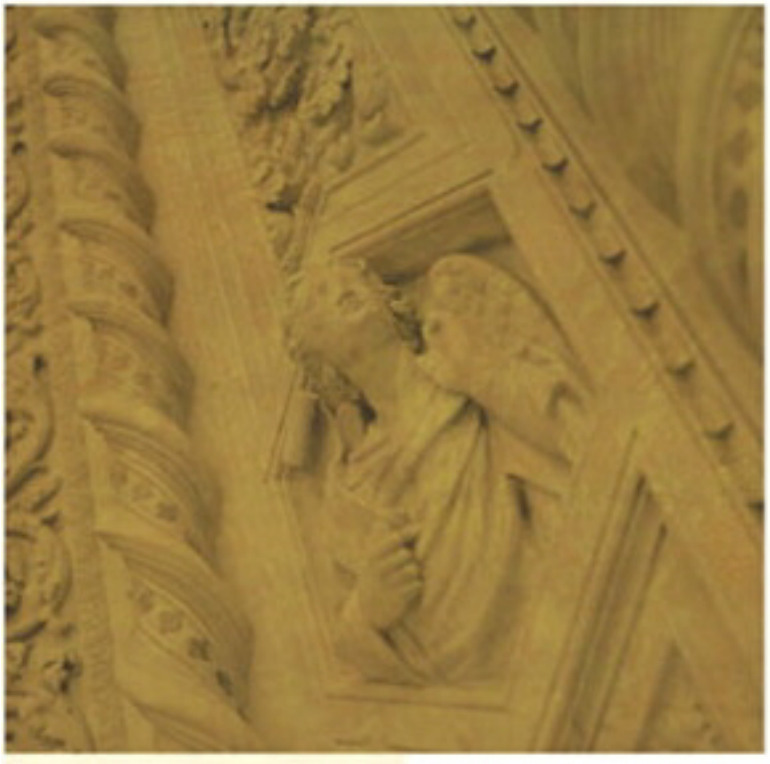
"The 'must-have' PC architecture reference set."

—*PC Magazine*'s "Read Only" column

Universal Serial Bus System Architecture



Second Edition

Includes an 85-minute USB 2.0 overview video by Don Anderson on CD-ROM

MindShare, Inc.

Don Anderson / Dave Dzatko

PC System Architecture Series

MINDSHARE
BRINGING LIFE TO KNOWLEDGE

world-class technical training

Are your company's technical training needs being addressed in the most effective manner?

MindShare has over 25 years experience in conducting technical training on cutting-edge technologies. We understand the challenges companies have when searching for quality, effective training which reduces the students' time away from work and provides cost-effective alternatives. MindShare offers many flexible solutions to meet those needs. Our courses are taught by highly-skilled, enthusiastic, knowledgeable and experienced instructors. We bring life to knowledge through a wide variety of learning methods and delivery options.

training that fits your needs

MindShare recognizes and addresses your company's technical training issues with:

- Scalable cost training
- Just-in-time training
- Training in a classroom, at your cubicle or home office
- Customizable training options
- Overview and advanced topic courses
- Reducing time away from work
- Training delivered effectively globally
- Concurrently delivered multiple-site training

MindShare training courses expand your technical skillset

- PCI Express 2.0®
- Intel Core 2 Processor Architecture
- AMD Opteron Processor Architecture
- Intel 64 and IA-32 Software Architecture
- Intel PC and Chipset Architecture
- PC Virtualization
- USB 2.0
- Wireless USB
- Serial ATA (SATA)
- Serial Attached SCSI (SAS)
- DDR2/DDR3 DRAM Technology
- PC BIOS Firmware
- High-Speed Design
- Windows Internals and Drivers
- Linux Fundamentals

... and many more.

All courses can be customized to meet your group's needs. Detailed course outlines can be found at **www.mindshare.com**

bringing life to knowledge.

real-world tech training put into practice worldwide

*PCI Express ® is a registered trademark of the PCISIG

MindShare Learning Options

MindShare Classroom

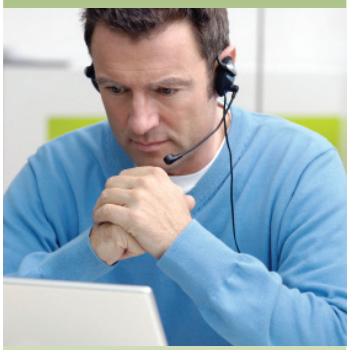
In-House Training

Public Training

Classroom Training

Invite MindShare to train you in-house, or sign-up to attend one of our many public classes held throughout the year and around the world. No more boring classes, the 'MindShare Experience' is sure to keep you engaged.

MindShare Virtual Classroom



Virtual In-House Training

Virtual Public Training

Virtual Classroom Training

The majority of our courses live over the web in an interactive environment with WebEx and a phone bridge. We deliver training cost-effectively across multiple sites and time zones. Imagine being trained in your cubicle or home office and avoiding the hassle of travel. Contact us to attend one of our public virtual classes.

MindShare eLearning

Intro eLearning Modules

Comprehensive eLearning Modules

eLearning Module Training

MindShare is also an eLearning company. Our growing list of interactive eLearning modules include:

- **Intro to Virtualization Technology**
- **Intro to IO Virtualization**
- **Intro to PCI Express 2.0 Updates**
- **PCI Express 2.0**
- **USB 2.0**
- **AMD Opteron Processor Architecture**
- **Virtualization Technology**

...and more

MindShare Press

Books

eBooks

MindShare Press

Purchase our books and eBooks or publish your own content through us. MindShare has authored over 25 books and the list is growing. Let us help make your book project a successful one.

Engage MindShare

Have knowledge that you want to bring to life? MindShare will work with you to "Bring Your Knowledge to Life." Engage us to transform your knowledge and design courses that can be delivered in classroom or virtual classroom settings, create online eLearning modules, or publish a book that you author.

We are proud to be the preferred training provider at an extensive list of clients that include:

ADAPTEC • AMD • AGILENT TECHNOLOGIES • APPLE • BROADCOM • CADENCE • CRAY • CISCO • DELL • FREESCALE
GENERAL DYNAMICS • HP • IBM • KODAK • LSI LOGIC • MOTOROLA • MICROSOFT • NASA • NATIONAL SEMICONDUCTOR
NETAPP • NOKIA • NVIDIA • PLX TECHNOLOGY • QLOGIC • SIEMENS • SUN MICROSYSTEMS SYNOPSYS • TI • UNISYS

MINDSHARE
BRINGING LIFE TO KNOWLEDGE

4285 SLASH PINE DRIVE COLORADO SPRINGS, CO 80908 USA
M 1.602.617.1123 O 1.800.633.1440 ravi@mindshare.com

www.mindshare.com

USB System Architecture (USB 2.0)

MINDSHARE, INC.

Don Anderson

ADDISON-WESLEY DEVELOPER'S PRESS

Reading, Massachusetts • Harlow, England • Menlo Park, California

Berkeley, California • Don Mills, Ontario • Sydney

Bonn • Amsterdam • Tokyo • Mexico City

Library of Congress Cataloging-in-Publication Data

ISBN: 0-201-46137-4

Sponsoring Editor:
Project Manager:
Cover Design:
Set in 10 point Palatino by MindShare, Inc.

1 2 3 4 5 6 7 8 9-MA-999897
First Printing, March, 2001

Addison-Wesley books available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government, and Special Sales Department at (800) 238-9682.

Find A-W Developer's Press on the World-Wide Web at:
http://www.awl.com/devpress/

About This Book

Part One
Overview of USB 2.0

Chapter 1: Design Goals of USB

Chapter 2: The Big Picture

Chapter 3: Cables and Connectors

Chapter 4: USB Cable Power Distribution

Part Two
Low- & Full-Speed Device Operation

Chapter 5: LS/FS Signaling Environment

Chapter 6: LS/FS Transfer Types & Scheduling

Chapter 7: Packets & Transactions

Contents

Chapter 8: Error Recovery

Chapter 9: USB Power Conservation

Part Three
High Speed Device Operation

Chapter 10: Overview of HS Device Operation

Chapter 11: The High-Speed Signaling Environment

Chapter 12: HS Transfers, Transactions, & Scheduling

Chapter 13: HS Error Detection and Handling

Chapter 14: HS Suspend and Resume

Part Four
USB 2.0 Hub Operation with LS/FS/HS Devices

Chapter 15: HS Hub Overview

Chapter 16: 2.0 Hubs During HS Transactions

Chapter 17: 2.0 Hubs During LS/FS Transactions

Part Five
USB Device Configuration

Chapter 18: Configuration Process

Chapter 19: USB Device Configuration

Chapter 20: Hub Configuration

Chapter 21: Device Classes

Part Six
USB Software Overview

Chapter 22: Overview of USB Host Software

Appendix

Appendix A: Standard Device Requests

Appendix B: Hub Requests

Appendix C: Universal Host Controller

Appendix D: Open Host Controller

1 Design Goals of USB

This Chapter

Many PCs designed today still implement peripheral devices based on interfaces used in the original IBM PC designs of the early 1980s. These implementations have numerous shortcomings that cause both designers and users considerable frustration. This chapter discusses the primary design goals of USB 2.0 and reviews the shortcomings of the legacy implementation.

The Next Chapter

The next chapter provides an overview of the primary concepts of USB transfers and describes the interaction between USB system software, system hardware, and USB devices for USB 1.x systems and for USB 2.0 systems. The USB communications process is described, including the concept of the device framework. Each hardware and software element in a USB system is introduced and its primary functions are described.

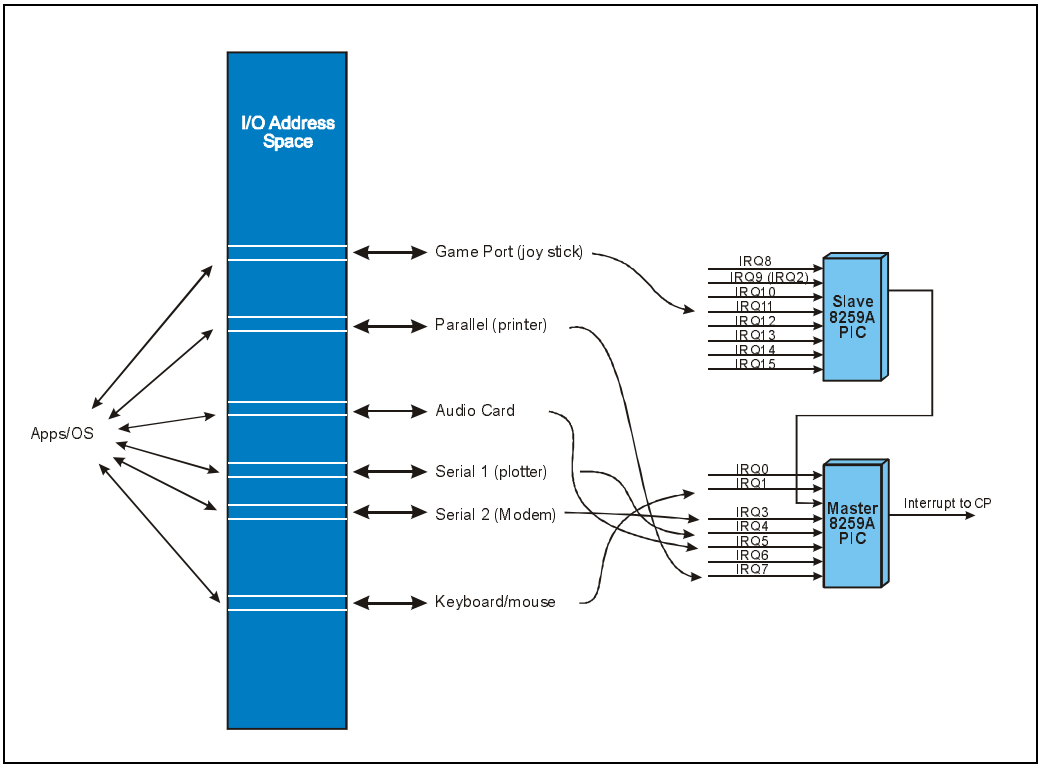
Shortcomings of the Original PC I/O Paradigm

USB emerged as a result of the difficulties associated with the cost, configuration, and attachment of peripheral devices in the personal computer environment. In short, USB creates a method of attaching and accessing peripheral devices that reduces overall cost, simplifies the attachment and configuration from the end-user perspective, and solves several technical issues associated with old style peripherals. The following sections detail the various problems associated with PC peripherals today and investigate the challenges that the USB standard faces.

Limited System Resources

Figure 1-1 on page 14 illustrates the legacy I/O paradigm where peripheral devices were typically mapped into the CPU's I/O address space and assigned a specific IRQ line, and in some cases a DMA channel. These system resources were assigned to particular peripheral devices by IBM and other manufacturers and became the standard I/O locations, IRQs, and DMA channels used by software developers to access a given device. Figure 1-1 illustrates the I/O address space interrupt assignments that are used in the PC environment, making these system resources scarce while complicating device configuration.

Figure 1-1: System Resources Used by Legacy Peripheral Devices



Another limitation in the legacy PC environment is the limited number of peripheral devices that can be attached to the standard connectors. For example, the serial and parallel connectors support single devices only, thereby limiting the number of peripherals that can be easily and inexpensively attached.

Interrupts

Perhaps the most critical system resource problem revolves around the allocation of interrupts required by the myriad of devices that are typically implemented in PCs. This is particularly true of peripheral devices that attach via the ISA bus, since the ISA bus does not reliably support sharable interrupts. Table 1-1 lists each IRQ line and the devices that typically use it. As can be seen, many of the IRQ lines are dedicated to particular devices based on legacy conventions, while other IRQ lines may be used by a variety of peripheral devices. In PCI-based systems that also contain an ISA bus, the interrupt shortage can become a major problem, because several of the IRQ lines ideally should be left available for ISA expansion cards that might require them.

Table 1-1: Typical Legacy Interrupt Lines Used by Standard Devices

IRQ Line	Devices
IRQ0	system timer (dedicated on system board)
IRQ1	keyboard (dedicated on system board)
IRQ2	cascade channel for slave interrupt controller (not available for peripheral devices)
IRQ3	serial mouse, modem, plotter, serial printer, game port, pen, infrared port
IRQ4	serial mouse, modem, plotter, serial printer
IRQ5	bus mouse, parallel printer, sound card, LAN adapter, tape drive, game port
IRQ6	floppy drive
IRQ7	parallel printer
IRQ8	RTC alarm (dedicated on system board)
IRQ9	LAN adapter, video adapter, tape drive, game port
IRQ10	LAN adapter, sound card
IRQ11	LAN adapter, SCSI controller, PCMCIA controller
IRQ12	PS/2 mouse, PCMCIA controller

Table 1-1: Typical Legacy Interrupt Lines Used by Standard Devices

IRQ Line	Devices
IRQ13	numeric coprocessor errors (dedicated on system board)
IRQ14	hard drive
IRQ15	SCSI controller, PCMCIA controller

I/O Addresses

I/O address conflicts are also quite common in the PC environment. Note that peripheral devices usually require a block of I/O address locations to report status information and to issue commands to the device. While it's true that x86 processors have the ability to access 64KB of I/O address locations (more than enough for all peripheral devices), legacy ISA expansion cards typically decode only 10 of the 16 address lines available. This yields a maximum 1KB block of address space that is usable by ISA expansion devices. Furthermore, the limited decode creates the well known aliasing effect that renders the upper 768 bytes of each aligned 1KB of I/O space unusable by other devices. See MindShare's *ISA System Architecture* book, published by Addison-Wesley, for details.

Non-shareable Interfaces

Standard PC peripheral interfaces (e.g., serial and parallel connections) support the attachment of a single device. Since only one peripheral device can be attached at any given time, the flexibility of such connections is minimized. This limitation frequently leads to the costly decision of building an expansion card that plugs into an expansion bus (e.g., ISA or PCI) to create an attachment point for a new peripheral design.

End User Concerns

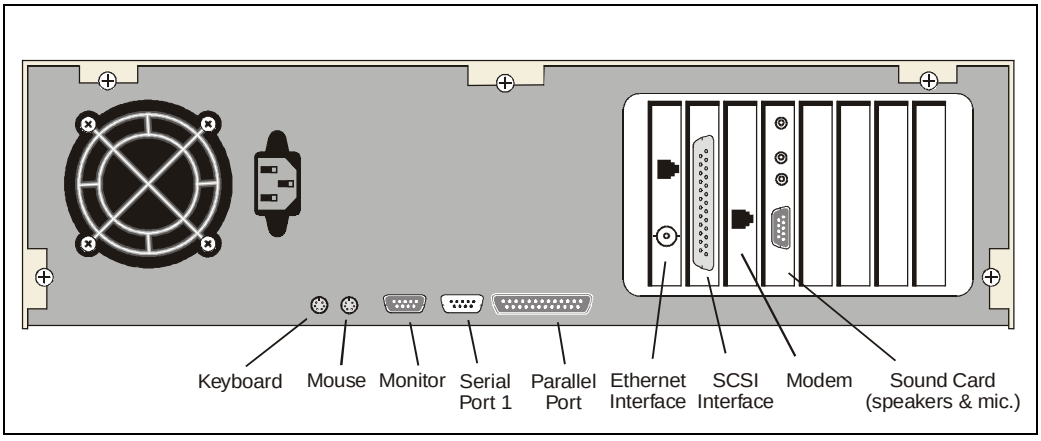
End users are faced with a variety of problems when connecting peripherals to their PCs. These concerns include:

- Too many connector/cable types
- System must be shut down to attach most peripherals
- System must be restarted to install/load software
- Cost

Cable Crazed

Dedicated cables are required for the mouse, keyboard, printer, external modem, Zip drive, plotter, etc., most of which are completely different. Figure 1-2 on page 17 illustrates the backplane of a typical PC before USB. The variety of different connectors and cables required to connect particular peripheral devices is inconvenient and confusing.

Figure 1-2: Connectors at Backplane



Installation and Configuration of Expansion Cards

When peripherals are purchased, many of them require the installation of expansion cards. This, of course, may involved removing the cover of the PC, setting the switches and jumpers to configure the card, inserting the card, and replacing the cover. The trouble only begins there. Once the system is powered up the software for this device may have to be installed from diskette, which can also be a frustrating process for novice and experienced user alike.

No Hot Attachment of Peripherals

When most legacy I/O devices are attached to the system, they will not work without first restarting the system. Restarting the system is required so that the new peripheral can be detected by software. In the process, system resources must be selected and assigned to the new device (e.g., I/O space, IRQ line, and DMA channel) in order for it to work correctly and to ensure that the resource selected is not already being used by another device in the system.

2 The Big Picture

The Previous Chapter

Many PCs designed today still implement peripheral devices based on interfaces used in the original IBM PC designs of the early 1980s. These implementations have numerous shortcomings that cause both designers and users considerable frustration. The previous chapter discussed the primary design goals of USB 2.0 and reviewed the shortcomings of the legacy implementation.

This Chapter

This chapter provides an overview of the primary concepts of USB transfers and describes the interaction between USB system software, system hardware, and USB devices for USB 1.x systems and for USB 2.0 systems. The USB communications process is described, including the concept of the device framework. Each hardware and software element in a USB system is introduced and its primary functions are described.

The Next Chapter

USB defines a single connector type for attaching all USB peripherals to the host system. The next chapter introduces the physical aspects of USB connectors and cables.

Overview

Figure 2-1 on page 26 provides a system view of USB implemented in a PCI-based system. In this implementation the USB host controller resides on the PCI bus. The controller acting as a bus master obtains data structures from memory that describe the USB transactions that have been scheduled by system software for delivery over the USB.

Figure 2-2 on page 27 depicts a hub-oriented chip set with the USB controller integrated into the I/O Hub chip. The high-speed link between the I/O Hub and the Memory Hub permit higher bandwidth between the I/O subsystem

and memory than a typical PCI bus and may be better suited to meet the bandwidth needs of USB 2.0.

Figure 2-1: USB System Implemented in a PCI-Based Platform

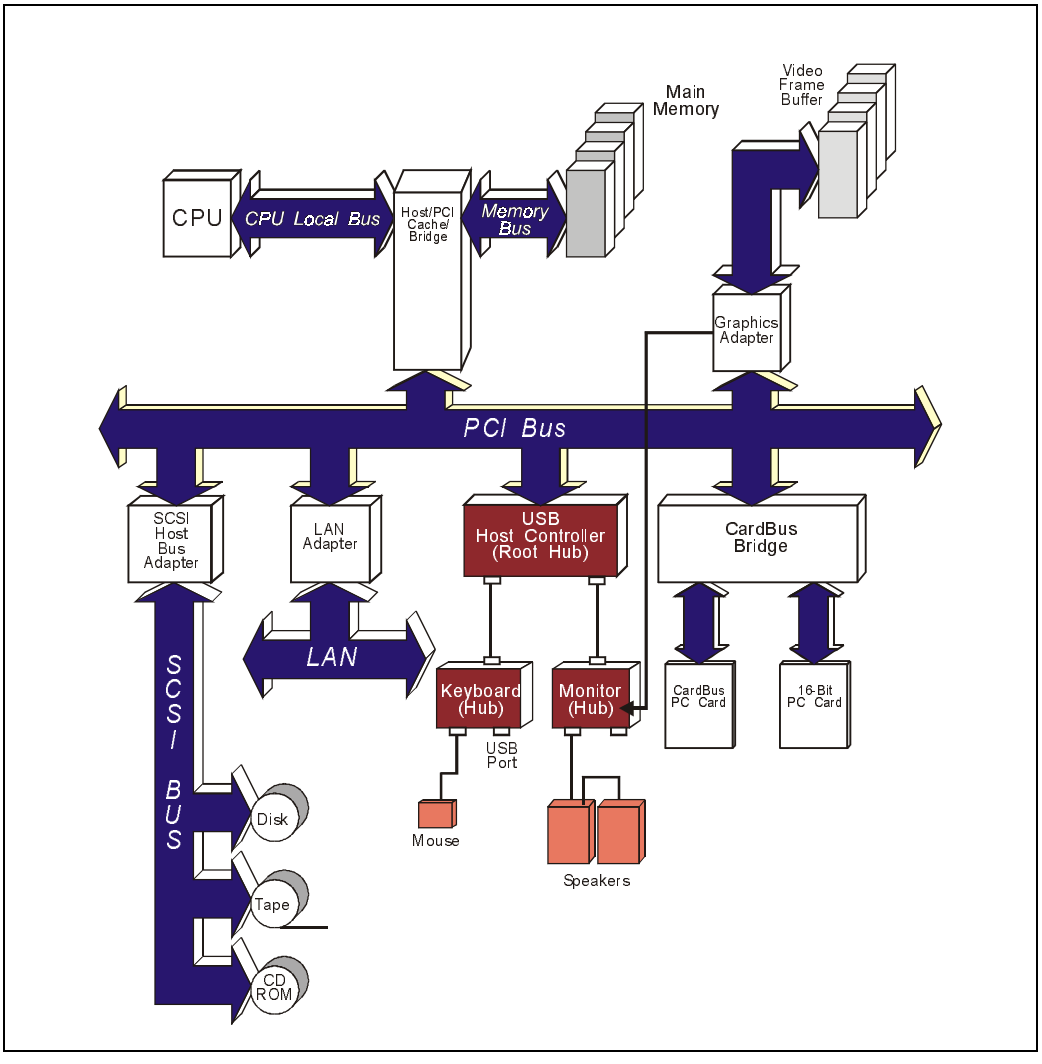
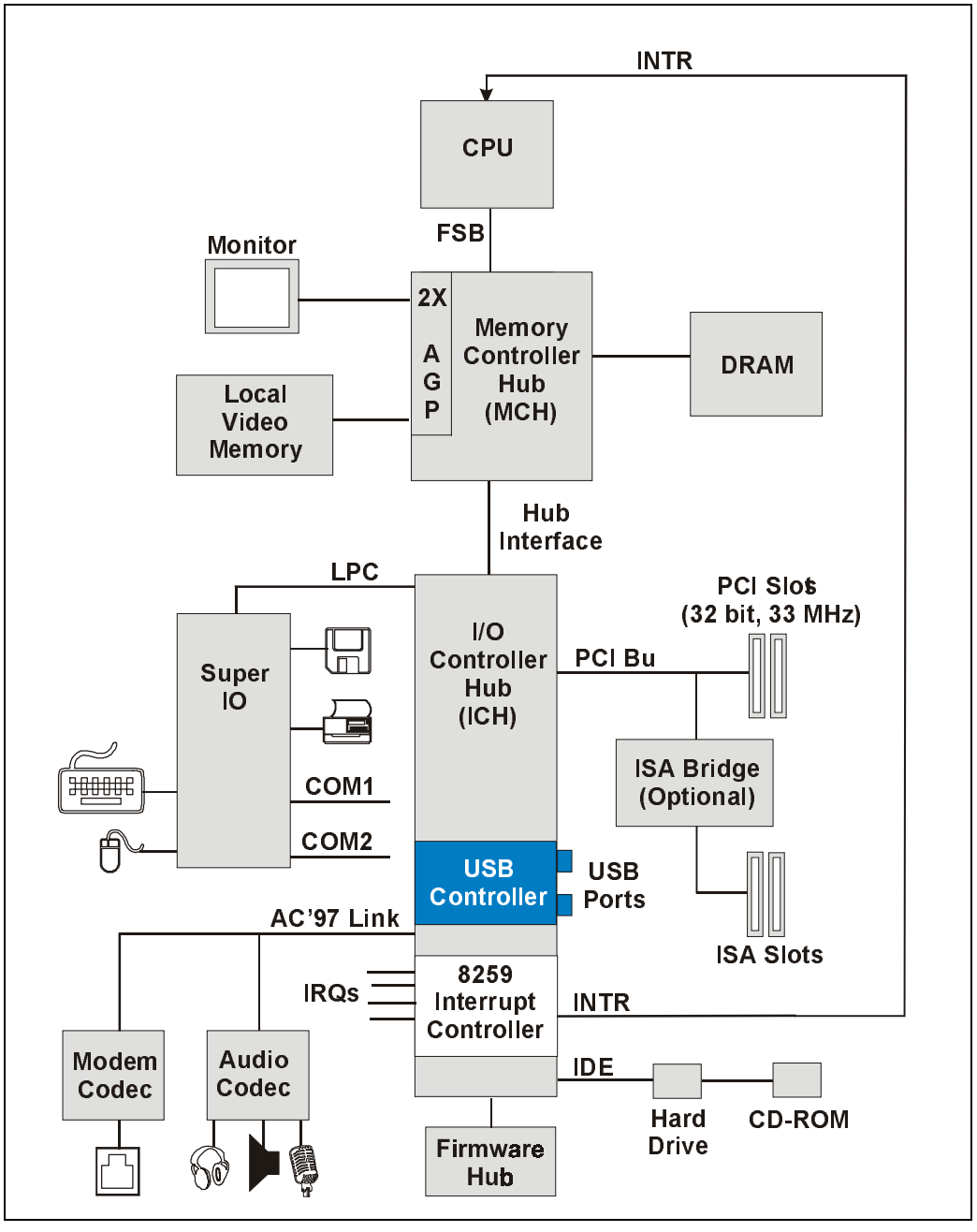


