchronous schedule until the either an End-Of-micro-Frame event occurs, or an empty list is detected. If the event is an End-of-micro-Frame, go attempt to traverse the Periodic schedule. If the event is an empty list, then set a sleep timer and go to a *schedule sleep* state.
- When the sleep timer expires, set working context to the Asynchronous Schedule start condition and go to *schedule active* state. The start context allows the HC to reload *Nakcnt* fields, etc. so the HC has a chance to run for more than one iteration through the schedule.

This process simply repeats itself each micro-frame. Figure 4-12 illustrates a sample state machine to manage the active and sleep states of the Asynchronous Schedule traversal policy. There are three states: Actively traversing the Asynchronous schedule, Sleeping, and Not Active. The last two are similar in terms of interaction with the Asynchronous schedule, but the Not Active state means that the host controller is busy with the Periodic schedule or the Asynchronous schedule is not enabled. The Sleeping state is specifically a special state where the host controller is just waiting for a period of time before resuming execution of the Asynchronous schedule.

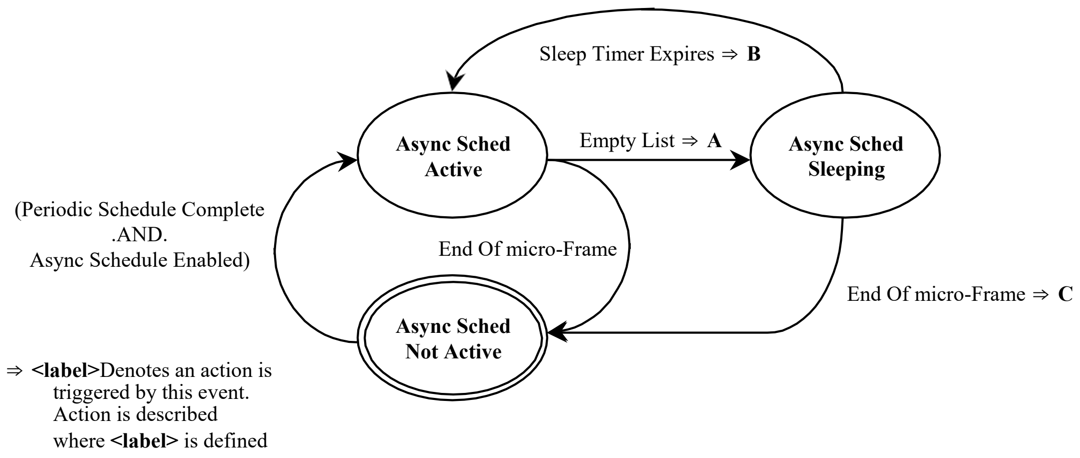

Sleep Timer Expires ⇒ **B**
Async Sched Active
Empty List ⇒ **A**
Async Sched Sleeping
(Periodic Schedule Complete .AND. Async Schedule Enabled)
End Of micro-Frame
End Of micro-Frame ⇒ **C**
Async Sched Not Active
⇒ <label>Denotes an action is triggered by this event. Action is described where <label> is defined

Figure 4-12. Example State Machine for Managing Asynchronous Schedule Traversal

The actions referred to in Figure 4-12 are defined in Table 4–7.

Table 4–7. Asynchronous Schedule SM Transition Actions

Action Label	Action Description
A	On detection of the empty list, the host controller sets the *AsynchronousTraversalSleepTimer* to *AsyncSchedSleepTime*.
B	When the *AsynchronousTraversalSleepTimer* expires, the host controller sets the *Reclamation* bit in the USBSTS register to a one and moves the Nak Counter reload state machine to **WaitForListHead** (see Section 4.9).
C	The host controller cancels the sleep timer (*AsynchronousTraversalSleepTimer*).

4.8.4.1.1 Async Sched Not Active

This is the initial state of the traversal state machine after a host controller reset. The traversal state machine will not leave this state when the *Asynchronous Schedule Enable* bit in the USBCMD register is a zero.

This state is entered from **Async Sched Active** or **Async Sched Sleeping** states when the end-of-micro-frame event is detected.

4.8.4.1.2 Async Sched Active

This state is entered from the **Async Sched Not Active** state when the periodic schedule is not active. It is also entered from the **Async Sched Sleeping** states when the *AsyncrhonousTraversalSleepTimer* expires.

On every transition into this state, the host controller sets the *Reclamation* bit in the USBSTS register to a one.

While in this state, the host controller will continually traverse the asynchronous schedule until either the end of micro-frame or an empty list condition is detected.

4.8.4.1.3 Async Sched Sleeping

The state is entered from the **Async Sched Active** state when a schedule empty condition is detected. On entry to this state, the host controller sets the *AsynchronousTraversalSleepTimer* to *AsyncSchedSleepTime*.

4.8.4.2 Example Derivation for *AsyncSchedSleepTime*

The derivation is based on analysis of what work the host controller could be doing next. It assumes the host controller does not keep any state about what work is possibly pending in the asynchronous schedule. The schedule could contain any mix of the possible combinations of high- full- or low-speed control and bulk requests. Table 4–8 summarizes some of the typical 'next transactions' that could be in the schedule, and the amount of time (e.g. *footprint*, or *wall clock*) the transaction will take to complete.

Table 4–8. Typical Low-/Full-speed Transaction Times

Transaction Attributes		Footprint (time)	Description
Speed	HS	11.9 µs	Maximum foot print for a worst-case, full-sized bulk data transaction.
Size	512	9.45 µs	Maximum footprint for an approximate best-case, full-sized bulk data transaction.
Type	Bulk		
Speed	FS	~50 µs	Approximate typical for full-sized bulk data. An 8-byte low-speed is about 2x, or between 90 and 100 µs.
Size	64		
Type	Bulk		
Speed	FS	~12 µs	Approximate typical for 8-byte bulk/control (i.e. setup)
Size	8		
Type	Cntrl		

A *AsyncSchedSleepTime* value of 10 µs provides a reasonable relaxation of the system memory load and still provides a good level of service for the various transfer types and payload sizes. For example, say we detect an empty list after issuing a start-split for a 64-byte full-speed bulk request. Assuming this is the only thing in the list, the host controller will get the results of the full-speed transaction from the hub during the fifth complete-split request. If the full-speed transaction was an IN and it nak'd, the 10µs sleep period would allow the host controller to get the NAK results on the first complete-split.

4.8.5 Asynchronous Schedule Traversal : *Start Event*

Once the HC has *idled* itself via the empty schedule detection (Section 4.8.3), it will naturally *activate* and begin processing from the Periodic Schedule at the beginning of each micro-frame. In addition, it may have idled itself early in a micro-frame. When this occurs (idles early in the micro-frame) the HC must occasionally *re-activate* during the micro-frame and traverse the asynchronous schedule to determine whether any progress can be made. The requirements and method for this restart are described in Section 4.8.4. Asynchronous schedule *Start Event*s are defined to be:

- Whenever the host controller transitions from the periodic schedule to the asynchronous schedule. If the periodic schedule is disabled and the asynchronous schedule is enabled, then the beginning of the micro-frame is equivalent to the transition from the periodic schedule, or
- The asynchronous schedule traversal restarts from a sleeping state (see Section 4.8.4).

4.8.6 Reclamation Status Bit (USBSTS Register)

The operation of the empty asynchronous schedule detection feature (Section 4.8.3) depends on the proper management of the *Reclamation* bit in the USBSTS register. The host controller tests for an empty schedule just after it fetches a new queue head while traversing the asynchronous schedule (See Section 4.10.1).

It is required that the host controller sets the *Reclamation* bit to a one whenever an asynchronous schedule traversal *Start Event*, as documented in Section 4.8.5, occurs. The *Reclamation* bit is also set to a one whenever the host controller executes a transaction while traversing the asynchronous schedule (see Section 4.10.3).

The host controller sets the *Reclamation* bit to a zero whenever it finds a queue head with its *H-bit* set to a one. Software should only set a queue head's *H-bit* if the queue head is in the asynchronous schedule. If software sets the *H-bit* in an interrupt queue head to a one, the resulting behavior is undefined. The host controller may set the *Reclamation* bit to a zero when executing from the periodic schedule.

4.9 Operational Model for Nak Counter

This section describes the operational model for the *NakCnt* field defined in a queue head (see Section 3.6). Software should not use this feature for interrupt queue heads. This rule is not required to be enforced by the host controller.

USB protocol has built-in flow control via the Nak response by a device. There are several scenarios, beyond the Ping feature, where an endpoint may naturally Nak or Nyet the majority of the time. An example is the host controller management of the split transaction protocol for control and bulk endpoints. All bulk endpoints (High- or Full-speed) are serviced via the same asynchronous schedule. The time between the *Start-split* transaction and the first *Complete-split* transaction could be very short (i.e. like when the endpoint is the only one in the asynchronous schedule). The hub NYETs (effectively Naks) the *Complete-split* transaction until the classic transaction is complete. This could result in the host controller thrashing memory, repeatedly fetching the queue head and executing the transaction to the Hub, which will not complete until after the transaction on the classic bus completes.

There are two component fields in a queue head to support the throttling feature: a counter field (*NakCnt*), and a counter reload field (*RL*). *NakCnt* is used by the host controller as one of the criteria to determine whether or not to execute a transaction to the endpoint. There are two operational modes associated with this counter:

- **Not Used.** This mode is set when the *RL* field is zero. The host controller ignores the *NakCnt* field for any execution of transactions through a queue head with an *RL* field of zero. Software must use this selection for interrupt endpoints.
- **Nak Throttle Mode.** This mode is selected when the *RL* field is non-zero. In this mode, the value in the *NakCnt* field represents the maximum number of Nak or Nyet responses the host controller will tolerate on each endpoint. In this mode, the HC will decrement the *NakCnt* field based on the token/handshake criteria listed in Table 4–9. The host controller must reload *NakCnt* when the endpoint successfully moves data (e.g. policy to reward device for moving data).

Table 4–9. NakCnt Field Adjustment Rules

Token	Handshake	
	NAK	NYET
IN/PING	decrement *NakCnt*	N/A (protocol error)
OUT	decrement *NakCnt*	No Action[1]
Start Split	decrement *NakCnt*	N/A (protocol error)
Complete Split	No Action	Decrement *NakCnt*

[1] Recommended behavior on this response is to reload *NakCnt*.

In summary, system software enables the counter by setting the reload field (*RL*) to a non-zero value. The host controller may execute a transaction if *NakCnt* is non-zero. The host controller will not execute a transaction if *NakCnt* is zero.

The reload mechanism is described in detail in Section 4.9.1.

Note that when all queue heads in the Asynchronous Schedule either exhausts all transfers or all NakCnt's go to zero, then the host controller will detect an empty Asynchronous Schedule and idle schedule traversal (see Section 4.8.3).

Any time the host controller begins a new traversal of the Asynchronous Schedule, a *Start Event* is assumed, see Section 4.8.5. Every time a Start-Event occurs, the Nak Count reload procedure is enabled.

4.9.1 Nak Count Reload Control

When the host controller reaches the *Execute Transaction* state for a queue head (meaning that it has an active operational state), it checks to determine whether the *NakCnt* field should be reloaded from *RL* (see Section 4.10.3). If the answer is yes, then *RL* is copied into *NakCnt*. After the reload or if the reload is not active, the host controller evaluates whether to execute the transaction.

The host controller must reload nak counters (*NakCnt* see Figure 3-7) in queue heads during the first pass through the reclamation list after an asynchronous schedule Start Event (see Section 4.8.5 for the definition of the Start Event). The Asynchronous Schedule should have at most one queue head marked as the head (see Figure 4-11). Figure 4-13 illustrates an example state machine that satisfies the operational requirements of the host controller detecting the first pass through the Asynchronous Schedule. This state machine is maintained internal to the host controller and is only used to gate reloading of the nak counter during the queue head traversal state: **Execute Transaction** (Figure 4-14).

The host controller does not perform the nak counter reload operation if the RL field (see Figure 3-7) is set to zero.

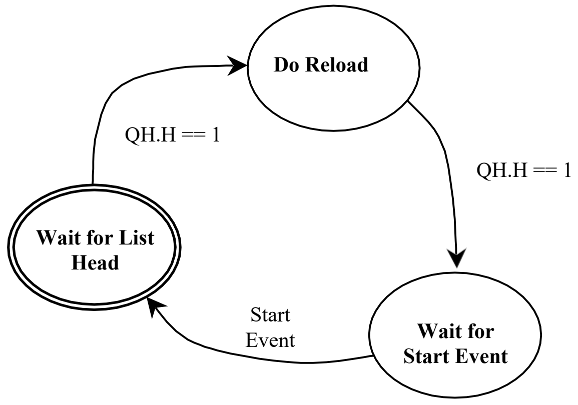


Figure 4-13. Example HC State Machine for Controlling Nak Counter Reloads

4.9.1.1 Wait for List Head

This is the initial state. The state machine enters this state from **Wait for Start Event** when a start event as defined in Section 4.8.5 occurs. The purpose of this state is to wait for the first observation of the head of the Asynchronous Schedule. This occurs when the host controller fetches a queue head whose *H-bit* is set to a one.

4.9.1.2 Do Reload

This state is entered from the **Wait for List Head** state when the host controller fetches a queue head with the *H-bit* set to a one. While in this state, the host controller will perform nak counter reloads for every queue head visited that has a non-zero nak reload value (*RL*) field.

4.9.1.3 Wait for Start Event

This state is entered from the *Do Reload* state when a queue head with the *H-bit* set to a one is fetched. While in this state, the host controller will not perform nak counter reloads.

4.10 Managing Control/Bulk/Interrupt Transfers via Queue Heads

This section presents an overview of how the host controller interacts with queuing data structures.

Queue heads use the Queue Element Transfer Descriptor (qTD) structure defined in Section 3.5. One queue head is used to manage the data stream for one endpoint. The queue head structure contains static endpoint characteristics and capabilities. It also contains a working area from where individual bus transactions for an endpoint are executed (see Overlay area defined in Figure 3-7). Each qTD represents one or more bus transactions, which is defined in the context of this specification as a *transfer*.

The general processing model for the host controller's use of a queue head is simple:

- read a queue head,
- execute a transaction from the overlay area,
- write back the results of the transaction to the overlay area
- move to the next queue head.

If the host controller encounters errors during a transaction, the host controller will set one (or more) of the error reporting bits in the queue head's *Status* field. The *Status* field accumulates all errors encountered during the execution of a qTD (e.g. the error bits in the queue head *Status* field are 'sticky' until the transfer (qTD) has completed). This state is always written back to the source qTD when the transfer is complete.

On transfer (e.g. buffer or halt conditions) boundaries, the host controller must auto-advance (without software intervention) to the next qTD. Additionally, the hardware must be able to halt the queue so no additional bus transactions will occur for the endpoint and the host controller will not advance the queue.

An example host controller operational state machine of a queue head traversal is illustrated in Figure 4-14. This state machine is a model for how a host controller should traverse a queue head. The host controller must be able to advance the queue from the *Fetch QH* state in order to avoid all hardware/software race conditions. This simple mechanism allows software to simply link qTDs to the queue head and *activate* them, then the host controller will always *find* them if/when they are reachable.

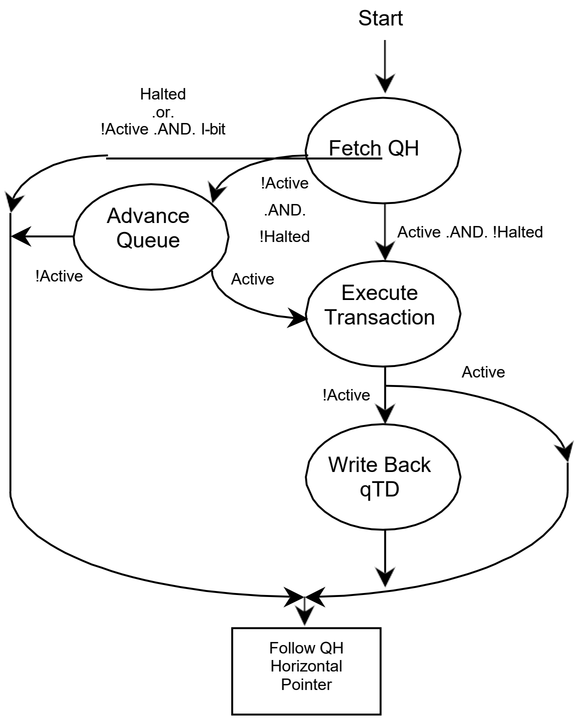


Figure 4-14. Host Controller Queue Head Traversal State Machine

This traversal state machine applies to all queue heads, regardless of transfer type or whether split transactions are required. The following sections describe each state. Each state description describes the entry criteria. The **Execute Transaction** state (Section 4.10.3) describes the basic requirements for all endpoints. Sections 4.12.1 and 4.12.2 describe details of the required extensions to the **Execute Transaction** state for endpoints requiring split transactions.

Note: Prior to software placing a queue head into either the periodic or asynchronous list, software must ensure the queue head is properly initialized. Minimally, the queue head should be initialized to the following (see Section 3.6 for layout of a queue head):

- Valid static endpoint state
- For the very first use of a queue head, software may zero-out the queue head transfer overlay, then set the *Next qTD Pointer* field value to reference a valid qTD.

4.10.1 Fetch Queue Head

A queue head can be referenced from the physical address stored in the ASYNCLISTADDR Register (Section 2.3.7). Additionally, it may be referenced from the *Next Link Pointer* field of an iTD, siTD, FSTN or another Queue Head. If the referencing link pointer has the *Typ* field set to indicate a queue head, it is assumed to reference a queue head structure as defined in Figure 3-7.

While in this state, the host controller performs operations to implement empty schedule detection (Section 4.8.3) and Nak Counter reloads (Section 4.9).

After the queue head has been fetched, the host controller conducts the following queries for empty schedule detection:

- If queue head is not an interrupt queue head (i.e. *S-mask* is a zero), and
- The *H-bit* is a one, and
- The *Reclamation* bit in the USBSTS register is a zero.

When these criteria are met, the host controller will stop traversing the asynchronous list (as described in Section 4.8.3). When the criteria are not met, the host controller continues schedule traversal.

If the queue head is not an interrupt and the *H-bit* is a one and the *Reclamation* bit is a one, then the host controller sets the *Reclamation* bit in the USBSTS register to a zero before completing this state.

The operations for reloading of the Nak Counter are described in detail in Section 4.9.

This state is complete when the queue head has been read on-chip.

4.10.2 Advance Queue

To advance the queue, the host controller must find the next qTD, adjust pointers, perform the overlay and write back the results to the queue head.

This state is entered from the **FetchQHD** state if the overlay *Active* and *Halt* bits are set to zero. On entry to this state, the host controller determines which next pointer to use to fetch a qTD, fetches a qTD and determines whether or not to perform an overlay. Note that if the *I-bit* is a one and the *Active* bit is a zero, the host controller immediately skips processing of this queue head, exits this state and uses the horizontal pointer to the next schedule data structure.

If the field *Bytes to Transfer* is not zero and the *T-bit* in the *Alternate Next qTD Pointer* is set to zero, then the host controller uses the *Alternate Next qTD Pointer*. Otherwise, the host controller uses the *Next qTD Pointer*. If *Next qTD Pointer*'s *T-bit* is set to a one, then the host controller exits this state and uses the horizontal pointer to the next schedule data structure.

Using the selected pointer the host controller fetches the referenced qTD. If the fetched qTD has it's *Active* bit set to a one, the host controller moves the pointer value used to reach the qTD (*Next* or *Alternate Next*) to the *Current qTD Pointer* field, then performs the overlay. If the fetched qTD has its *Active* bit set to a zero, the host controller aborts the queue advance and follows the queue head's horizontal pointer to the next schedule data structure.

The host controller performs the overlay based on the following rules:

- The value of the data toggle (*dt*) field in the overlay area depends on the value of the *data toggle control* (*dtc*) bit (see Table 3-19).
- If the *EPS* field indicates the endpoint is a high-speed endpoint, the *Ping* state field is preserved by the host controller. The value of this field is not changed as a result of the overlay.
- *C-prog-mask* field is set to zero (field from incoming qTD is ignored, as is the current contents of the overlay area).
- *Frame Tag* field is set to zero (field from incoming qTD is ignored, as is the current contents of the overlay area).
- *NakCnt* field in the overlay area is loaded from the *RL* field in the queue head's Static Endpoint State.
- All other areas of the overlay are set by the incoming qTD.

The host controller exits this state when it has committed the write to the queue head.

4.10.3 Execute Transaction

The host controller enters this state from the **Fetch Queue Head** state only if the *Active* bit in *Status* field of the queue head is set to a one.

On entry to this state, the host controller executes a few pre-operations, then checks some pre-condition criteria before committing to executing a transaction for the queue head.

The pre-operations performed and the pre-condition criteria depend on whether the queue head is an interrupt endpoint. The host controller can determine that a queue head is an interrupt queue head when the queue head's *S-mask* field contains a non-zero value. It is the responsibility of software to ensure the *S-mask* field is appropriately initialized based on the transfer type. There are other criteria that must be met if the *EPS* field indicates that the endpoint is a low- or full-speed endpoint, see Sections 4.12.1 and 4.12.2.

- **Interrupt Transfer Pre-condition Criteria**

  If the queue head is for an interrupt endpoint (e.g. non-zero *S-mask* field), then the FRINDEX[2:0] field must identify a bit in the *S-mask* field that has a one in it. For example, an *S-mask* value of 00100000b would evaluate to true only when FRINDEX[2:0] is equal to 101b. If this condition is met then the host controller considers this queue head for a transaction**.**

- **Asynchronous Transfer Pre-operations and Pre-condition Criteria**

  If the queue head is not for an interrupt endpoint (e.g. a zero *S-mask* field), then the host controller performs one pre-operation and then evaluates one pre-condition criteria: The pre-operation is:

  - Checks the Nak counter reload state (Section 4.9). It may be necessary for the host controller to reload the Nak Counter field. The reload is performed at this time.

  The pre-condition evaluated is:

  - Whether or not the *NakCnt* field has been reloaded, the host controller checks the value of the *NakCnt* field in the queue head. If *NakCnt* is non-zero, or if the *Reload Nak Counter* field is zero, then the host controller considers this queue head for a transaction.

- **Transfer Type Independent Pre-operations**

  Regardless of the transfer type, the host controller always performs at least one pre-operation and evaluates one pre-condition. The pre-operation is:

  - A host controller internal transaction (down) counter *qHTransactionCounter* is loaded from the queue head's *Mult* field. A host controller implementation is allowed to ignore this for queue heads on the asynchronous list. It is mandatory for interrupt queue heads. Software should ensure that the *Mult* field is set appropriately for the transfer type.

  The pre-conditions evaluated are:

  - The host controller determines whether there is enough time in the micro-frame to complete this transaction (see Section 4.4.1.1 for an example evaluation method). If there is not enough time to complete the transaction, the host controller exits this state.
  - If the value of *qHTransactionCounter* for an interrupt endpoint is zero, then the host controller exits this state.

When the pre-operations are complete and pre-conditions are met, the host controller sets the *Reclamation* bit in the USBSTS register to a one and then begins executing one or more transactions using the endpoint information in the queue head. The host controller iterates *qHTransactionCounter* times in this state executing transactions. After each transaction is executed, *qHTransactionCounter* is decremented by one. The host controller will exit this state when one of the following events occurs:

- The *qHTransactionCounter* decrements to zero, or
- The endpoint responds to the transaction with any handshake other than an ACK,[4] or
- The transaction experiences a transaction error, or
- The *Active* bit in the queue head goes to a zero, or
- There is not enough time in the micro-frame left to execute the next transaction (see Section 4.4.1.1 for example method for implementing the frame boundary test).

[4] Note that for a high-bandwidth interrupt OUT endpoint, the host controller may optionally immediately retry the transaction if it fails.

The results of each transaction is recorded in the on-chip overlay area. If data was successfully moved during the transaction, the transfer state in the overlay area is advanced. To advance queue head's transfer state, the *Total Bytes to Transfer* field is decremented by the number of bytes moved in the transaction, the data toggle bit (*dt*) is toggled, the current page offset is advanced to the next appropriate value (e.g. advanced by the number of bytes successfully moved), and the *C_Page* field is updated to the appropriate value (if necessary). See Section 4.10.6.

Note that the *Total Bytes To Transfer* field may be zero when all the other criteria for executing a transaction are met. When this occurs, the host controller will execute a zero-length transaction to the endpoint. If the *PID_Code* field indicates an IN transaction and the device delivers data, the host controller will detect a packet babble condition, set the *babble* and *halted* bits in the *Status* field, set the *Active* bit to a zero, write back the results to the source qTD, then exit this state.

In the event an IN token receives a data PID mismatch response, the host controller must ignore the received data (e.g. not advance the transfer state for the bytes received). Additionally, if the endpoint is an interrupt IN, then the host controller must record that the transaction occurred (e.g. decrement *qHTransactionCounter*). It is recommended (but not required) the host controller continue executing transactions for this endpoint if the resultant value of *qHTransactionCounter* is greater than one.

If the response to the IN bus transaction is a Nak (or Nyet) and *RL* is non-zero, *NakCnt* is decremented by one. If *RL* is zero, then no write-back by the host controller is required (for a transaction receiving a Nak or Nyet response and the value of *CErr* did not change). Software should set the *RL* field to zero if the queue head is an interrupt endpoint. Host controller hardware is not required to enforce this rule or operation.

After the transaction has finished and the host controller has completed the post processing of the results (advancing the transfer state and possibly *NakCnt*, the host controller writes back the results of the transaction to the queue head's overlay area in main memory.