Figure 2-2: USB Controller Integrated into I/O Hub Chip



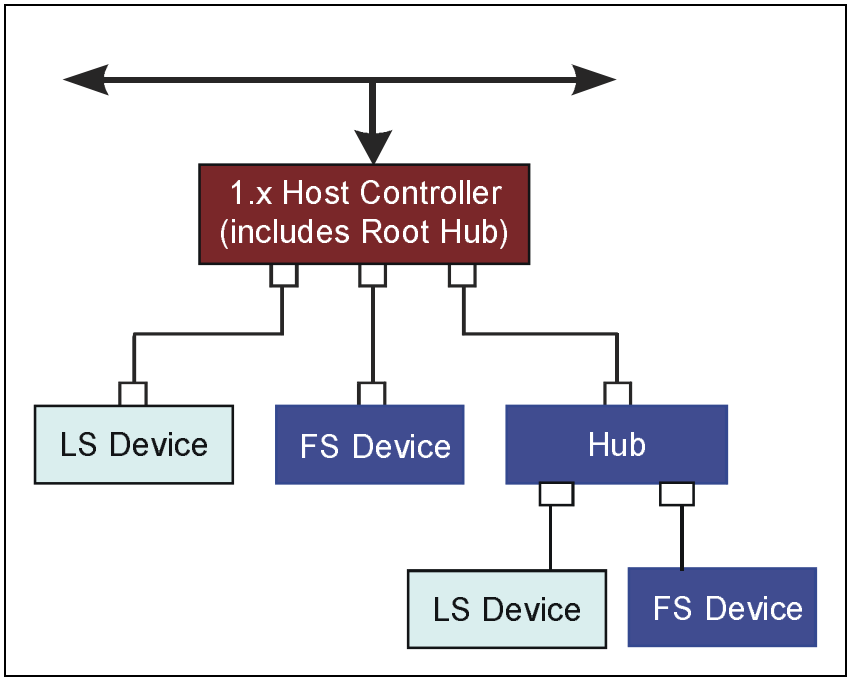
USB 1.x Systems and Devices

This section provides an overview of low- and full-speed system and device operation. Later portions of the book provide much greater detail regarding the implementation of these devices.

Low-Speed and Full-Speed Devices

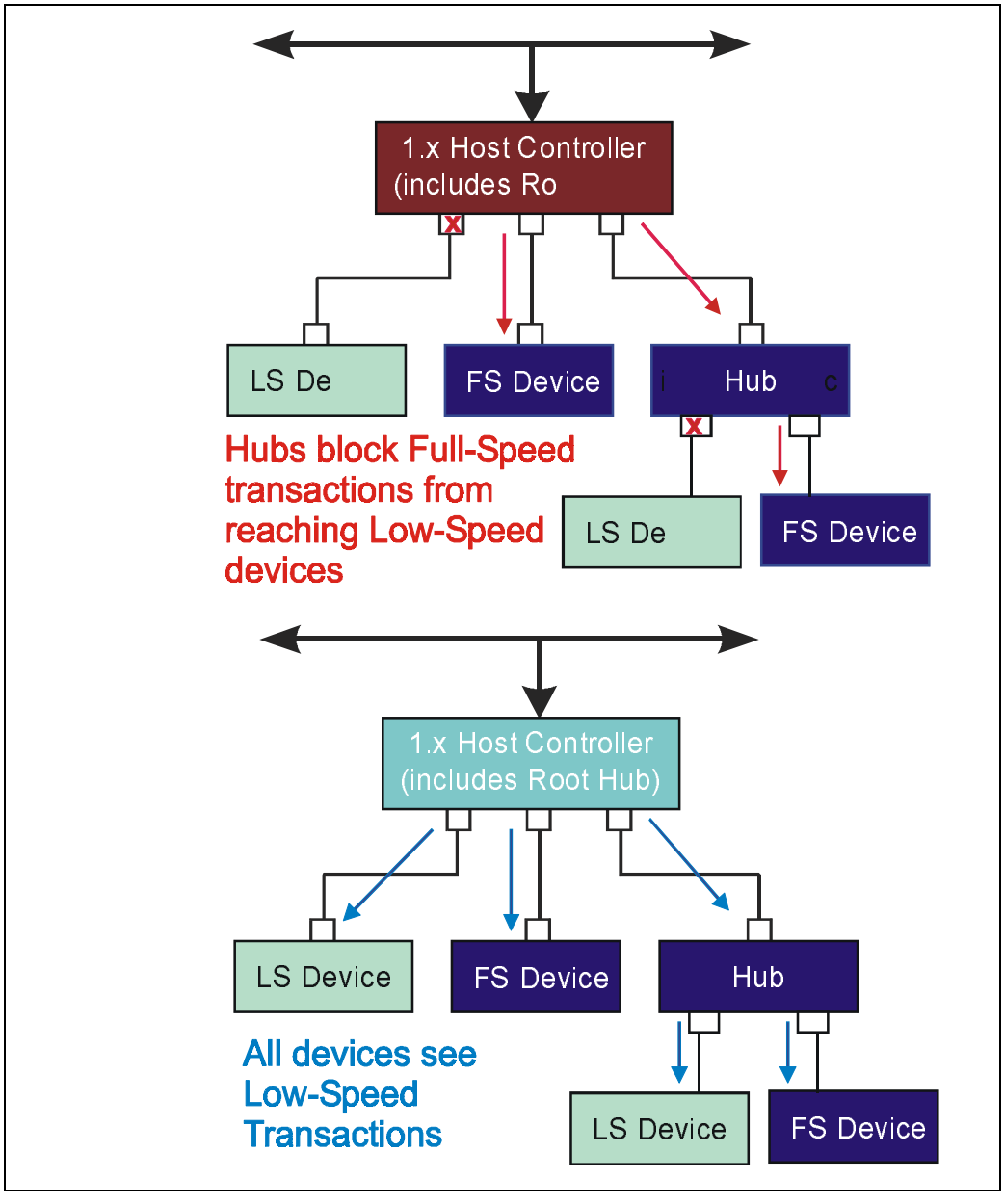
USB 1.0 and 1.1 (i.e., 1.x) systems can support only 1.5Mb/s (low speed) and 12Mb/s (full-speed) transactions as illustrated in Figure 2-3. The host delivers low- or full-speed transactions depending on the speed of the device being accessed.

Figure 2-3: 1.x Systems Support Only Low- and Full-Speed Devices



When full-speed transactions are performed, these transactions are prevented from reaching the low-speed devices that otherwise might be confused by a full-speed transaction. Conversely, low-speed transactions can safely be transferred to full-speed devices. (See Figure 2-4 on page 29.)

Figure 2-4: Full-Speed Transactions Do Not Reach Low-Speed Devices



3 Cables and Connectors

The Previous Chapter

The previous chapter provided an overview of the primary concepts of USB transfers and described the interaction between USB system software, system hardware, and USB devices for USB 1.x systems and for USB 2.0 system. The USB communications process is described, including the concept of the device framework. Each hardware and software element in a USB system is introduced and its primary functions are described.

This Chapter

USB defines a single connector type for attaching all USB peripherals to the host system. This chapter introduces the primary mechanical elements of USB connectors and cables.

The Next Chapter

The next chapter discusses USB power distribution, along with issues related to bus-powered devices and the operation of self-powered devices. The chapter also discusses the role of host software in detecting and reporting power-related problems.

The Connectors

USB connectors are designed to permit any USB peripheral device to be attached to a hub port. Hub ports will be located at the back of the computer or may be associated with other peripheral devices such as monitors and printers, or are available on stand-alone hub devices.

Many USB peripherals have the USB cable permanently attached, while others have detachable USB cables. If the same connector were used on both ends of a

USB cable, it would be possible to connect the cable between two USB ports. To prevent a detachable cable from being plugged into two USB ports at the same time, a separate connector has been designed for the peripheral cable connection. The two connector types are described below:

- Series A connectors — provide the USB port connection to the USB peripheral cable. The series A receptacle is implemented as the hub port connector, while the series A plug is attached to the peripheral cable, permitting attachment of a USB peripheral device.
- Series B connectors — provide the cable connection to the USB peripheral device when a detachable cable is implemented. The series B receptacle is implemented at the peripheral and the series B plug is attached to the cable. A small form-factor B connector called the "Mini-B" connector has been defined. See MindShare's website for a direct link to the specification for this new connector.

Figure 3-1: A View of the Series A Plug

Each connector has four contacts: two for carrying differential data and two for powering the USB device. Note that the power contacts are longer than the data contacts to ensure that a USB device receives power prior to the data contacts mating (power pins are 7.41mm and the data pins are 6.41mm).

The connector contacts are numbered and the cable conductors are color coded for easy identification, as listed in Table 3-1.

Table 3-1: Connector Pin Designations

Contact Number	Signal Name	Cable Color
1	Vcc	Red
2	-Data	White
3	+Data	Green
4	Ground	Black

Series A Connectors

Series A connectors are used to connect a peripheral cable to a USB hub port. The receptacle comes in four variants and can be obtained in through-hole or Surface Mount Technology (SMT) versions. The four variant are:

- Vertical Mount
- Right Angle Mount
- Stacked Right Angle Mount
- Panel Mount

Series B Connectors

The series B connector is implemented in peripherals that have detachable cables. The specification does not define mounting variations for the series B receptacle; however, the author suspects that the same variants defined for the series A receptacle apply to the series B receptacle.

Cables

The USB specification defines two cables for compliant signaling. The low-speed cable is defined for 1.5Mb/s signaling and the full-speed cable defined by the 1.1 USB specification that supports both full- and high-speed transmission. The low speed cable permits a more economical cable implementation for low-speed/low-cost peripherals such as mice and keyboards. The following sections detail the characteristics of each cable type.

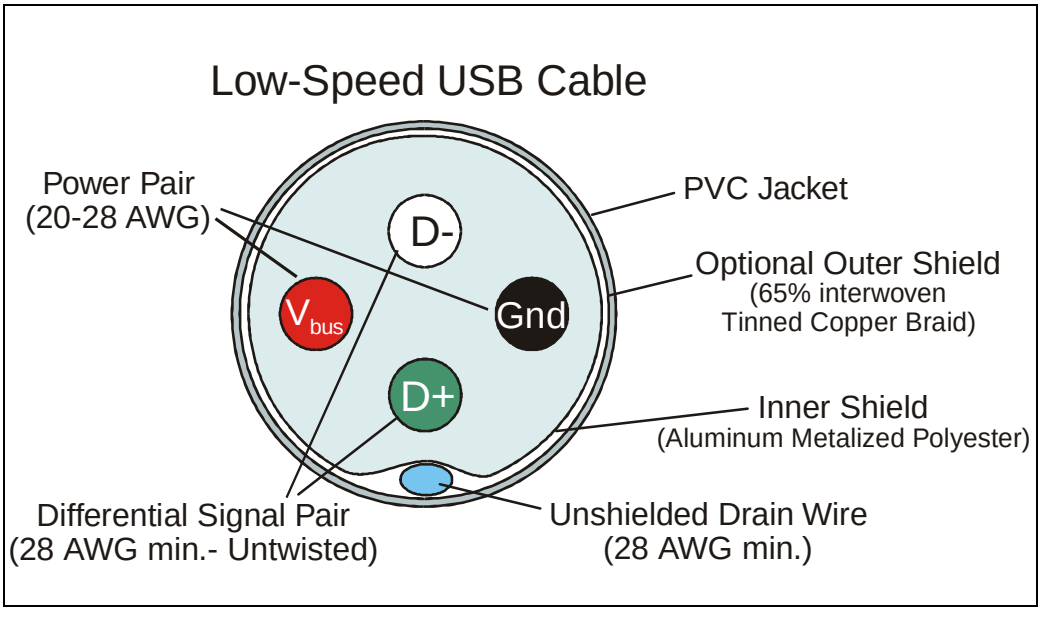
Low-Speed Cables

Figure 3-2 illustrates the cross section of a low-speed cable, sometimes referred to as a sub-channel cable. These cables are intended only for 1.5Mb/s signaling and are used in applications where the wider bandwidths are not required.

The differential data signaling pair may be non-twisted 28 AWG stranded conductors. In addition, low-speed cables require an inner shield (with the conducting side out) and drain wire that contacts the inner shield. The drain wire is attached to the plug and socket case. The outer shield is recommended but not required by the specification.

Low-speed cables are limited in the specification to 3.0 meters and must have the maximum propagation delay no greater than 18ns (one-way). The maximum cable length is a function of the maximum rise and fall times defined for low-speed signaling and the capacitive load seen by the low-speed drivers. Refer to the specification for details regarding these parameters.

Figure 3-2: Cross Section of a Low-Speed Cable Segment



Full- and High-Speed Cables

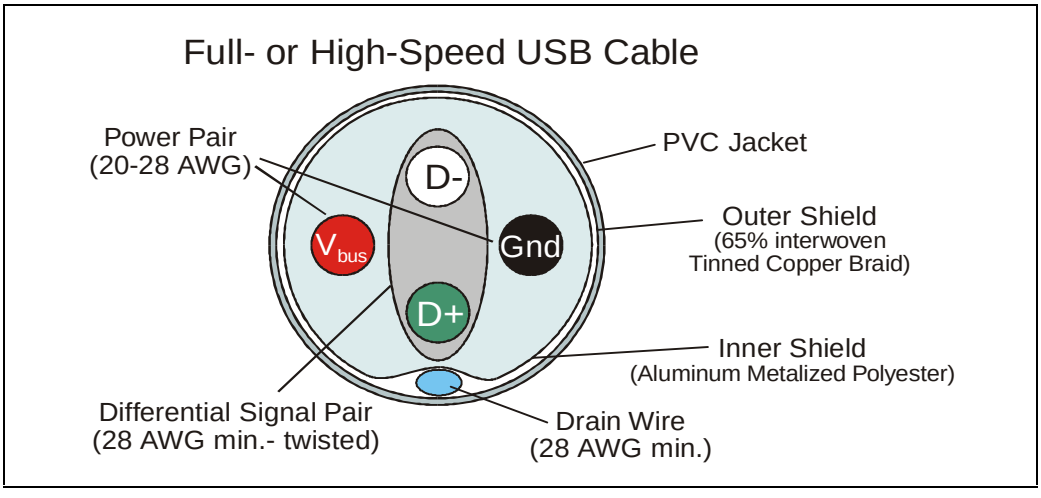
Full-speed and high-speed USB devices require “twisted pair” for the differential data lines, along with inner and outer shielding and the drain wire as illustrated in Figure 3-3. The maximum propagation delay must be equal to or less than 26ns over the length of the cable when operating in the frequency range of 1-480MHz. If the cable cannot meet the propagation delay limit of 26ns then the cable must be shortened as shown in Table 3-2 (see the specification for details).

Table 3-2: Cable Propagation Delay

Cable Propagation Delay	Maximum Cable Length
9.0ns / m	3.3m
8.0ns / m	3.7m
7.0ns / m	4.3m
6.5ns / m	4.6m

The maximum cable length supported for full- and high-speed cables is 5.0 meters. This length is determined by the propagation delay of the cable as mentioned above and the attenuation of the signal pair.

Figure 3-3: Cross Section of a High-Speed Cable Segment



4 USB Cable Power Distribution

The Previous Chapter

USB defines a single connector type for attaching all USB peripherals to the host system. The previous chapter introduced some of the physical and electrical properties of USB connectors and cables.

This Chapter

This chapter discusses USB power distribution, along with issues related to bus-powered devices and the operation of self-powered devices. The chapter also discusses the role of host software in detecting and reporting power-related problems.

The Next Chapter

USB employs NRZI encoding and differential signaling to transfer information across USB cables. The next chapter discusses the low- and full-speed signaling environment, including the differential signaling and NRZI encoding techniques used by the USB. The signaling environment must also support a wide range of other signal-related functions such as: detecting device attachment and removal, suspending and resuming operation, resetting a device, and others, all of which are discussed in this chapter. Note that high-speed devices are discussed later in this book.

USB Power

All USB ports supply power for devices that are attached. Peripheral devices and hubs can be attached to a given port and use the available cable power or can implement their own power supply. This chapter details both cable- and self-powered hubs and devices.

Hubs

A major function associated with a hub is the distribution and control of cable power. Hubs may derive all power from the upstream cable or may include their own power supply for powering downstream ports. Hubs must also provide voltage regulation and ensure current is limited to downstream ports for safety considerations.

Hubs, like all USB devices, must include descriptors to specify their capabilities. A major portion of a hub's descriptor definition relates to power-related issues.

Current Budget

A fully rated port must be able to provide five units of current (500ma) to the attached device. Self-powered hubs (including the root hub) having their own local power supply can provide the maximum rated power to each port. However, bus-powered hubs have only the bus power that they receive from the upstream cable to distribute to all of their USB ports. This can severely limit the amount of current that is available for USB devices that attach to bus-powered hub ports. The minimum current available at a port is 100ma.

Hubs specify whether they are self-powered as part of their configuration descriptor as shown in Table 4-1. The shaded area shows a bit-mapped "Attribute" field that defines the hub's power implementation. Note that bit 7 is now reserved and must be set to 1. This setting in the 1.x environment indicated whether the device was bus powered.

If a device loses its external power source, it must not consume additional power from the bus to make up the deficit such that the bus power consumed exceeds the amount reported in the "MaxPower" field of the configuration descriptor. If the device cannot continue to operate with local power removed, it will no longer respond to accesses and software will be notified of the failure. USB system software may be able to detect that local power has been removed by checking device status.

Table 4-1: Source of Hub Power Defined in Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Number	Size of this descriptor in bytes.
1	DescriptorType	1	02h	CONFIGURATION = 2
2	TotalLength	2	Number	Total length of data returned for this configuration. Includes the combined length of all descriptors (configuration, interface, endpoint, and class or vendor specific) returned for this configuration.
4	NumInterfaces	1	Number	Number of interfaces supported by this configuration.
5	Configuration-Value	1	Number	Value to use as an argument to Set Configuration to select this configuration.
6	Configuration	1	Index	Index of string descriptor describing this configuration.
7	Attributes	1	Bitmap	Configuration characteristics D7 Reserved (must be set to 1) (Bus Powered bit in 1.x) D6 Self Powered D5 Remote Wakeup D4:0 Reserved (reset to 0) A device configuration that uses power from the bus and a local source must have a non-zero value in the MaxPower field. If a device configuration supports remote wakeup, D5 is set to one (1).
8	MaxPower	1	X2ma	Maximum amount of bus power this hub will consume in this configuration (value based on 2ma increments).

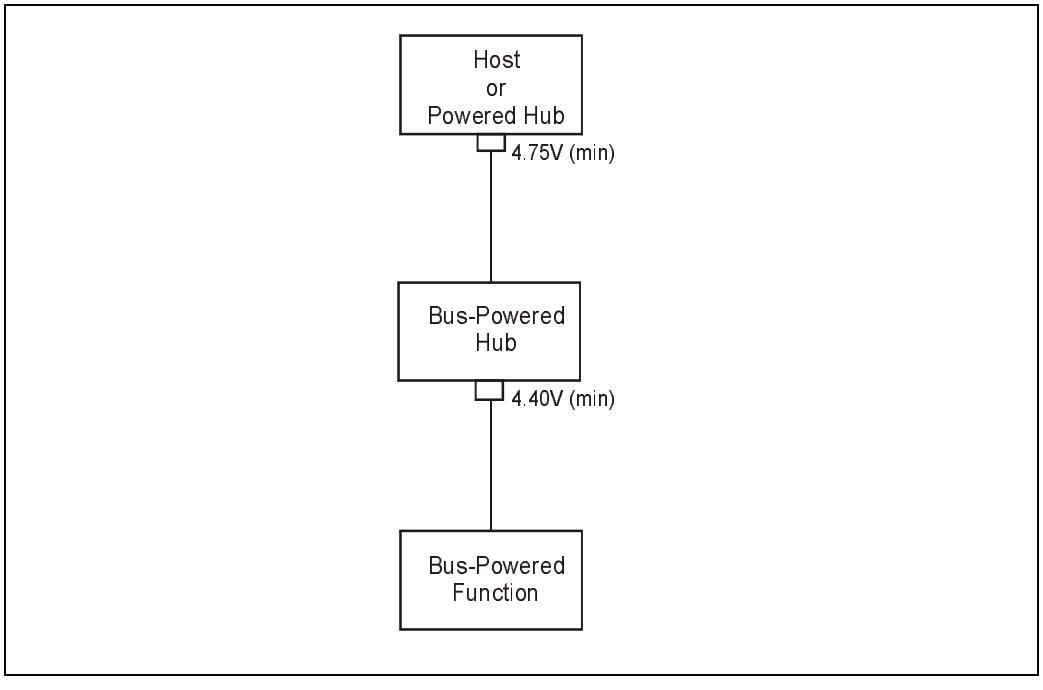
Over-Current Protection

USB ports must be current limited due to safety regulations. No more than five amps of current can be supplied to a single port due to personal safety concerns. Current protection can be ganged to multiple ports or done on a per port basis as long as the current protection satisfies the 5a limit. Note also that a bus-powered hub has only the power it receives from the cable to distribute to USB ports. In this instance, no current limiting is needed.