The number of bytes moved during an IN transaction depends on how much data the device endpoint delivers. The maximum number of bytes a device can send is *Maximum Packet Size*. The number of bytes moved during an OUT transaction is either *Maximum Packet Length* bytes or *Total Bytes to Transfer*, whichever is less.

If there was a transaction error during the transaction, the transfer state (as defined above) is not advanced by the host controller. The *CErr* field is decremented by one and the status field is updated to reflect the type of error observed. Transaction errors are summarized in Section 4.15.1.1.

The following events will cause the host controller to clear the *Active* bit in the queue head's overlay status field. When the *Active* bit transitions from a one to a zero, the transfer in the overlay is considered complete. The reason for the transfer completion (clearing the *Active* bit) determines the next state.

- *CErr* field decrements to zero. When this occurs the *Halted* bit is set to a one and *Active* is set to a zero. This results in the hardware not advancing the queue and the pipe halts. Software must intercede to recover.
- The device responds to the transaction with a STALL PID. When this occurs, the *Halted* bit is set to a one and the *Active* bit is set to a zero. This results in the hardware not advancing the queue and the pipe halts. Software must intercede to recover.
- The *Total Bytes to Transfer* field is zero after the transaction completes. Note that for a zero length transaction, it was zero before the transaction was started. When this condition occurs, the *Active* bit is set to zero.
- The PID code is an IN, and the number of bytes moved during the transaction is less than the *Maximum Packet Length*. When this occurs, the *Active* bit is set to zero and a short packet condition exists. The short-packet condition is detected during the **Advance Queue** state. Refer to Section 4.12 for additional rules for managing low- and full-speed transactions.
- The *PID Code* field indicates an IN and the device sends more than the expected number of bytes (e.g. *Maximum Packet Length* or *Total Bytes to Transfer* bytes, whichever is less) (e.g. a packet babble). This results in the host controller setting the *Halted* bit to a one.

With the exception of a NAK response (when *RL* field is zero), the host controller always writes the results of the transaction back to the overlay area in main memory. This includes when the transfer completes. For a high-speed endpoint, the queue head information written back includes minimally the following fields:

- *NakCnt*, *dt*, *Total Bytes to Transfer*, *C_Page*, *Status*, *CERR*, and *Current Offset*

For a low- or full-speed device the queue head information written back also includes the fields:

- *C-prog-mask*, *FrameTag* and *S-bytes*.

The duration of this state depends on the time it takes to complete the transaction(s) and the status write to the overlay is committed.

4.10.3.1 Halting a Queue Head

A halted endpoint is defined only for the transfer types that are managed via queue heads (control, bulk and interrupt). The following events indicate that the endpoint has reached a condition where no more activity can occur without intervention from the driver:

- An endpoint may return a STALL handshake during a transaction,
- A transaction had three consecutive error conditions, or
- A Packet Babble error occurs on the endpoint.

When any of these events occur (for a queue head) the Host Controller halts the queue head and set the USBERRINT status bit in the USBSTS register to a one. To halt the queue head, the *Active* bit is set to a zero and the *Halted* bit is set to a one. There may be other error status bits that are set when a queue is halted. The host controller always writes back the overlay area to the source qTD when the transfer is complete, regardless of the reason (normal completion, short packet or halt). The host controller will not advance the transfer state on a transaction that results in a *Halt* condition (e.g. no updates necessary for *Total Bytes to Transfer*, *C_Page*, *Current Offset,* and *dt*). The host controller must update *CErr* as appropriate.

When a queue head is halted, the *USB Error Interrupt* bit in the USBSTS register is set to a one. If the *USB Error Interrupt Enable* bit in the USBINTR register is set to a one, a hardware interrupt is generated at the next interrupt threshold.

4.10.3.2 Asynchronous Schedule Park Mode

Asynchronous Schedule Park mode is a special execution mode that can be enabled by system software, where the host controller is permitted to execute more than one bus transaction from a high-speed queue head in the Asynchronous schedule before continuing horizontal traversal of the Asynchronous schedule. This feature has no effect on queue heads or other data structures in the Periodic schedule. This feature is similar in intent as the *Mult* feature that is used in the Periodic schedule. Where-as the *Mult* feature is a characteristic that is tunable for each endpoint; park-mode is a policy that is applied to all high-speed queue heads in the asynchronous schedule. It is essentially the specification of an iterator for consecutive bus transactions to the same endpoint. All of the rules for managing bus transactions and the results of those as defined in Section 4.10.3 apply. This feature merely specifies how many consecutive times the host controller is permitted to execute from the same queue head before moving to the next queue head in the Asynchronous List. This feature should allow the host controller to attain better bus utilization for those devices that are capable of moving data at maximum rate, while at the same time providing a fair service to all endpoints.

A host controller exports its capability to support this feature to system software by setting the *Asynchronous Schedule Park Capability* bit in the HCCPARAMs register to a one. This information keys system software that the *Asynchronous Schedule Park Mode Enable* and *Asynchronous Schedule Park Mode Count* fields in the USBCMD register are modifiable. System software enables the feature by writing a one to the *Asynchronous Schedule Park Mode Enable* bit.

When park-mode is not enabled (e.g. *Asynchronous Schedule Park Mode Enable* bit in the USBCMD register is a zero), the host controller must not execute more than one bus transaction per high-speed queue head, per traversal of the asynchronous schedule.

When park-mode is enabled, the host controller must not apply the feature to a queue head whose *EPS* field indicates a Low/Full-speed device (i.e. only one bus transaction is allowed from each Low/Full-speed queue head per traversal of the asynchronous schedule). Park-mode may only be applied to queue heads in the Asynchronous schedule whose *EPS* field indicates that it is a high-speed device.

The host controller must apply park mode to queue heads whose *EPS* field indicates a high-speed endpoint. The maximum number of consecutive bus transactions a host controller may execute on a high-speed queue head is determined by the value in the *Asynchronous Schedule Park Mode Count* field in the USBCMD register. Software must not set *Asynchronous Schedule Park Mode Enable* bit to a one and also set *Asynchronous Schedule Park Mode Count* field to a zero. The resulting behavior is not defined. An example behavioral example describes the operational requirements for the host controller implementing park-mode.

This feature does not affect how the host controller handles the bus transaction as defined in Section 4.10.3. It only effects how many consecutive bus transactions for the current queue head can be executed. All boundary conditions, error detection and reporting applies as usual. This feature is similar in concept to the use of the *Mult* field for high-bandwidth Interrupt for queue heads in the Periodic Schedule.

The host controller effectively loads an internal down-counter *PM-Count* from *Asynchronous Schedule Park Mode Count* when *Asyncrhonous Schedule Park Mode Enable* bit is a one, and a high-speed queue head is first fetched and meets all the criteria for executing a bus transaction. After the bus transaction, *PM-Count* is decremented. The host controller may continue to execute bus transactions from the current queue head until *PM-Count* goes to zero, an error is detected, the buffer for the current transfer is exhausted or the endpoint responds with a flow-control or STALL handshake. Table 4–10 summarizes the responses that effect whether the host controller continues with another bus transaction for the current queue head.

Table 4–10. Actions for Park Mode, based on Endpoint Response and Residual Transfer State

PID	Endpoint Response	Transfer State after Transaction		Action
		PM-Count	Bytes to Transfer	
IN	DATA[0,1] w/Maximum Packet sized data	Not zero	Not Zero	Allowed to perform another bus transaction.[1, 2]
		Not zero	Zero	Retire qTD and move to next QH
		Zero	Don't care	Move to next QH.
	DATA[0,1] w/short packet	Don't care	Don't care	Retire qTD and move to next QH.
	NAK	Don't care	Don't care	Move to next QH.
	STALL, XactErr	Don't care	Don't care	Move to next QH.
OUT	ACK	Not zero	Not Zero	Allowed to perform another bus transaction. [2]
		Not zero	Zero	Retire qTD and move to next QH
		Zero	Don't' care	Move to next QH.
	NYET, NAK	Don't care	Don't care	Move to next QH.
	STALL, XactErr	Don't care	Don't care	Move to next QH

Table 4–10. Actions for Park Mode, based on Endpoint Response and Residual Transfer State (cont.)

Direction	Endpoint Response	Transfer State after Transaction		Action
		PM-Count	Bytes to Transfer	
PING	ACK	Not Zero	Not Zero	Allowed to perform another bus transaction. [2]
	NAK	Don't care	Don't care	Move to next QH
	STALL, XactErr	Don't care	Don't care	Move to next QH

[1] Note, the host controller may continue to execute bus transactions from the current high-speed queue head (if *PM-Count* is not equal to zero), if a PID mismatch is detected (e.g. expected DATA1 and received DATA0, or visa-versa).

[2] Note, this specification does not *require* that the host controller execute another bus transaction when *PM-Count* is non-zero. Implementations are encouraged to make appropriate complexity and performance trade-offs.

4.10.4 Write Back qTD

This state is entered from the **Execute Transaction** state when the *Active* bit is set to a zero. The source data for the write-back is the transfer results area of the queue head overlay area (see Figure 3-7). The host controller uses the *Current qTD Pointer* field as the target address for the qTD. The queue head transfer result area is written back to the transfer result area of the target qTD. This state is also referred to as: qTD retirement. The fields that must be written back to the source qTD include *Total Bytes to Transfer*, *Cerr*, and *Status*.

The duration of this state depends on when the qTD write-back has been committed.

4.10.5 Follow Queue Head Horizontal Pointer

The host controller must use the horizontal pointer in the queue head to the next schedule data structure when any of the following conditions exist:

- If the *Active* bit is a one on exit from the **Execute Transaction** state, or
- When the host controller exits the **Write Back qTD** state, or
- If the **Advance Queue** state fails to advance the queue because the target qTD is not active, or
- If the *Halted* bit is a one on exit from the **Fetch QH** state.

There is no functional requirement that the host controller wait until the current transaction is complete before using the horizontal pointer to read the next linked data structure. However, it must wait until the current transaction is complete before executing the next data structure.

4.10.6 Buffer Pointer List Use for Data Streaming with qTDs

A qTD has an array of buffer pointers, which is used to reference the data buffer for a transfer. This specification requires that the buffer associated with the transfer be *virtually contiguous*. This means: if the buffer spans more than one physical page, it must obey the following rules (Figure 4-15 illustrates an example):

- The first portion of the buffer must begin at some offset in a page and extend through the end of the page.
- The remaining buffer cannot be allocated in small chunks scattered around memory. For each 4K chunk beyond the first page, each buffer portion matches to a full 4K page. The final portion, which may only be large enough to occupy a portion of a page, must start at the top of the page and be contiguous within that page.

The buffer pointer list in the qTD is long enough to support a maximum transfer size of 20K bytes. This case occurs when all five buffer pointers are used and the first offset is zero. A qTD handles a 16Kbyte buffer with any starting buffer alignment.

The host controller uses the field *C_Page* field as an index value to determine which buffer pointer in the list should be used to start the current transaction. The host controller uses a different buffer pointer for each physical page of the buffer. This is always true, even if the buffer is physically contiguous.

The host controller must detect when the current transaction will span a page boundary and automatically move to the next available buffer pointer in the page pointer list. The next available pointer is reached by incrementing *C_Page* and pulling the next page pointer from the list. Software must ensure there are sufficient buffer pointers to move the amount of data specified in the *Bytes to Transfer* field.

Figure 4-15 illustrates a nominal example of how System software would initialize the buffer pointers list and the *C_Page* field for a transfer size of 16383 bytes. *C_Page* is set to zero. The upper 20-bits of Page 0 references the start of the physical page. *Current Offset* (the lower 12-bits of queue head Dword 7) holds the offset in the page e.g. 2049 (e.g. 4096-2047). The remaining page pointers are set to reference the beginning of each subsequent 4K page.

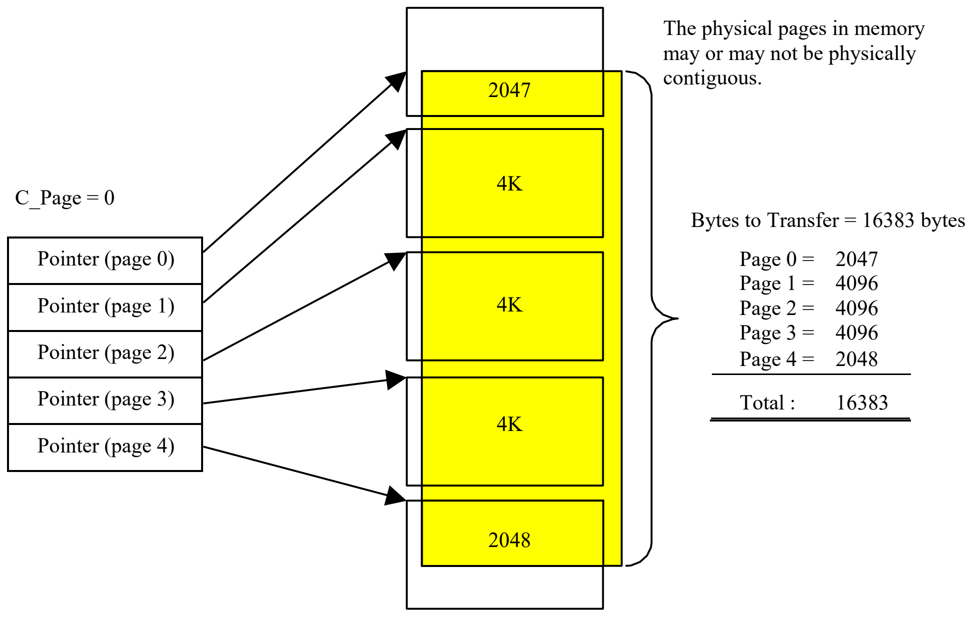


Figure 4-15. Example Mapping of qTD Buffer Pointers to Buffer Pages

For the first transaction on the qTD (assuming a 512-byte transaction), the host controller uses the first buffer pointer (page 0 because *C_Page* is set to zero) and concatenates the *Current Offset* field. The 512 bytes are moved during the transaction, the *Current Offset* and *Total Bytes to Transfer* are adjusted by 512 and written back to the queue head working area.

During the 4th transaction, the host controller needs 511 bytes in page 0 and one byte in page 1. The host controller will increment *C_Page* (to 1) and use the page 1 pointer to move the final byte of the transaction. After the 4th transaction, the active page pointer is the page 1 pointer and *Current Offset* has rolled to one, and both are written back to the overlay area. The transactions continue for the rest of the buffer, with the host controller automatically moving to the next page pointer (i.e. *C_Page*) when necessary.

There are three conditions for how the host controller handles *C_Page*.

- The current transaction does not span a page boundary. The value of *C_Page* is not adjusted by the host controller.
- The current transaction does span a page boundary. The host controller must detect the page cross condition and advance to the next buffer while streaming data to/from the USB.

- The current transaction completes on a page boundary (i.e. the last byte moved for the current transaction is the last byte in the page for the current page pointer). The host controller must increment *C_Page* before writing back status for the transaction.

Note that the only valid adjustment the host controller may make to *C_Page* is to increment by one.

4.10.7 Adding Interrupt Queue Heads to the Periodic Schedule

The link path(s) from the periodic frame list to a queue head establishes in which frames a transaction can be executed for the queue head. Queue heads are linked into the periodic schedule so they are polled at the appropriate rate. System software sets a bit in a queue head's *S-Mask* to indicate which micro-frame with-in a 1 millisecond period a transaction should be executed for the queue head. Software must ensure that all queue heads in the periodic schedule have *S-Mask* set to a non-zero value. An *S-mask* with a zero value in the context of the periodic schedule yields undefined results.

If the desired poll rate is greater than one frame, system software can use a combination of queue head linking and *S-Mask* values to spread interrupts of equal poll rates through the schedule so that the periodic bandwidth is allocated and managed in the most efficient manner possible. Some examples are illustrated in Table 4–11.

Table 4–11. Example Periodic Reference Patterns for Interrupt Transfers with 2ms Poll Rate

Frame # Reference Sequence	Description
0, 2, 4, 6, 8, etc. *S-Mask* = 01h	A queue head for the *bInterval* of 2 milliseconds (16 micro-frames) is linked into the periodic schedule so that it is reachable from the periodic frame list locations indicated in the previous column. In addition, the *S-Mask* field in the queue head is set to 01h, indicating that the transaction for the endpoint should be executed on the bus during micro-frame 0 of the frame.
0, 2, 4, 6, 8, etc. *S-Mask* = 02h	Another example of a queue head with a *bInterval* of 2 milliseconds is linked into the periodic frame list at exactly the same interval as the previous example. However, the *S-Mask* is set to 02h indicating that the transaction for the endpoint should be executed on the bus during micro-frame 1 of the frame.

4.10.8 Managing Transfer Complete Interrupts from Queue Heads

The host controller will set an interrupt to be signaled at the next interrupt threshold when the completed transfer (qTD) has an *Interrupt on Complete* (*IOC*) bit set to a one, or whenever a transfer (qTD) completes with a short packet. If system software needs multiple qTDs to complete a client request (i.e. like a control transfer) the intermediate qTDs do not require interrupts. System software may only need a single interrupt to notify it that the complete buffer has been transferred. System software may set IOC's to occur more frequently. A motivation for this may be that it wants early notification so that interface data structures can be re-used in a timely manner.

4.11 Ping Control

USB 2.0 defines an addition to the protocol for high-speed devices called Ping. Ping is required for all USB 2.0 High-speed bulk and control endpoints. Ping is not allowed for a split-transaction stream. This extension to the protocol eliminates the bad side-effects of Naking OUT endpoints. The *Status* field has a *Ping State* bit, which the host controller uses to determine the *next* actual PID it will use in the next transaction to the endpoint (see Table 3-16). The Ping State bit is only managed by the host controller for queue heads that meet the following criteria:

- Queue head is not an interrupt and

- *EPS* field equals High-Speed and
- *PIDCode* field equals OUT

Table 4–12 illustrates the state transition table for the host controller's responsibility for maintaining the PING protocol. Refer to Chapter 8 in the USB Specification Revision 2.0 for detailed description on the Ping protocol.

Table 4–12. Ping Control State Transition Table

Current	Event		Next
	Host	Device	
Do Ping	PING	Nak	Do Ping
Do Ping	PING	Ack	Do OUT
Do Ping	PING	XactErr[1]	Do Ping
Do Ping	PING	Stall	N/C[2]
Do OUT	OUT	Nak	Do Ping
Do OUT	OUT	Nyet	Do Ping[3]
Do OUT	OUT	Ack	Do OUT
Do OUT	OUT	XactErr[1]	Do Ping
Do OUT	OUT	Stall	N/C[2]

[1] Transaction Error (XactErr) is any time the host misses the handshake.

[2] No transition change required for the Ping State bit. The Stall handshake results in the endpoint being halted (e.g. *Active* set to zero and *Halt* set to a one). Software intervention is required to restart queue.

[3] A Nyet response to an OUT means that the device has accepted the data, but cannot receive any more at this time. Host must advance the transfer state and additionally, transition the Ping State bit to **Do Ping**.

The *Ping State* bit has the following encoding:

Value	Meaning
0B	*Do OUT* The host controller will use an OUT PID during the next bus transaction to this endpoint.
1B	*Do Ping* The host controller will use a PING PID during the next bus transaction to this endpoint.

The defined ping protocol (see USB 2.0 Specification, Chapter 8) allows the host to be *imprecise* on the initialization of the ping protocol (i.e. start in *Do OUT* when we don't know whether there is space on the device or not).

The host controller manages the *Ping State* bit. System software sets the initial value in the queue head when it initializes a queue head. The host controller preserves the *Ping State* bit across all queue advancements. This means that when a new qTD is written into the queue head overlay area, the previous value of the *Ping State* bit is preserved.

4.12 Split Transactions

USB 2.0 defines extensions to the bus protocol for managing USB 1.x data streams through USB 2.0 Hubs. This section describes how the host controller uses the interface data structures to manage data streams with full- and low-speed devices, connected below USB 2.0 hub, utilizing the split transaction protocol. Refer to USB 2.0 Specification for the complete definition of the split transaction protocol.

Full- and Low-speed devices are enumerated identically as high-speed devices, but the transactions to the Full- and Low-speed endpoints use the split-transaction protocol on the high-speed bus. The split transaction protocol is an encapsulation of (or wrapper around) the Full- or Low-speed transaction. The high-speed wrapper portion of the protocol is addressed to the USB 2.0 Hub and Transaction Translator below which the Full- or Low-speed device is attached.

The EHCI interface uses dedicated data structures for managing full-speed isochronous data streams (see Section 3.4). Control, Bulk and Interrupt are managed using the queuing data structures (see Sections 3.6). The interface data structures need to be programmed with the device address and the Transaction Translator number of the USB 2.0 Hub operating as the Low-/Full-speed host controller for this link. The following sections describe the details of how the host controller must process and manage the split transaction protocol.

4.12.1 Split Transactions for Asynchronous Transfers

A queue head in the asynchronous schedule with an *EPS* field indicating a full-or low-speed device indicates to the host controller that it must use split transactions to stream data for this queue head. All full-speed bulk and full-, low-speed control are managed via queue heads in the asynchronous schedule.

Software must initialize the queue head with the appropriate device address and port number for the transaction translator that is serving as the full/low-speed host controller for the links connecting the endpoint. Software must also initialize the split transaction state bit (*SplitXState*) to **Do-Start-Split**. Finally, if the endpoint is a control endpoint, then system software must set the *Control Transfer Type* (*C*) bit in the queue head to a one. If this is not a control transfer type endpoint, the *C* bit must be initialized by software to be a zero. This information is used by the host controller to properly set the Endpoint Type (ET) field in the split transaction bus token. When the *C* bit is a zero, the split transaction token's ET field is set to indicate a bulk endpoint. When the *C* bit is a one, the split transaction token's ET field is set to indicate a control endpoint. Refer to Chapter 8 of USB Specification Revision 2.0 for details.

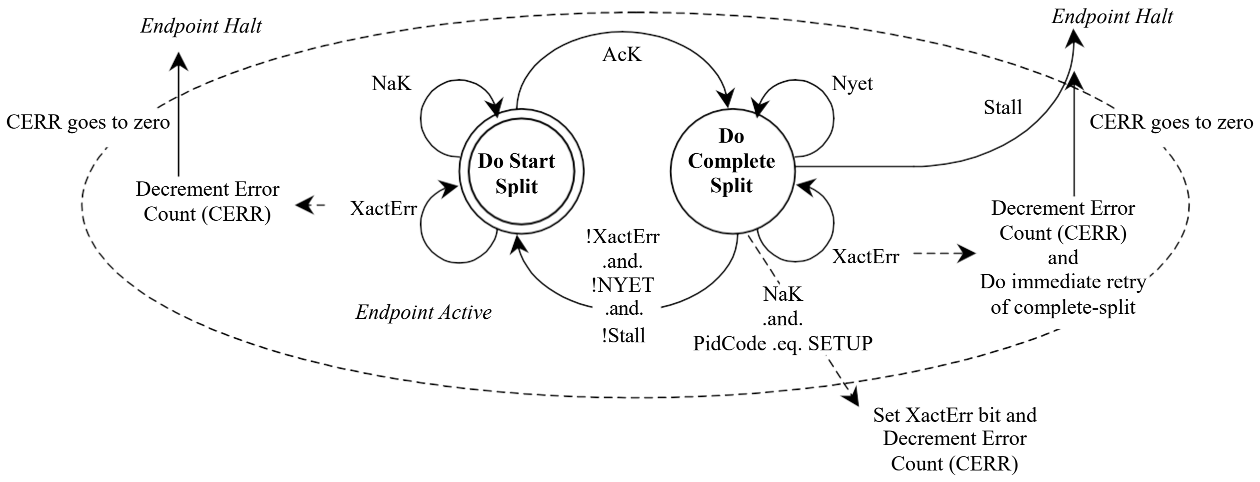

Endpoint Halt
Endpoint Halt
AcK
NaK
Nyet
Stall
CERR goes to zero
CERR goes to zero
Do Start Split
Do Complete Split
Decrement Error Count (CERR)
XactErr
Decrement Error Count (CERR) and Do immediate retry of complete-split
XactErr
!XactErr .and. !NYET .and. !Stall
NaK .and. PidCode .eq. SETUP
Endpoint Active
Set XactErr bit and Decrement Error Count (CERR)

Figure 4-16. Host Controller Asynchronous Schedule Split-Transaction State Machine

4.12.1.1 Asynchronous - Do Start Split

This is the state which software must initialize a full- or low-speed asynchronous queue head. This state is entered from the **Do Complete Split** state only after a complete-split transaction receives a valid response from the transaction translator that is not a Nyet handshake.

For queue heads in this state, the host controller will execute a start-split transaction to the appropriate transaction translator. If the bus transaction completes without an error and *PidCode* indicates an IN or OUT transaction, then the host controller will reload the error counter (*CErr*). If it is a successful bus transaction and the *PidCode* indicates a SETUP, the host controller will not reload the error counter. If the transaction translator responds with a Nak, the queue head is left in this state, and the host controller proceeds to the next queue head in the asynchronous schedule.

If the host controller times out the transaction (no response, or bad response) the host controller decrements *Cerr* and proceeds to the next queue head in the asynchronous schedule.

4.12.1.2 Asynchronous - Do Complete Split

This state is entered from the **Do Start Split** state only after a start-split transaction receives an Ack handshake from the transaction translator.