Voltage Drop Budget

Power may be supplied to USB peripheral devices via the cable. Voltage at a powered hub port can be no less than 4.75Vdc, while voltage at a bus-powered hub may be no lower than 4.40Vdc as illustrated in Figure 4-1. Consequently, USB devices must operate properly with as little as 4.40 volts at the upstream end of their cable.

Figure 4-1: Minimum Cable Voltage and Voltage Drop Budget



Power Switching

Hubs apply power to ports in one of the following ways:

- Direct power — not switched
- Ganged switching to all ports in common
- Individual port power switching

The power switching supported by a given hub is specified within the hub's endpoint zero descriptor. Table 4-2 on page 79 shows a portion of the hub's endpoint descriptor. Note that data bits D0 and D1 at offset 3 define the power switching mode supported by the hub.

Table 4-2: Power Switching Mode Supported Is Defined by the Hub Class Descriptor

Offset	Field	Size	Description
0	DescLength	1	Number of bytes in the descriptor, including this byte.
1	Descriptor-Type	1	Descriptor Type
2	NbrPorts	1	Number of downstream ports that this hub supports.
3	HubCharacteristics	2	D1:D0 Power Switching Mode 00 Ganged power switching (all ports powered at once) 01 Individual port power switching 1X No power switching (ports always powered on when hub is on, and off when hub is off). D15:D2 Defines other hub characteristics

5 LS/FS Signaling Environment

The Previous Chapter

The previous chapter discussed USB power distribution, along with issues related to bus powered devices and the operation of self-powered devices. The chapter also discussed the role of host software in detecting and reporting power-related problems.

This Chapter

USB employs NRZI encoding and differential signaling to transfer information across USB cables. This chapter discusses the low- and full-speed signaling environment, including the differential signaling and NRZI encoding techniques used by the USB. The signaling environment must also support a wide range of other signal-related functions such as: detecting device attachment and removal, suspending and resuming operation, resetting a device, and others, all of which are discussed in this chapter.

The Next Chapter

USB supports four transfer types: interrupt, bulk, isochronous, and control. These transfer types and the process used to initiate and perform them are described in the next chapter.

Overview

The LS/FS signaling interface provides the means to support a wide variety of events including:

- Detection of LS device attachment
- Detection of FS device attachment
- Resetting the device

- Start of Packet (SOP) recognition
- End of Packet (EOP) recognition
- Differential data signaling
- Data encoding and recovery

The signaling interfaces for full-speed and low-speed devices are similar but have a few important differences that will be discussed in the following sections. Figure 5-1 illustrates the signaling interface for a hub and an attached FS device. The primary elements of the signaling interface include:

- Differential drivers
- Differential receivers
- Single-ended receivers
- Pull-down resistors on data lines at hub (15KΩ)
- Pull-up resistor on D+ at full-speed device (1.5KΩ)

Detecting Device Attachment and Speed Detect

Before transferring information to or from a given USB device, host software must first detect its presence. USB is designed to detect the attachment of devices to the USB, after which transactions may be initiated by host software to configure the device for normal operation.

USB hubs monitor each port to observe a connect or disconnect event. Device attachment can only be detected when power has been applied to the port. Figure 5-2 illustrates a USB port interface when no device is attached. The pull-down resistors on the D+ and D- lines ensure that both data lines are near ground. When no device is attached, the single-ended receivers detect an electrical low on both data lines. USB devices must include a pull-up resistor on either D+ or D- (depending on its speed) to enable connect detection.

Figure 5-1: Signaling Interface USB Hub and Attached USB Full-Speed Device

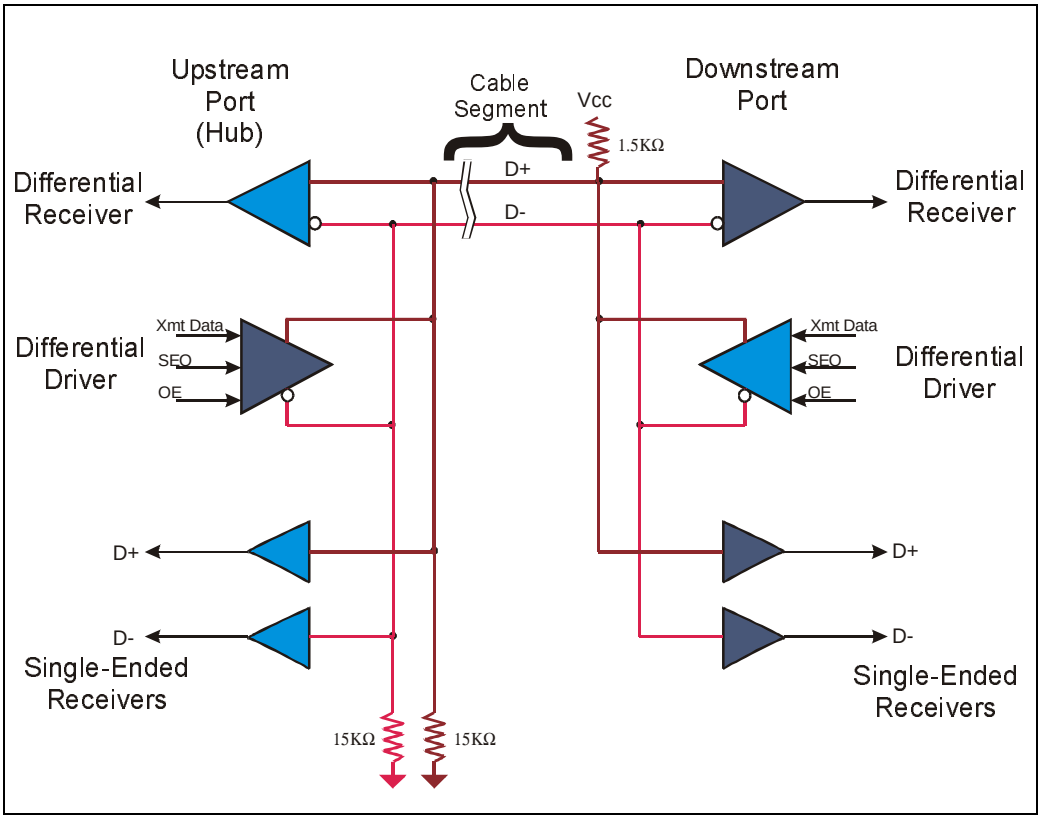


Figure 5-2: Hub Port with No Device Connected

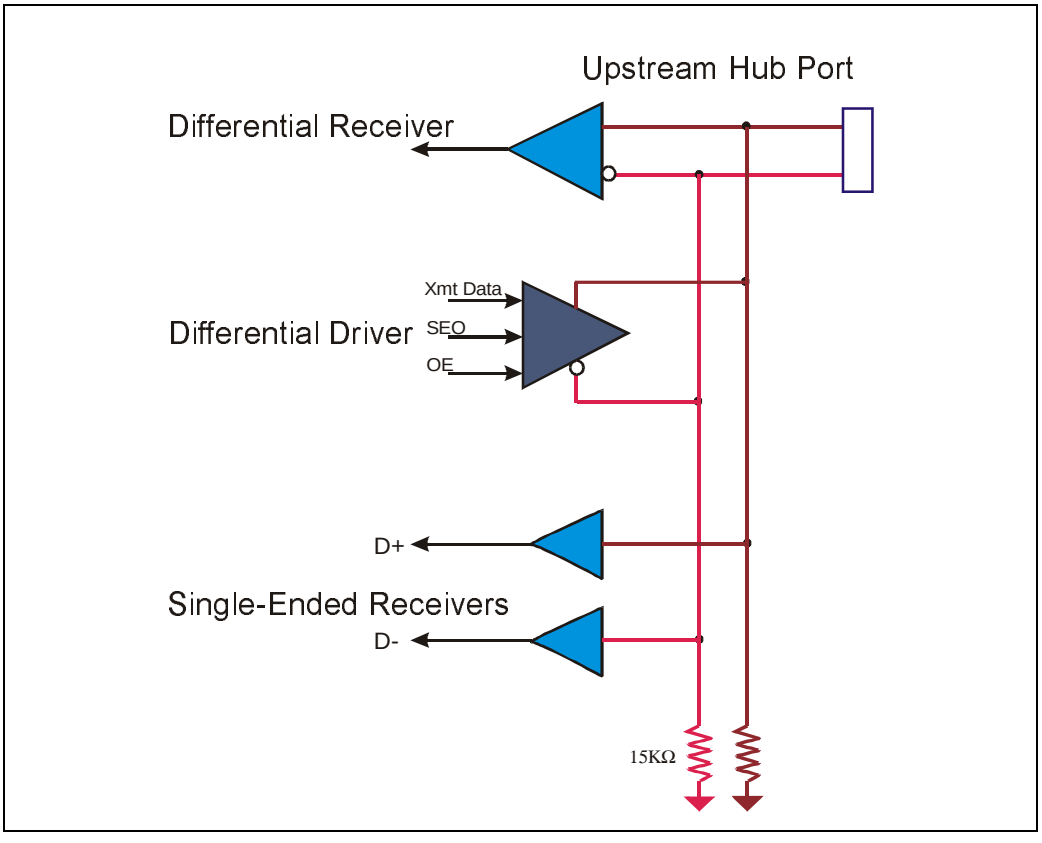
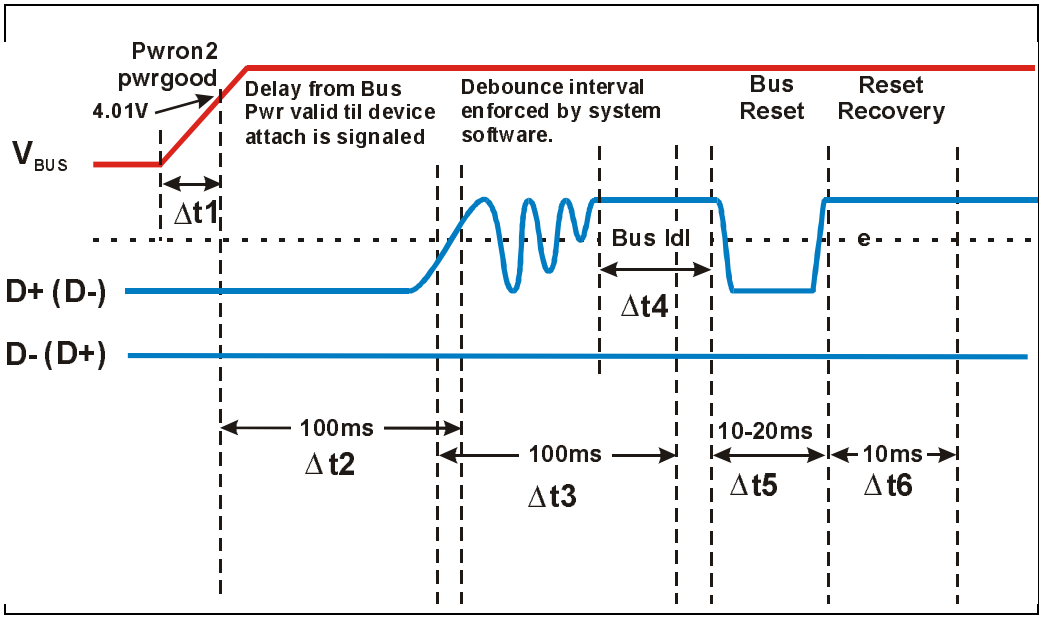


Figure 5-3 on page 97 depicts the D+ and D- state changes that occur during the attachment sequence. The sequence presumes that a full-speed device has been previously attached to a hub port prior to power being applied to the port (this would be the case during system power-up). The initial state of the port is unpowered and the sequence begins with software applying power to the port. Several delays exist that software must be aware of so that hub status is not checked prior to the device signaling connect and the hub detecting the connection event and setting connect status. These timing events are:

- Δt1 — specifies the time required for a hub to apply valid power to a port once a SetPortPower command has been issued. This value is reported by hubs via their hub class descriptor.

- Δt2 — specifies the delay from port power valid till D+(D-) is pulled above V_{IH} at the hub port receiver.
- Δt3 — this interval ensures that the signals are debounced. This debounce interval is enforced by software, after which software can safely check hub status to determine if a device is attached to the port just powered. Note that this paramter is defined for hot plug purposes. In this instance, the power pins make contact first when a device is plugged into a hub port and the data lines connect last. The data lines will make and break contact due to the connectors scraping together as the plug is inserted, thus causing the signals to bounce.
- Δt4 — after D+(D-) has settled (during Δt3) the bus assumes the idle state until Δt3 has expired. During this time the device enters the suspended state after 3ms of bus idle.
- Δt5 — software issues a PortReset command to the hub, which in turn signals RESET by driving D+ and D- on low for >10 ms and <20ms. This forces the device into its default state.
- Δt6 — this interval is called reset recovery time. The hub sets status when it is ready for access following reset. During this interval the device will enter its suspend state.

Figure 5-3: Connect Sequence from Port Power through Device Reset



6 LS/FS Transfer Types & Scheduling

The Previous Chapter

USB employs NRZI encoding and differential signaling to transfer information across USB cables. The previous chapter discussed the low- and full-speed signaling environment, including the differential signaling and NRZI encoding techniques used by the USB. The signaling environment must also support a wide range of other signal-related functions such as: detecting device attachment and removal, suspending and resuming operation, resetting a device, and others, all of which were discussed in the previous chapter.

This Chapter

USB supports four transfer types: interrupt, bulk, isochronous, and control. These transfer types and the process used to initiate and perform them are described in this chapter.

The Next Chapter

Every transfer broadcast over the USB consists of a combination of packets. These packets are combined to define individual transactions that are performed as part of a larger transfer. Each transaction type is defined, along with the individual packets that compose it.

Overview

Each endpoint within a given USB device has particular characteristics that dictate how it must be accessed. The transfer characteristics relate to the requirements of the application. The following four transfer types have been defined by the USB specification, each of which reflects the nature of transfers that may be required by a USB device endpoint:

- Interrupt transfer — an interrupt transfer is used for devices that are typically thought of as interrupt-driven devices in legacy PC implementations. USB devices that are interrupt driven must be polled periodically to see if the device has data to transfer. For example, in legacy PC systems a hardware interrupt is generated each time a key is pressed on the keyboard to notify the processor that it must execute a software interrupt routine to service the keyboard, whereas a USB keyboard is polled periodically to determine if keyboard data (e.g., resulting from a key being pressed) is ready to be transferred. The polling rate, of course, is critical; it must be frequent enough to ensure that data is not lost but not so frequent that bus bandwidth is needlessly reduced.
- Bulk transfer — a bulk transfer is used for transferring large blocks of data that have no periodic or transfer rate requirement. An example of a bulk transfer is a print job being transferred to a USB printer. While transfer speed is important for performance, a print job delivered slowly does not result in lost or corrupted data.
- Isochronous transfer — an isochronous transfer requires a constant delivery rate. Devices that use isochronous transfers must ensure that rate matching between the sender and receiver can be accomplished. For example, a USB microphone and speaker would use isochronous transfers to ensure that no frequency distortion results from transferring data across the USB.
- Control transfers — control transfers are used to transfer specific requests to USB devices and are most commonly used during device configuration. A special transfer sequence is used to pass requests (commands) to a device, sometimes followed by a data transfer, and always ending with completion status.

A given device may have a collection of endpoints, each of which may support a different transfer type. For example, when a file manager program accesses a USB-based CD-ROM, the data endpoint is defined as a bulk transfer endpoint, whereas accesses performed by a CD audio program would require isochronous transfers be performed from a data endpoint.

Client Initiates Transfer

During the configuration process, the USB driver reads the device descriptors to determine the type of endpoints that a given device requires for its function. The USB driver determines if sufficient bus bandwidth is available to accommodate all endpoint transfer requirements. If the bandwidth is available, the USB driver establishes a communications pipe with the bus bandwidth reservation associated with the pipe. The driver also apprises the appropriate USB device driver that the communications pipe exists for its use.

Communications Pipes

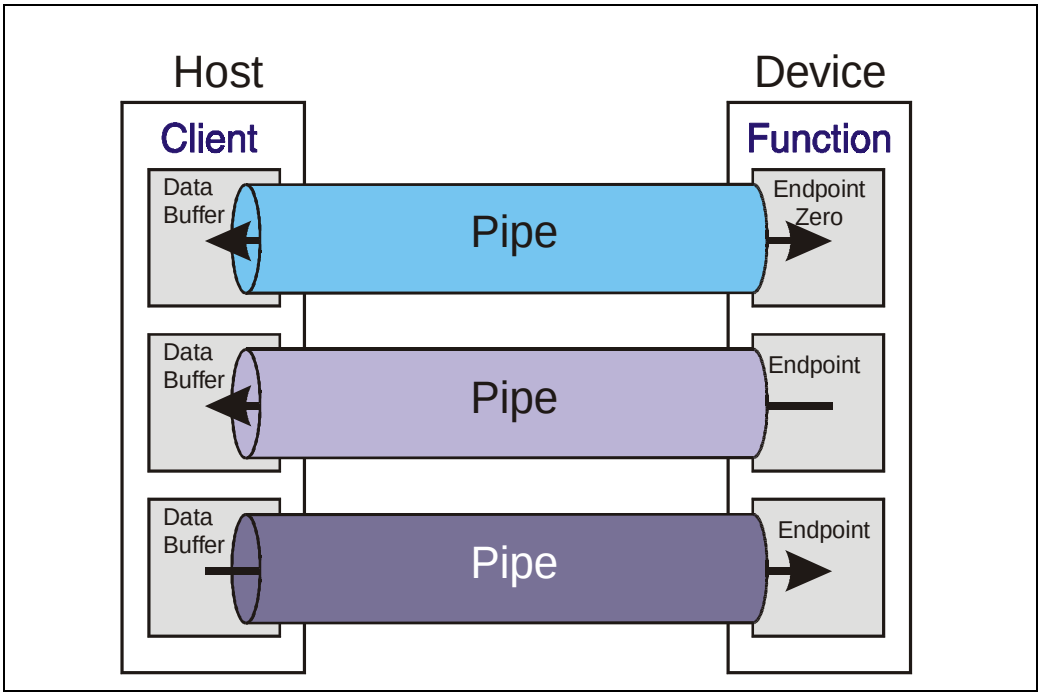
Figure 6-1 on page 120 illustrates the concept of the communications pipes that have been established for the client driver to communicate with its function. The communications pipes remain inactive until the client requests a transfer via one of the pipes. Note also that the USB device driver knows only about communications pipes and the endpoints that it must communicate with, and is not aware of the nature of the low-level USB transfer mechanisms.

The USB specification classifies a communications pipe as either a stream or a message pipe:

- Streaming pipes — The characteristic of a streaming pipe is that USB imposes no particular format relative to the actual data being transferred. Isochronous, Interrupt, and bulk endpoints fall into this category. Data delivered to or from these endpoints may have specific structures or formats, but these would be class or vendor-specific.
- Message pipes — A message pipe has a specific structure defined by USB. Communication with a control endpoint requires a specific structure and sequence, and the data patterns sent to the endpoint define requests or commands that are issued to a device. These requests specify that the device must take some action.

Note that in Figure 6-1 on page 120 the pipe at the top of the illustration is targeting endpoint zero (the default control endpoint) that every device must implement. This communications pipe is bidirectional to allow messages to be passed in both directions. The other transfer pipes are always unidirectional. This means that a device such as a read / write mass storage device must implement two endpoints for bi-directional data transfer: one for reads and one for writes.

Figure 6-1: Communications Pipes Between Client Software's Memory Buffer and Device Endpoints

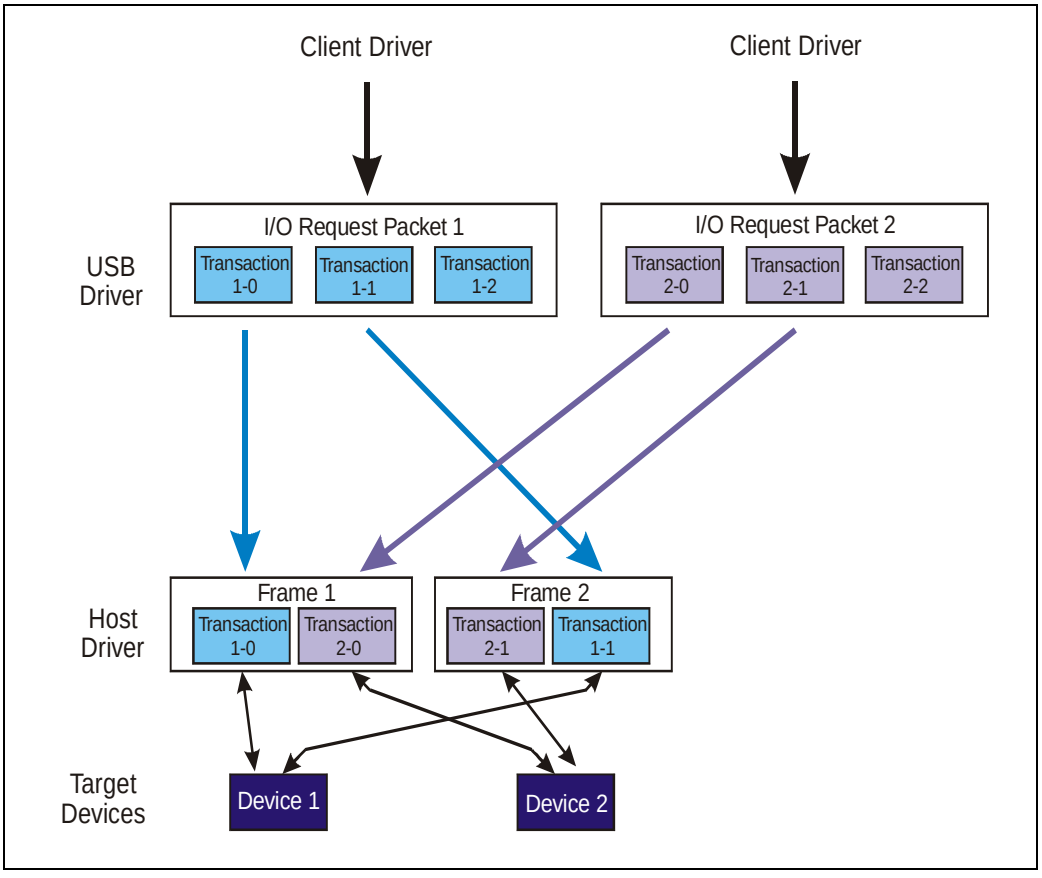


Communication Initiated by I/O Request Packets

Refer to Figure 6-2 on page 121. A USB device driver calls the USB driver to start a transfer. The client transfer request is referred to as an I/O Request Packet, or IRP. The IRP results in a transfer being performed over the USB. The USB driver allocates the amount of bus bandwidth that is given to this transfer during each frame. The transfer request may take numerous frames to complete.

The host controller driver schedules all IRPs that are presented to it by the USB driver. The host controller then performs the individual transfers that have been scheduled, by performing multiple transactions (reads from or writes to the target USB device) during each frame until the transfer completes.