For queue heads in this state, the host controller will execute a complete-split transaction to the appropriate transaction translator. If the transaction translator responds with a Nyet handshake, the queue head is left in this state, the error counter is reset and the host controller proceeds to the next queue head in the asynchronous schedule. When a Nyet handshake is received for a bus transaction where the queue head's *PidCode* indicates an IN or OUT, the host controller will reload the error counter (*CErr*). When a Nyet handshake is received for a complete-split bus transaction where the queue head's *PidCode* indicates a SETUP, the host controller must not adjust the value of *CErr*.

Independent of *PIDCode*, the following responses have the effects:

- Transaction Error (XactErr). Timeout or data CRC failure, etc. The error counter (*Cerr*) is decremented by one and the complete split transaction is *immediately* retried (if possible). If there is not enough time in the micro-frame to execute the retry, the host controller MUST ensure that the next time the host controller begins executing from the Asynchronous schedule, it must begin executing from this queue head. If another start-split (for some other endpoint) is sent to the transaction translator before the complete-split is really completed, the transaction translator could dump the results (which were never delivered to the host). This is why the core specification states the retries must be immediate. A method to accomplish this behavior is to not advance the asynchronous schedule. When the host controller returns to the asynchronous schedule in the next micro-frame, the first transaction from the schedule will be the retry for this endpoint. If *Cerr* went to zero, the host controller must halt the queue.
- NAK. The target endpoint Nak'd the full- or low-speed transaction. The state of the transfer is not advanced and the state is exited.

  If the *PidCode* is a SETUP, then the Nak response is a protocol error. The *XactErr* status bit is set to a one and the *CErr* field is decremented.
- STALL. The target endpoint responded with a STALL handshake. The host controller sets the *halt* bit in the status byte, retires the qTD but does not attempt to advance the queue.

If the *PidCode* indicates an IN, then any of following responses are expected:

- DATA0/1. On reception of data, the host controller ensures the PID matches the expected data toggle and checks CRC. If the packet is *good*, the host controller will advance the state of the transfer, e.g. move the data pointer by the number of bytes received, decrement *BytesToTransfer* field by the number of bytes received, and toggle the *dt* bit. The host controller will then exit this state. The response and advancement of transfer may trigger other processing events, such as retirement of the qTD and advancement of the queue.

  If the data sequence PID does not match the expected, the data is ignored, the transfer state is not advanced and this state is exited.

If the *PidCode* indicates an OUT/SETUP, then any of following responses are expected:

- ACK. The target endpoint accepted the data, so the host controller must advance the state of the transfer. The *Current Offset* field is incremented by *Maximum Packet Length* or *Bytes to Transfer*, whichever is less. The field *Bytes To Transfer* is decremented by the same amount and the data toggle bit (*dt*) is toggled. The host controller will then exit this state.

  Advancing the transfer state may cause other processing events such as retirement of the qTD and advancement of the queue (see Section 4.10).

4.12.2 Split Transaction Interrupt

Split-transaction Interrupt-IN/OUT endpoints are managed via the same data structures used for high-speed interrupt endpoints. They both co-exist in the periodic schedule. Queue heads/qTDs offer the set of features required for reliable data delivery, which is characteristic to interrupt transfer types. The split-transaction protocol is managed completely within this defined functional transfer framework. For example, for a high-speed endpoint, the host controller will visit a queue head, execute a high-speed transaction (if criteria are met) and advance the transfer state (or not) depending on the results of the entire transaction. For low- and full-speed endpoints, the details of the *execution* phase are different (i.e. takes more than one bus transaction to complete), but the remainder of the operational framework is intact. This means that the transfer advancement, etc. occurs as defined in Section 4.10, but only occurs on the completion of a split transaction.

4.12.2.1 Split Transaction Scheduling Mechanisms for Interrupt

Full- and low-speed Interrupt queue heads have an *EPS* field indicating full- or low-speed and have a non-zero *S-mask* field. The host controller can detect this combination of parameters and assume the endpoint is a periodic endpoint. Low- and full-speed interrupt queue heads require the use of the split transaction protocol. The host controller sets the Endpoint Type (ET) field in the split token to indicate the transaction is an interrupt. These transactions are managed through a transaction translator's periodic pipeline. Software should not set these fields to indicate the queue head is an interrupt unless the queue head is used in the periodic schedule.

System software manages the per/transaction translator periodic pipeline by budgeting and scheduling exactly during which micro-frames the start-splits and complete-splits for each endpoint will occur. The characteristics of the transaction translator are such that the high-speed transaction protocol must execute during explicit micro-frames, or the data or response information in the pipeline is lost. Figure 4-17 illustrates the general scheduling boundary conditions that are supported by the EHCI periodic schedule and queue head data structure. The **S** and **C**$_X$ labels indicate micro-frames where software can schedule start-splits and complete splits (respectively).

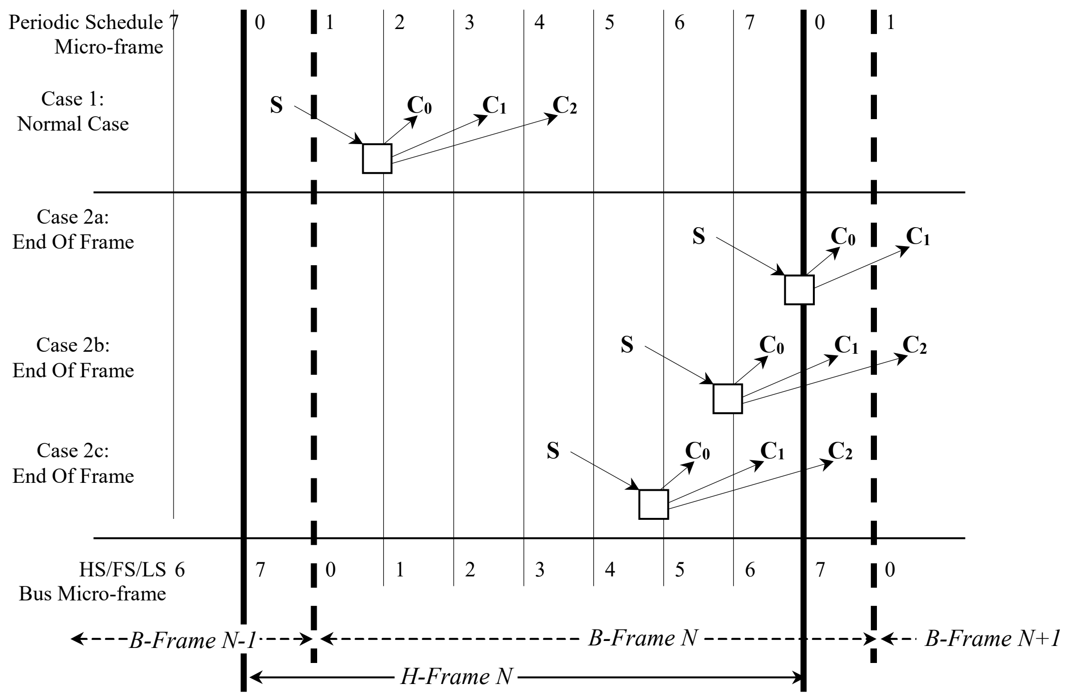


Figure 4-17. Split Transaction, Interrupt Scheduling Boundary Conditions

The scheduling cases are:

- Case 1: The normal scheduling case is where the entire split transaction is completely bounded by a frame (*H-Frame* in this case).
- Case 2a through Case 2c: The USB 2.0 Hub pipeline rules states clearly, when and how many complete-splits must be scheduled to account for earliest to latest execution on the full/low-speed link. The complete-splits may span the *H-Frame* boundary when the start-split is in micro-frame 4 or later. When this occurs, the *H-Frame* to *B-Frame* alignment requires that the queue head be reachable from consecutive periodic frame list locations. System software cannot build an efficient schedule that satisfies this requirement unless it uses FSTNs. Figure 4-18 illustrates the general layout of the periodic schedule.

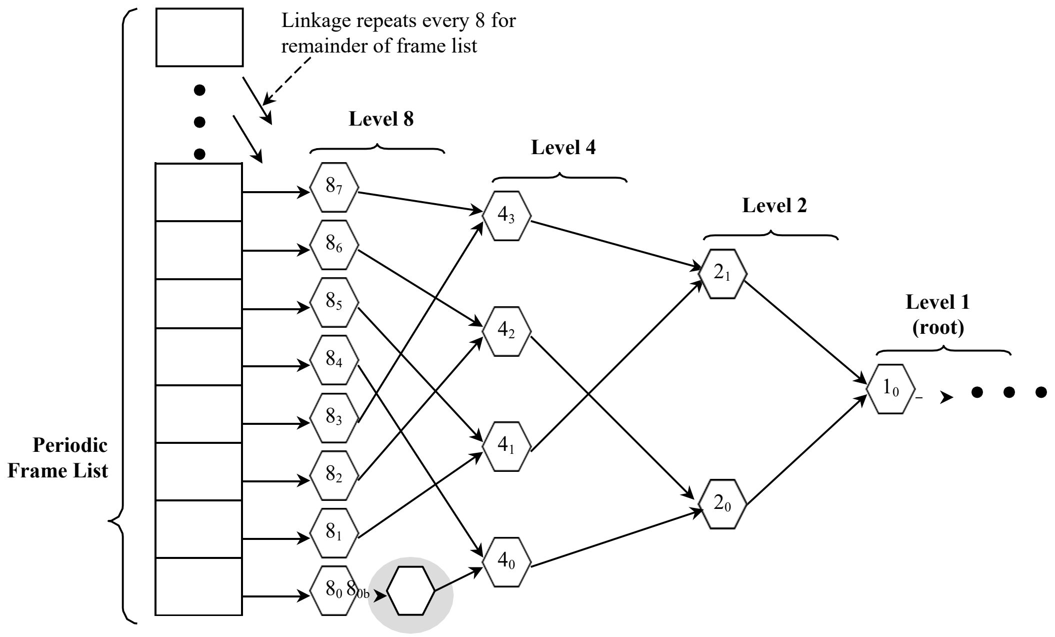


Figure 4-18. General Structure of EHCI Periodic Schedule Utilizing Interrupt Spreading

The periodic frame list is effectively the leaf level a binary tree, which is always traversed leaf to root. Each level in the tree corresponds to a 2^N poll rate. Software can efficiently manage periodic bandwidth on the USB by *spreading* interrupt queue heads that have the same poll rate requirement across all the available paths from the frame list. For example, system software can schedule eight poll rate 8 queue heads and account for them once in the high-speed bus bandwidth allocation.

When an endpoint is allocated an execution footprint that spans a frame boundary, the queue head for the endpoint must be reachable from consecutive locations in the frame list. An example would be if 8_{0b} where such an endpoint. Without additional support on the interface, to get 8_{0b} reachable at the correct time, software would have to link 8_1 to 8_{0b}. It would then have to move 4_1 and everything linked after into the same path as 4_0. This upsets the integrity of the binary tree and disallows the use of the spreading technique.

FSTN data structures are used to preserve the integrity of the binary-tree structure and enable the use of the spreading technique. Section 4.12.2.2 defines the hardware and software operational model requirements for using FSTNs.

The following queue head fields are initialized by system software to instruct the host controller when to execute portions of the split-transaction protocol.

- *SplitXState*. This is a single bit residing in the *Status* field of a queue head (see Table 3-16). This bit is used to track the current state of the split transaction.
- µ*Frame S-mask*. This is a bit-field where-in system software sets a bit corresponding to the micro-frame (within an *H-Frame*) that the host controller should execute a start-split transaction. This is always

qualified by the value of the *SplitXState* bit in the *Status* field of the queue head. For example, referring to Figure 4-17, case one, the *S-mask* would have a value of 00000001b indicating that if the queue head is traversed by the host controller, and the *SplitXState* indicates **Do_Start**, and the current micro-frame as indicated by FRINDEX[2:0] is 0, then execute a start-split transaction.

- *µFrame C-mask.* This is a bit-field where system software sets one or more bits corresponding to the micro-frames (within an *H-Frame*) that the host controller should execute complete-split transactions. The interpretation of this field is always qualified by the value of the *SplitXState* bit in the *Status* field of the queue head. For example, referring to Figure 4-17, case one, the *C-mask* would have a value of 00011100b indicating that if the queue head is traversed by the host controller, and the *SplitXState* indicates **Do_Complete**, and the current micro-frame as indicated by FRINDEX[2:0] is 2, 3, or 4, then execute a complete-split transaction.

It is software's responsibility to ensure that the translation between *H-Frames* and *B-Frames* is correctly performed when setting bits in *S-mask* and *C-mask*

4.12.2.2 Host Controller Operational Model for FSTNs

The FSTN data structure is used to manage Low/Full-speed interrupt queue heads that need to be reached from consecutive frame list locations (i.e. boundary cases 2a through 2c). An FSTN is essentially a *back pointer*, similar in intent to the back pointer field in the siTD data structure (see Section 3.4.5). This feature provides software a simple primitive to save a schedule position, redirect the host controller to traverse the necessary queue heads in the previous frame, then restore the original schedule position and complete normal traversal.

There are four components to the use of FSTNs:

1. FSTN data structure, defined in Section 3.7.
2. A *Save Place* indicator. This is always an FSTN with its *Back Path Link Pointer. T-bit* set to zero.
3. A *Restore* indicator. This is always an FSTN with its *Back Path Link Pointer.T-bit* set to a one.
4. Host controller FSTN traversal rules.

4.12.2.2.1 Host Controller Operational Model for FSTNs

When the host controller encounters an FSTN during micro-frames 2 through 7 it simply follows the node's *Normal Path Link Pointer* to access the next schedule data structure. Note that the FSTN's *Normal Path Link Pointer.T-bit* may set to a one, which the host controller must interpret as the end of periodic list mark.

When the host controller encounters a *Save-Place* FSTN in micro-frames 0 or 1, it will save the value of the *Normal Path Link Pointer* and set an internal flag indicating that it is executing in *Recovery Path* mode. *Recovery Path* mode modifies the host controller's rules for how it traverses the schedule and limits which data structures will be considered for execution of bus transactions. The host controller continues executing in *Recovery Path* mode until it encounters a *Restore* FSTN or it determines that it has reached the end of the micro-frame (see details in the list below). The rules for schedule traversal and limited execution while in *Recovery Path* mode are:

- Always follow the *Normal Path Link Pointer* when it encounters an FSTN that is a *Save-Place* indicator. The host controller must not recursively follow *Save-Place* FSTNs. Therefore, while executing in *Recovery Path* mode, it must never follow an FSTN's *Back Path Link Pointer*.
- Do not process an siTD or, iTD data structure. Simply follow its *Next Link Pointer*.
- Do not process a QH (Queue Head) whose *EPS* field indicates a high-speed device. Simply follow its *Horizontal Link Pointer*.
- When a QH's *EPS* field indicates a Full/Low-speed device, the host controller will only consider it for execution if its *SplitXState* is **DoComplete** (note: this applies whether the *PID Code* indicates an IN or an OUT). See Sections 4.10.3 and 4.12.2.3 for a complete list of additional conditions that

must be met in general for the host controller to issue a bus transaction. Note that the host controller must not execute a Start-split transaction while executing in *Recovery Path* mode. See Section 4.12.2.4.2 for special handling when in *Recovery Path* mode.

- Stop traversing the *recovery path* when it encounters an FSTN that is a *Restore* indicator. The host controller unconditionally uses the saved value of the *Save-Place* FSTN's *Normal Path Link Pointer* when returning to the normal path traversal. The host controller must clear the context of executing a *Recovery Path* when it restores schedule traversal to the *Save-Place* FSTN's *Normal Path Link Pointer*.

  If the host controller determines that there is not enough time left in the micro-frame to complete processing of the periodic schedule, it abandons traversal of the recovery path, and clears the context of executing a recovery path. The result is that at the start of the next consecutive micro-frame, the host controller starts traversal at the frame list.

An example traversal of a periodic schedule that includes FSTNs is illustrated in Figure 4-19.

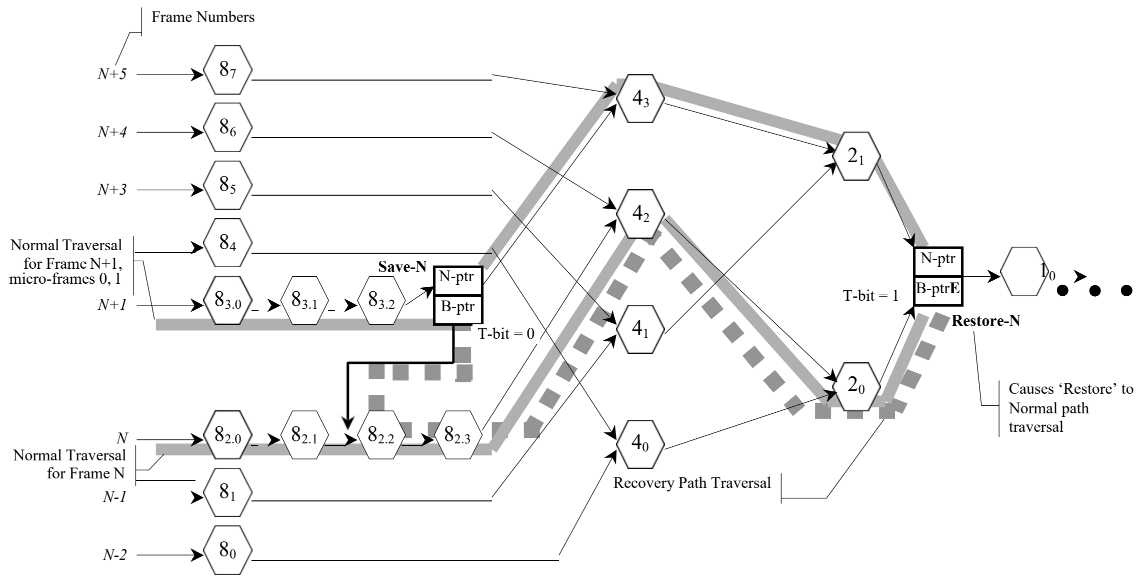


Figure 4-19. Example Host Controller Traversal of Recovery Path via FSTNs

In frame N (micro-frames 0-7), for this example, the host controller will traverse all of the schedule data structures utilizing the *Normal Path Link Pointers* in any FSTNs it encounters. This is because the host controller has not yet encountered a *Save-Place* FSTN so it not executing in *Recovery Path* mode. When it encounters the *Restore* FSTN, (Restore-N), during micro-frames 0 and 1, it uses Restore-N.Normal Path Link Pointer to traverse to the next data structure (i.e. normal schedule traversal). This is because the host controller must use a Restore FSTN's *Normal Path Link Pointer* when not executing in a *Recovery-Path* mode. The nodes traversed during frame N include: {$8_{2.0}$, $8_{2.1}$, $8_{2.2}$, $8_{2.3}$, 4_2, 2_0, Restore-N, 1_0 …}.

In frame N+1 (micro-frames 0 and 1), when the host controller encounters Save-Path FSTN (Save-N), it observes that Save-N.Back Path Link Pointer.T-bit is zero (definition of a Save-Path indicator). The host controller saves the value of Save-N.Normal Path Link Pointer and follows Save-N.Back Path Link Pointer. At the same time, it sets an internal flag indicating that it is now in *Recovery Path* mode (the recovery path is annotated in Figure 4-19 with a large dashed line). The host controller continues traversing data structures on the recovery path and executing only those bus transactions as noted above, on the recovery path until it reaches Restore FSTN (Restore-N). Restore-N.Back Path Link Pointer.T-bit is set to a one (definition of a Restore indicator), so the host controller exits *Recovery Path* mode by clearing the internal *Recovery Path* mode flag and commences (restores) schedule traversal using the saved value of the *Save-Place* FSTN's *Normal Path Link Pointer* (e.g. Save-N.Normal Path Link Pointer). The nodes traversed during these micro-frames include: {$8_{3.0}$, $8_{3.1}$, $8_{3.2}$, Save-A, $\mathbf{8_{2.2}}$, $\mathbf{8_{2.3}}$, $\mathbf{4_2}$, $\mathbf{2_0}$, **Restore-N**, 4_3, 2_1, Restore-N, 1_0 …}. The nodes on the recovery-path are bolded.

In frame N+1 (micro-frames 2-7), when the host controller encounters Save-Path FSTN Save-N, it will unconditionally follow Save-N.Normal Path Link Pointer. The nodes traversed during these micro-frames include: {$8_{3.0}$, $8_{3.1}$, $8_{3.2}$, Save-A, 4_3, 2_1, Restore-N, 1_0 …}.

4.12.2.2.2 Software Operational Model for FSTNs

Software must create a consistent, coherent schedule for the host controller to traverse. When using FSTNs, system software must adhere to the following rules:

- Each *Save-Place* indicator requires a matching *Restore* indicator.

  The *Save-Place* indicator is an FSTN with a valid *Back Path Link Pointer and T-bit* equal to zero. Note that *Back Path Link Pointer.Typ* field must be set to indicate the referenced data structure is a queue head. The *Restore* indicator is an FSTN with its *Back Path Link Pointer.T-bit* set to a one.

  A *Restore* FSTN may be matched to one or more *Save-Place* FSTNs. For example, if the schedule includes a poll-rate 1 level, then system software only needs to place a *Restore* FSTN at the beginning of this list in order to match all possible *Save-Place* FSTNs.

- If the schedule does not have elements linked at a poll-rate level of one, and one or more *Save-Place* FSTNs are used, then System Software must ensure the *Restore* FSTN's *Normal Path Link Pointer*'s *T-bit* is set to a one, as this will be use to mark the end of the periodic list.

- When the schedule does have elements linked at a poll rate level of one, a *Restore* FSTN must be the first data structure on the poll rate one list. All traversal paths from the frame list converge on the poll-rate one list. System software must ensure that *Recovery Path* mode is exited before the host controller is allowed to traverse the poll rate level one list.

- A *Save-Place* FSTN's *Back Path Link Pointer* must reference a queue head data structure. The referenced queue head must be reachable from the previous frame list location. In other words, if the *Save-Place* FSTN is reachable from frame list offset N, then the FSTN's *Back Path Link Pointer* must reference a queue head that is reachable from frame list offset N-1.

Software should make the schedule as efficient as possible. What this means in this context is that software should have no more than one *Save-Place* FSTN reachable in any single frame. Note there will be times when two (or more, depending on the implementation) could exist as full/low-speed footprints change with bandwidth adjustments. This could occur, for example when a bandwidth rebalance causes system software to move the *Save-Place* FSTN from one poll rate level to another. During the transition, software must preserve the integrity of the previous schedule until the new schedule is in place.

4.12.2.3 Tracking Split Transaction Progress for Interrupt Transfers

To correctly maintain the data stream, the host controller must be able to detect and report errors where data is lost. For interrupt-IN transfers, data is lost when it makes it into the USB 2.0 hub, but the USB 2.0 host system is unable to get it from the USB 2.0 Hub and into the system before it expires from the transaction translator pipeline. When a lost data condition is detected, the queue must be halted, thus signaling system software to recover from the error. A data-loss condition exists whenever a start-split is issued, accepted and successfully executed by the USB 2.0 Hub, but the complete-splits get unrecoverable errors on the high-speed link, or the complete-splits do not occur at the correct times. One reason complete-splits might not occur at the right time would be due to host-induced system hold-offs that cause the host controller to miss bus transactions because it cannot get timely access to the schedule in system memory.

The same condition can occur for an interrupt-OUT, but the result is not an endpoint halt condition, but rather effects only the progress of the transfer.

The queue head has the following fields to track the progress of each split transaction. These fields are used to keep incremental state about which (and when) portions have been executed.

- *C-prog-mask*. This is an eight-bit bit-vector where the host controller keeps track of which complete-splits have been executed. Due to the nature of the Transaction Translator periodic pipeline, the complete-splits need to be executed in-order. The host controller needs to detect when the complete-splits have not been executed in order. This can only occur due to system hold-offs where the host controller cannot get to the memory-based schedule. *C-prog-mask* is a simple bit-vector that the host controller sets one of the *C-prog*-mask bits for each complete-split executed. The bit position is determined by the micro-frame number in which the complete-split was executed. The host controller always checks *C-prog-mask* before executing a complete-split transaction. If the previous complete-splits have not been executed then it means one (or more) have been skipped and data has potentially been lost.

- *FrameTag*. This field is used by the host controller during the complete-split portion of the split transaction to tag the queue head with the frame number (*H-Frame* number) when the next complete split must be executed.
- *S-bytes*. This field can be used to store the number of data payload bytes sent during the start-split (if the transaction was an OUT). The *S-bytes* field must be used to accumulate the data payload bytes received during the complete-splits (for an IN).

4.12.2.4 Split Transaction Execution State Machine for Interrupt

In the following presentation, all references to micro-frame are in the context of a micro-frame within an *H-Frame.*

As with asynchronous Full- and Low-speed endpoints, a split-transaction state machine is used to manage the split transaction sequence. Aside from the fields defined in the queue head for scheduling and tracking the split transaction, the host controller calculates one internal mechanism that is also used to manage the split transaction. The internal calculated mechanism is:

- *cMicroFrameBit*. This is a single-bit encoding of the current micro-frame number. It is an eight-bit value calculated by the host controller at the beginning of every micro-frame. It is calculated from the three least significant bits of the *FRINDEX* register (i.e. *cMicroFrameBit* = (1 shifted-left(*FRINDEX*[2:0]))). The *cMicroFrameBit* has at most one bit asserted, which always corresponds to the current micro-frame number. For example, if the current micro-frame is 0, then *cMicroFrameBit* will equal 00000001b.

  The variable *cMicroFrameBit* is used to compare against the *S-mask* and *C-mask* fields to determine whether the queue head is marked for a start- or complete-splt transaction for the current micro-frame.