Figure 6-2: Client Request Converted to USB Transactions



Frame-Based Transfers

Since the USB is shared by a wide variety of devices, a mix of USB transfer types will likely be performed during each 1ms frame. Since interrupt and isochronous transfers must occur at fixed intervals, they have a special priority during the execution of each frame. The specification states that a maximum of 90% of the USB bandwidth can be devoted to periodic (interrupt and isochronous) transfers, while control transfers have a 10% reservation during each frame. Bulk transfers are allocated the remainder of the available bandwidth.

7 Packets & Transactions

The Previous Chapter

The previous chapter introduced and described the four transfer types supported by USB: interrupt, bulk, control, and isochronous. The purpose of the transfers and their capabilities and limitations were also discussed.

This Chapter

Every transfer broadcast over the USB consists of a combination of packets. These packets are combined to define individual transactions that are performed as part of a larger transfer. Each transaction type is defined, along with the individual packets that compose it.

The Next Chapter

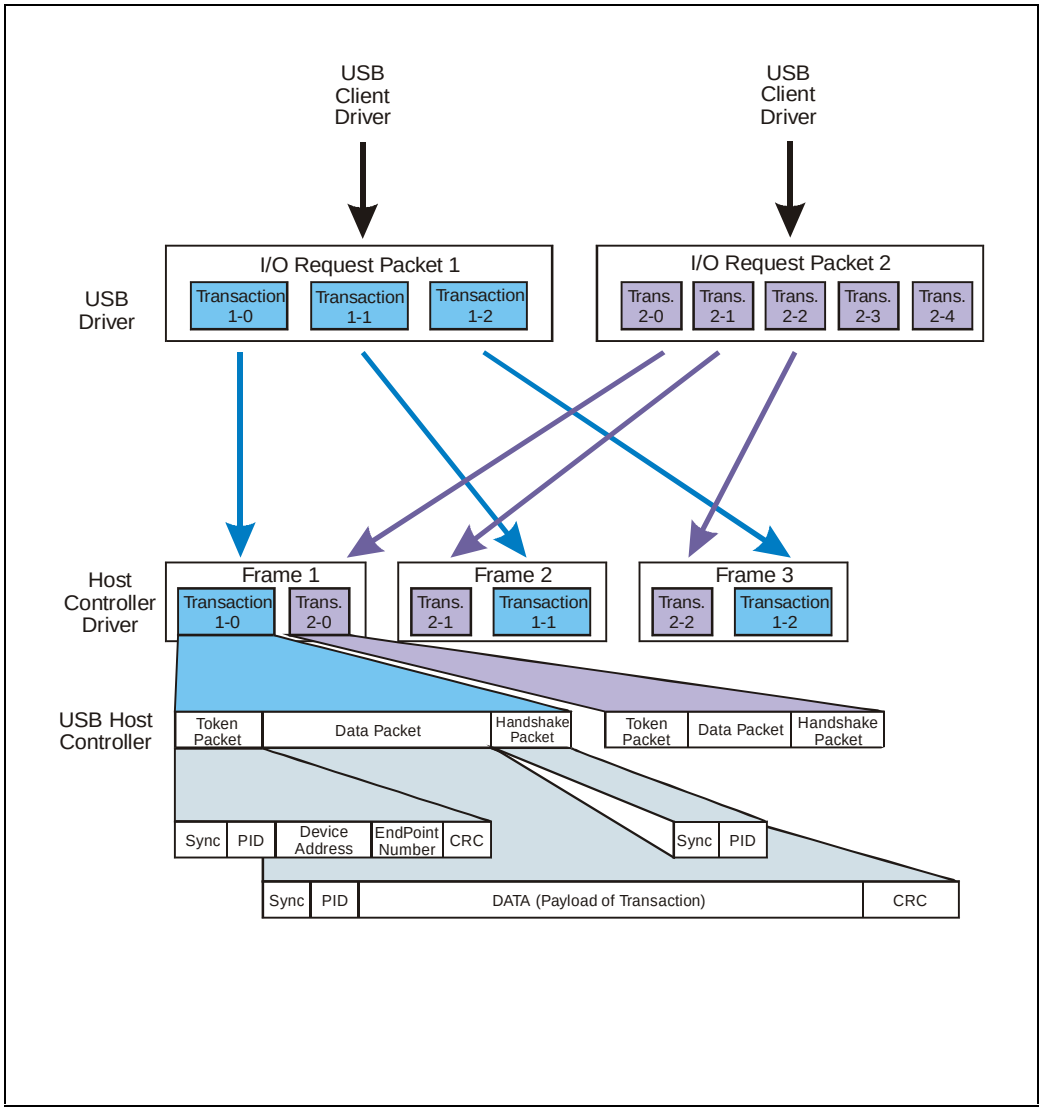
Interrupt, bulk, and control transfers require that the successful delivery of data be verified by USB. CRC and other error checking is performed to verify data delivery and if errors occur retries of the failed transmission are performed. This chapter discusses the various sources of errors and the error detection mechanisms used by USB to identify them, and the error recovery that is performed to overcome them.

Overview

The previous chapter discussed the various transfer types used to communicate with endpoints within a device. Transfers are performed across the USB using one or more transactions consisting of a series of packets. Figure 7-1 on page 142 illustrates the relationships between the various layers involved in performing a transfer — from the USB device driver's request to perform a transfer (IRP) to the resulting packets that are transmitted across the USB wire to or from the device. This chapter deals with individual transactions that are initiated by the

host to transfer data to or from the target USB device. Each transaction consists of one or more packets that are transmitted over the USB wire.

Figure 7-1: The Layers Involved in USB Transfers



Packets — The Basic Building Blocks of USB Transactions

Transactions typically consist of three packets as illustrated in Figure 7-2. However, a transaction may consist of one, two, or three packets depending on the type:

- Token Packet — Each transaction begins with a token packet that defines the target device, endpoint number, and the direction of the data transfer. The Start of Frame (SOF) token contains the current frame number that is broadcast to all full-speed devices. This is the only token packet that does not target a specific device.
- Data Packet — The data consists of a data packet that carries the payload associated with the transfer. A data packet can carry a maximum payload of 1023 bytes of data (isochronous transactions) during a single transaction; while the other transfer types have maximum data payloads of 64 bytes at full speed.
- Handshake Packet — all USB transfers (except isochronous) are implemented to guarantee data delivery, and include a handshake packet to verify a successful data transfer. If a packet error occurs, no handshake packet is returned to the sender and an error is flagged. The host is permitted to retry transactions that incur errors until the error count reaches three. (See Chapter 8, entitled "Error Recovery," on page 167 for a detailed explanation of the error handling.)

Figure 7-2: Many USB Transactions Consist of Three Phases

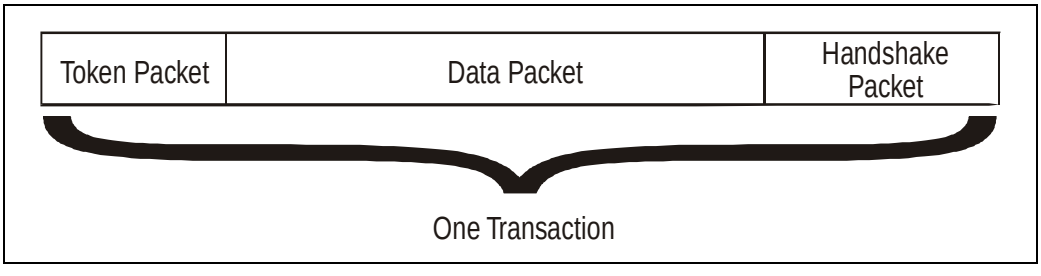
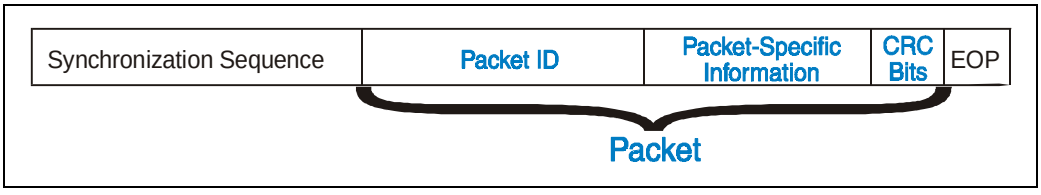


Figure 7-3 illustrates the basic format of a USB packet. Immediately preceding each packet is a synchronization sequence that permits USB devices to synchronize to the data rate of the incoming bits within the packet. The type of packet is defined by a bit pattern called a packet ID (PID). Following the PID is packet-specific information (e.g., an address or data) that varies depending on the

packet type. Finally, each packet ends with a sequence of Cyclic Redundancy Check (CRC) bits, used to verify correct delivery of the packet-specific information. The end of each packet is identified by an end of packet (EOP). Each type of packet is detailed in the following sections.

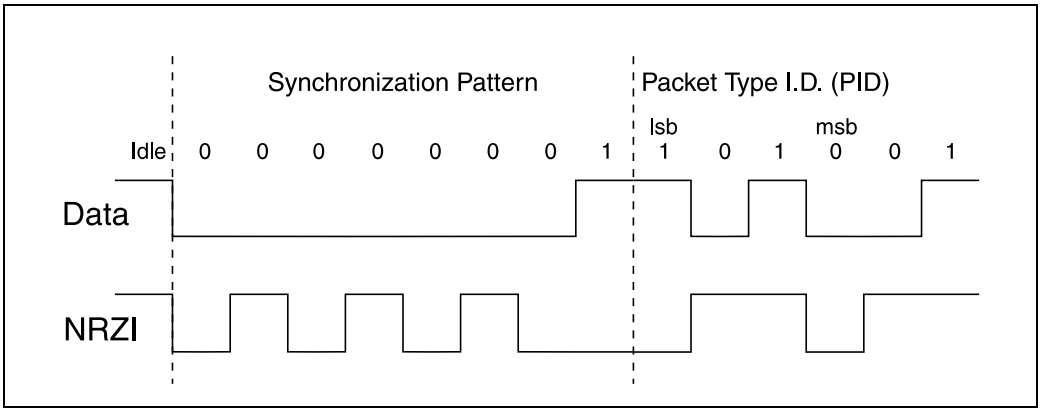
Figure 7-3: Packet Format



Synchronization Sequence

Figure 7-4 illustrates the synchronization sequence. The synchronization sequence consists of eight bits starting with seven consecutive logic 0s and ending with a logic 1. Since zeros are encoded with transitions of the differential data lines, the seven zeros each create a transition during each bit time, thus providing a clock that can be synchronized to. The synchronization sequence also alerts USB receivers that a packet is being sent, which will immediately follow the 8-bit synchronization sequence.

Figure 7-4: Synchronization Sequence



Packets can be broadcast at a full speed (12Mb/s) bit rate or a slow speed (1.5Mb/s) bit rate depending on the speed of the device being accessed. The USB receiver must detect the logic state of each bit value within the packet by sampling the data lines at the correct point during each bit time. The synchronization sequence is transmitted at the transfer speed being used, allowing the receiver to synchronize to either incoming data rate.

A variation to the packet sequence occurs when the host sends a low-speed packet to a low-speed target device. Recall that slow-speed USB devices cannot communicate at full speed. As a result, low-speed cables carry only low-speed transactions. USB hubs prevent full-speed packets from reaching low-speed devices by keeping low-speed ports disabled until a low-speed transaction is to be performed. This is accomplished with a special packet called a preamble packet that is used specifically to command hubs to enable their low-speed ports. Further, hubs must be given time to enable their low-speed ports after they receive the preamble packet. See “Preamble Packet” on page 154 for details.

When the low-speed packet transfer completes, the hub ports to which low-speed devices are attached once again are disabled from sending packets to low-speed devices. Note that the hub ports remain enabled in the upstream direction so that low-speed devices can respond to any form of host request.

Packet Identifier

Packet identifiers define the purpose and thus the format and content of a given packet. Packets are grouped into three major categories:

- Token Packets — Token packets are sent at the beginning of a USB transaction to define the target device and endpoint address (i.e., endpoint number + direction of transfer).
- Data Packets — These packets follow token packets during transactions that require data payloads be transferred to or from USB devices.
- Handshake Packets — Handshake packets are typically send by the device that receives the data, thus providing feedback to the sender to verify the success of the data transfer. In some cases, the USB device being requested to send data to the system may send a handshake packet to indicate that it currently has no data to send.
- Special Packets — Currently the only special packet defined for low- or full-speed devices is the preamble packet used to enable low-speed ports prior to sending a downstream low-speed packet.

8 Error Recovery

The Previous Chapter

Every transfer broadcast over the USB consists of a combination of packets. These packets are combined to define individual transactions that are performed as part of a larger transfer. Each transaction type was defined, along with the individual packets that compose it.

This Chapter

Interrupt, bulk, and control transfers require that the successful delivery of data be verified by USB. CRC and other error checking is performed to verify data delivery and if errors occur, retries of the failed transmission are performed. This chapter discusses the various sources of errors and the error detection mechanisms used by USB to identify them, and the error recovery that is performed to overcome them.

The Next Chapter

USB devices support power conservation by entering a suspended state. The next chapter discusses the ways that devices are placed into the suspended state under software control. It also discusses how software re-awakens devices, and how a device such as a modem can initiate a wakeup remotely.

Overview

A variety of error conditions are detectable by hardware during the transfer of data across the USB. The previous chapter introduced the handshake packet that has been designed into the USB transaction protocol to verify that a packet has been successfully received. This chapter details all the USB error checking mechanisms and describes the related error recovery procedures. Error checking mechanisms supported by the USB include:

- Packet error checks
- False EOP

- Bus time-out (no response)
- Data toggle error checks
- Babble — transactions occurring beyond end of frame
- LOA — loss of activity on bus

Packet Errors

The USB devices detect three types of packet errors:

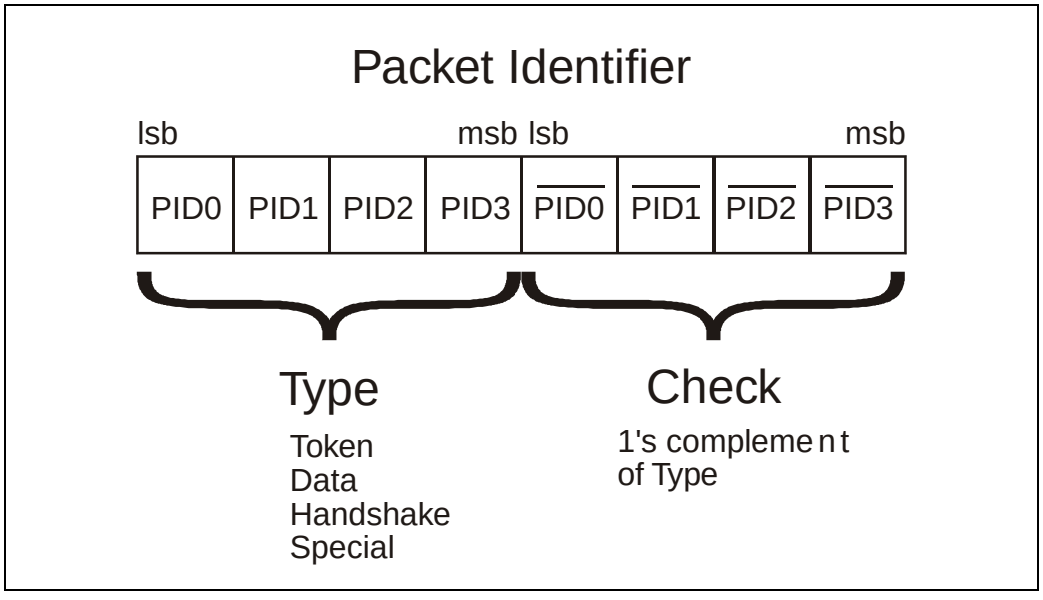
- Packet ID (PID) checks
- Cyclic Redundancy Checks (CRC)
- Bit stuff errors

If any of these error conditions exist, the receiver of the packet must ignore the packet and not respond to it in any manner. The receiver consequently never sends a packet back to the transmitter if the packet just received contains an error. Note that the type of packet error detected is not significant to the USB devices or host as it relates to error recovery. However, the host system may capture statistics regarding the nature of packet failures. The following sections discuss each form of packet-related error.

PID Checks

Each packet broadcast over the USB starts with a Packet ID (PID) consisting of four bits and is followed by a PID check field as illustrated in Figure 8-1. The check field is the PID inverted (1's complement). All potential USB target devices must perform the PID check and ignore the packet if an error is detected, since the definition of the packet is unknown.

Figure 8-1: PID Check



CRC Errors

Each packet contains CRC bits used to validate the information sent following the Packet ID field. The nature of this information varies depending on the packet type. Each packet contains either 5 or 16 CRC bits, which is determined by the packet's potential size and its type. Refer to Table 8-1.

Table 8-1: Packet Type and CRC

Packet Type	Fields	Max. Size of Fields	Number of CRC bits
Start of Frame	frame number	11 bits	5
IN	device and endpoint address	11 bits	5
OUT	device and endpoint address	11 bits	5

Table 8-1: Packet Type and CRC

Packet Type	Fields	Max. Size of Fields	Number of CRC bits
SETUP	device and endpoint address	11 bits	5
DATA0	data payload	1023 bytes	16
DATA1	data payload	1023 bytes	16
ACK	NA — packet ID only	NA	NA
NAK	NA — packet ID only	NA	NA
STALL	NA — packet ID only	NA	NA
PREAMBLE	NA — packet ID only	NA	NA

The 5-bit CRC field for the token packets is based on a generator polynomial:

$G(X) = X^5 + X^2 + 1$

The bit pattern representing this polynomial is 00101b. The 5-bit residual at the receiver will be 01100b is all bits are received correctly.

The 16-bit CRC field for the data packets is based on the generator polynomial:

$G(X) = X^{16} + X^{15} + X^2 + 1$

The bit pattern representing this polynomial is 1000000000000101b. If the data is received without errors, then the 16-bit residual will be 1000000000001101.

Note that the CRC bit stream will contain stuffed bits if the CRC contains six consecutive 1s.

Bit Stuff Errors

Bit stuffing ensures that the sender and receiver of NRZI data maintain synchronization by forcing a transition into the data stream after detecting six consecutive 1s. See the heading entitled “Bit Stuffing” on page 112 for details.

USB receivers expect to see a guaranteed transition (stuffed bit) in the data stream after six consecutive 1s. If a stuffed bit is not present, this indicates that

the packet has been corrupted or that the sender is not properly generating stuffed bits, or that the receiver is not decoding the NRZI data correctly.

Packet-Related Error Handling

Any one of the packet-related errors results in the same device behavior. In every instance, the receiver of a corrupted packet must ignore the packet and not respond. The error handling method, however, varies depending on the transaction and which packet of a given transaction is corrupted.

Token Packet Errors

IN Packet Errors. Consider an IN packet during which some form of packet error occurs. Since an error is detected in the token packet, it is ignored by the target device and no response is returned to the host. The host expects either a data or handshake packet to be returned in response to the IN token. However, since the target does not respond, the host will detect a bus time-out and the transaction fails. The host is then responsible for retrying the failed transaction.

OUT or SETUP Packet Errors. If an OUT or SETUP packet is being broadcast by the host when a packet error is detected, the host will follow the token packet with a data packet. The target may decode the data packet without error, but may not be able to verify that it is the intended recipient (due to an address CRC error), or may not be able to detect the meaning of the data packet (due to a PID check error). Since the target does not respond to the OUT token nor the following data packet, the host detects a bus time-out and knows and that the transaction has failed. The host then reschedules the transaction.

Data Packet Errors

During OUT or SETUP Transactions. Errors occurring within data packets cause the receiving device to discard the data and not respond to the sender. During an OUT transaction, the target, having detected a data packet error, will not respond with a handshake. The resulting time-out tells the host of the failed write. The host then retries the transaction later.

During IN Transactions. During IN transactions, valid data is returned by the target but the host receives a corrupted data packet. Since the host does not respond with an ACK handshake, the target is informed that the host did not receive the data. The host then must retry the transaction.

9 USB Power Conservation

The Previous Chapter

Interrupt, bulk, and control transfers require that the successful delivery of data be verified by USB. CRC and other error checking is performed to verify data delivery, and if errors occur, retries of the failed transmission are performed. The previous chapter discussed the various sources of errors and the error detection mechanisms used by USB to identify them.

This Chapter

USB devices support power conservation by entering a suspend state. This chapter discusses the ways that devices are placed into the suspend state under software control. It also discusses how software re-awakens devices, and how a device such as a modem can initiate a wakeup remotely.

The Next Chapter

The next chapter provides a brief introduction to high-speed device operation and sets the stage for a detailed discussion of the high-speed environment.

Power Conservation — Suspend

Suspend is designed to reduce overall power consumption under software control. USB supports two types of suspend:

- Global suspend — all USB devices are placed into the suspend state
- Selective suspend — selected devices are placed into the suspend state

When a device enters its suspend state it must consume no more than 500μa of current. Devices enter suspend after 3ms of no bus activity. Normally devices are kept from entering the suspend state because they receive a SOF token at the beginning of every 1ms frame, even if no other USB traffic is occurring.

Low-speed devices however, see only low-speed traffic, meaning that they do not see the SOF packet nor any full-speed transactions. Consequently, hubs must signal an idle to K state transition on all low-speed ports at intervals of less than 3.0ms to prevent low-speed devices from inadvertently entering the suspend state. To this end the specification minimally requires that the hub signal a low-speed EOP to all low-speed devices at the beginning of each frame.

Device Response to Suspend

When devices enter the suspend state, they must preserve their state and consume no more than an average of 500µa of current, or 2.5ma for high-power devices that are enable to generate remote wakeup. The specified current draw must not exceed the specified current limit when averaged over an interval of 1 second. A momentary current spike is permitted for configured devices during the averaging interval, but must not exceed the device's power allocation (specified in the max power field of the configuration descriptor). This limit includes the current draw associated with the pull-up and pull-down resistors on the D- and D+ lines.

Some devices may need to awaken the system in response to an external event. For example, a modem function could be designed to awaken the system when it receives a "ring indicate" from an external line. It would then be necessary to notify system software that the modem requires attention.

Hub Response to Suspend

When a hub detects greater than 3ms of inactivity on its upstream port, it must also enter the suspend state. Since the hub has detected no bus activity for 3.0ms, by definition no activity will have been broadcast to any of the hub's downstream ports for 3ms. All downstream ports, along with the hub itself, will detect the suspend state at approximately the same time.

Upon detecting suspend, hubs take the following actions:

- place their repeaters into the wait for start of packet (WFSOP) state
- float all output drivers
- maintain static values of all control and status bits
- preserve current state info for all downstream ports

All internal clocks are stopped and power consumption from the hub function is reduced to a minimum.

Since each device attached downstream also enters the suspend state, each device can each draw up to 500µa. This means that the hub itself may draw Configured bus-powered hubs may also draw up to 2.5ma with devices attached to it downstream ports. Unconfigured hubs like all other unconfigured devices must limit current draw to 500µa. Further, when in the suspend state, hubs must be able to supply the maximum current defined for their downstream ports to support remote wakeup. That is, a device may drive the bus during suspend to wake up the system as will be discussed later in this chapter.

Global Suspend

Global suspend places the entire USB network of devices into the suspend state from the top down. This provides for minimum power consumption from the USB. The host initiates global suspend typically in response to a prolonged period of bus inactivity.

Initiating Global Suspend

Global suspend is initiated when all downstream traffic from the root hub is terminated. This is done under software control by issuing a global suspend request to the root hub's control endpoint. All USB devices (hubs and functions) automatically enter the suspend state when they encounter 3.0ms of inactivity.