Figure 4-20 illustrates the state machine for managing a complete interrupt split transaction. There are two phases to each split transaction. The first is a single start-split transaction, which occurs when the *SplitXState* is at **Do_Start** and the single bit in *cMicroFrameBit* has a corresponding bit active in *QH.S-mask*. The transaction translator does not acknowledge the receipt of the periodic start-split, so the host controller unconditionally transitions the state to **Do_Complete**. Due to the available jitter in the transaction translator pipeline, there will be more than one complete-split transaction scheduled by software for the **Do_Complete** state. This translates simply to the fact that there are multiple bits set to a one in the *QH.C-mask* field.

The host controller keeps the queue head in the **Do_Complete** state until the split transaction is complete (see definition below), or an error condition triggers the *three-strikes-rule* (e.g. after the host tries the same transaction three times, and each encounters an error, the host controller will stop retrying the bus transaction and halt the endpoint, thus requiring system software to detect the condition and perform system-dependent recovery).

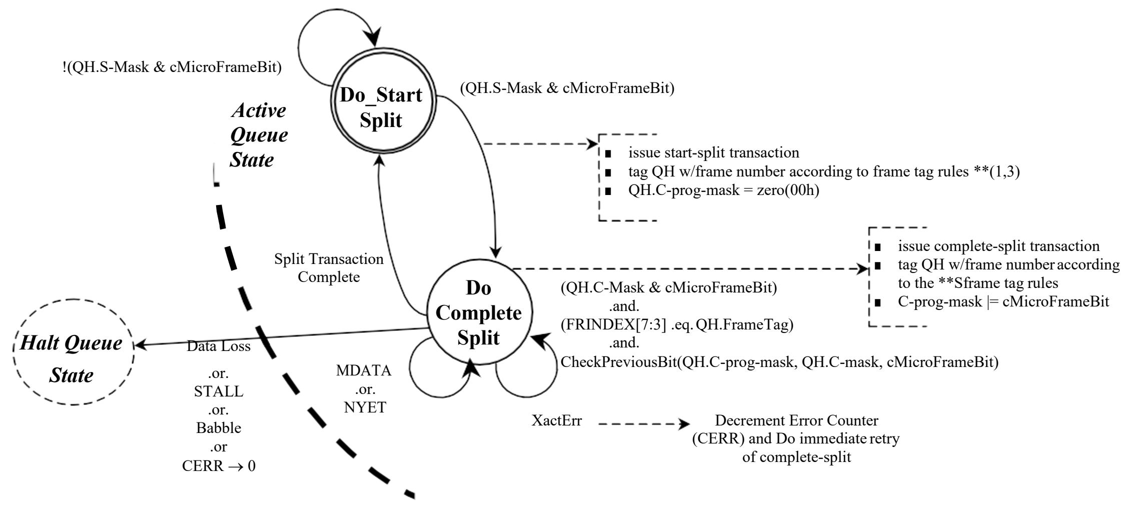

!(QH.S-Mask & cMicroFrameBit)
Do_Start Split
(QH.S-Mask & cMicroFrameBit)
Active Queue State
issue start-split transaction
tag QH w/frame number according to frame tag rules **(1,3)
QH.C-prog-mask = zero(00h)
issue complete-split transaction
tag QH w/frame number according to the **Sframe tag rules
C-prog-mask |= cMicroFrameBit
Split Transaction Complete
Do Complete Split
(QH.C-Mask & cMicroFrameBit) .and. (FRINDEX[7:3] .eq. QH.FrameTag) .and. CheckPreviousBit(QH.C-prog-mask, QH.C-mask, cMicroFrameBit)
Halt Queue State
Data Loss .or. STALL .or. Babble .or CERR → 0
MDATA .or. NYET
XactErr
Decrement Error Counter (CERR) and Do immediate retry of complete-split

Figure 4-20. Split Transaction State Machine for Interrupt

**See Section 4.12.2.4.3 for the frame tag management rules.

4.12.2.4.1 Periodic Interrupt - Do Start Split

This is the state software must initialize a full- or low-speed interrupt queue head *StartXState* bit. This state is entered from the **Do_Complete Split** state only after the split transaction is complete. This occurs when one of the following events occur:

The transaction translator responds to a complete-split transaction with one of the following:

- NAK. A NAK response is a propagation of the full- or low-speed endpoint's NAK response.
- ACK. An ACK response is a propagation of the full- or low-speed endpoint's ACK response. Only occurs on an OUT endpoint.
- DATA 0/1. Only occurs for INs. Indicates that this is the last of the data from the endpoint for this split transaction.
- ERR. The transaction on the low-/full-speed link below the transaction translator had a failure (e.g. timeout, bad CRC, etc.).
- NYET (and Last). The host controller issued the last complete-split and the transaction translator responded with a NYET handshake. This means that the start-split was not correctly received by the transaction translator, so it never executed a transaction to the full- or low-speed endpoint, see Section 4.12.2.4.2 for the definition of 'Last'.

Each time the host controller visits a queue head in this state (once within the **Execute Transaction** state), it performs the following test to determine whether to execute a start-split.

- *QH.S-mask* is bit-wise anded with *cMicroFrameBit.*

If the result is non-zero, then the host controller will issue a start-split transaction. If the *PIDCode* field indicates an IN transaction, the host controller must zero-out the *QH.S-bytes* field. After the split-transaction has been executed, the host controller sets up state in the queue head to track the progress of the complete-split phase of the split transaction. Specifically, it records the expected frame number into *QH.FrameTag* field (see Section 4.12.2.4.3), set *C-prog-mask* to zero (00h), and exits this state. Note that the host controller must not adjust the value of *CErr* as a result of completion of a start-split transaction.

4.12.2.4.2 Periodic Interrupt - Do Complete Split

This state is entered unconditionally from the **Do Start Split** state after a start-split transaction is executed on the bus. Each time the host controller visits a queue head in this state (once within the **Execute Transaction** state), it checks to determine whether a complete-split transaction should be executed now.

There are four tests to determine whether a complete-split transaction should be executed.

- **Test A**. *cMicroFrameBit* is bit-wise anded with *QH.C-mask* field. A non-zero result indicates that software scheduled a complete-split for this endpoint, during this micro-frame.
- **Test B**. *QH.FrameTag* is compared with the current contents of *FRINDEX*[7:3]. An equal indicates a match.
- **Test C**. The complete-split progress bit vector is checked to determine whether the previous bit is set, indicating that the previous complete-split was appropriately executed. An example algorithm for this test is provided below:

```
Algorithm Boolean CheckPreviousBit(QH.C-prog-mask, QH.C-mask, cMicroFrameBit)
Begin
  -- Return values:
  -- TRUE - no error
  -- FALSE - error
  --

  Boolean rvalue = TRUE;

  previousBit = cMicroframeBit logical-rotate-right(1)

  -- Bit-wise anding previousBit with C-mask indicates whether there was an intent
  -- to send a complete split in the previous micro-frame. So, if the
  -- 'previous bit' is set in C-mask, check C-prog-mask to make sure it
  -- happened.

  If (previousBit bitAND QH.C-mask)then
     If not(previousBit bitAND QH.C-prog-mask) then
        rvalue = FALSE;
     End if
  End If

  -- If the C-prog-mask already has a one in this bit position, then an aliasing
  -- error has occurred. It will probably get caught by the FrameTag Test, but
  -- at any rate it is an error condition that as detectable here should not allow
  -- a transaction to be executed.

  If (cMicroFrameBit bitAND QH.C-prog-mask) then
     rvalue = FALSE;
  End if
  return (rvalue)
End Algorithm
```

- **Test D.** Check to see if a start-split should be executed in this micro-frame. Note this is the same test performed in the **Do Start Split** state (see Section 4.12.2.4.1). Whenever it evaluates to TRUE and the controller is NOT processing in the context of a *Recovery Path* mode, it means a start-split should occur in this micro-frame. Test D and Test A evaluating to TRUE at the same time is a system software error. Behavior is undefined.

If (A .and. B .and. C .and. not(D)) then the host controller will execute a complete-split transaction. When the host controller commits to executing the complete-split transaction, it updates *QH.C-prog-mask* by bit-ORing with *cMicroFrameBit*. On completion of the complete-split transaction, the host controller records the result of the transaction in the queue head and sets *QH.FrameTag* to the expected *H-Frame* number (see Section 4.12.2.4.3). The effect to the state of the queue head and thus the state of the transfer depends on the response by the transaction translator to the complete-split transaction. The following responses have the effects (note that any responses that result in decrementing of the *CErr* will result in the queue head being halted by the host controller if the result of the decrement is zero):

- NYET (and Last). On each NYET response, the host controller checks to determine whether this is the last complete-split for this split transaction. Last is defined in this context as the condition where all of the scheduled complete-splits have been executed. If it is the last complete-split (with a NYET response), then the transfer state of the queue head is not advanced (never received any data) and this state exited. The transaction translator must have responded to all the clompete-splits with NYETs, meaning that the start-split issued by the host controller was not received. The start-split should be retried at the next poll period.

  The test for whether this is the **Last** complete split can be performed by XOR *QH.C-mask* with *QH.C-prog-mask*. If the result is all zeros then all complete-splits have been executed. When this condition occurs, the *XactErr* status bit is set to a one and the *CErr* field is decremented.

- NYET (and not Last). See above description for testing for **Last**. The complete-split transaction received a NYET response from the transaction translator. Do not update any transfer state (except for *C-prog-mask* and *FrameTag*) and stay in this state. The host controller must not adjust *CErr* on this response.

- Transaction Error (XactErr). Timeout, data CRC failure, etc. The *CErr* field is decremented and the *XactErr* bit in the *Status* field is set to a one. The complete split transaction is *immediately* retried (if *Cerr* is non-zero). If there is not enough time in the micro-frame to complete the retry and the endpoint is an IN, or *CErr* is decremented to a zero from a one, the queue is halted. If there is not enough time in the micro-frame to complete the retry and the endpoint is an OUT and *CErr* is not zero, then this state is exited (i.e. return to **Do Start Split**). This results in a retry of the entire OUT split transaction, at the next poll period. Refer to Chapter 11 Hubs (specifically the section full- and low-speed Interrupts) in the USB Specification Revision 2.0 for detailed requirements on why these errors must be immediately retried.

- ACK. This can only occur if the target endpoint is an OUT. The target endpoint ACK'd the data and this response is a propagation of the endpoint ACK up to the host controller. The host controller must advance the state of the transfer. The *Current Offset* field is incremented by *Maximum Packet Length* or *Bytes to Transfer*, whichever is less. The field *Bytes To Transfer* is decremented by the same amount. And the data toggle bit (*dt*) is toggled. The host controller will then exit this state for this queue head. The host controller must reload *CErr* with maximum value on this response.

  Advancing the transfer state may cause other process events such as retirement of the qTD and advancement of the queue (see Section 4.10).

- MDATA. This response will only occur for an IN endpoint. The transaction translator responded with zero or more bytes of data and an MDATA PID. The incremental number of bytes received is accumulated in *QH.S-bytes*. The host controller must not adjust *CErr* on this response.

- DATA0/1. This response may only occur for an IN endpoint. The number of bytes received is added to the accumulated byte count in *QH.S-bytes*. The state of the transfer is advanced by the result and the host controller will exit this state for this queue head.

  Advancing the transfer state may cause other processing events such as retirement of the qTD and advancement of the queue (see Section 4.10).

  If the data sequence PID does not match the expected, the entirety of the data received in this split transaction is ignored, the transfer state is not advanced and this state is exited.

- NAK. The target endpoint Nak'd the full- or low-speed transaction. The state of the transfer is not advanced, and this state is exited. The host controller must reload *CErr* with maximum value on this response.

- ERR. There was an error during the full- or low-speed transaction. The ERR status bit is set to a one, *Cerr* is decremented, the state of the transfer is not advanced, and this state is exited.

- STALL. The queue is halted (an exit condition of the **Execute Transaction** state). The status field bits: *Active* bit is set to zero and the *Halted* bit is set to a one and the qTD is retired.

Responses which are not enumerated in the list or which are received out of sequence are illegal and may result in undefined host controller behavior.

The other possible combinations of tests A, B, C, and D may indicate that data or response was lost. Table 4–13 lists the possible combinations and the appropriate action.

Table 4–13. Interrupt IN/OUT Do Complete Split State Execution Criteria

Condition	Action	Description

not(A)
not(D)
	Ignore QHD	Neither a start nor complete-split is scheduled for the current micro-frame. Host controller should continue walking the schedule.

A not(C)	If PIDCode = IN Halt QHD If PIDCode = OUT Retry start-split	Progress bit check failed. These means a complete-split has been missed. There is the possibility of lost data. If *PIDCode* is an IN, then the Queue head must be halted. If *PIDCode* is an OUT, then the transfer state is not advanced and the state exited (e.g. start-split is retried). This is a host-induced error and does not effect *CERR*. In either case, set the *Missed Micro-frame* bit in the status field to a one.
A not(B) C	If PIDCode = IN Halt QHD If PIDCode = OUT Retry start-split	*QH.FrameTag* test failed. This means that exactly one or more *H-Frames* have been skipped. This means complete-splits and have missed. There is the possibility of lost data. If *PIDCode* is an IN, then the Queue head must be halted. If *PIDCode* is an OUT, then the transfer state is not advanced and the state exited (e.g. start-split is retried). This is a host-induced error and does not effect *CERR*. In either case, set the *Missed Micro-frame* bit in the status field to a one.
A B C not(D)	Execute complete-split	This is the non-error case where the host controller executes a complete-split transaction.
D	If PIDCode = IN Halt QHD If PIDCode = OUT Retry start-split	This is a degenerate case where the start-split was issued, but all of the complete-splits were skipped and all possible intervening opportunities to detect the missed data failed to fire. If *PIDCode* is an IN, then the Queue head must be halted. If *PIDCode* is an OUT, then the transfer state is not advanced and the state exited (e.g. start-split is retried). This is a host-induced error and does not effect *CERR*. In either case, set the *Missed Micro-frame* bit in the status field to a one. Note: When executing in the context of a *Recovery Path* mode, the host controller is allowed to process the queue head and take the actions indicated above, or it may wait until the queue head is visited in the normal processing mode. Regardless, the host controller must not execute a start-split in the context of a executing in a *Recovery Path* mode.

4.12.2.4.3 Managing *QH.FrameTag* Field

The *QH.FrameTag* field in a queue head is completely managed by the host controller. The rules for setting *QH.FrameTag* are simple:

- **Rule 1:** If transitioning from **Do Start Split** to **Do Complete Split** and the current value of *FRINDEX*[2:0] is 6 *QH.FrameTag* is set to *FRINDEX*[7:3] + 1. This accommodates split transactions whose start-split and complete-splits are in different *H-Frames* (case 2a, see Figure 4-17).
- **Rule 2:** If the current value of *FRINDEX*[2:0] is 7, *QH.FrameTag* is set to *FRINDEX*[7:3] + 1. This accommodates staying in **Do Complete Split** for cases 2a, 2b, and 2c (Figure 4-17).
- **Rule 3:** If transitioning from **Do_Start Split** to **Do Complete Split** and the current value of *FRINDEX*[2:0] is not 6, or currently in **Do Complete Split** and the current value of (*FRINDEX*[2:0]) is not 7, *FrameTag* is set to *FRINDEX*[7:3]. This accommodates all other cases (Figure 4-17).

4.12.2.5 Rebalancing the Periodic Schedule

System software must occasionally adjust a periodic queue head's S-mask and C-mask fields during operation. This need occurs when adjustments to the periodic schedule create a new bandwidth budget and one or more queue head's are assigned new execution footprints (i.e. new S-mask and C-mask values). It is imperative that System software must not update these masks to new values in the midst of a split transaction. In order to avoid any race conditions with the update, the EHCI host controller provides a simple assist to system software.

System software sets the *Inactivate-on-next-Transaction* (*I*) bit to a one to signal the host controller that it intends to update the S-mask and C-mask on this queue head. System software will then wait for the host controller to observe the *I-bit* is a one and transition the *Active* bit to a zero. The rules for how and when the host controller sets the *Active* bit to zero are enumerated below:

- If the *Active* bit is a zero, no action is taken. The host controller does not attempt to advance the queue when the *I-bit* is a one.
- If the *Active* bit is a one and the *SplitXState* is **DoStart** (regardless of the value of *S-mask*), the host controller will simply set *Active* bit to a zero. The host controller is not required to write the transfer state back to the *current* qTD. Note that if the *S-mask* indicates that a start-split is scheduled for the current micro-frame, the host controller must not issue the start-split bus transaction. It must set the *Active* bit to zero.

System software must save transfer state before setting the *I-bit* to a one. This is required so that it can correctly determine what transfer progress (if any) occurred after the *I-bit* was set to a one and the host controller executed it's final bus-transaction and set *Active* to a zero.

After system software has updated the S-mask and C-mask, it must then reactivate the queue head. Since the *Active* bit and the *I-bit* cannot be updated with the same write, system software needs to use the following algorithm to coherently re-activate a queue head that has been stopped via the *I-bit*.

1. Set the *Halted* bit to a one, then
2. Set the *I-bit* to a zero, then
3. Set the *Active* bit to a one and the *Halted* bit to a zero in the same write.

Setting the *Halted* bit to a one inhibits the host controller from attempting to advance the queue between the time the *I-bit* goes to a zero and the *Active* bit goes to a one.

4.12.3 Split Transaction Isochronous

Full-speed isochronous transfers are managed using the split-transaction protocol through a USB 2.0 transaction translator in a USB2.0 Hub. The EHCI controller utilizes siTD data structure to support the special requirements of isochronous split-transactions. This data structure uses the scheduling model of isochronous TDs (iTD, Section 3.3) (see Section 4.7 for the operational model of iTDs) with the contiguous data feature provided by queue heads. This simple arrangement allows a single isochronous scheduling model and adds the additional feature that all data received from the endpoint (per split transaction) must land into a contiguous buffer.

4.12.3.1 Split Transaction Scheduling Mechanisms for Isochronous

Full-speed isochronous transactions are managed through a transaction translator's periodic pipeline. As with full- and low-speed interrupt, system software manages each transaction translator's periodic pipeline by budgeting and scheduling exactly during which micro-frames the start-splits and complete-splits for each full-speed isochronous endpoint occur. The requirements described in Section 4.12.2.1 apply. Figure 4-21 illustrates the general scheduling boundary conditions that are supported by the EHCI periodic schedule. The **S**$_X$ and **C**$_X$ labels indicate micro-frames where software can schedule start- and complete-splits (respectively). The *H-Frame* boundaries are marked with a large, solid bold vertical line. The *B-Frame* boundaries are marked with a large, bold, dashed line. The bottom of the figure illustrates the relationship of an siTD to the *H-Frame*.

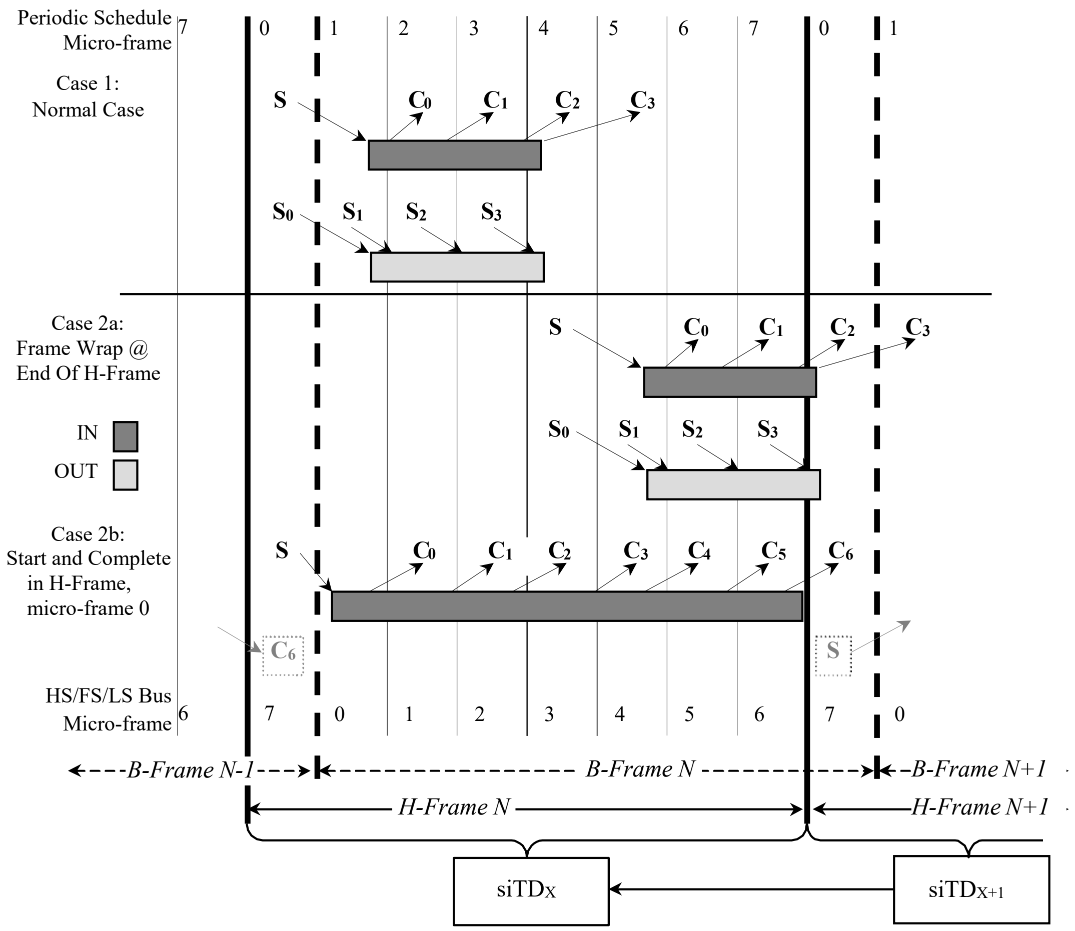


Figure 4-21. Split Transaction, Isochronous Scheduling Boundary Conditions

When the endpoint is an isochronous OUT, there are only start-splits, and no complete-splits. When the endpoint is an isochronous IN, there is at most one start-split and one to *N* complete-splits. The scheduling boundary cases are:

- *Case 1*: The entire split transaction is completely bounded by an *H-Frame.* For example: the start-splits and complete-splits are all scheduled to occur in the same *H-Frame*.
- *Case 2a*: This boundary case is where one or more (at most two) complete-splits of a split transaction IN are scheduled across an *H-Frame* boundary. This can only occur when the split transaction has the possibility of moving data in *B-Frame,* micro-frames 6 or 7 (*H-Frame* micro-frame 7 or 0). When an *H-Frame* boundary wrap condition occurs, the scheduling of the split transaction spans more than one location in the periodic list. (e.g. it takes two siTDs in adjacent periodic frame list locations to fully describe the scheduling for the split transaction).

  Although the scheduling of the split transaction may take two data structures, all of the complete-splits for each full-speed IN isochronous transaction must use only one data pointer. For this reason, siTDs contain a back pointer, the use of which is described below.

  Software must never schedule full-speed isochronous OUTs across an *H-Frame* boundary.
- *Case 2b*: This case can only occur for a very large isochronous IN. It is the only allowed scenario where a start-split and complete-split for the same endpoint can occur in the same micro-frame. Software must enforce this rule by scheduling the large transaction first. Large is defined to be anything larger than 579 byte maximum packet size.

A subset of the same mechanisms employed by full- and low-speed interrupt queue heads are employed in siTDs to schedule and track the portions of isochronous split transactions. The following fields are initialized by system software to instruct the host controller when to execute portions of the split transaction protocol.

- *SplitXState*. This is a single bit residing in the *Status* field of an siTD (see Table 3-11). This bit is used to track the current state of the split transaction. The rules for managing this bit are described in Section 4.12.3.3.
- μ*Frame S-mask*. This is a bit-field where-in system software sets a bit corresponding to the micro-frame (within an *H-Frame*) that the host controller should execute a start-split transaction. This is always qualified by the value of the *SplitXState* bit. For example, referring to the IN example in Figure 4-21, case one, the *S-mask* would have a value of 00000001b indicating that if the siTD is traversed by the host controller, and the *SplitXState* indicates **Do Start Split**, and the current micro-frame as indicated by FRINDEX[2:0] is 0, then execute a start-split transaction.
- μ*Frame C-mask.* This is a bit-field where system software sets one or more bits corresponding to the micro-frames (within an *H-Frame*) that the host controller should execute complete-split transactions. The interpretation of this field is always qualified by the value of the *SplitXState* bit. For example, referring to the IN example in Figure 4-21, case one, the *C-mask* would have a value of 00111100b indicating that if the siTD is traversed by the host controller, and the *SplitXState* indicates **Do Complete Split**, and the current micro-frame as indicated by *FRINDEX*[2:0] is 2, 3, 4, or 5, then execute a complete-split transaction.
- *Back Pointer*. This field in a siTD is used to complete an IN split-transaction using the previous *H-Frame*'s siTD. This is only used when the scheduling of the complete-splits span an *H-Frame* boundary.

There exists a one-to-one relationship between a high-speed isochronous split transaction (including all start- and complete-splits) and one full-speed isochronous transaction. An siTD contains (amongst other things) buffer state and split transaction scheduling information. An siTD's buffer state always maps to one full-speed isochronous data payload. This means that for any full-speed transaction payload, a single siTD's data buffer must be used. This rule applies to both IN an OUTs. An siTD's scheduling information usually also maps to one high-speed isochronous split transaction. The exception to this rule is the *H-Frame* boundary wrap cases mentioned above.

The siTD data structure describes at most, one frame's worth of high-speed transactions and that description is strictly bounded within a frame boundary. Figure 4-22 illustrates some examples. On the top are examples of the full-speed transaction footprints for the boundary scheduling cases described above. In the middle are time-frame references for both the *B-Frames* (HS/FS/LS Bus) and the *H-Frames*. On the bottom is illustrated the relationship between the scope of an siTD description and the time references. Each *H-Frame* corresponds to a single location in the periodic frame list. The implication is that each siTD is reachable from a single periodic frame list location at a time.

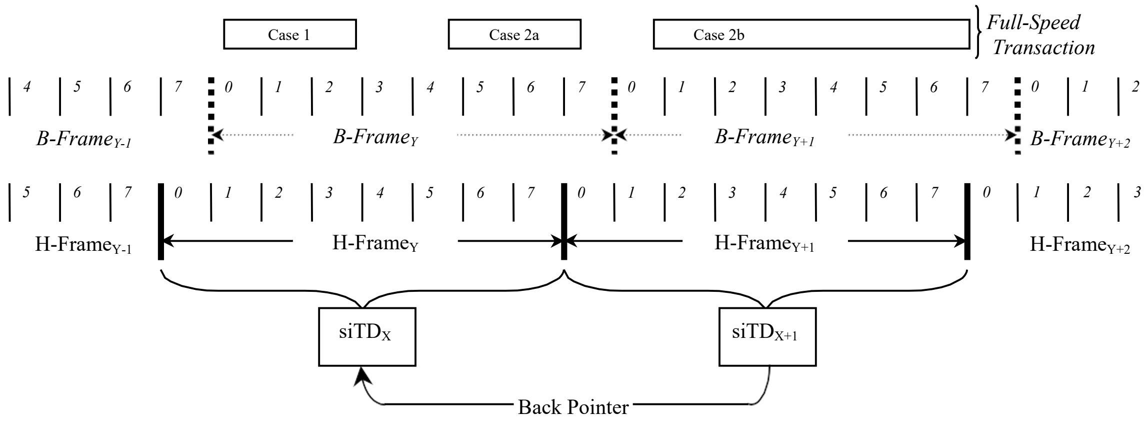


Figure 4-22. siTD Scheduling Boundary Examples

Each case is described below:

- *Case 1*: One siTD is sufficient to describe and complete the isochronous split transaction because the whole isochronous split transaction is tightly contained within a single *H-Frame*.
- *Case 2a, 2b*: Although both INs and OUTs can have these footprints, OUTs always take only one siTD to schedule. However, INs (for these boundary cases) require two siTDs to complete the scheduling of the isochronous split transaction. siTD$_X$ is used to always issue the start-split and the first *N* complete-splits. The full-speed transaction (for these cases) can deliver data on the full-speed bus segment during micro-frame 7 of *H-Frame$_{Y+1}$*, or micro-frame 0 of *H-Frame$_{Y+2}$*. The complete splits are scheduled using siTD$_{X+2}$ (not shown). The complete-splits to extract this data must use the buffer pointer from siTD$_{X+1}$. The only way for the host controller to reach siTD$_{X+1}$ from *H-Frame$_{Y+2}$* is to use siTD$_{X+2}$'s back pointer. The host controller rules for when to use the back pointer are described is Section 4.12.3.3.2.1.

Software must apply the following rules when calculating the schedule and linking the schedule data structures into the periodic schedule:

- Software must ensure that an isochronous split-transaction is started so that it will complete before the end of the *B-Frame*.
- Software must ensure that for a single full-speed isochronous endpoint, there is never a start-split and complete-split in *H-Frame, micro-frame 1*. This is mandated as a rule so that case 2a and case 2b can be discriminated. According to the core USB specification, the long isochronous transaction illustrated in Case 2b, could be scheduled so that the start-split was in micro-frame 1 of *H-Frame* N and the last complete-split would need to occur in micro-frame 1 of H-Frame N+1. However, it is impossible to discriminate between cases 2a and case 2b, which has significant impact on the complexity of the host controller.

4.12.3.2 Tracking Split Transaction Progress for Isochronous Transfers

To correctly maintain the data stream, the host controller must be able to detect and report errors where device to host data is lost. Isochronous endpoints do not employ the concept of a halt on error, however the host is required to identify and report per-packet errors observed in the data stream. This includes schedule traversal problems (skipped micro-frames), timeouts and corrupted data received.

In similar kind to interrupt split-transactions, the portions of the split transaction protocol must execute in the micro-frames they are scheduled. The queue head data structure used to manage full- and low-speed interrupt has several mechanisms for tracking when portions of a transaction have occurred. Isochronous transfers use siTDs, for their transfers, and the data structures are only reachable via the schedule in the exact micro-frame in which they are required (so all the mechanism employed for tracking in queue heads is not required for siTDs). Software has the option of reusing siTD several times in the complete periodic schedule. However, it must ensure that the results of split transaction N are consumed and the siTD re-initialized (activated) before the host controller gets back to the siTD (in a future micro-frame).

Split-transaction isochronous OUTs utilize a low-level protocol to indicate which portions of the split transaction data have arrived. Control over the low-level protocol is exposed in an siTD via the fields *Transaction Position* (*TP*) and *Transaction Count* (*T-count*). If the entire data payload for the OUT split transaction is larger than 188 bytes, there will be more than one start-split transaction, each of which require proper annotation. If host hold-offs occur, then the sequence of annotations received from the host will not be complete, which is detected and handled by the transaction translator. See Section 4.12.3.3.1 for a description on how these fields are used during a sequence of start-split transactions.

The fields *siTD.T-Count* and *siTD.TP* are used by the host controller to drive and sequence the transaction position annotations. It is the responsibility of system software to properly initialize these fields in each siTD. Once the budget for a split-transaction isochronous endpoint is established, *S-mask*, *T-Count*, and *TP* initialization values for all the siTD associated with the endpoint are constant. They remain constant until the budget for the endpoint is recalculated by software and the periodic schedule adjusted.

For IN-endpoints, the transaction translator simply annotates the response data packets with enough information to allow the host controller to identify the last data. As with split transaction Interrupt, it is the host controller's responsibility to detect when it has missed an opportunity to execute a complete-split. The following field in the siTD is used to track and detect errors in the execution of a split transaction for an IN isochronous endpoint.

- *C-prog-mask*. This is an eight-bit bit-vector where the host controller keeps track of which complete-splits have been executed. Due to the nature of the Transaction Translator periodic pipeline, the complete-splits need to be executed in-order. The host controller needs to detect when the complete-splits have not been executed in order. This can only occur due to system hold-offs where the host controller cannot get to the memory-based schedule. *C-prog-mask* is a simple bit-vector that the host controller sets a bit for each complete-split executed. The bit position is determined by the micro-frame (FRINDEX[2:0]) number in which the complete-split was executed. The host controller always checks *C-prog-mask* before executing a complete-split transaction. If the previous complete-splits have not been executed, then it means one (or more) have been skipped and data has potentially been lost. System software is required to initialize this field to zero before setting an siTD's *Active* bit to a one.

If a transaction translator returns with the final data before all of the complete-splits have been executed, the state of the transfer is advanced so that the remaining complete-splits are not executed. Refer to Section 4.12.3.3.2 for a description on how the state of the transfer is advanced. It is important to note that an IN siTD is retired based solely on the responses from the Transaction Translator to the complete-split transactions. This means, for example, that it is possible for a transaction translator to respond to a complete-split with an MDATA PID. The number of bytes in the MDATA's data payload could cause the siTD field *Total Bytes to Transfer* to decrement to zero. This response can occur, before all of the scheduled complete-splits have been executed. In other interface, data structures (e.g. high-speed data streams through queue heads), the transition of *Total Bytes to Transfer* to zero signals the end of the transfer and results in setting of the *Active* bit to zero. However, in this case, the result has not been delivered by the Transaction Translator and the host must continue with the next complete-split transaction to extract the residual transaction state. This scenario occurs because of the pipeline rules for a Transaction Translator (see Chapter 11 of the Universal Serial Bus Revision 2.0). In summary the periodic pipeline rules require that on a micro-frame boundary, the Transaction Translator will hold the final two bytes received (if it has not seen an End Of Packet (EOP)) in the full-speed bus pipe stage and give the remaining bytes to the high-speed pipeline stage. At the micro-frame boundary, the Transaction Translator could have received the entire packet (including both CRC bytes) but not received the packet EOP. In the next micro-frame, the Transaction Translator will respond with an MDATA and send all of the data bytes (with the two CRC bytes being held in the full-speed pipeline stage). This could cause the siTD to decrement it's *Total Bytes to Transfer* field to zero, indicating it has received all expected data. The host must still execute one more (scheduled) complete-split transaction in order to extract the results of the full-speed transaction from the Transaction Translator (for example, the Transaction Translator may have detected a CRC failure, and this result must be forwarded to the host).

If the host experiences hold-offs that cause the host controller to skip one or more (but not all) scheduled split transactions for an isochronous OUT, then the protocol to the transaction translator will not be consistent and the transaction translator will detect and react to the problem. Likewise, for host hold-offs that cause the host controller to skip one or more (but not all) scheduled split transactions for an isochronous IN, the *C-prog-mask* is used by the host controller to detect errors. However, if the host experiences a hold-off that causes it to skip all of an siTD, or an siTD expires during a host hold off (e.g. a hold-off occurs and the siTD is no longer reachable by the host controller in order for it to report the hold-off event), then system software must detect that the siTDs have not been processed by the host controller (e.g. state not advanced) and report the appropriate error to the client driver.

4.12.3.3 Split Transaction Execution State Machine for Isochronous

In the following presentation, all references to micro-frame are in the context of a micro-frame within an *H-Frame.*

If the *Active* bit in the *Status* byte is a zero, the host controller will ignore the siTD and continue traversing the periodic schedule. Otherwise the host controller will process the siTD as specified below.

A split transaction state machine is used to manage the split-transaction protocol sequence. The host controller uses the fields defined in Section 4.12.3.2, plus the variable *cMicroFrameBit* defined in Section 4.12.2.4 to track the progress of an isochronous split transaction.

Figure 4-23 illustrates the state machine for managing an siTD through an isochronous split transaction. Bold, dotted circles denote the state of the *Active* bit in the *Status* field of a siTD. The Bold, dotted arcs denote the transitions between these states. Solid circles denote the states of the split transaction state machine and the solid arcs denote the transitions between these states. Dotted arcs and boxes reference actions that take place either as a result of a transition or from being in a state.

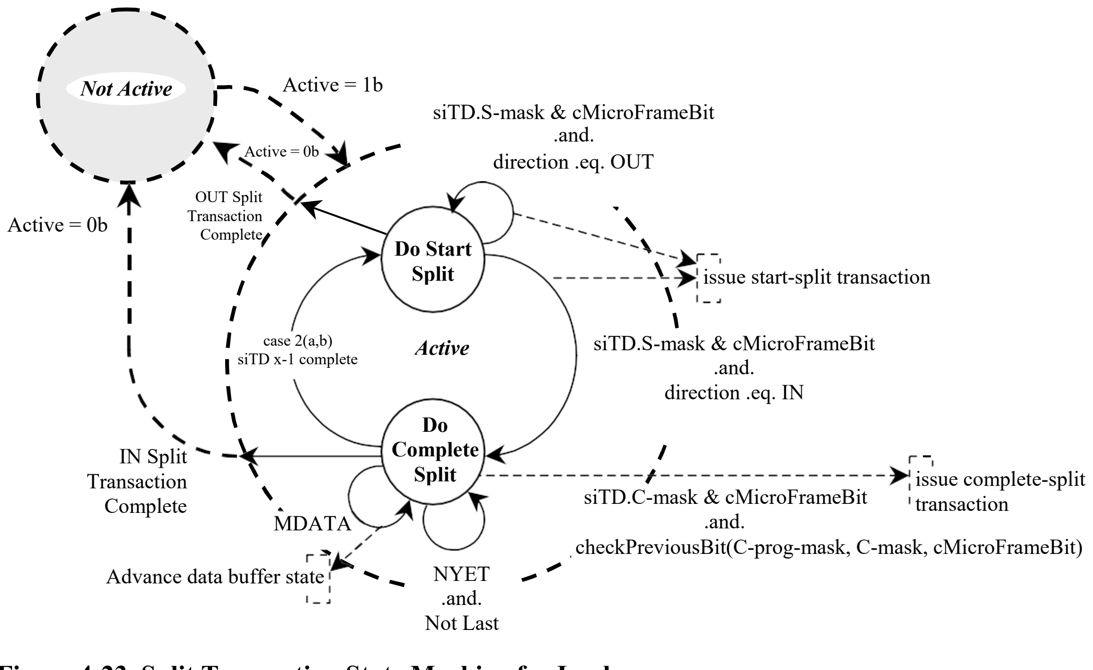

Not Active
Active = 1b
Active = 0b
Active = 0b
siTD.S-mask & cMicroFrameBit .and. direction .eq. OUT
OUT Split Transaction Complete
Do Start Split
issue start-split transaction
case 2(a,b) siTD x-1 complete
Active
siTD.S-mask & cMicroFrameBit .and. direction .eq. IN
IN Split Transaction Complete
Do Complete Split
issue complete-split transaction
siTD.C-mask & cMicroFrameBit .and. checkPreviousBit(C-prog-mask, C-mask, cMicroFrameBit)
MDATA
Advance data buffer state
NYET .and. Not Last

Figure 4-23. Split Transaction State Machine for Isochronous

4.12.3.3.1 Periodic Isochronous - Do Start Split

Isochronous split transaction OUTs use only this state. An siTD for a split-transaction isochronous IN is either initialized to this state, or the siTD transitions to this state from **Do Complete Split** when a case 2a (IN) or 2b scheduling boundary isochronous split-transaction completes.

Each time the host controller reaches an active siTD in this state, it checks the *siTD.S-mask* against *cMicroFrameBit*. If there is a one in the appropriate position, the siTD will execute a start-split transaction. By definition, the host controller cannot *reach* an siTD at the wrong time.

If the *I/O* field indicates an IN, then the start-split transaction includes only the extended token plus the full-speed token. Software must initialize the *siTD.Total Bytes To Transfer* field to the number of bytes expected. This is usually the maximum packet size for the full-speed endpoint. The host controller exits this state when the start-split transaction is complete.

The remainder of this section is specific to an isochronous OUT endpoint (i.e. the *I/O* field indicates an OUT). When the host controller executes a start-split transaction for an isochronous OUT it includes a data payload in the start-split transaction. The memory buffer address for the data payload is constructed by concatenating *siTD.Current Offset* with the page pointer indicated by the page selector field (*siTD.P*). A zero in this field selects Page 0 and a 1 selects Page 1. During the start-split for an OUT, if the data transfer crosses a page boundary during the transaction, the host controller must detect the page cross, update the *siTD.P*-bit from a zero to a one, and begin using the *siTD.Page 1* with *siTD.Current Offset* as the memory address pointer.

The field *siTD.TP* is used to annotate each start-split transaction with the indication of which part of the split-transaction data the current payload represents (ALL, BEGIN, MID, END). In all cases the host controller simply uses the value in *siTD.TP* to mark the start-split with the correct transaction position code.

T-Count is always initialized to the number of start-splits for the current frame. *TP* is always initialized to the first required transaction position identifier. The scheduling boundary case (see Figure 4-22) is used to determine the initial value of *TP*. The initial cases are summarized in Table 4–14.

Table 4–14. Initial Conditions for OUT siTD's TP and T-count Fields

Case	T-count	TP	Description
1, 2a	=1	ALL	When the OUT data payload is less than (or equal to) 188 bytes, only one start-split is required to move the data. The one start-split must be marked with an ALL.
1, 2a	!=1	BEGIN	When the OUT data payload is greater than 188 bytes more than one start-split must be used to move the data. The initial start-split must be marked with a BEGIN.

After each start-split transaction is complete, the host controller updates *T-Count* and *TP* appropriately so that the next start-split is correctly annotated. Table 4–15 illustrates all of the *TP* and *T-count* transitions, which must be accomplished by the host controller.

Table 4–15. Transaction Position (TP)/Transaction Count (T-Count) Transition Table

TP	T-count next	TP next	Description
ALL	0	N/A	Transition from ALL, to done.
BEGIN	1	END	Transition from BEGIN to END. Occurs when *T-count* starts at 2.
BEGIN	!=1	MID	Transition from BEGIN to MID. Occurs when *T-count* starts at greater than 2.
MID	!=1	MID	*TP* stays at MID while *T-count* is not equal to 1 (e.g. greater than 1). This case can occur for any of the scheduling boundary cases where the *T-count* starts greater than 3.
MID	1	END	Transition from MID to END. This case can occur for any of the scheduling boundary cases where the *T-count* starts greater than 2.

The start-split transactions do not receive a handshake from the transaction translator, so the host controller always advances the transfer state in the siTD after the bus transaction is complete. To advance the transfer state the following operations take place:

- The *siTD.Total Bytes To Transfer* and the *siTD.Current Offset* fields are adjusted to reflect the number of bytes transferred.
- The *siTD.P* (page selector) bit is updated appropriately.
- The *siTD.TP* and *siTD.T-count* fields are updated appropriately as defined in Table 4–15.

These fields are then written back to the memory based siTD.

The *S-mask* is fixed for the life of the current budget. As mentioned above, *TP* and *T-count* are set specifically in each siTD to reflect the data to be sent from this siTD. Therefore, regardless of the value of *S-mask*, the actual number of start-split transactions depends on *T-count* (or equivalently, *Total Bytes to Transfer*). The host controller must set the *Active* bit to a zero when it detects that all of the schedule data has been sent to the bus. The preferred method is to detect when *T-Count* decrements to zero as a result of a start-split bus transaction. Equivalently, the host controller can detect when *Total Bytes to Transfer* decrements to zero. Either implementation must ensure that if the initial condition is *Total Bytes to Transfer* equal to zero and *T-count* is equal to a one, then the host controller will issue a single start-split, with a zero-length data payload. Software must ensure that *TP*, *T-count* and *Total Bytes to Transfer* are set to deliver the appropriate number of bus transactions from each siTD. An inconsistent combination will yield undefined behavior.

If the host experiences hold-offs that cause the host controller to skip start-split transactions for an OUT transfer, the state of the transfer will not progress appropriately. The transaction translator will observe protocol violations in the arrival of the start-splits for the OUT endpoint (i.e. the transaction position annotation will be incorrect as received by the transaction translator).

Example scenarios are described in Section 4.12.3.4.

A host controller implementation can optionally track the progress of an OUT split transaction by setting appropriate bits in the *siTD.C-prog-mask* as it executes each scheduled start-split. The *checkPreviousBit*() algorithm defined in Section 4.12.3.3.2 can be used prior to executing each start-split to determine whether start-splits were skipped. The host controller can use this mechanism to detect missed micro-frames. It can then set the siTD's *Active* bit to zero and stop execution of this siTD. This saves on both memory and high-speed bus bandwidth.

4.12.3.3.2 Periodic Isochronous - Do Complete Split

This state is only used by a split-transaction isochronous IN endpoint. This state is entered unconditionally from the **Do Start State** after a start-split transaction is executed for an IN endpoint. Each time the host controller visits an siTD in this state, it conducts a number of tests to determine whether it should execute a complete-split transaction. The individual tests are listed below. The sequence they are applied depends on which micro-frame the host controller is currently executing which means that the tests might not be applied until after the siTD referenced from the back pointer has been fetched.

- **Test A**. *cMicroFrameBit* is bit-wise anded with *siTD.C-mask* field. A non-zero result indicates that software scheduled a complete-split for this endpoint, during this micro-frame. This test is always applied to a newly fetched siTD that is in this state.
- **Test B**. The *siTD.C-prog-mask* bit vector is checked to determine whether the previous complete splits have been executed. An example algorithm is below (this is slightly different than the algorithm used in Section 4.12.2.4.2). The sequence in which this test is applied depends on the current value of FRINDEX[2:0]. If FRINDEX[2:0] is 0 or 1, it is not applied until the back pointer has been used. Otherwise it is applied immediately.

```
Algorithm Boolean CheckPreviousBit(siTD.C-prog-mask, siTD.C-mask, cMicroFrameBit)
Begin
  Boolean rvalue = TRUE;

  previousBit = cMicroFrameBit rotate-right(1)

  -- Bit-wise anding previousBit with C-mask indicates whether there was an intent
  -- to send a complete split in the previous micro-frame. So, if the
  -- 'previous bit' is set in C-mask, check C-prog-mask to make sure it
  -- happened.

  if previousBit bitAND siTD.C-mask then
     if not (previousBit bitAND siTD.C-prog-mask) then
        rvalue = FALSE
     End if
  End if

  Return rvalue
End Algorithm
```

If Test A is true and FRINDEX[2:0] is zero or one, then this is a case 2a or 2b scheduling boundary (see Figure 4-21). See Section 4.12.3.3.2.1 for details in handling this condition.

If Test A and Test B evaluate to true, then the host controller will execute a complete-split transaction using the transfer state of the current siTD. When the host controller commits to executing the complete-split transaction, it updates *QH.C-prog-mask* by bit-ORing with *cMicroFrameBit*. The transfer state is advanced based on the completion status of the complete-split transaction. To advance the transfer state of an IN siTD, the host controller must:

- Decrement the number of bytes received from *siTD.Total Bytes To Transfer,*
- Adjust *siTD.Current Offset* by the number of bytes received,
- Adjust *siTD.P* (page selector) field if the transfer caused the host controller to use the next page pointer, and
- Set any appropriate bits in the *siTD.Status* field, depending on the results of the transaction.

Note that if the host controller encounters a condition where *siTD.Total Bytes To Transfer* is zero, and it receives more data, the host controller must not write the additional data to memory. The *siTD.Status.Active* bit must be set to zero and the *siTD.Status.Babble Detected* bit must be set to a one. The fields *siTD.Total Bytes To Transfer*, *siTD.Current Offset*, and *siTD.P* (page selector) are not required to be updated as a result of this transaction attempt.

The host controller must accept (assuming good data packet CRC and sufficient room in the buffer as indicated by the value of *siTD.Total Bytes To Transfer*) MDATA and DATA0/1 data payloads up to and including 192 bytes. A host controller implementation may optionally set *siTD.Status Active* to a zero and *siTD.Status.Babble Detected* to a one when it receives and MDATA or DATA0/1 with a data payload of more than 192 bytes.

The following responses have the noted effects:

- ERR. The full-speed transaction completed with a time-out or bad CRC and this is a reflection of that error to the host. The host controller sets the *ERR* bit in the *siTD.Status* field and sets the *Active* bit to a zero.
- Transaction Error (XactErr). The complete-split transaction encounters a Timeout, CRC16 failure, etc. The *siTD.Status* field *XactErr* field is set to a one and the complete-split transaction must be retried immediately. The host controller must use an internal error counter to count the number of retries as a counter field is not provided in the siTD data structure. The host controller will not retry more than two times. If the host controller exhausts the retries or the end of the micro-frame occurs, the *Active* bit is set to zero.
- DATAx (0 or 1). This response signals that the final data for the split transaction has arrived. The transfer state of the siTD is advanced and the *Active* bit is set to a zero. If the *Bytes To Transfer* field has

not decremented to zero (including the reception of the data payload in the DATAx response), then less data than was expected, or allowed for was actually received. This *short packet* event does not set the USBINT status bit in the USBSTS register to a one. The host controller will not detect this condition.

- NYET (and Last). On each NYET response, the host controller also checks to determine whether this is the last complete-split for this split transaction. Last was defined in Section 4.12.2.4.2. If it is the last complete-split (with a NYET response), then the transfer state of the siTD is not advanced (never received any data) and the *Active* bit is set to a zero. No bits are set in the *Status* field because this is essentially a skipped transaction. The transaction translator must have responded to all the scheduled clompete-splits with NYETs, meaning that the start-split issued by the host controller was not received. This result should be interpreted by system software as if the transaction was completely skipped.

  The test for whether this is the last complete split can be performed by XORing C-mask with C-prog-mask. A zero result indicates that all complete-splits have been executed.

- MDATA (and Last). See above description for testing for **Last**. This can only occur when there is an error condition. Either there has been a babble condition on the full-speed link, which delayed the completion of the full-speed transaction, or software set up the *S-mask* and/or *C-mask*s incorrectly. The host controller must set *XactErr* bit to a one and the *Active* bit is set to a zero.
- NYET (and not Last). See above description for testing for **Last**. The complete-split transaction received a NYET response from the transaction translator. Do not update any transfer state (except for *C-prog-mask*) and stay in this state.
- MDATA (and not Last). The transaction translator responds with an MDATA when it has partial data for the split transaction. For example, the full-speed transaction data payload spans from micro-frame X to X+1 and during micro-frame X, the transaction translator will respond with an MDATA and the data accumulated up to the end of micro-frame X. The host controller advances the transfer state to reflect the number of bytes received.

If Test A succeeds, but Test B fails, it means that one or more of the complete-splits have been skipped. The host controller sets the *Missed Micro-Frame* status bit and sets the *Active* bit to a zero.

4.12.3.3.2.1 Complete-Split for Scheduling Boundary Cases 2a, 2b

Boundary cases 2a and 2b (INs only) (see Figure 4-21) require that the host controller use the transaction state context of the previous siTD to finish the split transaction. Table 4–16 enumerates the transaction state fields.