Resume from Global Suspend

Devices awaken from their suspend state when they detect resume signaling on the bus. Resume is signaled by a non-idle state (K state), initiated in the following ways:

- by the root hub, causing the resume signaling to occur on all downstream cable segments.
- by a device on any downstream cable segment attached to an enabled port, causing the hub to reflect the resume signaling back to that port, to all other enabled ports downstream, and upstream to its root port.
- by device attachment.
- by device detachment.
- by reset, causing all devices to be reconfigured.

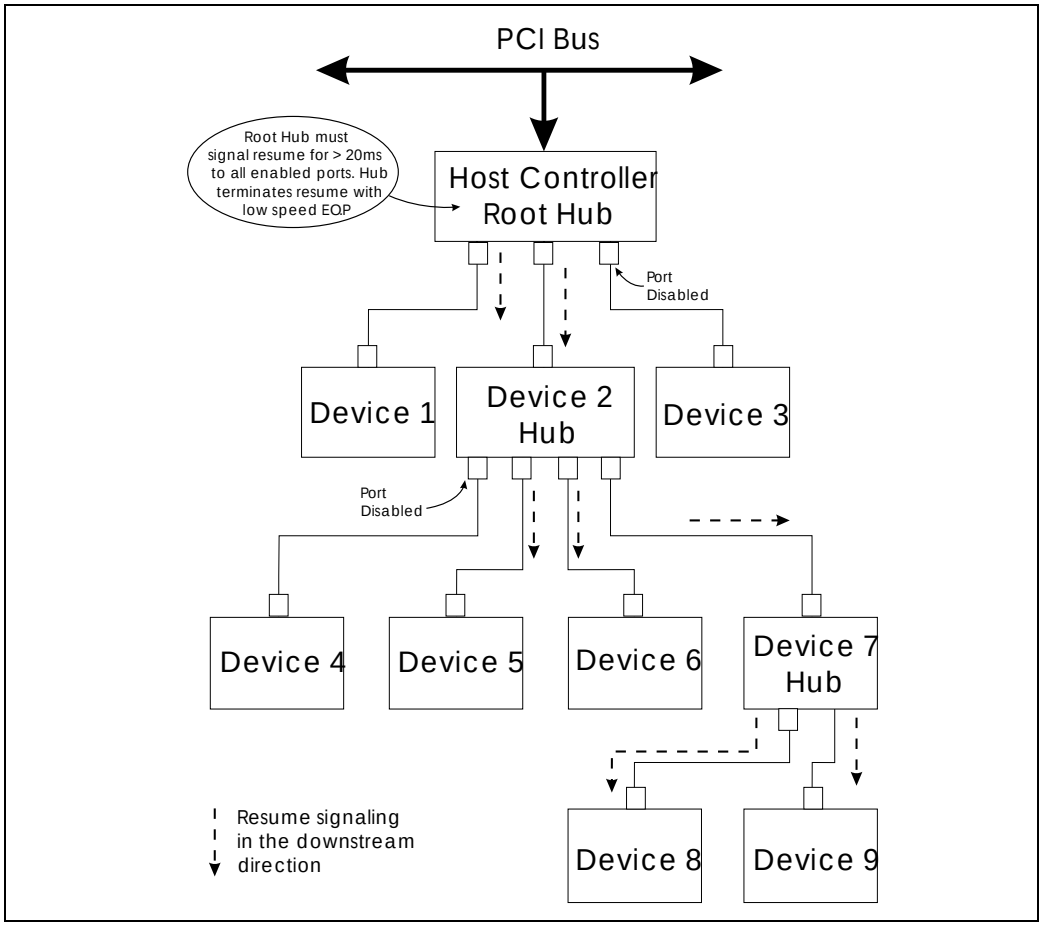
The following sections discuss the methods of handling suspend and resume.

Resume Initiated by Host

Host software may initiate USB wakeup by issuing a resume request to the root hub. The root hub responds by signaling resume to all downstream ports that are enabled as illustrated in Figure 9-1 on page 198. The resume signaling must be maintained for 20ms to give each attached device sufficient time to recover from suspend and be ready to receive transactions. The root hub ends resume signaling by driving an EOP for two low-speed bit times.

Downstream hubs that receive the 20ms of resume signaling must also propagate the resume signaling to all of their enabled downstream ports as illustrated in Figure 9-1.

Figure 9-1: Host Initiated Resume



Remote Wakeup from Device

A device in the suspend state may respond to an external event by signaling a wakeup to the system via the USB. Resume signaling is initiated by the device by driving a K state (non-idle) onto the bus. The actions taken by the hub are:

- signal resume to upstream port
- signal resume to all enabled downstream ports
- reflect resume signaling back to the originating device

Figure 9-2 illustrates the sequence of events and describes the timing associated with device initiated resume. The following steps and timing apply:

1. A K state is signaled by the device to the hub port (t_0).
2. The port detects the resume signaling.
3. Resume signaling is broadcast by hub 7 to its upstream port and to all enabled downstream ports within 900µs of receiving the resume (t_1).
4. Hub 2 signals resume to all of its enabled ports within 900µs of receiving the resume from hub 7.
5. Hub 7 ceases driving resume in the upstream direction within 1-15ms of t_0. but not earlier than 100µs and reverses connectivity. This is referred to as t_2.
6. Hub 2 ceases driving resume in the upstream direction within 10ms of receiving resume from hub 7, but not earlier than 50µs.
7. When the root hub detects the resume signaling, it initiates 20ms of resume signaling downstream to all ports, including the originating port.
8. The root hub terminates resume signaling by driving two low-speed EOPs.

Note that during the interval between t_0 and t_2, the host may have started driving resume signaling from the upstream direction, while the downstream hub is still driving resume signaling. Since it is acceptable to drive both ends of a bus segment to the same state, no problem is encountered.

Remote Wakeup via Hub Port Event

Remote wakeup may be signaled by a hub if it is enabled to produce a remote wakeup. In this case, when a hub detects a status change event (a bit is set in either the hub port status or status change register), it signals wakeup to the system in the same fashion as described for remote wakeup. See “Port Status Fields” on page 457 and “Port Change Fields” on page 459 for definition of the events that can cause a hub to generate remote wakeup.

10 Overview of HS Device Operation

The Previous Chapter

USB devices support power conservation by entering a suspend state. The previous chapter introduced the ways that devices are placed into the suspend state under software control. It also discussed how software re-awakens devices, and how a device such as a modem can initiate a wakeup remotely.

This Chapter

This chapter provides a brief introduction to high-speed device operation and sets the stage for a detailed discussion of the high-speed environment.

The Next Chapter

High-speed capable devices must also be able to communicate in the full-speed signaling environment. High-speed devices add many extensions to the full-speed environment to permit reliable signaling at a 480Mb/s rate. The next chapter introduces the principles associated with USB high-speed signaling and the methods used to switch between full- and high-speed operation.

Overview

High-speed device operation is different from low- and full-speed device operation in many ways beyond the obvious difference of transmission rate. Perhaps the most important difference is that low- and full-speed devices must operate at both high- and full-speed. This ensures that high-speed devices operate in systems that are USB 1.x compliant as well as USB 2.0 systems.

New High-Speed Device Features

High-speed devices can only take advantage of their HS capability when installed in 2.0 compliant systems. New high-speed devices may also function in 1.x systems but full-function operation is not required. Below is a list of new features and major changes associated with HS capability.

- New host controller designs are required.
- New system software including the USB bus driver and host controller driver is required to support high-speed devices.
- New client drivers for high-speed devices are required.
- Periodic transactions are scheduled on the basis of 125µs intervals called microframes, rather than the 1ms frames.
- New maximum data packet payloads are defined for isochronous, interrupt, and bulk transfers.
- New packet types are defined for the data, handshake, and special packet categories.
- High bandwidth transactions are defined for isochronous and interrupt endpoints.
- Error detection mechanisms are the same concepts as 1.x; however, timing-related parameters have been adjusted for the higher bit rate.
- High-speed transceivers have been changed to support the faster bit rate, while maintaining compatibility with full- and low-speed signaling.
- High-speed hubs must support all three device speeds.
- High-speed hubs use split transactions when communicating with full- and low-speed devices.
- New device descriptors have been added, and some modifications have been made to existing descriptors.
- New control transfers requests have been defined.

1.x USB Device Support

USB 2.0 provides two aspects of 1.x support:

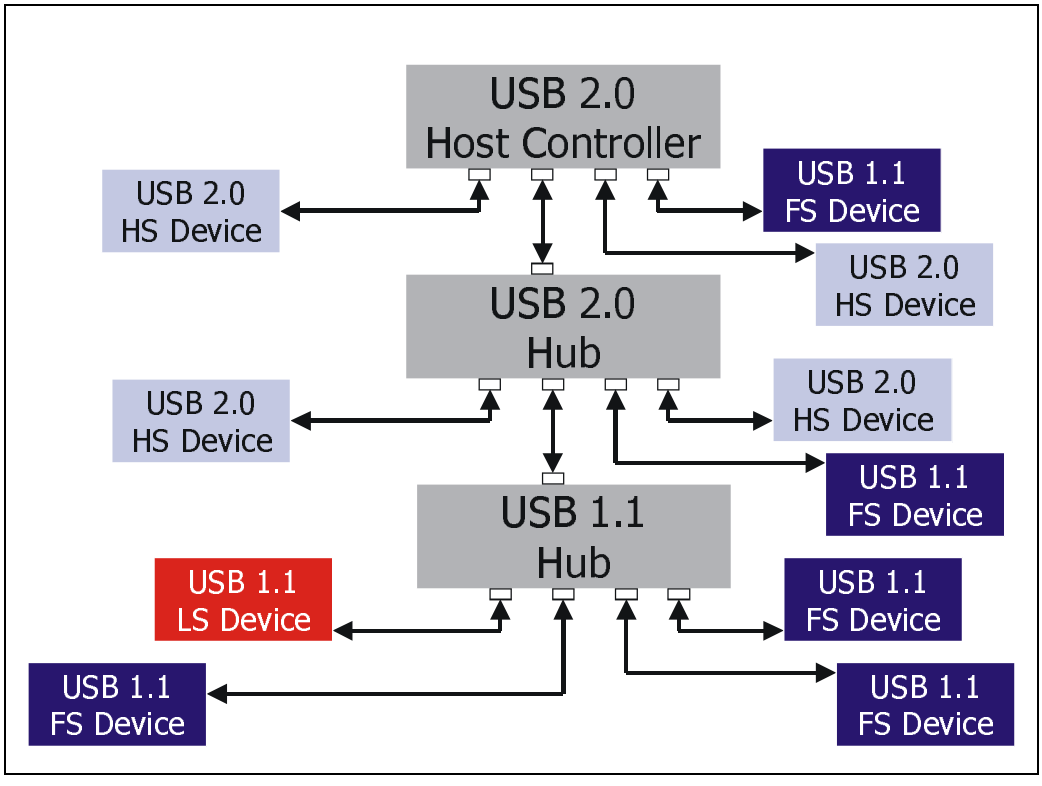
- High-speed devices can be attached to 1.x hub ports and accessed at full speed.
- Low- and full-speed devices can be attached to high-speed capable ports and accessed at their native speed.

Figure 10-1 on page 215 depicts a 2.0 system topology that includes both high-speed and full-speed hub ports.

High-speed devices operate normally when attached to high-speed ports. However, when a high-speed device is attached to a 1.x hub port it can only operate at full-speed. Note that the device may not have full functionality in this case.

When a 1.0 or 1.1 compliant device is attached to a high-speed capable device, the hub configures its signaling interface so that it can communicate at the required speed. In this way, high-speed capable hub ports provide complete back compatibility to older version USB devices.

Figure 10-1: USB 2.0 Example Topology



The 2.0 Host Controller

The 2.0 host controller (Enhanced Host Controller Interface) specification was released at the time of this writing. After the host controller specification is released, MindShare will document the features and functions of the controller and make it available for download. See www.mindshare.com for details.

11 The High-Speed Signaling Environment

The Previous Chapter

The previous chapter provided a brief introduction to high-speed device operation and was intended to set the stage for a detailed discussion of the high-speed environment starting with this chapter and others that follow.

This Chapter

High-speed capable devices must also be able to communicate in the full-speed signaling environment. High-speed devices add many extensions to the full-speed environment to permit reliable signaling at a 480Mb/s rate. This chapter introduces the principles associated with USB high-speed signaling and the methods used to switch between full- and high-speed operation.

The Next Chapter

The next chapter introduces the changes brought about by the USB 2.0 specification. The transfers defined in USB 1.0 have the same primary characteristics in the high-speed environment; however, some changes have been made such as new packet sizes. Also, new features have been added to the high-speed environment such as high-bandwidth transfers and the ping protocol. These and other changes are reviewed in this chapter.

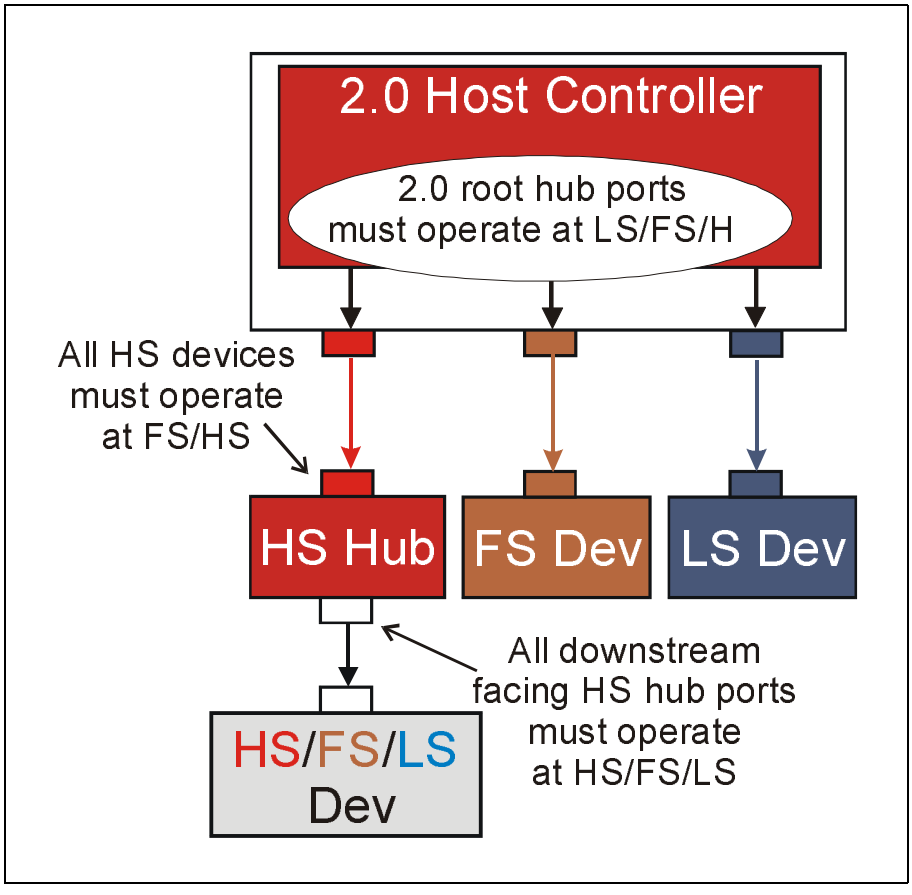
Overview

High-speed devices must be able to operate at both full-speed and high-speed. This ability ensures that any high-speed device attached to a 1.x port (i.e., root ports in 1.x system or 1.x hub ports) will work properly. This requirement is

important for compatibility with the large number of existing 1.x hubs and systems. Similarly, high-speed hubs must support low-, full-, and high-speed devices attached to their downstream ports to ensure compatibility with all USB devices. Figure 11-1 on page 218 summarizes the required port speed capabilities of each high-speed hub and device transceiver.

High-speed hubs must be able to detect whether they are connected to a full-speed or high-speed port and operate correctly at that speed. When the high-speed hub is connected to a high-speed port, it must be able to detect whether a high-speed / full-speed / low-speed device is attached and perform transactions to devices at their speed.

Figure 11-1: High-Speed Capable Ports Must Support a Variety of Speeds



The high-speed signaling interface must include the necessary features to support the appropriate upstream port speed and perform a variety of other signaling events. The high-speed interface must be capable of:

- Dynamic transition between different interface speeds
- High-speed device detection (handshake between high-speed hub and device)
- Detecting high-speed device disconnect (hub ports only)
- Signaling (hubs only) and detecting high-speed reset
- Supporting high-speed differential signaling (including impedance matching)
- Signaling and detecting high-speed start of packet
- Signaling and detecting high-speed EOP
- Supporting suspend and resume operation and signaling

Each of the signaling functions listed above will be discussed later in this chapter. Figure 11-2 illustrates the high-speed transceiver interface. The interface components in the lighter shade are used in the 1.x transceivers, and the darker shade represents components that have been added to support high-speed signaling. The differential receivers are illustrated with both shades because in this example they operate at all three speeds.

Detecting High-Speed Device Attachment

High-speed hub ports must be able to detect when a low-speed, full-speed, or high-speed device has been connected to the port. High-speed devices initially appear as full-speed devices when attached to a high-speed port. The high-speed port and the high-speed devices must then perform a handshake to identify the high-speed device. If the handshake fails, the high-speed device defaults to full-speed operation. The sequence of events is illustrated in Figure 11-3 and presumes that power has already been applied to the port. This sequence is identical to that which occurs when a full-speed device is attached except for the handshake (called the chirp sequence) that occurs during device RESET.

Figure 11-2: High-Speed signaling Interface

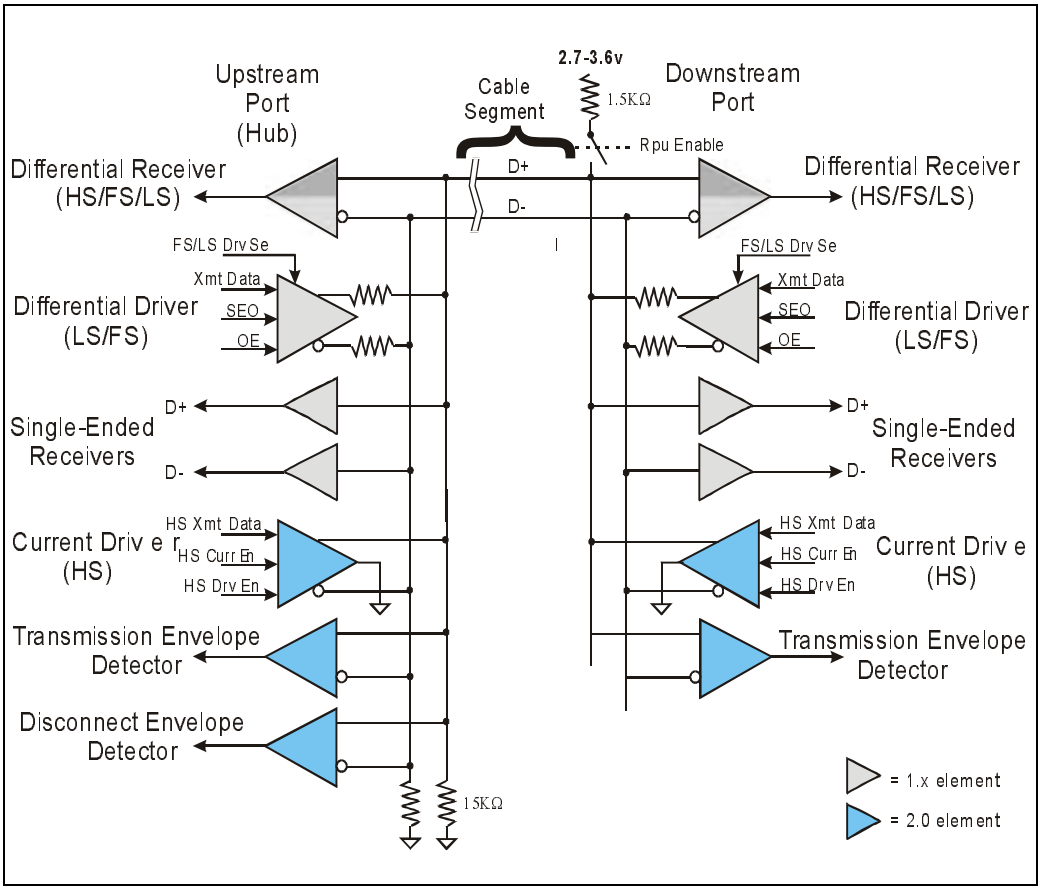
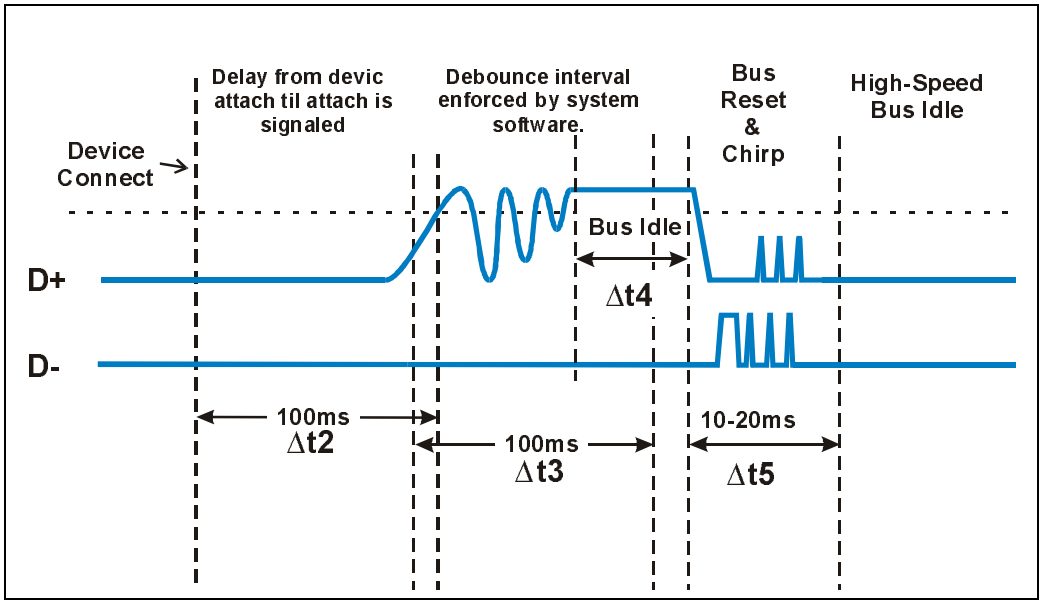


Figure 11-3: Sequence of Events from Device Connect to High-Speed Operation



Initial Device Detection

Full- and high-speed devices both have a 1.5KΩ on the D+ line. When the device is connected, power is applied to the cable, the device, and the pull-up on D+, causing the voltage to rise above V_{IH} at the hub receiver. Software then polls the hub and detects full-speed device attachment.

Device Reset and the Chirp Sequence

After software detects that a full-speed device is connected, it issues RESET to the hub via the ResetPort command, which causes the hub to drive a SE0 (both D+ and D- low) for >10ms. If a high-speed capable device is attached, the chirp sequence begins. Figure 11-4 on page 223 depicts the chirp sequence. Each step in the chirp sequence is detailed in the following enumerated list:

1. Hub drives RESET, which is T_0 for the chirp sequence.
2. The high-speed device detects RESET and signals a Chirp K. signaling

12 HS Transfers, Transactions, & Scheduling

The Previous Chapter

High-speed capable devices must also be able to communicate in the full-speed signaling environment. High-speed devices add many extensions to the full-speed environment to permit reliable signaling at a 480Mb/s rate. The previous chapter introduced the principles associated with USB high-speed signaling and the methods used to switch between full- and high-speed operation.