Table 4–16. Summary siTD Split Transaction State

Buffer State	Status	Execution Progress
Total Bytes To Transfer P (page select) Current Offset TP (transaction position) T-count (transaction count)	All bits in the status field	C-prog-mask

Note: *TP* and *T-count* are used only for Host to Device (OUT) endpoints.

If software has budgeted the schedule of this data stream with a frame wrap case, then it must initialize the *siTD.Back Pointer* field to reference a valid siTD and will have the *siTD.Back Pointer.T-bit* in the *siTD.Back Pointer* field set to a zero. Otherwise, software must set the *siTD.Back Pointer.T-bit* in the *siTD.Back Pointer* field to a one. The host controller's rules for interpreting when to use the *siTD.Back Pointer* field are listed below. These rules apply only when the siTD's *Active* bit is a one and the *SplitXState* is **Do Complete Split**.

- When cMicroFrameBit is a 1h and the $siTD_X$*.Back Pointer.T-bit* is a zero, or
- If cMicroFrameBit is a 2h and $siTD_X$*.S-mask*[0] is a zero

When either of these conditions apply, then the host controller must use the transaction state from *siTD*$_{X-1}$.

In order to access *siTD*$_{X-1}$, the host controller reads on-chip the siTD referenced from *siTD*$_X$*.Back Pointer*. The host controller must save the entire state from *siTD*$_X$ while processing *siTD*$_{X-1}$. This is to accommodate for case 2b processing. The host controller must not recursively walk the list of *siTD.Back Pointers*.

If *siTD*$_{X-1}$ is active (*Active* bit is a one and *SplitXStat* is **Do Complete Split**), then both Test A and Test B are applied as described above. If these criteria to execute a complete-split are met, the host controller executes the complete split and evaluates the results as described above. The transaction state (see Table 4–16) of *siTD*$_{X-1}$ is appropriately advanced based on the results and written back to memory. If the resultant state of *siTD*$_{X-1}$'s *Active* bit is a one, then the host controller returns to the context of *siTD*$_X$, and follows its next pointer to the next schedule item. No updates to *siTD*$_X$ are necessary.

If *siTD*$_{X-1}$ is active (*Active* bit is a one and *SplitXStat* is **Do Start Split**), then the host controller must set *Active* bit to a zero and *Missed Micro-Frame* status bit to a one and the resultant status written back to memory.

If *siTD*$_{X-1}$'s *Active* bit is a zero, (because it was zero when the host controller first visited *siTD*$_{X-1}$ via *siTD*$_X$'s back pointer, it transitioned to zero as a result of a detected error, or the results of *siTD*$_{X-1}$'s complete-split transaction transitioned it to zero), then the host controller returns to the context of *siTD*$_X$ and transitions its *SplitXState* to **Do Start Split**. The host controller then determines whether the case 2b start split boundary condition exists (i.e. if *cMicroframeBit* is a 1b and *siTD*$_X$*.S-mask*[0] is a 1b). If this criterion is met the host controller immediately executes a start-split transaction and appropriately advances the transaction state of *siTD*$_X$, then follows *siTD*$_X$*.Next Pointer* to the next schedule item. If the criterion is not met, the host controller simply follows *siTD*$_X$*.Next Pointer* to the next schedule item. Note that in the case of a 2b boundary case, the split-transaction of *siTD*$_{X-1}$ will have its *Active* bit set to zero when the host controller returns to the context of *siTD*$_X$. Also, note that software should not initialize an siTD with *C-mask* bits 0 and 1 set to a one and an *S-mask* with bit zero set to a one. This scheduling combination is not supported and the behavior of the host controller is undefined.

4.12.3.4 Split Transaction for Isochronous - Processing Examples

There is an important difference between how the hardware/software manages the isochronous split transaction state machine and how it manages the asynchronous and interrupt split transaction state machines. The asynchronous and interrupt split transaction state machines are encapsulated within a single queue head. The progress of the data stream depends on the progress of each split transaction. In some respects, the split-transaction state machine is sequenced via the **Execute Transaction** queue head traversal state machine (see Figure 4-14).

Isochronous is a pure time-oriented transaction/data stream. The interface data structures are optimized to efficiently describe transactions that need to occur at specific times. The isochronous split-transaction state machine must be managed across these time-oriented data structures. This means that system software must correctly describe the scheduling of split-transactions across more than one data structure. Then the host controller must make the appropriate state transitions at the appropriate times, in the correct data structures.

For example, Table 4–17 illustrates a couple of frames worth of scheduling required to schedule a case 2a full-speed isochronous data stream.

Table 4–17. Example Case 2a - Software Scheduling siTDs for an IN Endpoint

siTD$_X$		Micro-Frames								Initial
#	Masks	0	1	2	3	4	5	6	7	SplitXState
X	S-Mask					1				Do Start Split
	C-Mask	1	1					1	1	
X+1	S-Mask					1				Do Complete Split
	C-Mask	1	1					1	1	
X+2	S-Mask					1				Do Complete Split
	C-Mask	1	1					1	1	
X+3	S-Mask	Repeats previous pattern								Do Complete Split
	C-Mask									

This example shows the first three siTDs for the transaction stream. Since this is the case-2a frame-wrap case, *S-mask*s of all siTDs for this endpoint have a value of 10h (a one bit in micro-frame 4) and *C-mask* value of C3h (one-bits in micro-frames 0,1, 6 and 7). Additionally, sofware ensures that the *Back Pointer* field of each siTD references the appropriate siTD data structure (and the *Back Pointer T-bits* are set to zero).

The initial *SplitXState* of the first siTD is **Do Start Split**. The host controller will visit the first siTD eight times during frame X. The C-mask bits in micro-frames 0 and 1 are ignored because the state is **Do Start Split**. During micro-frame 4, the host controller determines that it can run a start-split (and does) and changes *SplitXState* to **Do Complete Split**. During micro-frames 6 and 7, the host controller executes complete-splits. Notice the siTD for frame X+1 has it's *SplitXState* initialized to **Do Complete Split**. As the host controller continues to traverse the schedule during *H-Frame* X+1, it will visit the second siTD eight times. During micro-frames 0 and 1 it will detect that it must execute complete-splits.

During *H-Frame* X+1, micro-frame 0, the host controller detects that siTD$_{X+1}$'s *Back Pointer.T-bit* is a zero, saves the state of siTD$_{X+1}$ and fetches siTD$_X$. It executes the complete split transaction using the transaction state of siTD$_X$. If the siTD$_X$ split transaction is complete, siTD's *Active* bit is set to zero and results written back to siTD$_X$. The host controller retains the fact that siTD$_X$ is retired and transitions the *SplitXState* in the siTD$_{X+1}$ to **Do Start Split.** At this point, the host controller is prepared to execute the start-split for siTD$_{X+1}$ when it reaches micro-frame 4. If the split-transaction completes early (transaction-complete is defined in Section 4.12.3.3.2), i.e. before all the scheduled complete-splits have been executed, the host controller will transition *siTD$_X$.SplitXState* to **Do Start Split** early and naturally skip the remaining scheduled complete-split transactions. For this example, siTD$_{X+1}$ does not receive a DATA0 response until *H-Frame* X+2, micro-frame 1.

During *H-Frame X+2,* micro-frame 0, the host controller detects that siTD$_{X+2}$'s *Back Pointer.T-bit* is a zero, saves the state of siTD$_{X+2}$ and fetches siTD$_{X+1}$. As described above, it executes another split transaction, receives an MDATA response, updates the transfer state, but does not modify the *Active* bit. The host controller returns to the context of siTD$_{X+2}$, and traverses it's next pointer without any state change updates to siTD$_{X+2}$. S

During *H-Frame X+2*, micro-frame 1, the host controller detects siTD$_{X+2}$'s *S-mask[0]* is a zero, saves the state of siTD$_{X+2}$ and fetches siTD$_{X+1}$. It executes another complete-split transaction, receives a DATA0 response, updates the transfer state and sets the *Active* bit to a zero. It returns to the state of siTD$_{X+2}$ and changes its *SplitXState* to **Do Start Split**. At this point, the host controller is prepared to execute start-splits for siTD$_{X+2}$ when it reaches micro-frame 4.

<TBD… describe how software detects that there was missing micro-frames (don't think we care about missing out micro-frames. There is enough residual state to identify than not all transactions were executed.).

4.13 Host Controller Pause

When the host controller's *HCHalted* bit in the USBSTS register is a zero, the host controller is sending SOF (Start OF Frame) packets down all enabled ports. When the schedules are enabled, the EHCI host controller will access the schedules in main memory each micro-frame. This constant pinging of main memory is known to create CPU power management problems for mobile systems. Specifically, mobile systems aggressively manage the state of the CPU, based on recent history usage. In the more aggressive power saving modes, the CPU can disable its caches. Current PC architectures assume that bus-master accesses to main memory must be cache-coherent. So, when bus masters are busy touching memory, the CPU power management software can detect this activity over time and inhibit the transition of the CPU into its lowest power savings mode. USB controllers are bus-masters and the frequency at which they access their memory-based schedules keeps the CPU power management software from placing the CPU into its lowest power savings state.

USB Host controllers don't access main memory when they are suspended. However, there are a variety of reasons why placing the USB controllers into suspend won't work, but they are beyond the scope of this document. The base requirement is that the USB controller needs to be kept out of main memory, while at the same time, the USB bus is kept from going into suspend.

EHCI controllers provide a large-grained mechanism that can be manipulated by system software to change the memory access pattern of the host controller. System software can manipulate the schedule enable bits in the USBCMD register to turn on/off the scheduling traversal. A software heuristic can be applied to implement an on/off duty cycle that allows the USB to make reasonable progress and allow the CPU power management to get the CPU into its lowest power state. This method is not intended to be applied at all times to throttle USB, but should only be applied in very specific configurations and usage loads. For example, when only a keyboard or mouse is attached to the USB, the heuristic could detect times when the USB is attempting to move data only very infrequently and can adjust the duty cycle to allow the CPU to reach it's low power state for longer periods of time. Similarly, it could detect increases in the USB load and adjust the duty cycle appropriately, even to the point where the schedules are never disabled. The assumption here is that the USB is moving data and the CPU will be required to process the data streams.

It is suggested that in order to provide a complete solution for the system, the companion host controllers should also provide a similar method to allow system software to inhibit the companion host controller from accessing it's shared memory based data structures (schedule lists or otherwise).

4.14 Port Test Modes

EHCI host controllers must implement the port test modes **Test J_State**, **Test K_State**, **Test_Packet**, **Test Force_Enable**, and **Test SE0_NAK** as described in the USB Specification Revision 2.0. The system is only allowed to test ports that are owned by the EHCI controller (e.g. *CF-bit* is a one and *PortOwner* bit is a zero). System software is allowed to have at most one port in test mode at a time. Placing more than one port in test mode will yield undefined results. The required, per port test sequence is (assuming the *CF-bit* in the CONFIGFLAG register is a one):

- Disable the periodic and asynchronous schedules by setting the *Asynchronous Schedule Enable* and *Periodic Schedule Enable* bits in the USBCMD register to a zero.
- Place all enabled root ports into the suspended state by setting the *Suspend* bit in each appropriate PORTSC register to a one.
- Set the *Run/Stop* bit in the USBCMD register to a zero and wait for the *HCHalted* bit in the USBSTS register, to transition to a one. Note that an EHCI host controller implementation may optionally allow port testing with the *Run/Stop* bit set to a one. However, all host controllers must support port testing with *Run/Stop* set to a zero and *HCHalted* set to a one.
- Set the *Port Test Control* field in the port under test PORTSC register to the value corresponding to the desired test mode. If the selected test is **Test_Force_Enable**, then the *Run/Stop* bit in the USBCMD register must then be transitioned back to one, in order to enable transmission of SOFs out of the port under test.

- When the test is complete, system software must ensure the host controller is halted (*HCHalted* bit is a one) then it terminates and exits test mode by setting *HCReset* to a one.

4.15 Interrupts

The EHCI Host Controller hardware provides interrupt capability based on a number of sources. There are several general groups of interrupt sources:

- Interrupts as a result of executing transactions from the schedule (success and error conditions),
- Host controller events (Port change events, etc.), and
- Host Controller error events

All transaction-based sources are maskable through the Host Controller's Interrupt Enable register (USBINTR, see Table 2-11). Additionally, individual transfer descriptors can be marked to generate an interrupt on completion. This section describes each interrupt source and the processing that occurs in response to the interrupt.

During normal operation, interrupts may be immediate or deferred until the next interrupt threshold occurs. The interrupt threshold is a tunable parameter via the *Interrupt Threshold Control* field in the USBCMD register. The value of this register controls when the host controller will generate an interrupt on behalf of normal transaction execution. When a transaction completes during an interrupt interval period, the interrupt signaling the completion of the transfer will not occur until the interrupt threshold occurs. For example, the default value is eight micro-frames. This means that the host controller will not generate interrupts any more frequently than once every eight micro-frames.

Section 4.15.2.4 details effects of a host system error.

If an interrupt has been scheduled to be generated for the current interrupt threshold interval, the interrupt is not signaled until after the status for the last complete transaction in the interval has been written back to host memory. This may sometimes result in the interrupt not being signaled until the next interrupt threshold.

Initial interrupt processing is the same, regardless of the reason for the interrupt. When an interrupt is signaled by the hardware, CPU control is transferred to host controller's USB interrupt handler. The precise mechanism to accomplish the transfer is OS specific. For this discussion it is just assumed that control is received. When the interrupt handler receives control, its first action is to reads the USBSTS (USB Status Register). It then acknowledges the interrupt by clearing all of the interrupt status bits by writing ones to these bit positions. The handler then determines whether the interrupt is due to schedule processing or some other event. After acknowledging the interrupt, the handler (via an OS-specific mechanism), schedules a deferred procedure call (DPC) which will execute later. The DPC routine processes the results of the schedule execution. The precise mechanisms used are beyond the scope of this document.

Note: the host controller is not required to de-assert a currently active interrupt condition when software sets the interrupt enables (in the USBINR register, see Section 2.3.3) to a zero. The only reliable method software should use for acknowledging an interrupt is by transitioning the appropriate status bits in the USBSTS register (Section 2.3.2) from a one to a zero.

4.15.1 Transfer/Transaction Based Interrupts

These interrupt sources are associated with transfer and transaction progress. They are all dependent on the next interrupt threshold.

4.15.1.1 Transaction Error

A transaction error is any error that caused the host controller to think that the transfer did not complete successfully. Table 4–18 lists the events/responses that the host can observe as a result of a transaction. The effects of the error counter and interrupt status are summarized in the following paragraphs. Most of these errors set the *XactErr* status bit in the appropriate interface data structure.

There is a small set of protocol errors that relate only when executing a queue head and fit under the umbrella of a WRONG PID error that are significant to explicitly identify. When these errors occur, the *XactErr* status bit in the queue head is set and the *CErr* field is decremented. When the *PIDCode* indicates a SETUP, the following responses are protocol errors and result in *XactErr* bit being set to a one and the *CErr* field being decremented.

- *EPS* field indicates a high-speed device and it returns a Nak handshake to a SETUP.
- *EPS* field indicates a high-speed device and it returns a Nyet handshake to a SETUP.
- *EPS* field indicates a low- or full-speed device and the complete-split receives a Nak handshake.

Table 4–18 Summary of Transaction Errors

Event / Result	Queue Head/qTD/iTD/siTD Side-effects		USB Status Register (USBSTS)
	Cerr	Status Field	USBERRINT
CRC	-1	XactErr set to a one.	1[1]
Timeout	-1	XactErr set to a one.	1[1]
Bad PID[2]	-1	XactErr set to a one.	1[1]
Babble	N/A	Section 4.15.1.1.1	1
Buffer Error	N/A	Section 4.15.1.1.2	

[1] If occurs in a queue head, then *USBERRINT* is asserted only when *CErr* counts down from a one to a zero. In addition the queue is halted, see Section 4.10.3.1.

[2] The host controller received a response from the device, but it could not recognize the PID as a valid PID.

4.15.1.1.1 Serial Bus Babble

When a device transmits more data on the USB than the host controller is expecting for this transaction, it is defined to be babbling. In general, this is called a *Packet Babble*. When a device sends more data than the *Maximum Length* number of bytes, the host controller sets the *Babble Detected* bit to a one and halts the endpoint if it is using a queue head (see Section 4.10.3.1). *Maximum Length* is defined as the minimum of *Total Bytes to Transfer* and *Maximum Packet Size*. The *CErr* field is not decremented for a packet babble condition (only applies to queue heads). A babble condition also exists if IN transaction is in progress at High-speed EOF2 point. This is called a frame babble. A frame babble condition is recorded into the appropriate schedule data structure. In addition, the host controller must disable the port to which the frame babble is detected.

The *USBERRINT* bit in the USBSTS register is set to a one and if the *USB Error Interrupt Enable* bit in the USBINTR register is a one, then a hardware interrupt is signaled to the system at the next interrupt threshold. The host controller must never start an OUT transaction that will babble across a micro-frame EOF.

NOTE: When a host controller detects a data PID mismatch, it must 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, it must ignore the received data and respond with an ACK handshake, in order to advance the transmitter's data sequence.

The EHCI interface 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 host controller 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 must disable the packet babble check when it observes the data PID mismatch.

4.15.1.1.2 Data Buffer Error

This event indicates that an overrun of incoming data or a underrun of outgoing data has occurred for this transaction. 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 error count in the queue head. When these errors do occur, the host controller records the fact the error occurred by setting the *Data Buffer Error* bit in the queue head, iTD or siTD.

If the data buffer error occurs on a non-isochronous IN, the host controller will 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 must 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 USB Specification Revision 2.0.

4.15.1.2 USB Interrupt (Interrupt on Completion (IOC))

Transfer Descriptors (iTDs, siTDs, and queue heads (qTDs)) contain a bit that can be set to cause an interrupt on their completion. The completion of the transfer associated with that schedule item causes the USB Interrupt (USBINT) bit in the USBSTS register to be set to a one. In addition, if a short packet is encountered on an IN transaction associated with a queue head, then this event also causes USBINT to be set to a one. If the USB Interrupt Enable bit in the USBINTR register is set to a one, a hardware interrupt is signaled to the system at the next interrupt threshold. If the completion is because of errors, the *USBERRINT* bit in the USBSTS register is also set to a one.

4.15.1.3 Short Packet

Reception of a data packet that is less than the endpoint's Max Packet size during Control, Bulk or Interrupt transfers signals the completion of the transfer. Whenever a short packet completion occurs during a queue head execution, the *USBINT* bit in the USBSTS register is set to a one. If the *USB Interrupt Enable* bit is set in the USBINTR register, a hardware interrupt is signaled to the system at the next interrupt threshold.

4.15.2 Host Controller Event Interrupts

These interrupt sources are independent of the interrupt threshold (with the one exception being the Interrupt on Async Advance, see Section 4.15.2.3).

4.15.2.1 Port Change Events

Port registers contain status and status change bits. When the status change bits are set to a one, the host controller sets the *Port Change Detect* bit in the USBSTS register to a one. If the *Port Change Interrupt Enable* bit in the USBINTR register is a one, then the host controller will issue a hardware interrupt. The port status change bits include:

- *Connect Status Change*
- *Port Enable/Disable Change*
- *Over-current Change*
- *Force Port Resume*

4.15.2.2 Frame List Rollover

This event indicates that the host controller has wrapped the frame list. The current programmed size of the frame list effects how often this interrupt occurs. If the frame list size is 1024, then the interrupt will occur every 1024 milliseconds, if it is 512, then it will occur every 512 milliseconds, etc.

When a frame list rollover is detected, the host controller sets the *Frame List Rollover* bit in the USBSTS register to a one. If the *Frame List Rollover Enable* bit in the USBINTR register is set to a one, the host controller issues a hardware interrupt. This interrupt is not delayed to the next interrupt threshold.

4.15.2.3 Interrupt on Async Advance

This event is used for deterministic removal of queue heads from the asynchronous schedule. Whenever the host controller advances the on-chip context of the asynchronous schedule, it evaluates the value of the *Interrupt on Async Advance Doorbell* bit in the USBCMD register. If it is a one, it sets the *Interrupt on Async Advance* bit in the USBSTS register to a one. If the *Interrupt on Async Advance Enable* bit in the USBINTR register is a one, the host controller issues a hardware interrupt at the next interrupt threshold. A detailed explanation of this feature is described in Section 4.8.2.

4.15.2.4 Host System Error

The host controller is a bus master and any interaction between the host controller and the system may experience errors. The type of host error may be catastrophic to the host controller (such as a Master Abort) making it impossible for the host controller to continue in a coherent fashion. In the presence of non-catastrophic host errors, such as parity errors, the host controller could potentially continue operation. The recommended behavior for these types of errors is to escalate it to a catastrophic error and halt the host controller. Host-based error must result in the following actions:

- The *Run/Stop* bit in the USBCMD register is set to a zero.
- The following bits in the USBSTS register are set:
  - *Host System Error* bit is to a one.
  - *HCHalted* bit is set to a one.
- If the *Host System Error Enable* bit in the USBINTR register is a one, then the host controller will issue a hardware interrupt. This interrupt is not delayed to the next interrupt threshold.

Table 4–19 summarizes the required actions taken on the various host errors.

Table 4–19. Summary Behavior of EHCI Host Controller on Host System Errors

Cycle Type	Master Abort	Target Abort	Data Phase Parity
Frame list pointer fetch (read)	Fatal	Fatal	Fatal [o]
siTD fetch (read)	Fatal	Fatal	Fatal [o]
siTD status write-back (write)	Fatal [o]	Fatal [o]	Fatal [o]
iTD fetch (read)	Fatal	Fatal	Fatal [o]
iTD status write-back (write)	Fatal [o]	Fatal [o]	Fatal [o]
qTD fetch (read)	Fatal	Fatal	Fatal [o]
qHD status write-back (write)	Fatal [o]	Fatal [o]	Fatal [o]
Data write	Fatal [o]	Fatal [o]	Fatal [o]
Data read	Fatal	Fatal	Fatal [o]

[o] Potentially, a host controller implementation could continue operation without a halt. However, the recommended behavior is to halt the host controller.

Note: After a *Host System Error*, Software must reset the host controller via *HCReset* in the USBCMD register before re-initializing and restarting the host controller.

Page intentionally left blank

5. EHCI Extended Capabilities

EHCI controllers export EHCI-specific extended capabilities utilizing a method similar to the PCI capabilities. If an EHCI controller implements any extended capabilities, it specifies a non-zero value in the *EHCI Extended Capabilities Pointer (EECP)* field of the HCCPARAMS register. This value is an offset into PCI Configuration space where the first extended capability register is located. Each capability register has the format illustrated in Table 5–1

Table 5–1. Format of EHCI Extended Capability Pointer Register

Bit	Description
31:16	**Capability Specific.** The definition and attributes of these bits depends on the specific capability.
15:8	**Next EHCI Extended Capability Pointer** — **RO.** This field points to the PCI configuration space offset of the next EHCI extended capability pointer. A value of 00h indicates the end of the extended capability list.
7:0	**Capability ID** — **RO.** This field identifies the EHCI Extended capability. See Table 5–2 for a list of the valid EHCI extended capabilities.

The Next field may contain a non-zero value. In which case it specifies the offset into PCI Configuration space where the next EHCI extended capability is located. The last EHCI extended capability in a linked list of capabilities must have its link field set to zero.

Table 5–2. EHCI Extended Capability Codes

ID	Name	Description
00h	Reserved	This value is reserved and must not be used.
01h	Pre-OS to OS Handoff Synchronization	OS Handoff Synchronization support capability. See Section 5.1 for details.

5.1 EHCI Extended Capability: 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 EHCI controller. The OS Handoff Synchronization capability provides the mechanisms to allow BIOS to enable SMI support for EHCI events and also a set of registers that are used to implement a semaphore to synchronize ownership changes of the EHCI controller. The hand-off mechanism must be clean and precise and each participant must adhere to the protocol defined below. Failure to do so will result in two software agents believing they each have exclusive ownership of the EHCI and attempt to use the controller concurrently. Note that these registers are not intended to support ownership exchange of the companion controllers.

The OS Handoff Synchronization EHCI extended capability includes two contiguous, 32-bit registers in PCI configuration space. The first register is the USB OS Handoff Synchronization EHCI Extended Capability register (USBLEGSUP), see Section 2.1.7 for the field definitions. This register is a standard EHCI extended capability pointer, including a EHCI Extended Capability ID field and a link to the next EHCI extended capability.

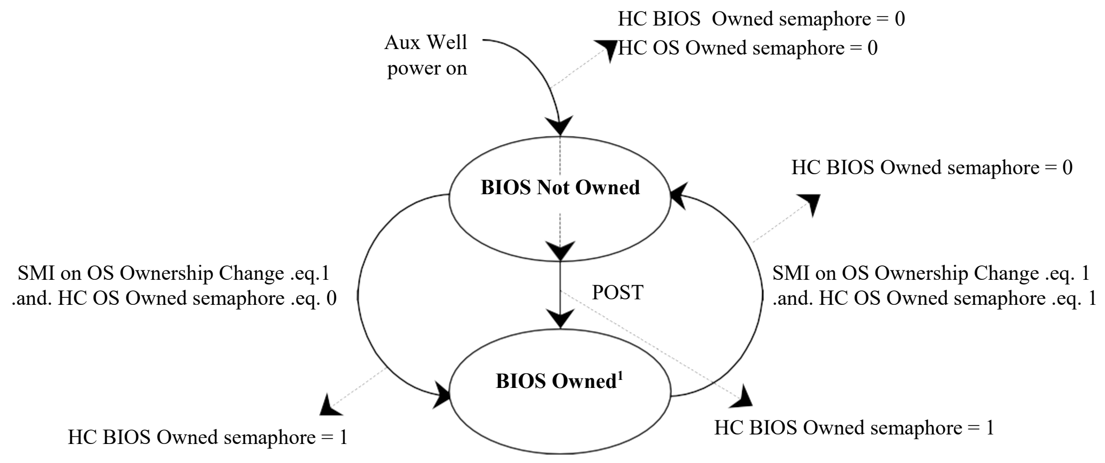
The upper 16 bits of this register contain ownership semaphores. One semaphore is for the operating system (OS) and one is for the BIOS. These semaphores are readable and writable. These fields are in adjacent bytes, which allows each agent (OS or BIOS) to update their respective semaphore without overwriting the other ownership semaphore.