This Chapter

This chapter introduces the changes brought about by the USB 2.0 specification. The transfers defined in USB 1.0 are also used in 2.0 implementations and have the same primary characteristics in the high-speed environment; however, some changes have been made such as new packet sizes. Furthermore, new features have been added to the high-speed environment such as high-bandwidth transfers and the ping protocol. These and other changes are reviewed in this chapter.

The Next Chapter

Error detection and handling during high-speed transactions is very similar in concept to the low- and full-speed error detection methods. However, due to the faster clock rates, several of the timing parameters must be changed to support error detection implementations such as time-out values and babble detect. These issues are discussed in the next chapter.

Overview

High-speed transfer types are the same as those used by full- and low-speed devices and include:

- isochronous
- interrupt
- bulk
- control

Although these transfers are the same as those defined by USB 1.x, in some cases the high-speed transactions have new characteristics that are designed to improve performance. Also, note that high-speed packets differ from their full-speed cousins in the following ways:

1. The synchronization sequence at the beginning of each packet is 4 bytes in length rather than 1 byte.
2. EOP is 1 byte rather than 2 bits.

Consequently, packet overhead associated with high-speed transaction are higher.

This chapter describes the characteristics of the high-speed transfers and transactions, and describes the mechanisms used for scheduling and executing them. Since isochronous and interrupt transfers have important characteristics in common, they will be discussed in the section entitled “Periodic Transfers,” while bulk and control transfers are discussed in the section entitled “Non-Periodic Transfers.”

High-Speed Transaction Scheduling

All high-speed devices share bus bandwidth when the devices are attached to ports of the same controller. High-speed transactions also use the same token, data, and handshake protocol that is used with full- and low-speed transactions. Software calculates the time required to perform transactions and schedules transactions to be completed during each microframe.

Microframes

High-speed bandwidth is allocated by software on the basis of 125µs intervals called microframes. Periodic transfers are restricted to no more than 80% of the 125µs bandwidth, and control transfers are guaranteed 20% of the bandwidth. Bulk transfers have no reserved bandwidth and will be performed after all scheduled transfers and after any control transfers that software has scheduled for the current microframe.

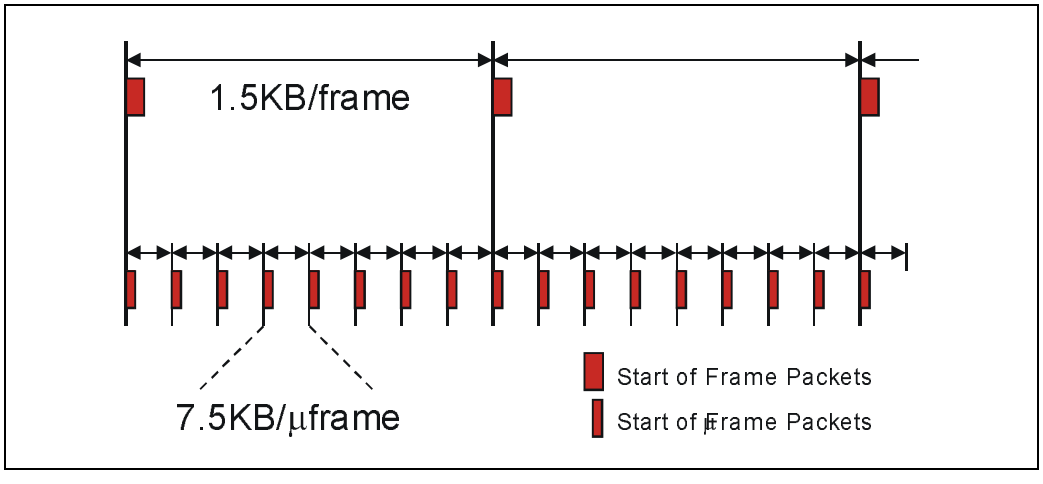
Theoretical HS Bandwidth

The transfer rate of 480Mbits/s yields 40 times more bandwidth than full-speed transfers at 12Mbits/s. The characteristics of high-speed transfers include the following statistics:

- ~2.08ns bit times
- 60,000 bit times/µframe (7,500 bytes)
- 480,000 bit times/1ms macroframe (60,000 bytes), or 60MB/s

Figure 12-1 graphically shows the bandwidth differences between the theoretical bandwidth of the full-speed and HS signaling environments. Five times more data can be transferred during a much shorter 125µs period at HS than a 1ms period at full speed.

Figure 12-1: Bandwidth Difference Between Full-Speed Frame and High-Speed Microframe



The theoretical HS bandwidth of 60MB/s is of course considerably more than actual data bandwidth. However, greater bus throughput and efficiency is possible with high-speed transfers due to larger packet sizes and new transactions defined for high-speed. Note that the same primary packets are used for high-speed, full-speed, and low-speed transactions; thus, the overhead associated with the packets themselves remains the same. Bandwidth issues associated with each transfer type are discussed in the corresponding sections later in this chapter.

Periodic Transfers

Both isochronous and interrupt transfers must be scheduled on a periodic basis. Isochronous transactions are scheduled every µframe, and interrupt transactions occur at specified polling intervals. These transfer types ensure throughput at the periodic rate, making them useful for USB applications that need specific bandwidth to perform adequately.

High-Speed Isochronous Transfers

Several changes have been made to the isochronous transfer type including:

- Maximum packet size changed from 1023 to 1024 bytes
- High bandwidth capability
- New data packet types used during high-bandwidth transfers

The following sections discuss these new features and summarize the overall characteristics of high-speed transfers.

Maximum Packet Size

Maximum packet size for HS isochronous transactions is 1024 bytes. The maximum packet size for FS isochronous transactions is 1023 bytes. Note that the 2.0 specification does not support software requests for isochronous data packets of zero bytes, but rather reserves a packet size zero encoding for a packet size of 1024 bytes.

Isochronous Bandwidth/Performance

A single isochronous transaction is permitted per endpoint during a single µframe in the standard implementation. The maximum data rate of an isochronous transfer is a function of the:

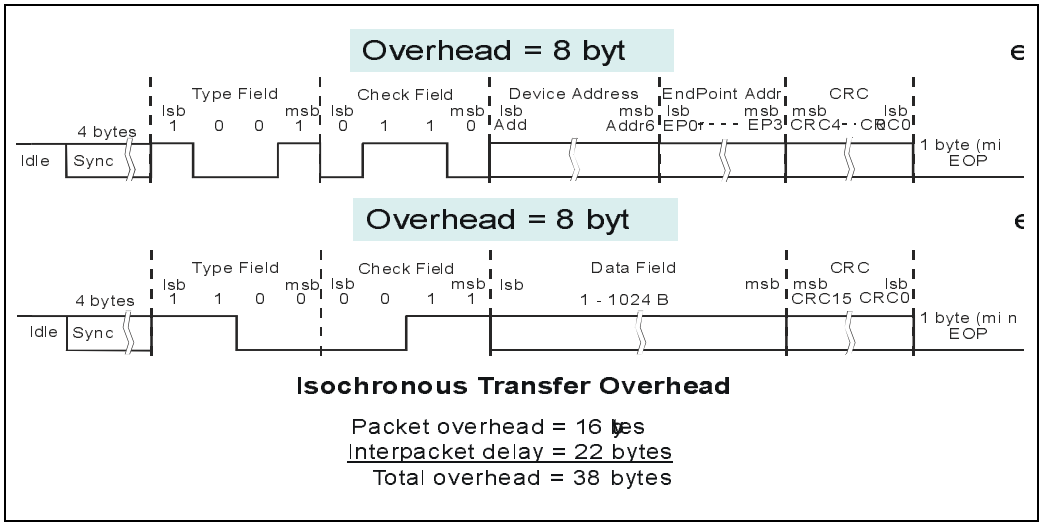
- packet overhead associated with an isochronous transaction
- propagation time of each packet (between host and target device)
- device response time
- size of the data payload

Figure 12-2 on page 245 illustrates the overhead from packets and lists the overhead associated with propagation delay. The overhead from the token packet and the data packet is 8 bytes each:

Token	Data
4 byte sync	4 byte sync
1 byte PID	1 byte PID
2 byte address + EP number + CRC	2 byte CRC
1 byte EOP	1 byte EOP

Interpacket delay consists of the propagation time for the packets to travel between the host and device, plus the host controller recovery time (during OUT transactions) or the response time of the device (during IN transactions). Bit stuffing time is not included in this overhead calculation.

Figure 12-2: Isochronous Packet Overhead



The bandwidth that is available for isochronous transactions during a given µframe is summarized in Table 12-1 on page 246. The table provides the following information:

13 HS Error Detection and Handling

The Previous Chapter

The previous chapter introduces the changes brought about by the USB 2.0 specification. The transfers defined in USB 1.0 are also used in 2.0 implementations and have the same primary characteristics in the high-speed environment. However, some changes have been made such as new packet sizes. Furthermore, new features have been added to the high-speed environment such as high-bandwidth transfers and the ping protocol. These and other changes are reviewed in this chapter.

This Chapter

Error detection and handling during high-speed transactions are very similar in concept to the low- and full-speed error detection methods. However, due to the faster clock rates, several of the timing parameters must be changed to support error detection implementations such as time-out values and babble detect.

The Next Chapter

This chapter discusses the changes required for high-speed devices to use the full-speed suspend and resume protocol and signaling conventions.

Overview

Error checking and recovery in the high-speed environment can be summarized as follows:

- same packet error detection as performed by low- and full-speed devices.
- transaction time-outs consist of the same round-trip delays, but the some of

the delays have been changed and delay timing is specified by high-speed bit times rather than low- or full-speed bit times.
- EOF1 and EOF2 babbling device detection uses the same principles as employed by 1.x hubs, but the EOF points are also referenced to high-speed bit times.
- packet error checking and reporting is modified as necessary to support split transactions.

High-Speed Bus Time-out

Like in the 1.x environment, high-speed devices report packet errors by remaining silent when a response is expected. No response is a positive indication to the device that is awaiting a return packet that the transfer has failed. The amount of time that a device waits for a response before detecting an error is specified by the worst-case round-trip delay between the host and target device. Figure 13-1 on page 267 depicts the topology that represents the worst delay. The round-trip delay is specified in high-speed bit times and is based on following:

- The maximum number of cable hops between host and device = 6
- Maximum number of hubs = 5
- Maximum response time of function = 192 high-speed bit times
- Maximum cable propagation delay = 26ns
- Maximum hub delay = 36 bit times + 4ns

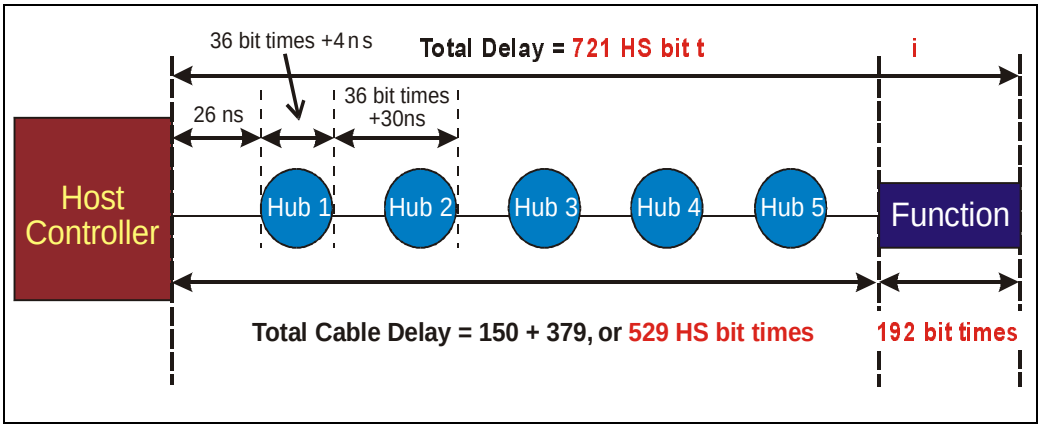
The total round-trip delay is calculated as:

- 12 cable crossings * 26ns = 312ns (~150 HS bit times)
- 10 hub crossings * 36 bit times + 4ns = 360 bit times + 40ns (~379 HS bit times)
- Maximum response time of function = 192 HS bit times

Total delay = 721 HS bit times

The specification requires that the time-out not occur any earlier than 736 HS bit times and any later than 816 HS bit times.

Figure 13-1: Worst-Case Round Trip Delay Between Host and Function



When the host or device transmits a packet requiring a response, a counter is started when the packet is completely transmitted and the counter stops when the start of the return packet is detected. The counter starts when the data lines return to the squelch level indicating packet end, and stops when the data lines leave the squelch level, indicating packet start. If the counter exceeds the specified time-out period, then an error is flagged. Behavior beyond this point is identical to the low- and full-speed implementations.

False EOP

High-speed EOPs are detected by a bit-stuffing error. The specification states that a receiver is required to interpret any bit-stuffing error as an EOP. If a non EOP bit-stuffing error occurs during the transmission of a packet, a false EOP is detected along with a CRC error.

Reaction to a false EOP by the host or device is the same for low-, full-, and high-speed operation, with one exception: host behavior after detecting a false EOP. Below is a summary of host behavior in the low- and full-speed environment and in the high-speed environment:

- In the low- or full-speed environment when a false EOP is detected, the host controller waits for the packet to end and then waits for an additional 16 bit times before sending the next packet downstream. The 16 bit time delay ensures that the transmitting device will time out and recognize that an

error has occurred in the packet just transmitted.

- In the high-speed environment if the host receives a corrupted high-speed data packet (including false EOP), it ignores any data until the data lines return to the squelch level. Upon detecting squelch, the next token packet can be sent immediately with normal inter-packet delays. Note that during high-speed operation, the host is not required to wait for the device to detect time-out before sending the next packet. In this case, the device has transferred data to the host and is unaware that a packet error has occurred, so the device is expecting a handshake packet that acknowledges successful transfer of data. However, the next packet sent by the host is a token packet that does not target the endpoint that expects the handshake. Since no handshake is received, the device presumes that the previous packet did not transfer correctly.

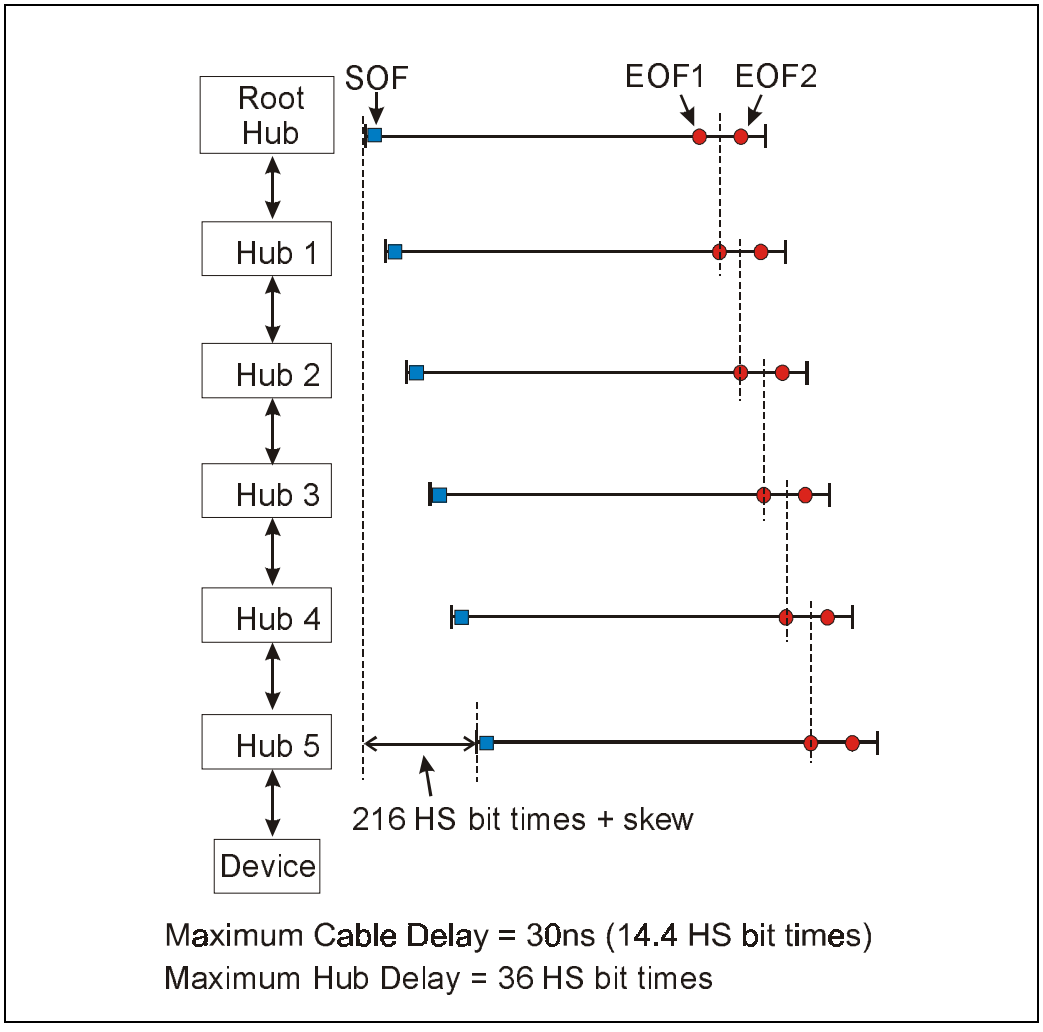
HS Babbling Device Detection

High-speed devices that are babbling or that have lost activity get detected via the same principle as full-speed implementations. However, the faster transmission rate and different signaling states lead to a number of changes.

Figure 13-2 on page 269 illustrates the alignment of frame timers within the hubs. The hub timers are skewed due to the propagation delay associated with the cable and hub repeaters. This ensures that the EOF1 sample points of each downstream hub fall between the EOF1 and EOF2 sample points of the hub immediately upstream. The specification states that when any hub reaches its EOF1 sample point, it “tears down upstream connectivity.” If an upstream packet is being transferred at EOF1, then the upstream port would detect abrupt packet termination, thus ensuring that no upstream packet will be observed when the EOF2 point is reached. Note that the hub port to which the babbling device is attached will detect upstream activity from the babbling device at its EOF2 point. The hub must disable the port to prevent it from interfering with the next microframe.

Note that the proper timing relationships between the EOF points from hub to hub are critical to the detection of a babbling device. The hub designer must ensure that delays associated with SOF decoding are offset by advancing the EOF points so that the timing deltas between EOF1 points are due solely to cable and hub delays.

Figure 13-2: Babbling Device Detection Model



14 HS Suspend and Resume

The Previous Chapter

Error detection and handling during high-speed transactions is very similar in concept to the low- and full-speed error detection methods. The previous chapter discussed the error detection changes that were made in order handle to the faster clock rates. Several of the timing parameters had to be changed to support error detection implementations such as time-out values and babble detect.

This Chapter

This chapter discusses the changes required for high-speed devices to use the full-speed suspend and resume protocol and signaling conventions.

The Next Chapter

This chapter introduces the primary characteristics of a high-speed hub. It must be able to operate when attached to both full-speed and high-speed ports, and must support all device speeds on its ports.

Overview

High-speed devices use the full-speed mechanisms for suspend and resume. In order to use the full-speed resume signaling, high-speed devices must transition to full-speed operation upon entering suspend and automatically return to high-speed signaling when returning to full-power operation. This section describes the transition from high-speed to full-speed and back.

Entering Device Suspend

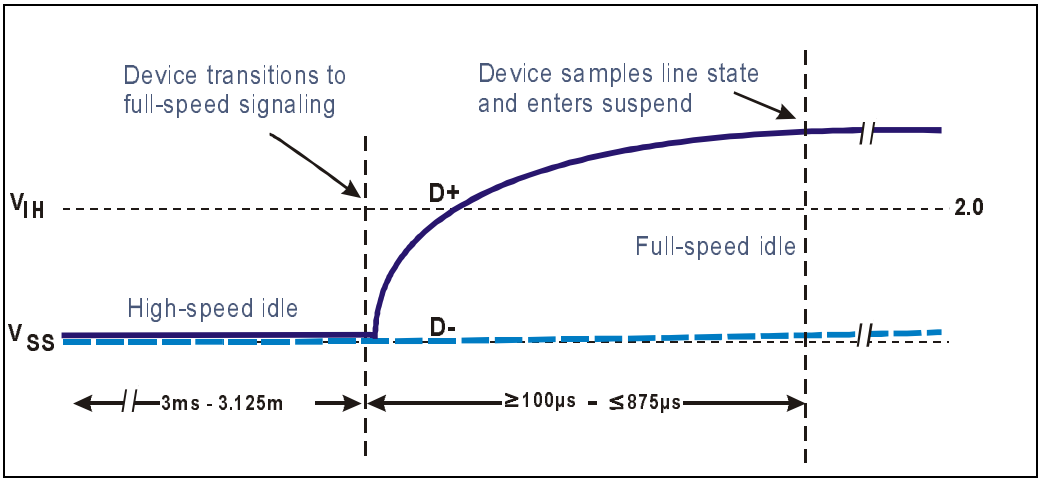
High-speed devices enter the suspend mode after detecting greater than 3ms of bus idle time, just as 1.x devices do. However, because both bus idle and device RESET are signaled via single-ended zero (SE0), additional logic must be employed to differentiate between these two possibilities. Another element common to both suspend and high-speed RESET is that they transition back to full-speed signaling.

- Suspend Detection — After 3ms of bus idle time the high-speed device knows that it is either being suspended or reset. If a suspend is being signaled, then the hub port will signal a full-speed idle (D+ pulled up and D- pulled down).
- RESET Detection — Recall that detecting RESET in the full-speed environment is done after observing a single-ended zero for greater than 2.5µs. This does not conflict with suspend signaling because bus idle and RESET are signaled differently. High-speed devices cannot detect RESET after 2.5µs of SE0 because this could also be a device suspend. If a RESET is being signaled by the hub port, then SE0 will continue for >10ms and <20ms.

Figure 14-1 on page 273 illustrates the process of determining whether a device will enter suspend or reset. The steps involved in transitioning to suspend are enumerated below:

1. The high-speed device transitions to full-speed operation after observing more than 3ms but before 3.125ms of HS bus idle. The device re-attaches the pull-up to D+ and turns off its full-speed drivers (thereby removing the high-speed terminations).
2. No sooner than 100ms and no later than 875ms after the transition to full-speed operation, the device samples D+ and D- to determine whether suspend or RESET is being signaled. If the full-speed bus is in the idle state (J state) then the device enters suspend and reduces current draw. If the full-speed bus is in the SE0 state, then the device detects RESET and begins its chirp sequence.

Figure 14-1: Device Detection and Entry into Suspend State



Device Resume

A device that enters suspend must remember that it was operating in the high-speed environment so that it can return to normal operation following resume.

When a device detects resume (>20ms of full-speed K followed by LS EOP), it transitions back to high-speed operation and can then increase current draw from the bus up to the maximum specified in its configuration descriptor.

Part Four

USB 2.0 Hub Operation with LS / FS / HS Devices

Part Four discusses the operation of high-speed hubs. Since USB 2.0 is backwardly compatible with low-, full-, and high-speed device operation, the issues associated with maintaining USB device compatibility is also discussed in Part Four. The chapters and topics included in Part Four are listed below:

- Chapter 15: USB 2.0 Hub Overview
- Chapter 16: 2.0 Hub Behavior During HS Transactions
- Chapter 17: 2.0 Hub Behavior During LS / FS Transactions

15 HS Hub Overview

The Previous Chapter

The previous chapter discussed the changes required for high-speed devices to use the full-speed suspend and resume protocol and signaling conventions.

This Chapter

This chapter introduces the primary characteristics of a high-speed hub. It must be able to operate when attached to both full-speed and high-speed ports, and must support all device speeds on its ports.

The Next Chapter

The next chapter discusses the 2.0 hub's behavior when it receives high-speed packets on its upstream and downstream ports. The chapter also details the operation of the high-speed repeater and discusses the delays associated with forwarding high-speed packets across the hub.