The second 32-bit register is the USB OS Handoff Synchronization Control/Status register (USBLEGCTLSTS), see Section 2.1.8 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 EHCI events that can be detected and enabled to generate an interrupt. The USBLEGCTLSTS register provides the mechanism for BIOS to map all EHCI 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 semphores) that BIOS and OS must adhere to in order to coherently request and/or relinquish ownership of the EHCI controller. 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 5-1 illustrates the protocol state machine for the BIOS ownership. The OS Handoff Synchronization registers are located in the Aux well, so any system event that removes power from the Aux well will result in these registers being reset to their default values when Aux well power is restored.


HC BIOS Owned semaphore = 0
HC OS Owned semaphore = 0
Aux Well power on
BIOS Not Owned
HC BIOS Owned semaphore = 0
SMI on OS Ownership Change .eq.1 .and. HC OS Owned semaphore .eq. 0
POST
SMI on OS Ownership Change .eq. 1 .and. HC OS Owned semaphore .eq. 1
BIOS Owned¹
HC BIOS Owned semaphore = 1
HC BIOS Owned semaphore = 1

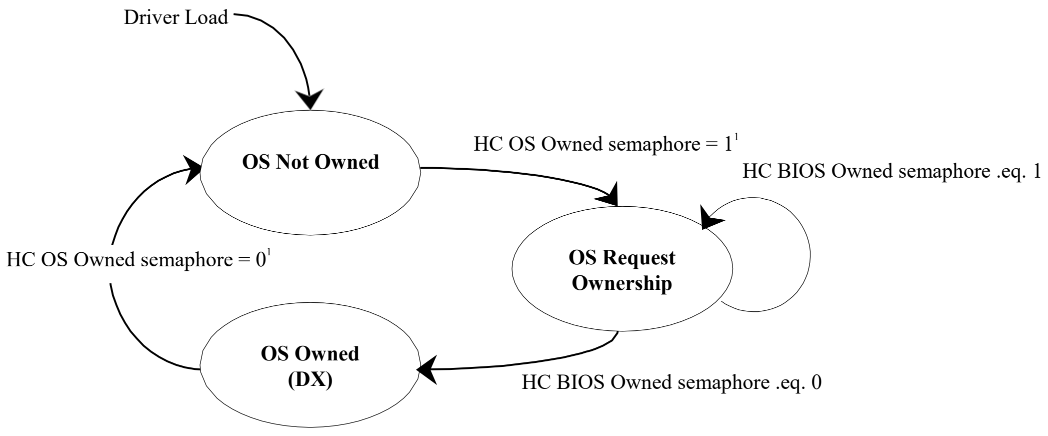
Notes:
[1] The BIOS is allowed to claim control of the EHCI as a result of POST (Power On System Test) or as a result of the OS relinquishing control of the EHCI. The BIOS must never attempt to claim the EHCI once it has relinquished control.

Figure 5-1. BIOS Ownership State Machine

When power is applied to the auxiliary power well, the *BIOS Owned* and *OS Owned* semaphores in the USBLEGSUP go to their default values (e.g. zeros). BIOS may take ownership of the EHCI controller by setting the *BIOS Owned* semaphore to a one. BIOS is only allowed to take ownership of the EHCI controller when the *OS Owned* bit is a zero. BIOS then may configure the SMI events it needs including the *SMI on OS Ownership Change*. The BIOS now owns the EHCI controller, 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 EHCI controller, it will need exclusive control over the EHCI controller. The OS driver must utilize the protocol defined in Figure 5-2 to request ownership of the host controller before it takes ownership and uses the controller. The OS driver initiates an ownership request by setting the *OS Owned* semaphore to a one. The OS waits for the *BIOS Owned* bit to go to a zero before attempting to use the EHCI controller. The time that OS must wait for BIOS to respond to the request for ownership is beyond the scope of this specification. 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 Change* in the USBLEGCTLSTS register to a one, it receives an SMI when the OS Driver sets the *OS Owned* semaphore to a one (above). BIOS observes that OS has changed the value of the *OS Owned* bit to a zero, there-by notifying BIOS that it intends to relinquish control of the EHCI..


Driver Load
OS Not Owned
HC OS Owned semaphore = 1[1]
HC BIOS Owned semaphore .eq. 1
OS Request Ownership
HC OS Owned semaphore = 0[1]
OS Owned (DX)
HC BIOS Owned semaphore .eq. 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.

Figure 5-2. OS Ownership State Machine

In the event that the OS driver unloads and/or wants to relinquish ownership of the EHCI controller, it must set the *OS Owned* semaphore to a zero. Again, if BIOS has set *SMI on OS Ownership Enable* in the USBLEGCTLSTS register to a one, it receives an SMI when the OS Driver sets the *OS Owned* semaphore to a zero. The BIOS observes that the OS has relinquished control and can then take over control of the EHCI controller as appropriate. Once system software has relinquished control of the controller, it must then request ownership as described above.

Note that this mechanism is intended only to ensure that an exchange of ownership of the host controller can be accomplished in a very deterministic and reliable manner.

Page intentionally left blank

Appendix A. EHCI PCI Power Management Interface

An advanced power management capabilities interface compliant with *PCI Bus Power Management Interface Specification Revision 1.1* is incorporated into the EHCI. This interface allows the EHCI to be placed in various power management states offering a variety of power savings for a host system.

Table A-3 highlights the EHCI support for power management states and features supported for each of the power management states. An EHCI 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 EHCI vendor to achieve the required behavior is implementation specific. The EHCI will assert PME# and retain chip context in accordance with the rules defined in the *PCI Bus Power Management Interface Specification Revision 1.1*and this specification

The controller software driver must place all enabled downstream USB ports of the EHCI 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-3. EHCI 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 EHCI 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 EHCI bus master capabilities disabled. All USB ports in suspended state.
D3hot	Required	Deep USB Sleep state with EHCI 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

EHCI implementations follow the PCI Power Management register interface specified in the PCI Power Management Specification Rev 1.1. Specific requirements and clarifications for EHCI implementations are:

- The host controller must 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 Auxiliary power), then the "PME_Support" bit for D3cold (bit 15 of the PMC Register) must 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 Aux_Current or Data Register value reported by the EHCI should represent the maximum current that the host controller device will consume. It must not include power consumed by devices connected to the downstream USB2 ports. Note that if the host controller has been configured to not generate PME# from D3cold, then Aux_Current field or Data Register (D3 Power Consumed, D3 Power Dissipated) must report "000".

All other registers and field should follow the PCI PM specification.

A.1.1 Power State Transitions

The EHCI enters the D0 power state from the D3cold power state when Vcc is applied and a hardware or software reset occurs. A software reset should 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 EHCI through D0, D1, D2, and D3hot power states via EHCI-owned PCI Power Management register accesses. Additional power management policy may be implemented to switch or continuously apply an auxiliary power supply, VAUX, to the EHCI when Vcc is removed. While in this power state, referred to as D3cold, the EHCI exhibits identical behavior as the D3hot power state (except that configuration space accesses are not supported) and no additional EHCI hardware is required to distinguish between D3hot and D3cold.

Per the PCI Power Management specification, the EHCI function asserts an internal reset during the D3hot to D0 transition. The host controller must 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 must be maintained. Additionally, the *PMC.PME_Support(D3cold)* bit must be maintained.

Additionally, the EHCI controller must 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 EHCI registers must not be reset during the D3hot to D0 transition *and* must be maintained in the auxiliary power well (see Section A.1.2 below):

- Configured Flag bit
- 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 must 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 mechanism in the USB Command Register.

A.1.2 Power State Definitions

This section defines the EHCI behavior per power state when programmed using *PMCSR.PowerState.* Power management software may use alternate register mechanisms to place the EHCI in similar states. The EHCI shall support the D0, D3hot, and D3cold power states and is recommended to support the D1, D2 power states.

Any wakeup events as specified in Table A-4 will set *PMCSR.PME_Status* when the EHCI is programmed with *PMCSR.Power_State* 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 EHCI 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 EHCI 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 EHCI out of the D0 power state.

All EHCI 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 must be retained.

The functional and wake-up characteristics for the EHCI power states are summarized in Table A-4.

Table A-4. EHCI Power State Summary

Power State	Functional Characteristics	Wake-up Characteristics (Associated Enables must be Set)
D0	▪ Fully functional EHCI device state ▪ Unmasked interrupts are fully functional	▪ Resume Detected on suspended port ▪ Connect or Disconnect detected on port ▪ Over Current detected on port
D1	▪ EHCI shall preserve PCI configuration ▪ EHCI 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	▪ EHCI shall preserve PCI configuration ▪ EHCI 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	▪ EHCI shall preserve PCI configuration ▪ EHCI 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 EHCI 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

A.1.3 PCI PME# Signal

The PCI PME# signal shall be implemented as an open drain, active low signal that is driven low by the EHCI 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 Power Management specification for more details.

Page intentionally left blank

Appendix B. EHCI 64-Bit Data Structures

This appendix lists the 64-bit versions of the data structures.

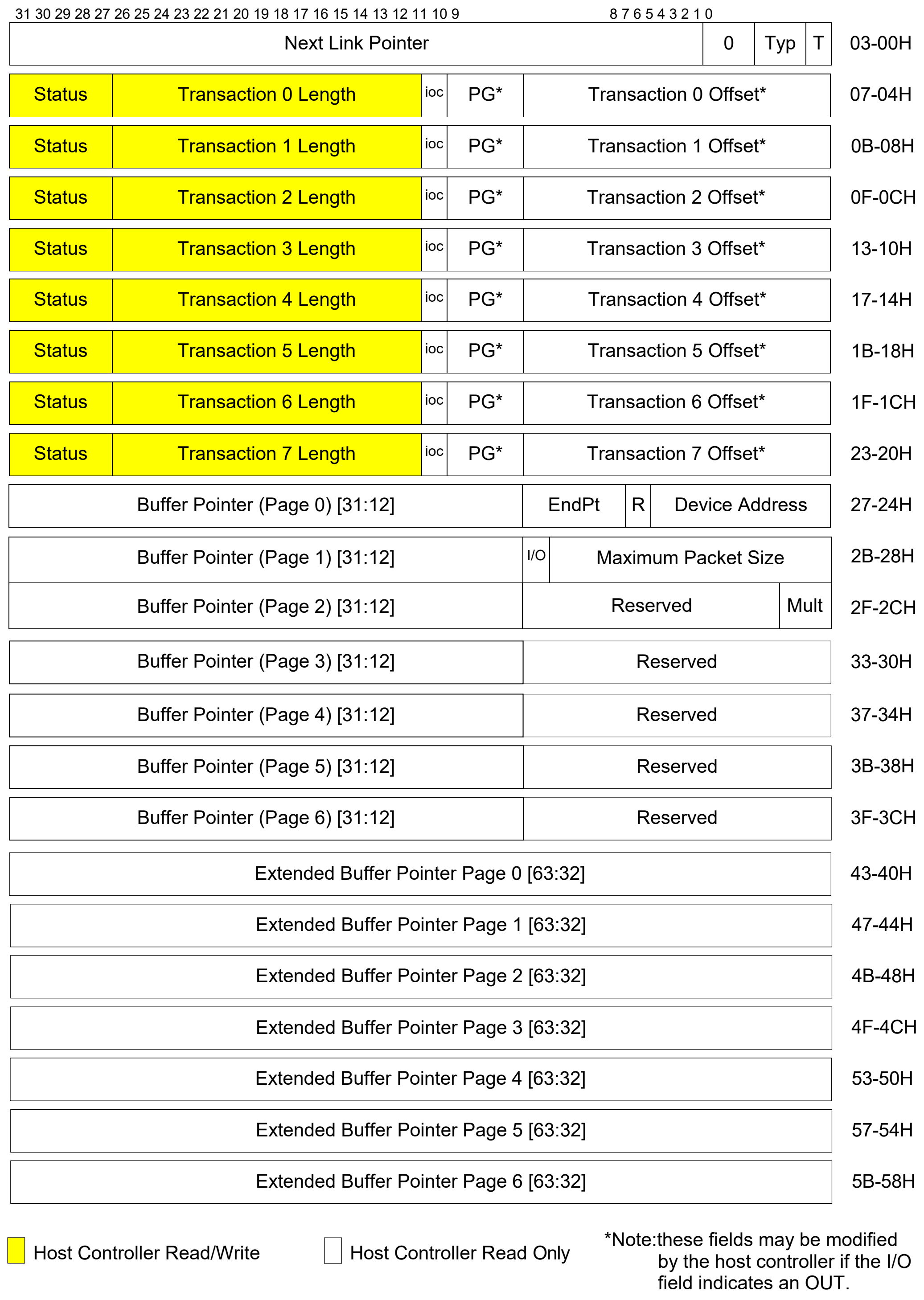

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9
8 7 6 5 4 3 2 1 0
Next Link Pointer | 0 | Typ | T | 03-00H
Status | Transaction 0 Length | ioc | PG* | Transaction 0 Offset* | 07-04H
Status | Transaction 1 Length | ioc | PG* | Transaction 1 Offset* | 0B-08H
Status | Transaction 2 Length | ioc | PG* | Transaction 2 Offset* | 0F-0CH
Status | Transaction 3 Length | ioc | PG* | Transaction 3 Offset* | 13-10H
Status | Transaction 4 Length | ioc | PG* | Transaction 4 Offset* | 17-14H
Status | Transaction 5 Length | ioc | PG* | Transaction 5 Offset* | 1B-18H
Status | Transaction 6 Length | ioc | PG* | Transaction 6 Offset* | 1F-1CH
Status | Transaction 7 Length | ioc | PG* | Transaction 7 Offset* | 23-20H
Buffer Pointer (Page 0) [31:12] | EndPt | R | Device Address | 27-24H
Buffer Pointer (Page 1) [31:12] | I/O | Maximum Packet Size | 2B-28H
Buffer Pointer (Page 2) [31:12] | Reserved | Mult | 2F-2CH
Buffer Pointer (Page 3) [31:12] | Reserved | 33-30H
Buffer Pointer (Page 4) [31:12] | Reserved | 37-34H
Buffer Pointer (Page 5) [31:12] | Reserved | 3B-38H
Buffer Pointer (Page 6) [31:12] | Reserved | 3F-3CH
Extended Buffer Pointer Page 0 [63:32] | 43-40H
Extended Buffer Pointer Page 1 [63:32] | 47-44H
Extended Buffer Pointer Page 2 [63:32] | 4B-48H
Extended Buffer Pointer Page 3 [63:32] | 4F-4CH
Extended Buffer Pointer Page 4 [63:32] | 53-50H
Extended Buffer Pointer Page 5 [63:32] | 57-54H
Extended Buffer Pointer Page 6 [63:32] | 5B-58H
🟨 Host Controller Read/Write
☐ Host Controller Read Only
*Note:these fields may be modified by the host controller if the I/O field indicates an OUT.

Figure B-1. 64-bit Isochronous Transaction Descriptor (iTD-64)

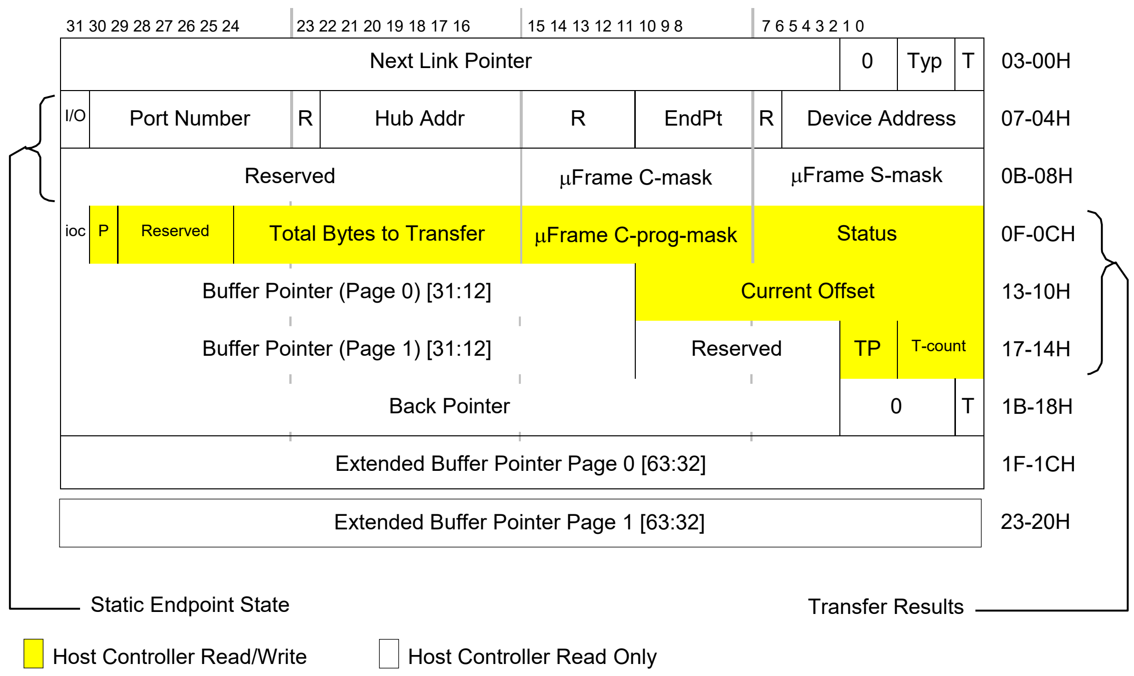


Figure B-2. 64-bit Split Transaction Isochronous Transaction Descriptor (siTD-64)

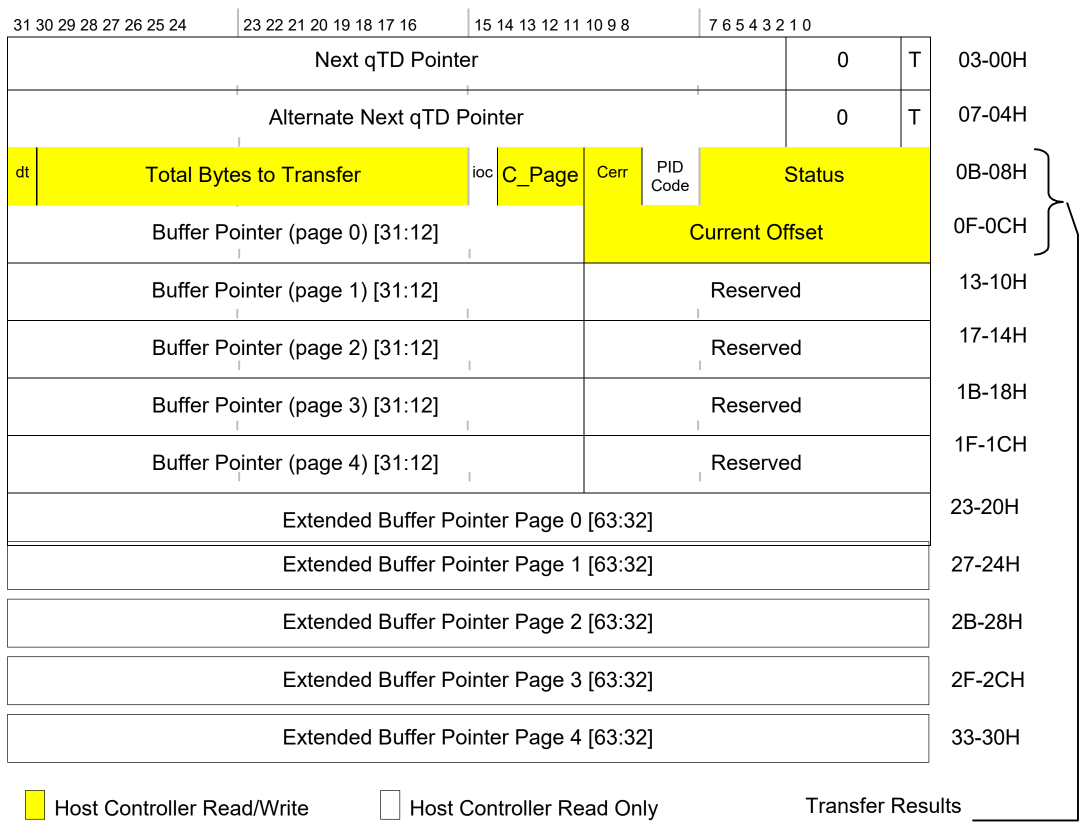

31 30 29 28 27 26 25 24
23 22 21 20 19 18 17 16
15 14 13 12 11 10 9 8
7 6 5 4 3 2 1 0
Next qTD Pointer
0
T
03-00H
Alternate Next qTD Pointer
0
T
07-04H
dt
Total Bytes to Transfer
ioc
C_Page
Cerr
PID Code
Status
0B-08H
Buffer Pointer (page 0) [31:12]
Current Offset
0F-0CH
Buffer Pointer (page 1) [31:12]
Reserved
13-10H
Buffer Pointer (page 2) [31:12]
Reserved
17-14H
Buffer Pointer (page 3) [31:12]
Reserved
1B-18H
Buffer Pointer (page 4) [31:12]
Reserved
1F-1CH
Extended Buffer Pointer Page 0 [63:32]
23-20H
Extended Buffer Pointer Page 1 [63:32]
27-24H
Extended Buffer Pointer Page 2 [63:32]
2B-28H
Extended Buffer Pointer Page 3 [63:32]
2F-2CH
Extended Buffer Pointer Page 4 [63:32]
33-30H
Host Controller Read/Write
Host Controller Read Only
Transfer Results

Figure B-3. 64-bit Queue Element Transaction Descriptor (qTD-64)

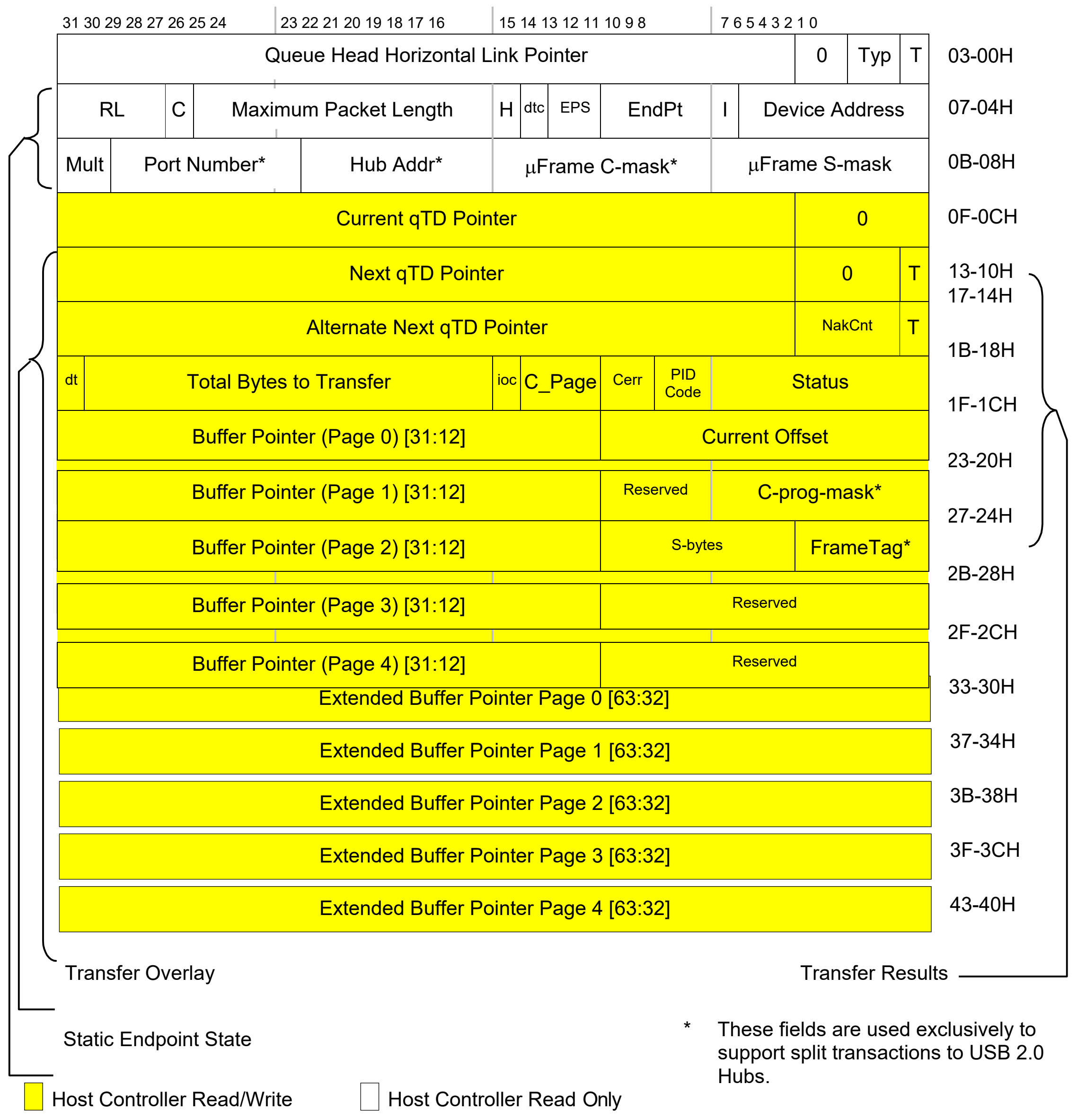

31 30 29 28 27 26 25 24
23 22 21 20 19 18 17 16
15 14 13 12 11 10 9 8
7 6 5 4 3 2 1 0
Queue Head Horizontal Link Pointer
0
Typ
T
03-00H
RL
C
Maximum Packet Length
H
dtc
EPS
EndPt
I
Device Address
07-04H
Mult
Port Number*
Hub Addr*
µFrame C-mask*
µFrame S-mask
0B-08H
Current qTD Pointer
0
0F-0CH
Next qTD Pointer
0
T
13-10H
Alternate Next qTD Pointer
NakCnt
T
17-14H
dt
Total Bytes to Transfer
ioc
C_Page
Cerr
PID Code
Status
1B-18H
Buffer Pointer (Page 0) [31:12]
Current Offset
1F-1CH
Buffer Pointer (Page 1) [31:12]
Reserved
C-prog-mask*
23-20H
Buffer Pointer (Page 2) [31:12]
S-bytes
FrameTag*
27-24H
Buffer Pointer (Page 3) [31:12]
Reserved
2B-28H
Buffer Pointer (Page 4) [31:12]
Reserved
2F-2CH
Extended Buffer Pointer Page 0 [63:32]
33-30H
Extended Buffer Pointer Page 1 [63:32]
37-34H
Extended Buffer Pointer Page 2 [63:32]
3B-38H
Extended Buffer Pointer Page 3 [63:32]
3F-3CH
Extended Buffer Pointer Page 4 [63:32]
43-40H
Transfer Overlay
Transfer Results
Static Endpoint State
* These fields are used exclusively to support split transactions to USB 2.0 Hubs.
Host Controller Read/Write
Host Controller Read Only

Figure B-4. 64-bit Queue Head Descriptor (QH-64)

Note: no 64-bit version of FSTNs are required. The upper 32-bits of the structure pointer must come from the CTRLDSSEGMENT register.