Overview

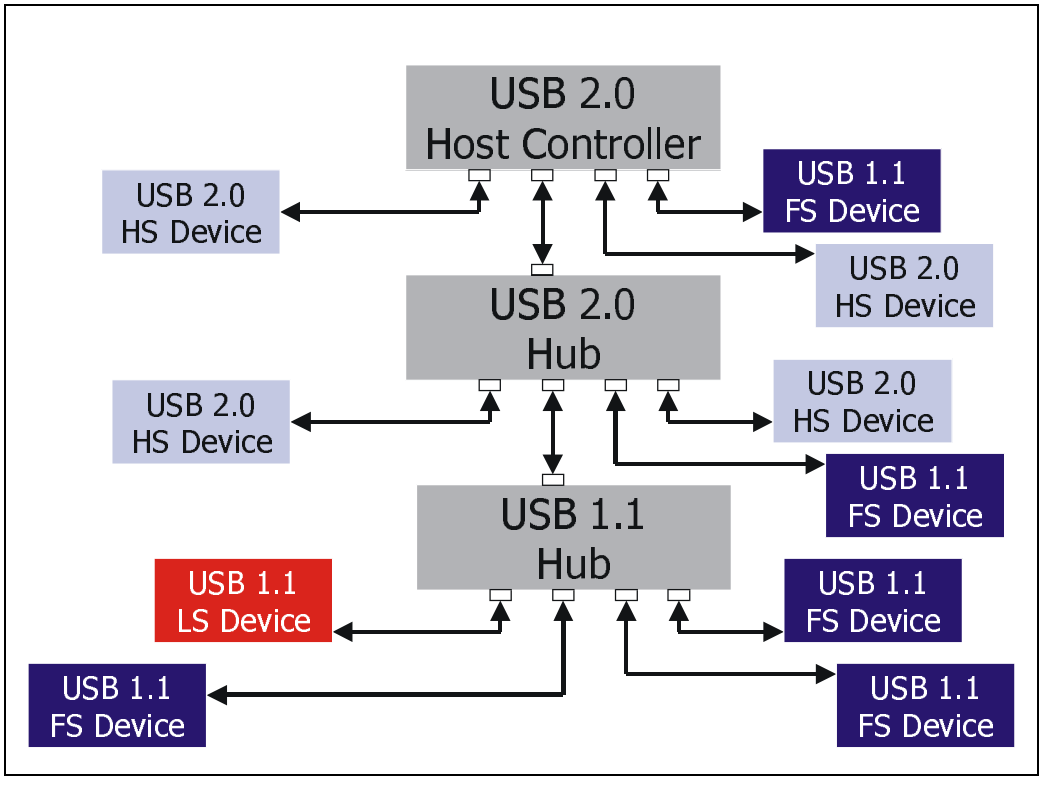
The operation of a 2.0 is dependent upon the speed of the upstream port to which it is attached. If it is attached to a high-speed port, then it enables its high-speed repeater and transaction translator logic. The repeater forwards all high-speed packets to the high-speed devices attached to the hub's downstream facing ports. The transaction translator handles transactions targeting all low- and full-speed devices attached to the hub's downstream facing ports.

If the hub attaches to a full-speed port, it must operate as a compliant full-speed hub. The high-speed repeater is disabled and low- and full-speed repeaters are enabled. Also, the transaction translator is disabled in this mode.

USB 2.0 Hub Attached to High-Speed Port

When in the high-speed mode, hubs must support low-, full-, and high-speed devices. Figure 15-1 on page 278 depicts a USB 2.0 topology with devices of all speeds attached to the hub's high-speed capable ports. Also, the third tier of this topology is shown to be a low- and full-speed tier, since all available ports at the third tier are provided by a 1.1 hub that is connected to the high-speed hub.

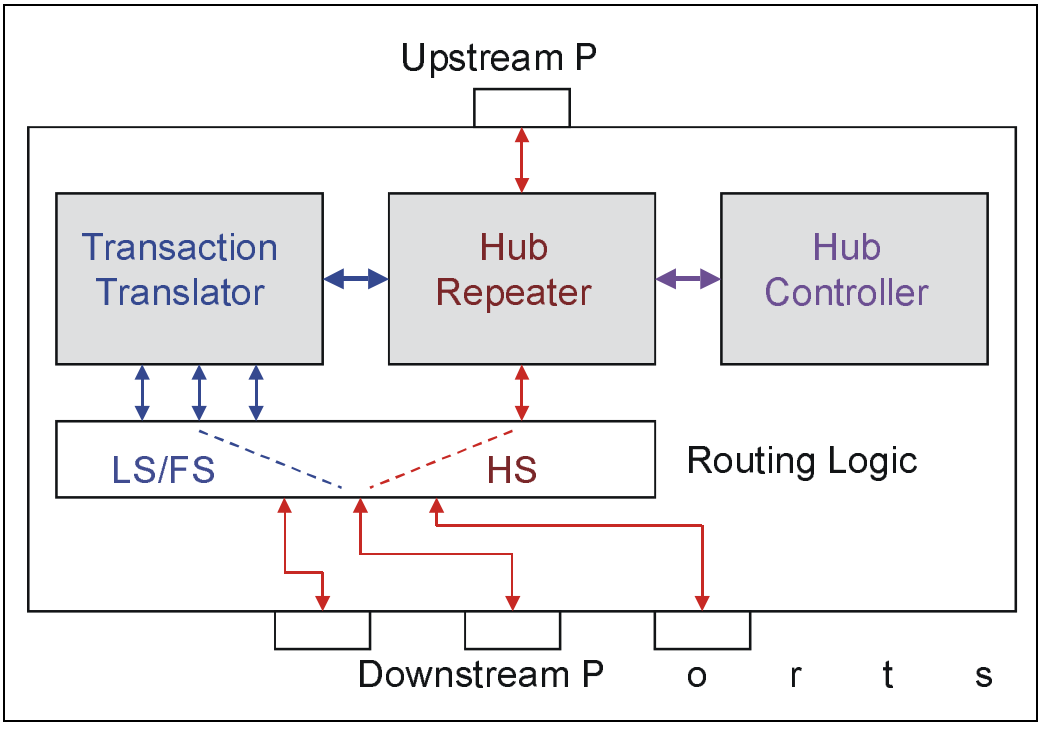
Figure 15-1: Example USB 2.0 Topology with Old and New Hubs



High-speed hubs detect the attachment and speed of each device and convert the port interface to support the particular device that has been attached. Figure 15-2 on page 279 illustrates four major blocks that are used to support all three device speeds.

- The hub **controller** collects status information associated with each port and makes this information available to software that can check the speed of each device attached to a hub port.
- The **repeater** forwards all high-speed packets to high-speed capable devices.
- The **transaction translator** accepts high-speed split transactions and performs the requested operation to the low- or full-speed device being targeted.
- The **routing** logic connects the repeater to all ports to which high-speed devices are attached, and connects the transaction translator to all port that have low- and full-speed devices attached.

Figure 15-2: Packet Routing Options for High-Speed Hub



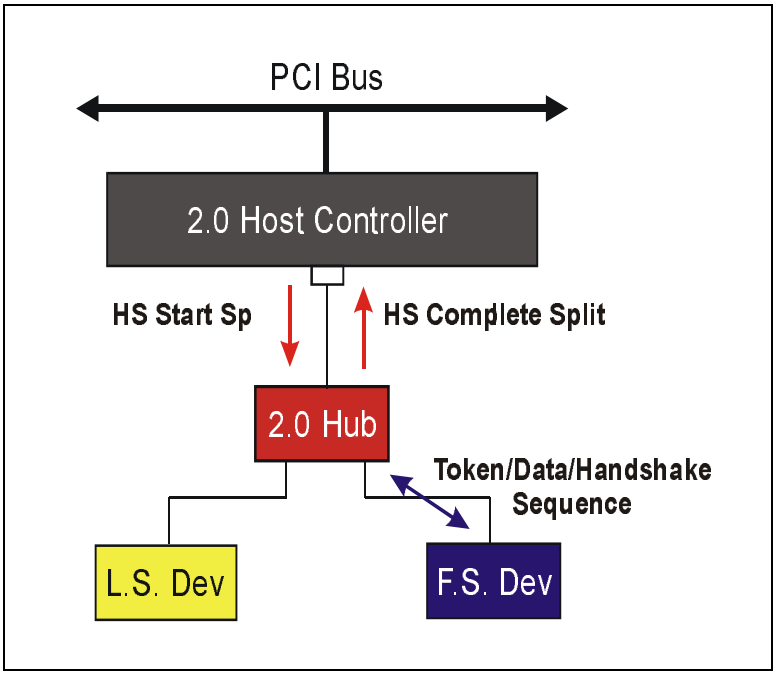
High-Speed Transactions

The high-speed hub repeater passes all high-speed packets to its high-speed ports, including high-speed split transactions. These packets are passed without performing any form of decode. Each packet is re-clocked to the downstream ports. Chapter 16, entitled "2.0 Hubs During HS Transactions," on page 283 details the operation of the hub when repeating high-speed packets.

Low- and Full-Speed Transactions

The host issues high-speed split transactions to access low- and full-speed devices that are attached to high-speed hub ports located downstream from the root hub (Figure 15-3). The host initiates a transaction via a start-split transaction that delivers the token packet to the hub and identifies the hub and port being targeted by the transaction. The hub then performs a normal low- or full-speed transaction to the device endpoint. This consists of the normal token, data, and handshake sequence. Depending on whether the transaction is an IN or OUT, the host schedules a complete-split transaction to fetch the data or obtain completion status (handshake). Chapter 17, entitled "2.0 Hubs During LS/FS Transactions," on page 289 details the operation of hubs when performing split transactions.

Figure 15-3: Split Transaction are Required to Access Low- or Full-Speed Devices That Attach to High-Speed Hubs



USB 2.0 Hub Attached to Full-Speed Port

High-speed capable hubs must operate properly when attached to a full-speed port. When in full-speed mode, they must support the attachment and complete operation of full- and low-speed devices just as a 1.x hub would.

In USB 2.0 the manner in which devices are attached to available ports can affect the performance of individual devices on USB. For example, if the user had reversed the positions of the 2.0 and 1.1 hubs shown in Figure 15-4 on page 282, then the 2.0 hub would be attached to a full-speed port and would only support low- and full-speed devices on its downstream ports. In this example, the only devices that would operate at high-speed are the two high-speed devices connected to the root hub ports. Software is required to detect and report these kinds of problems.

16 2.0 Hubs During HS Transactions

The Previous Chapter

The previous chapter introduced the primary characteristics of a high-speed hub, including the ability to operate when attached to both full-speed and high-speed ports and the ability to support all device speeds on its ports.

This Chapter

This chapter discusses the 2.0 hub's behavior when it receives high-speed packets on its upstream and downstream ports. This chapter details the operation of the high-speed repeater and discusses the delays associated with forwarding high-speed packets across the hub.

The Next Chapter

This chapter introduces the concept of split transactions that allow high-speed hubs to support low- and full-speed devices without sacrificing large amounts of bus time to access the slower devices. The operation of the transaction translator is described, along with the various forms of split transaction and the specific sequences employed by each.

Overview

When a USB 2.0 hub is connected to a high-speed port and has one or more high-speed devices attached to its downstream ports, it performs the straightforward functions of repeating packets in both directions. In this sense the function of a 2.0 hub is no different from a 1.x hub. However, several major differences exist between the two implementations:

1. 2.0 hub receivers have their receivers squelched until a high-speed packet is detected. Thus time is required to enable the receiver, causing some bits to

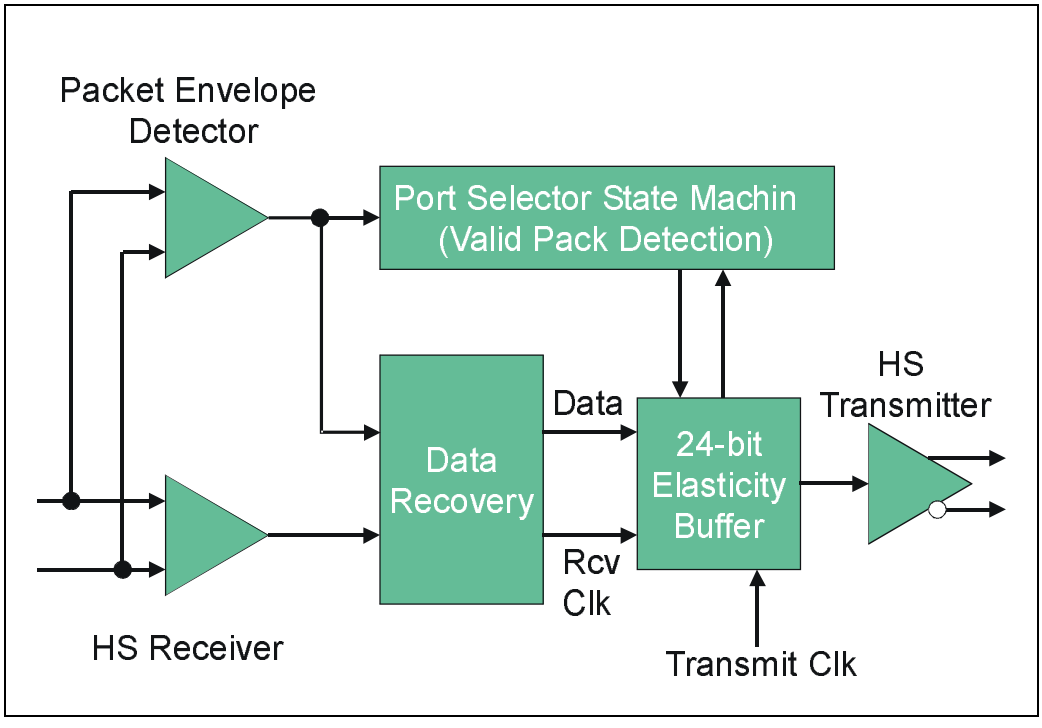
be dropped from the start of packet.

2. When a packet is received by a 2.0 hub, it re-clocks the data when sending it to the next device.
3. Re-clocking the data requires buffering some of the packet.

High-Speed Hub Repeater

Figure 16-1 provides a conceptual view of the receiver, repeater, and transmitter path. Each of these blocks are discussed below. Note that the maximum propagation delay through the buffer is specified as 36 bit times.

Figure 16-1: Repeater Function Within Hub



Receiver Squelch

As described in Chapter 11, high-speed receive circuitry is squelched when the bus is idle and enabled when a differential voltage of 150mv is detected by the packet envelope detector. The hub specification allows up to 4 bit times between detecting the packet and enabling the receiver. Thus, the first 4 bits of each packet can be dropped by a hub. This results in the synchronization clock being reduced by as many as 4 bits. Since the maximum number of hubs between any USB device and the root hub is 5, the maximum number of synchronization bits dropped is 20 out of the 32 produced at the originator.

The same 4 bit times affect the end of packet as well. When the receiving port returns to the idle state, the repeater is disabled; however, it may take up to 4 bit times to actually disable the repeater after EOP. This results in up to 4 random bits added to the end of the packet. These bits are termed dribble bits, but have no adverse effect on the detection of the packet at the receiver. They will accumulate with each hub crossing so that the fifth hub may send a packet with a maximum of 20 dribble bits. When the receiver of this packet detects EOP (via a bit stuffing error), it will check CRC and detect a valid value and the following dribble bits are simply discarded.

Re-clocking the Packet

Re-clocking packets is performed to reduce the jitter seen at a receiver so that jitter remains within the limits defined by the specification. Re-clocking involves extracting the data from the received NRZI stream and re-transmitting the stream using the hub's local clock.

Port Selector State Machine

The block represents a hub state machine whose job it is to verify that the incoming packet is valid. The state machine detects removal of squelch (which could be caused by noise) and awaits the priming of the elasticity buffer (12 bit times). The state machine then checks for the repeating pattern of "JK" or "KJ" within the elasticity buffer, which indicates the synchronization pattern and a valid packet. If no repeating pattern is detected, then transmission of the packet to the downstream ports is not enabled.

Elasticity Buffer

The elasticity buffer handles the frequency differences between the receive clock derived from the receive packet and transmit clock generated locally within the hub. The specification allows clock tolerance of 500ppm, resulting in the maximum difference between the receive and transmit clocks of 1000ppm. The elasticity buffer must handle the case where the receive clock is faster than the transmit clock and vice versa. To handle both conditions, the buffer must be filled half-way (primed) before data is clocked out of the buffer. In this way, if data is taken out of the buffer more quickly than it is being filled, no underrun will occur, and buffer space is also available to prevent overflow when data is stored faster than it is taken out.

The half-depth of the buffer must be equal to the maximum difference in clock rate over the length of a maximum-sized packet. Calculation of the half-depth of the buffer is as follows:

- given that the maximum clock difference is 1000ppm, and
- the maximum packet length is: 1024 byte data payload + total overhead including 20 dribble bits = 9644 bits;
- then the maximum overrun or underrun is approximately 10 bits (1000ppm * 9644 = 9.644 bits).

The specification requires a buffer half-depth of 12 bits to provide 2 bits of additional margin.

The Repeater State Machine

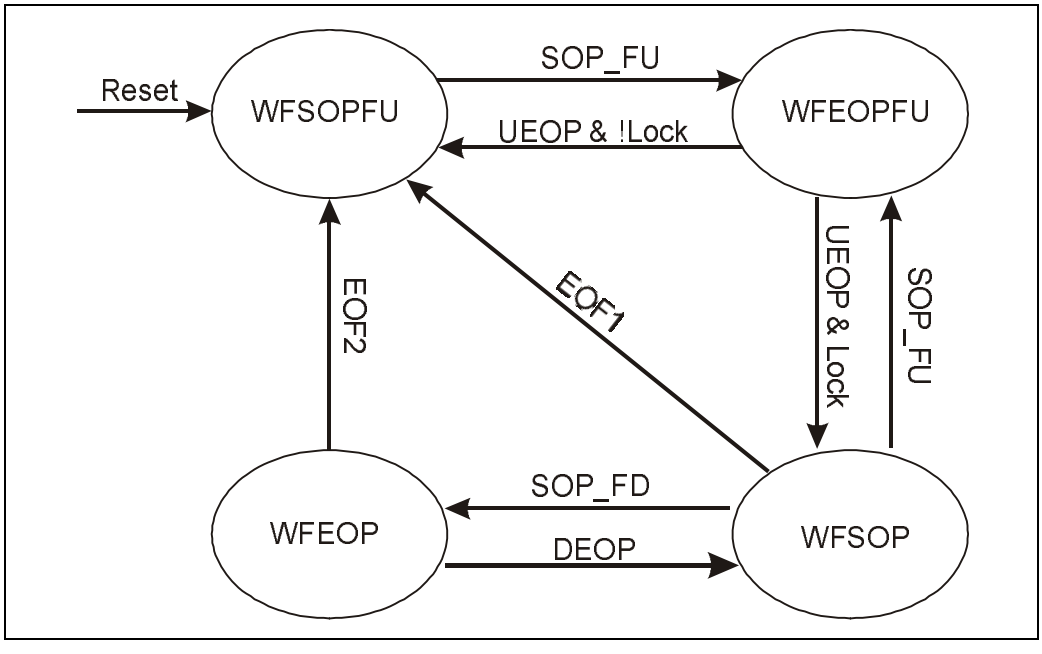
The hub repeater states are the same for low-, full-, and high-speed hub operation (See Figure 16-2 on page 287). However, when a hub is operating at high-speed, it repeats only high-speed packets. The operation of the high-speed state machine includes the following characteristics:

- Connectivity is established upon detecting a Start of High-Speed Packet (SOHP). Transitions caused by SOHP occur when the port selector state machine has ensured that a valid packet has been detected. See Figure 16-1 on page 284.
- Connectivity is torn down upon detecting a High-Speed End Of Packet (HEOP). The state transitions take place after the hub has repeated the last bit in the elasticity buffer.

Each state and transition is described in the following list:

- WFSOPFU (wait for start of packet from upstream) — This state is entered at reset and is also entered at the end of frame (EOF1 or EOF2). Thus, each microframe begins with the repeater in the WFSOPFU state, and the SOP from upstream being referred to is the start of frame (SOF) packet. Following a SOF packet from the host, the hub returns to the WFSOPFU state if the hub is not yet synchronized with (locked to) SOF timing.
- WFEOPFU (wait for end of packet from upstream — This state is entered when a start of packet from upstream (SOP_FU) is detected. This can occur from the WFSOPFU or WFSOP states.
- WFSOP (wait for start of packet) — In this state the hub is waiting for a packet from upstream or downstream. Transitions to this state occur when EOP is detected from downstream or from upstream (when the hub is locked to SOF). If, when waiting for a start of packet, the end of frame (EOF1) point is detected, the hub transitions to WFSOPFU.
- WFEOP (wait for end of packet) — In this state the hub awaits a packet from downstream. If EOP has not occurred when EOF2 is reached, then a transition to WFSOPFU occurs and the downstream port that had established the upstream connectivity is disabled.

Figure 16-2: Repeater State Machine



17 2.0 Hubs During LS/FS Transactions

The Previous Chapter

The previous chapter discussed the 2.0 hub's behavior when it receives high-speed packets on its upstream and downstream ports. The chapter also detailed the operation of the high-speed repeater and discussed the delays associated with forwarding high-speed packets across the hub.

This Chapter

This chapter introduces the concept of split transactions that allow high-speed hubs to support low- and full-speed devices without sacrificing large amounts of bus time to access the slower devices. The operation of the transaction translator is described, along with the various forms of split transaction and the specific sequences employed by each.

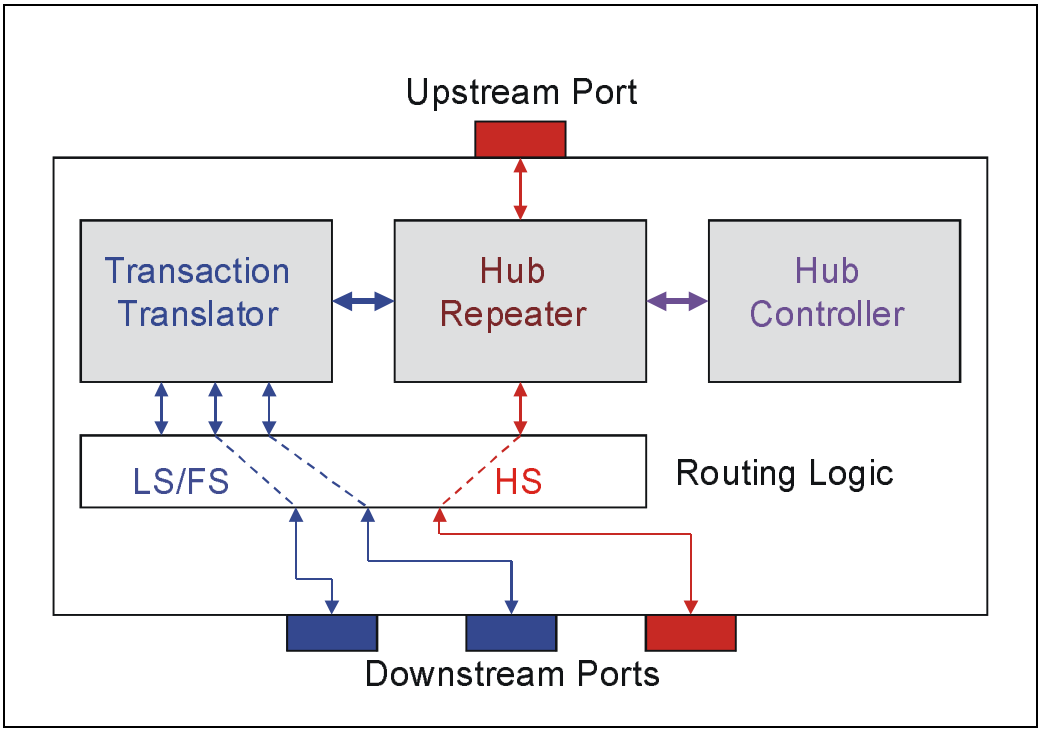
The Next Chapter

This chapter provides an overview of the configuration process. Each of the major steps involved in USB device enumeration are defined and discussed.