Page intentionally left blank

Appendix C. Debug Port

The debug port is an optional implementation feature. This appendix describes the required implementation and behavior of a USB2 Debug Port as part of an EHCI controller. Specific features of this implementation of a debug port are:

- Only works with a high-speed USB debug device
- Implemented for a specific port on the host controller
- Operational anytime the port is not suspended AND the host controller is in D0 power state.
- Capability is interrupted when port is driving USB RESET

C.1 Locating the Debug Port

The PCI Capability List[5] is used to provide a standard way for software to find and use the Debug Port. Figure C-1 illustrates the Debug Port capability layout, which consist of three fields. The CAP_ID is described in Table C-1, the NXT_PTR is described in Table C-2, and the DEBUG_PORT is described in Table C-3.

DEBUG_PORT	NXT_PTR	CAP_ID

Figure C-1. Debug Port Capability Register Layout

Table C-1. CAP_ID

Bits	Type	Field	Description
7:0	RO	CAP_ID	The value of 0Ah in this field identifies that the function supports a Debug Port.

Table C-2. NXT_PTR

Bits	Type	Field	Description
15:08	RO	NXT_PTR	Pointer to the next item in the capabilities list. Must be NULL for the final item in the list.

[5] PCI Capability List is defined in the PCI Local Bus Specification (Section 6.7 Capability List, Rev. 2.2, Dec. 18, 1999).

Table C-3. DEBUG_PORT

Bits	Type	Field	Description
28:16	RO	Offset	This 12 bit field indicates the byte offset (up to 4K) within the BAR indicated by *BAR#*. This offset is required to be DWORD aligned and therefore bits 16 and 17 are always zero.
31:29	RO	BAR #	A 3-bit field, which indicates which one of the possible 6 Base Address Register offsets contains the Debug Port registers. For example, a value of 1h indicates the first BAR (offset 10h) while a value of 5 indicates that the BAR at 20h. This offset is independent as to whether the BAR is 32 or 64 bit. For example, if the offset were 3 indicating that the BAR at offset 18h contains the Debug Port. BARs at offset 10 and 14h may or may not be implemented. This field is read only and only values 1-6h are valid. (A 64-bit BAR is allowed.) Only a memory BAR is allowed.

C.2 Using the Debug Port Fields

The Debug Port field is comprised of two sub-fields. The first is *OFFSET* and the second is *BAR#*. Figure C-2 illustrates how the fields are used by software to locate and access the Debug Port registers. The arrow going from *BAR#* of the Debug Port Capability Register to the BARs field in the PCI configuration Space indicates that *BAR#* contains a "pointer" to a one BAR of the device in which the debug port registers are mapped.

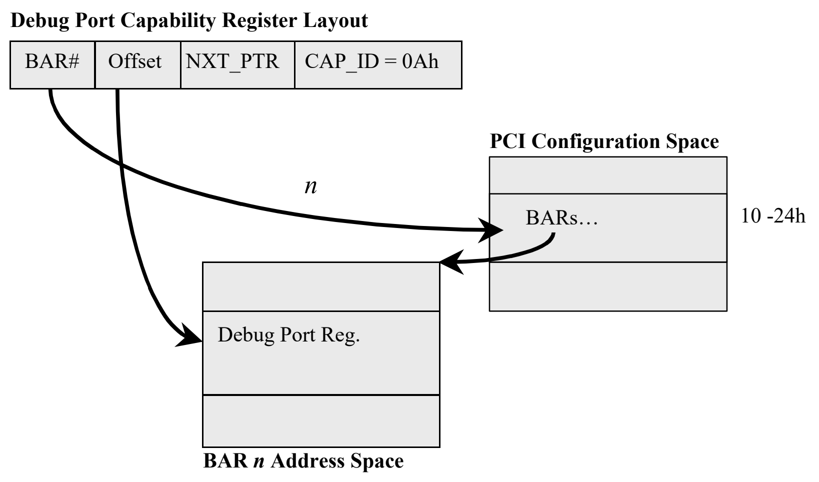


Figure C-2 Debug Port Capability Register

OFFSET of the Debug Port Capability register is used as an offset into the address space assigned by the BAR. Therefore software takes the base address and *OFFSET* to determine the start address of the Debug Port registers. For example, the BAR is programmed with an address of 3000 0000h (3 gigabyte) and the offset is 20h (32 bytes). Therefore, the base address of the Debug registers would be 3000 0020h.

C.3 USB2 Debug Port Register Interface

EHCI implementations use a PCI Capability structure, as defined above, to indicate that it provides a debug port. USB2 debug ports register are located in the same memory area, defined by the Base Address Register (BAR), as the standard EHCI registers. The specific EHCI port that supports this debug capability is indicated by a 4-bit field (bits 20-23) in the HCSPARAMS register of an EHCI controller.

Table C-4. Debug Port Control Register

Register Name	Offset
Control/Status	0x00h
USB PIDs	0x04h
Data Buffer (Bytes 3-0)	0x08h
Data Buffer (Bytes 7-4)	0x0Ch
Device Address	0x10h

C.3.1 Debug Port Control Register

The Debug Port Control register (offset 0h) allows software to interact with the USB2 Debug Port. The hardware associated with this register provides no checks to ensure that software programs the interface correctly. How the hardware behaves when programmed illegally is undefined. The definition of each bit or field in the control register is described in the table below. Note: all bits in the control register can be written or read simultaneously. Software is required to preserve reserved fields/bits by performing a read-modify-write operation. Registers which have reset values will reset based on a HCReset, a D3-D0 transition, or a PCI reset.

Table C-5. Control/Status Register Bits

Bits	Type	Field	Description
31	RO	Reserved	
30	R/W	Owner	When debug software writes a one to this bit, the ownership of the debug port is forced to the EHCI controller (ie. immediately taken away from the companion controller). If the port was already owned by the EHCI controller, then setting this bit is has no effect. This bit overrides all of the ownership related bits in the standard EHCI registers. Reset default = 0. Note that the value in this bit may not effect the value reported in the *Port Owner* bit in the associated PORTSC register.
29	RO	Reserved	
28	R/W	Enabled	This bit is a one if the debug port is enabled for operation. Software can clear this by writing a zero to it. The controller clears the bit for the same conditions where hardware sets the Port Enable/Disable Change bit (in the PORTSC register). (Note: this bit is not cleared when System Software clears the *Port Enabled/Disabled* bit (in the PORTSC register). Software can directly set this bit, if the port is already enabled in the associated Port Status and Control register (this is HW enforced). Reset default = 0.
27:17	RO	Reserved	
16	R/WC	Done	This bit is set by HW to indicate that the request is complete. Writing a 1 to this bit will clear it. Writing a 0 to this bit has no effect. Reset default = 0.
15:11	RO	Reserved	
10	R/W	In Use	Set by software to indicate that the port is in use. Cleared by software to indicate that the port is free and may be used by other software. Reset default = 0. (This bit has no affect on hardware.)

Table C-5. Control/Status Register Bits (cont.)

Bits	Type	Field	Description
9:7	RO	Exception	This field indicates the exception when *Error/Good#* is set. This field cannot be cleared by software. Reset default = 000b. **Value** / **Meaning** 000b None 001b Transaction error or babble: indicates the USB2 transaction had an error (CRC, bad PID, timeout, packet babble, etc.) 010b HW error. Request was attempted (or in progress) when port was suspended or reset. 011b-111b Reserved
6	RO	Error/ Good#	Updated by hardware at the same time it sets the *Done* bit. When set it indicates that an error occurred. Details of the error are provided in the *Exception* field. When cleared, it indicates that the request terminated successfully. Reset default = 0.
5	R/W	Go	Software sets this bit to cause the hardware to perform a request. Writing this bit to a 1 when the bit is already set may result in undefined behavior. Writing a 0 to this bit has no effect. When set, the hardware clears this bit when the hardware sets the *Done* bit. (Completion of a request is indicated by the *Done* bit.) Reset default = 0.
4	R/W	Write/ Read#	Software sets this bit to indicate that the current request is a write and clears it to indicate a read. Reset default = 0.
3:0	R/W	Data Length	For write operations, this field is set by software to indicate to the hardware how many bytes of data in *Data Buffer* are to be transferred to the console when *Write/Read#* is set when software sets *Go*. A value of 0h indicates that a zero-length packet should be sent. A value of 1-8 indicates 1-8 bytes are to be transferred. Values 9-Fh are illegal and how hardware behaves if used is undefined. For read operations, this field is set by hardware to indicate to software how many bytes in *Data Buffer* are valid in response to software setting *Go* when *Write/Read#* is cleared. A value of 0h indicates that a zero length packet was returned. (The state of *Data Buffer* is not defined.) A value of 1-8 indicates 1-8 bytes were received. Hardware is not allowed to return values in the range 9-Fh when the *Error/Good#* field is a zero (0b) after an IN transaction completes. The value in this field is not valid when the *Error/Good#* field is a one (1b) after an IN transaction completes. The transferring of data always starts with byte 0 in the data area and moves toward byte 7 until the transfer size is reached. Reset default = 0h.

C.3.2 USB PIDs Register

This Dword register is used to communicate PID information between the USB debug driver and the USB2 debug port. The debug port uses some of these fields to generate USB packets, and uses other fields to return PID information to the USB debug driver.

Table C-6. USB PIDs Register

Bits	Type	Field	Description
31:24	RO	Reserved	
23:16	RO	Received PID	The debug port controller updates this field with the received PID for transactions in either direction. When the controller is sending data (*Write/Read#* is asserted), this field is updated with the handshake PID that is received from the device. When the host controller is receiving data (*Write/Read#* is not asserted), this field is updated with the data packet PID (if the device sent data), or the handshake PID (if the device NAKs the request). This field is valid when the controller sets the *Done* bit to a one (1b) and the *Error/Good#* field in the **Control/Status** register is a zero. If a transaction completes (*Done* bit transitions to a one) with *Error/Good#* field set to a one, then the contents of this field are undefined. Reset default = undefined.
15:8	R/W	Send PID	The debug port controller sends this PID to begin the data packet when sending data to USB (ie. *Write/Read#* is asserted). Software will typically set this field to either DATA0 or DATA1 PID values. Reset default = undefined.
7:0	R/W	Token PID	The debug port controller sends this PID as the Token PID for each USB transaction. Software will typically set this field to either IN, OUT or SETUP PID values. Reset default = undefined.

C.3.3 Data Buffer

The data buffer consists of 8 bytes arranged as two consecutive Dwords located at offset 08h and 0Ch in the debug port's register space. Table C-7 provides more detail.

Table C-7. Data Buffer

Bits	Type	Field	Description
63:0	R/W	Data Buffer	The least significant byte is accessed at offset 08h and the most significant byte is accessed at offset 0Fh. Each byte in *Data Buffer* can be individually accessed. *Data Buffer* must be written with data before software initiates a write request. For a read request, *Data Buffer* contains valid data when *Done* is set, *Error/Good#* is a zero (0b), and *Data Length* specifies the number of bytes that are valid. If an IN transaction completes (*Done* bit transitions to a one) with *Error/Good#* field set to a one, then the contents of this field are undefined. Reset default = undefined.

C.3.4 Device Address Register

The Device Address register holds the information necessary to properly address the Debug Device when generating transactions. It specifies the USB device address of the debug device, and the endpoint address for USB tokens generated by the debug port.

Table C-8. Device Address Register

Bits	Type	Field	Description
31:15	RO	Reserved	
14:8	R/W	USB Address	7-bit field that identifies the USB device address used by the controller for all Token PID generation. This is a R/W field that is set to 7Fh after power-on reset. Reset default = 7Fh.
7:4	RO	Reserved	
3:0	R/W	USB Endpoint	4-bit field that identifies the endpoint used by the controller for all Token PID generation. Reset default = 01h.

C.4 Operational Model

There are two operational modes for the USB2 debug port. Mode 1 is when the EHCI host controller is not running from the viewpoint of a standard EHCI Driver (e.g. *HCHalted* bit in the USBSTS register is a one). In Mode 1, controller is required to generate a 'keepalive' packet less than 2 milliseconds apart to keep the attached debug device from suspending. The keepalive packet is a standalone 32-bit SYNC field.

Mode 2 is when the host controller is running (ie. *Run/Stop* bit is 1). In Mode 2, the normal transmission of SOF packets (or SYNC keepalives if the PORTSC register *Port Enable/Disable* bit is a zero) will keep the debug device from suspending.

In both modes, the controller must check for software requested debug transactions at least every 125usecs.

If the debug port is enabled by the debug driver, and the standard host controller driver resets the USB port, USB debug transactions are completed with an exception condition (0b010) for the duration of the reset and until after the first SOF is sent.

If the standard host controller driver suspends the USB port, then USB debug transactions are completed with an exception condition (0b010) for the duration of the suspend/resume sequence and until after the first SOF is sent.

The *Enabled* port control bit in the debug register space is independent of the similar port control bit in the associated Port Status and Control register.

The table below shows the debug port behavior related to the state of bits in the debug registers as well as bits in the associated Port Status and Control register.

Table C-9. Summary of Port Behavior vs. Register Settings

Debug bits		EHCI bits			Debug port behavior
Owner	Enabled	Port Enable	Run/ Stop	Suspend	
0	X	X	X	X	Debug port is not being used. Normal operation.
1	0	X	X	X	Debug port is not being used. Normal operation.
1	1	0	0	X	Debug port in Mode 1. SYNC keepalives sent plus debug traffic

Table C-9. Summary of Port Behavior vs. Register Settings (cont.)

Debug bits		EHCI bits			Debug port behavior
Owner	Enabled	Port Enable	Run/ Stop	Suspend	
1	1	0	1	X	Debug port in Mode 2. SYNC keepalives (or SOF's) sent plus debug transactions. Note that no other USB transactions are sent on this port, because the *Port Enabled* bit in the associated PORTSC register is a zero.
1	1	1	0	0	Debug port in Mode 2. SYNC keepalives sent plus debug traffic.
1	1	1	0	1	Port is suspended. No debug traffic sent.
1	1	1	1	0	Debug port in Mode 2. Debug traffic is interspersed with normal traffic.
1	1	1	1	1	Port is suspended. No debug traffic sent.

C.4.1 OUT/SETUP Transactions

When the debug port is enabled and the debug software sets the *Go* bit and the *Write/Read#* bit is set, then the debug controller is enabled to send a debug transaction that will send data to the debug device. The controller sends a token packet consisting of a SYNC, the Token PID field, the Device Address field, the Endpoint field, followed by a 5-bit CRC field. After sending the token packet, the controller sends a data packet consisting of a SYNC, the Send PID, *Data Length* bytes of data from the data registers, and a 16-bit CRC. Note that a *Data Length* value of zero is valid in which case no data bytes would be included in the packet. After sending the data packet, the controller waits for a handshake response from the debug device. If a handshake is received, the controller places the received PID in the *Received PID* register, resets the *Error/Good#* bit and set the *Done* bit. If no handshake PID is received, then the controller sets the *Exception field* to 001b, sets the *Error/Good#* bit, and sets the *Done* bit.

C.4.2 IN transactions

When the debug port is enabled and the debug software sets the *Go* bit and the *Write/Read#* bit is reset, then the debug controller is enabled to send a debug transaction that will receive data from the debug device. The controller sends a token packet consisting of a SYNC, the Token PID field, the Device Address field, and the Endpoint field, followed by a 5-bit CRC field. After sending the token packet, the controller waits for a response from the debug device. If a response is received, the received PID is placed into the *Received PID* register and any subsequent bytes are placed into the data registers. The *Data Length* field is updated to show the number of bytes that were received after the PID. If a valid packet was received from the device and it was one byte in total length (indicating it was a handshake packet) the controller resets the *Error/Good#* bit and sets the *Done* bit. If a valid packet was received from the device and it was more than one byte in total length (indicating it was a data packet), the controller transmits an ACK handshake packet, resets the *Error/Good#* bit and sets the *Done* bit. If no valid packet is received, then the controller sets the *Exception field* to 001b, sets the *Error/Good#* bit, and sets the *Done* bit.

C.4.3 Debug Software Startup

There are two general cases for debug software startup: 1) when the EHCI controller has not been initialized by the system host controller driver, and 2) when the EHCI controller 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 *Configured* bit in the EHCI *CONFIGFLAG* register. If the configure flag is set, that indicates that the system host controller driver has already initialized the EHCI controller. Generic startup procedures for the two cases are provided in the following sections.

C.4.3.1 Startup before System Host Controller Driver

Debug software can attempt to use the debug port if after setting the *Owner* bit, the *Current Connect Status* bit in the appropriate *PORTSC* register is set. Debug software should first reset the attached device by ensuring the host controller is running (*Run/Stop* bit in the USBCMD register is a one and *HCHalted* bit in the USBSTS register is a zero), then setting and then clearing the *Port Reset* bit in the *PORTSC register*. If a high-speed device is attached, the controller will automatically set the *Port Enabled/Disabled* bit in the *PORTSC* register and the debug software can proceed. Debug software should set the *Enabled* bit in the Debug Port *Control* register, and then reset the *Port Enabled/Disabled* bit in the *PORTSC* register (so that the system host controller driver doesn't see an enabled port when it is first loaded). When the port reset sequence is complete, Debug software can then turn off the host controller by setting the *Run/Stop* bit to a zero.

C.4.3.2 Startup after System Host Controller Driver

If there is a device attached (indicated by the *Current Connect Status* bit in the *PORTSC* register), debug software can set the *Owner* bit in the Debug Port *Control* register and then directly set the *Enabled* bit in the Debug Port *Control* register.

C.4.4 Finding the Debug Peripheral

After enabling the debug port functionality, debug software can determine if a debug peripheral is attached by attempting to send data to the debug peripheral. If all attempts result in an error (*Exception* field indicates Transaction Error), then the attached device is not a debug peripheral.

Appendix D. High Bandwidth Isochronous Rules

High bandwidth isochronous streams utilize addition PIDs in the USB protocol. The tables in this appendix completely enumerate all of the required responses an EHCI controller must 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:
  - **Direction**: indicates the direction of the transaction.
  - **Mult**: this is the value of the *Mult* field in an instantiation of an iTD. This is a constant value for the lifetime of the iTD. 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 Mult. 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 host controller as transactions are executed and data is moved.
- **Response:** lists the response from the device (PID code and data size) and the affects on the iTD status field bits and transaction iterator.
  - **PID/(data size)**: indicates the host stimulus, data PID or other response from the device.
- **Result:** list the effects of the response on the bits in the *Status* field and the iterator.

Each row in each table illustrates the required host controller behavior for all of the inputs/response combinations for a 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 D-1. High-Bandwidth Behavior for OUT Transactions

Inputs				Response	Results	
Dir	Mult	Cnt	Remaining Buffer	PID (data size)	Status	Explanation
Out	1 2 3	1	≥ Maxpacket	PID → DATA0 (Maxpacket) PID → DATA1 (Maxpacket) PID → DATA2 (Maxpacket)	Active → 0	Normal completion (for micro-frame) of 1, 2 or 3 high bandwidth transaction; send maxpacketsize bytes. Note that the ≥ Maxpacket where the > applies is just to account for the case where software has incorrectly programmed *Mult* or *Transaction Length*.
	1 2 3	1	< Maxpacket	PID → DATA0 (Xfer Length) PID → DATA1 (Xfer Length) PID → DATA2 (Xfer Length)	Active → 0	Normal completion (for frame) of 1, 2, or 3 high-bandwidth transaction; send as many bytes as are available in the buffer.
	3,2	2	> Maxpacket	PID → MDATA (Maxpacket)	Active → n/c	Intermediate transaction in high-bandwidth sequence; send maxpacketsize bytes with an MDATA PID.
	2 3	2	≤ Maxpacket	PID → DATA0 (Xfer Length) PID → DATA1 (Xfer Length)	Active → 0	Software did not have *Mult*MaxpacketSize* bytes to send for this transaction (micro-frame).
	3	3	> Maxpacket	PID → MDATA (Maxpacket)	Active → n/c	Intermediate transaction in high-bandwidth sequence; send maxpacketsize bytes with an MDATA PID.
	3	3	≤ Maxpacket	PID → DATA0 (Xfer Length)	Active → 0	Software did not have *Mult*MaxpacketSize* bytes to send for this transaction (micro-frame).
	3,2,1	>1	≥ Maxpacket	PID → MDATA (buffer error)	Active → 0 BuffErr → 1	host experienced a buffer error before being able to deliver all of the data. It must 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 one and immediately clear the *Active* status bit to a zero. 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 host controller'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 D-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 micro-frame is determined by the device's response to the first IN of the micro-frame. 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 micro-frame, 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 must be equal to maximum packet length (Maxpacket), and
2. The host controller must 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 Mult, then the received data PID is in the appropriate range. For example, if Mult is 2 and the device returns a DATA1, then Y=1 is less than Mult 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 micro-frame and the host controller must set the *Active* bit to a zero. 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 one.

Table D-2. High-Bandwidth Behavior for IN Transactions

Current state		Endpoint Response			Results	Next State	Explanation
	Cnt	PID[Y]		Data size			
Start	X	PID ← DATA[2,1]	Y < Mult	= Maxpacket		Cnt = [2,1]	Acceptable PID response. If no babble error, then go to Next state.
				< Maxpacket	Active → 0, XactErr → 1	Done	Data payload must be equal to maximum packet size.
				> Maxpacket	Active → 0, Babble → 1	Done	Data payloads larger than maximum packet size are a babble condition.
			Y ≥ Mult	Don't care	Active → 0, XactErr → 1	Done	Starting DATA PID is larger than allowed for this endpoint.
		PID ← DATA0		≤ Maxpacket	Active → 0	Done	Acceptable PID response. If no babble error, transaction completed normally.
				> Maxpacket	Active → 0, Babble → 1	Done	Data payloads larger than maximum packet size are a babble condition.
Next	2	PID ← DATA2		Don't care	Active → 0, XactErr → 1	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	Active → 0, XactErr → 1	Done	Data payload must be equal to maximum packet size.
				> Maxpacket	Active → 0, Babble → 1	Done	Data payloads larger than maximum packet size are a babble condition.
		PID ← DATA0		Don't care	Active → 0, XactErr → 1	Done	Device went from DATA2 to DATA0; invalid transition.
	1	PID ← DATA[2,1]		Don't care	Active → 0, XactErr → 1	Done	Endpoint repeated a DATA2 or DATA1 PID.
		PID ← DATA0		≤ Maxpacket	Active → 0	Done	Acceptable PID response. If no babble error, transaction sequence completed normally.
				> Maxpacket	Active → 0, Babble → 1	Done	Data payloads larger than maximum packet size are a babble condition.

In the *Next* state, the host controller 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 host controller must 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 host controller does a final babble check. If no babble error, the host controller remains in the *Next* state and executes another bus transaction. If there was an error, the host controller sets the *Active* bit to a zero. If the length check fails, the host controller sets the *XactErr* to a one. If the babble check fails, the host controller sets the *Babble* bit to a one.

When the received PID response is acceptable and is a DATA0, then the high-bandwidth transaction is complete for this micro-frame and the host controller must set the *Active* bit to a zero. 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 host controller will set the *Babble* bit to a one.

Any time the individual transaction completes in a Timeout, the host controller will set the status bit *Active* to a zero and status bit *XacErr* to a one.

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.