Overview

High-speed hubs must translate split transactions into either low- or full-speed transactions as discussed previously. This chapter details the operation of the hub and specifically the transaction translator that handles the conversion of high-speed packets to low- or full-speed. Figure 17-1 on page 290 illustrates the packet flow through a high-speed hub with low-, full- and high-speed devices attached to the hub's downstream port. Split transactions are handled by the transaction translator, resulting in a low- or full-speed transaction to the target hub port, while all high-speed packets (including split transactions) are repeated to the high-speed port.

Figure 17-1: Packet Flow Through Hub with LS/FS and HS Devices Attached



The Structure of Split Transactions

Host software issues split transactions to perform low- and full-speed transactions to devices attached to high-speed hub ports (but not root hub ports). Split transactions are used only for communication between the host controller and high-speed hubs. The actual recipient of a split transaction is the transaction translator that is responsible for converting high-speed split transactions into low- or full-speed transactions as required by the attached devices.

Split transactions are performed using two transaction types: Start Split and Complete Split. The following examples illustrate two split transactions that are part of an isochronous transfers that require no verification of data delivery. A second set of examples illustrates split transactions that do require verification of data delivery, because they are part of either an interrupt, bulk, or control transfer.

Isochronous Split Transaction Examples

The two examples that follow each describe the sequence of packets used by the host to perform an isochronous split transaction that has a relatively large data payload. In both cases, the data payload size is sufficiently large that the data transfer requires multiple microframes to complete. The first example is an isochronous OUT transaction and the second is an isochronous IN transaction.

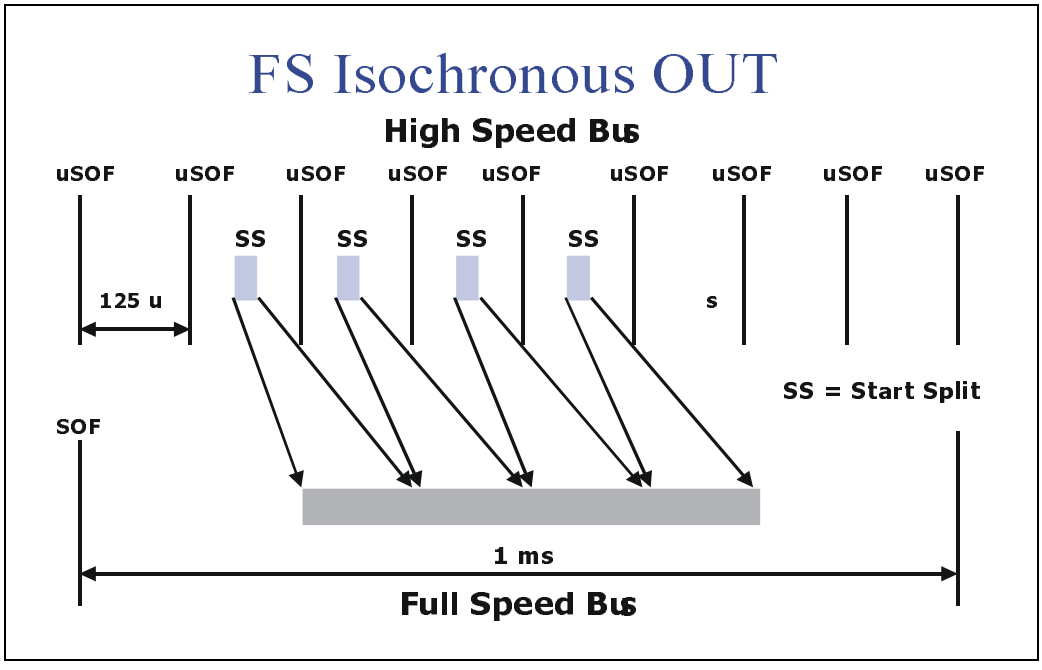
Example Split Isochronous OUT Transaction

Figure 17-2 on page 292 depicts the sequence of start split (SS) transactions associated with a lengthy isochronous OUT transaction. The first start split initiates the transaction and delivers the first block of data. Each microframe carries data at high speed that the hub will deliver to the full-speed bus during the next microframe interval. The data is delivered to the hub just in time for delivery across the full-speed bus during the following microframe interval. This method of delivering data has two benefits:

1. the data buffers within the hub can be smaller.
2. the distribution of data across multiple microframes keeps the bandwidth evened out across these frames, thereby making more bandwidth available to periodic transactions while meeting the needs of the full-speed bus.

Because no verification of data delivery is required during isochronous transactions, no complete split transaction is used in this case.

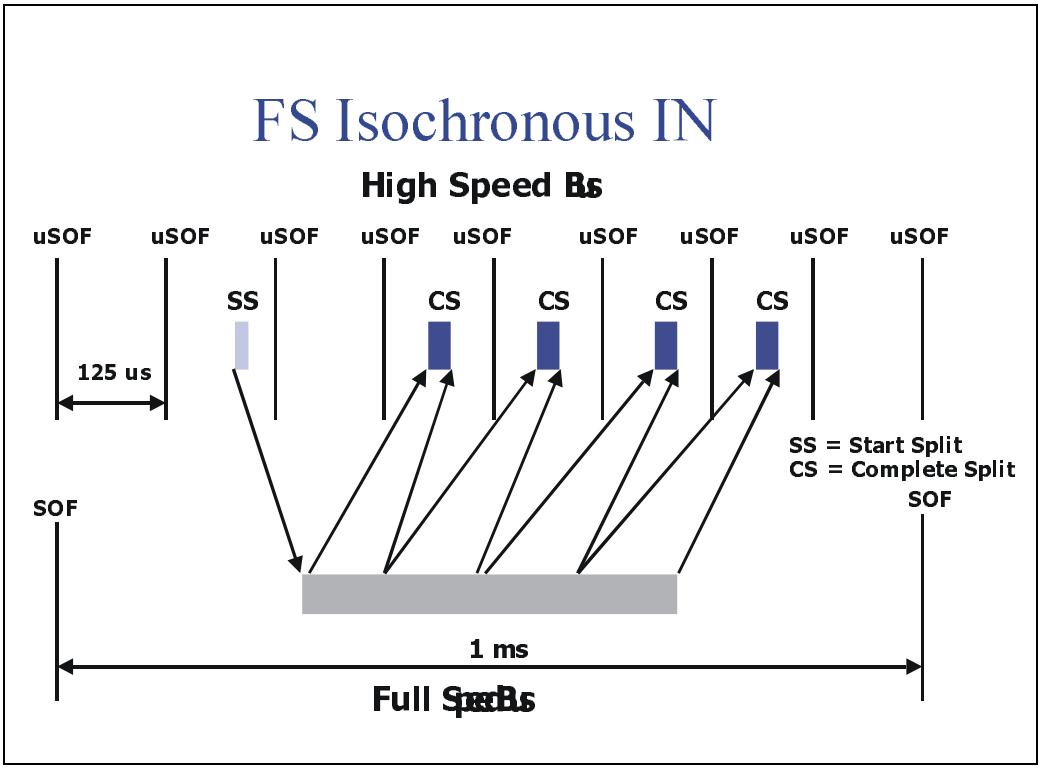
Figure 17-2: Example Isochronous OUT Split Transaction



Example Split Isochronous IN Transaction

Figure 17-3 on page 293 depicts the sequence of start split and complete split transactions required to complete the lengthy isochronous IN transaction. The start split transaction initiates the full-speed IN transaction. Then the host schedules a series of additional complete split (CS) transactions to read the data returned from the IN endpoint. The hub buffers read data returned during each microframe and send that data to the host during the following microframe via a complete split transaction.

Figure 17-3: Example Isochronous IN Split Transaction



Example Split Transactions with Data Verification

The following examples are designed to provide additional details regarding the split transactions and add verification of data delivery. These example transactions could be targeting control, bulk, or interrupt endpoints. Because the maximum packet size for these transfers is limited to 64 bytes in the full-speed environment and 8 bytes in the low-speed environment, the data transfer can complete in 125 microseconds (64 bytes @ 12Mb/s = ~42.6µs).

18 Configuration Process

The Previous Chapter

The previous chapter introduced the concept of split transactions that allow high-speed hubs to support low- and full-speed devices without sacrificing large amounts of bus time to access the slower devices. The operation of the transaction translator was also described, along with the various forms of split transaction and the specific sequences employed by each.

This Chapter

This chapter provides an overview of the configuration process. Each of the major steps involved in USB device enumeration are defined and discussed.

The Next Chapter

This chapter discusses configuration of USB devices that are attached to any USB port. The process is virtually the same for devices of any speed. Device descriptors and other characteristics and features that relate to configuring the device are also detailed and discussed.

Overview

Host software is responsible for detecting and configuring all devices attached to the root hub. The process of identifying and configuring a USB device is commonly referred to as USB device enumeration. Device enumeration begins at the root hub. Each hub port must be reset and enabled in turn. When powered, the hub determines if a low-, full-, or high-speed device is attached or no device at all. If a device is present, status bits within the root hub are set to reflect device attachment. Software recognizes that a device is attached, enables the port, resets the USB device, assigns a unique address, and completes the configuration. This operation is performed for each port until all devices attached to the root hub have been identified and configured.

The discussion in this chapter presumes that software has already initialized the USB host controller and that it is capable of generating USB transactions.

Different operating systems will define different software components involved in USB device configuration and will perform configuration in their own particular sequence. The following discussions identify the primary steps involved in the configuration process, but should not be interpreted as the specific sequence that is followed by a given host software solution. The following list specifies actions taken by host software and the root hub when configuring a device that is connected to a root hub port:

- Host requests power be applied to the ports, if not already powered.
- Hub detects device attachment and device speed, and sets status bits.
- Host polls hub and identifies that a device is attached.
- Host issues reset to port/USB device (minimum of 10ms). Chirp is performed during reset if a high-speed device is attached.
- Host checks hub status again if the device originally reported full-speed operation. This is done to see if the device is now operating at high speed.
- USB device now answers to default address (zero).
- Host performs GetDescriptor requests to fetch the standard descriptors that contain configuration information. These descriptors are parsed by software to determine the characteristics of the device. The configuration information includes bus power and bus bandwidth requirements, and device class information.
- Host assigns unique address to USB device.
- Host verifies that the USB resources needed by the device are available.
- Host issues a configuration value to the USB device specifying how it's to be used. When the configuration value is received, the device assumes the characteristics that it reported via the descriptors. The device is now ready to be accessed by client software and can draw the amount of bus power described in the configuration.

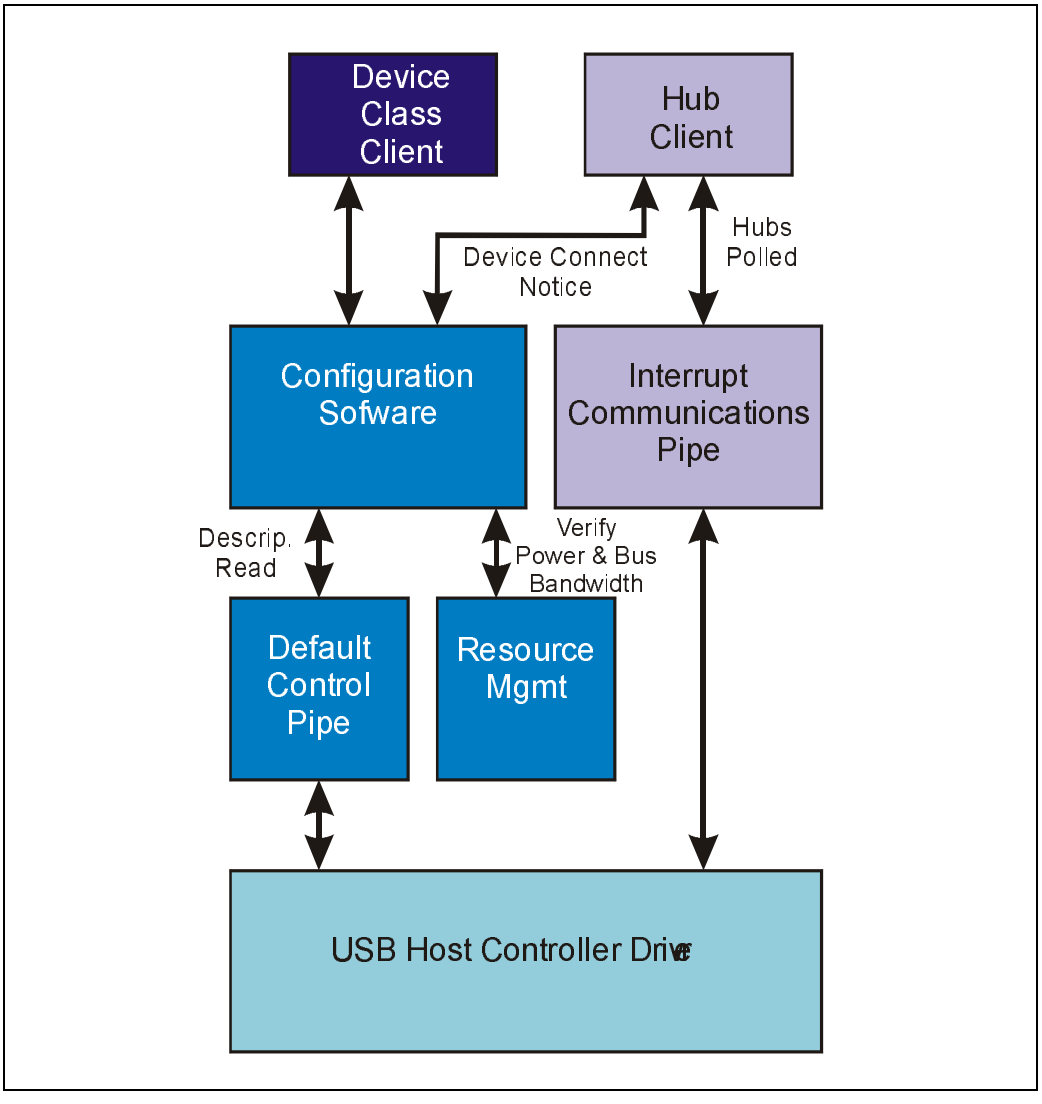
The operations performed above are primarily accomplished via control transfers. Software uses each device's control endpoint (endpoint zero) to access its descriptors and to configure the device. (The mechanisms used for performing control transfers to a device are described in the following chapters.)

The following sections provide an overview of the configuration process. More detail regarding device configuration can be found in the following chapters.

The Configuration Software Elements

Figure 18-1 illustrates the model that configuration is based upon and identifies the conceptual software elements involved.

Figure 18-1: The Software Elements Used During Configuration



USB Host Controller Driver

This software is typically the first USB-specific software to execute. The host controller must be initialized by this driver before USB becomes active. During configuration this driver also performs accesses to root hub status and control registers under the direction of the hub client.

Configuration Software

The hub client must notify configuration software that a new device has been attached. In response, configuration software must identify the newly attached device by reading its standard descriptors, using the default control pipe. Information read from the descriptors specifies the USB resources that are required by the device (e.g., bus bandwidth and power requirements). These descriptors must be interpreted by configuration software.

A device can support more than one configuration, requiring software to select the configuration of choice. Each configuration provides alternatives for configuring the device. For example, a device may support both high-power and low-power configurations. That is, if the hub port cannot support a high-power function, then configuration software could select the low-power configuration. Note that the client driver may also be involved in selecting the configuration.

Configuration software assigns a unique device address to the device and completes its hardware configuration by assigning a configuration to the device. When the configuration is assigned, the device is enabled and ready for operation.

The configuration software then finds the client driver software for this device based on its class or vendor-specific information that was previously read from the standard descriptors.

Default Control Pipe

The default control pipe accesses endpoint zero within each device. The hub client, configuration software, and device client software all share access to this communications pipe. The host system software is the default owner of the pipe; thus, hub clients and other device client drivers must request its use.

Resource Management

Configuration software must ensure that the available USB resources can support the device configuration that is selected. Resource management software must track the bus bandwidth consumed by all USB devices as they are configured. As each device is attached, resource management software must verify that the periodic endpoints can be supported based on their bandwidth needs and the bandwidth that remains.

Device bus power requirements must also be checked against the power available at the port to which the device is attached. Power available at a hub port must be determined by the hub client, because it has specific knowledge of the power characteristics of the hub. Port power capacity is requested by resource management software so that it can verify power requirements can be met.

If USB can provide the bus power and bandwidth needed by the device, then the device can be configured. If the USB cannot support the requested configuration, then alternate device configurations are checked; if none of the configurations can be supported, the device is not configured and the user is notified.

Device Client Software

Once configuration software has configured the device it is ready for operation from the hardware perspective. Client software is identified by configuration software and loaded. This software must read client-specific and/or class-specific descriptors to verify its capabilities. Client software can then initialize the device for use by the application layer.

Root Hub Configuration

Host software begins USB device enumeration by configuring the root hub. The root hub must implement a status change endpoint (see Figure 18-2) that can be used by the hub client to detect status changes pertaining to each port. Once the hub is configured, software can poll the status change endpoint to detect which ports currently have devices attached to them.

19 USB Device Configuration

The Previous Chapter

The previous chapter provided an overview of the configuration process. Each of the major steps involved in USB device enumeration were defined and discussed.

This Chapter

This chapter discusses configuration of USB devices that are attached to any USB port. The process is virtually the same for devices of any speed. Device descriptors and other characteristics and features that relate to configuring the device are also detailed and discussed.

The Next Chapter

Hub devices are configured like any other device attached to a USB port. Hub configuration differs in that it involves reporting whether or not other devices are attached to the downstream ports. The next chapter reviews the hub configuration process with the focus on the issues related to extending the bus through the hub's downstream facing ports.

Overview

During system boot and initialization, all USB hubs and devices will be detected and configured. Following initialization, devices may be detached or new devices may be connected. The hub client periodically polls all USB hubs to detect device attachment or detachment. If a status change results from a new device having been attached, the configuration process is triggered.

Summary of Configuration Process

Prior to configuring a device, the hub to which the device is attached must have already been configured and power must be applied to the port. Next, the hub and configuration software must detect the connected device:

- The hub recognizes that a device has been attached by monitoring the D- and D+ port signals.
- The hub sets status information for the port indicating device connect and speed.
- Configuration software reads port status and recognizes that a full-speed device is connected.
- Software then enables the port so that the hub will pass bus traffic to the device.
- Configuration software then issues a *Reset Port* request, forcing the device into its default state. In the default state the device is unconfigured and responds only to accesses targeted for device zero and endpoint zero.

Configuration software can now begin the device configuration process. This process is similar to that used when configuring a hub.

- Host queries device's control endpoint (zero) at address zero to determine maximum payload supported by the default pipe.
- Host assigns unique address to USB device.
- Host reads and evaluates configuration information from descriptors.
- Host verifies that the USB resources needed by the device are available.
- Host issues a configuration value to USB device specifying how it's to be used. When the configuration value is received, the device assumes its described characteristics. Device is now ready to be accessed by client software and can draw the amount of Vbus power described in the configuration.

How Software Detects Device Attachment & Speed

Hubs have specific knowledge of whether a device has been attached to one of their ports only when the port is powered. The software responsible for applying power to a hub port, for detecting that a device is attached to a port, for resetting the port, etc. is, of course, the hub client software. This software plays a pivotal role in device configuration. This section deals with how the hub client determines that a device is attached to a port and the speed at which it operates.

Configuration always begins at the root hub. The host controller driver actually performs the low-level accesses to registers within the controller to determine whether a device is attached to one of the root ports. However, the host controller driver software must create an abstraction of the regular USB hub so that the hub client can make requests to the host controller just as it does when accessing USB hubs. Because of this abstraction, the same process is described for both the root hub and USB hubs that reside on the USB. The actual mechanisms used to access status information is different, but only the USB mechanisms are described. This is because the host controller driver uses conventional memory or I/O transactions to read root hub status, whereas USB endpoints must be accessed to obtain status from other USB hubs.

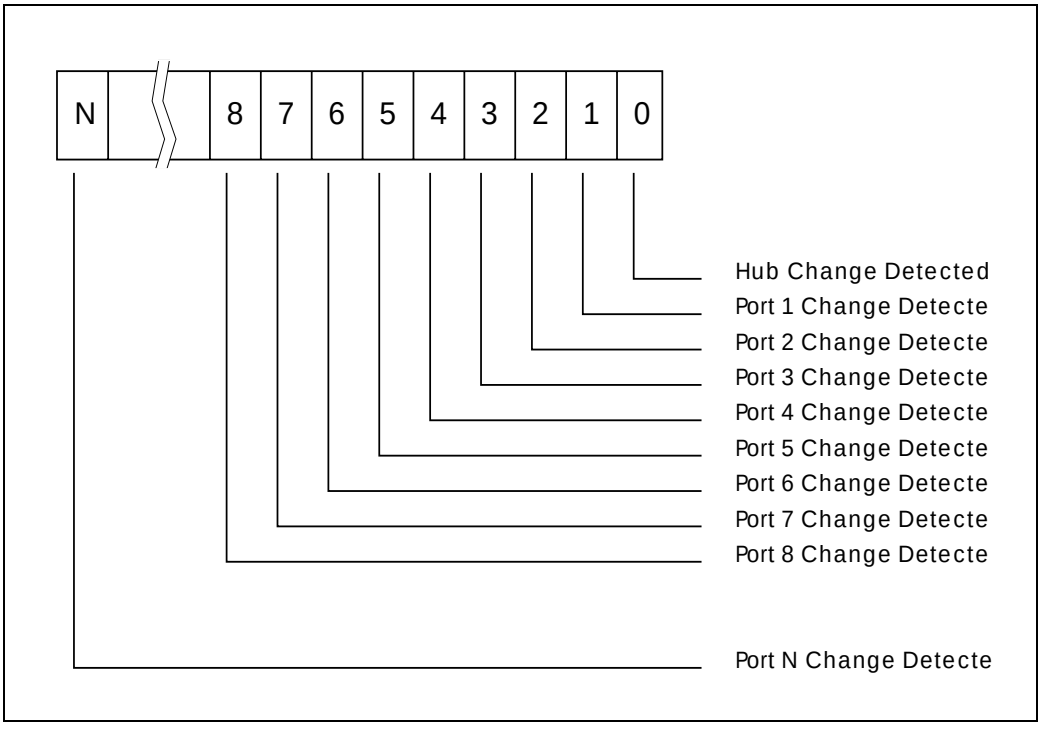
Polling the Status Change Endpoint

The first step in determining whether a device has been attached to a port is for the hub client to poll the hub's status change endpoint. Hubs implement an interrupt endpoint that keeps track of status change events by setting status bits. Figure 19-1 illustrates the definition of the information returned by the status change endpoint. Configuration software is aware of the number of ports supported by each hub, and therefore is aware of the size of the bitmap returned when the hub's status change endpoint is polled. Status is reported in byte-sized fields with zeros returned in the bit fields corresponding to ports that are not implemented. For most implementations a byte will be returned because hubs rarely have more than 7 ports.

When software polls the status change endpoint, the bitmap illustrated in Figure 19-1 is returned only if a status change has occurred. If, when polling the status change endpoint, it returns NAK during the IN transaction, then none of the bits are set, and no changes need to be reported.

To move ahead with the discussion, we must presume that at least one of the port bits is set. Upon detecting that a bit is set, the hub client must read status information from the port. This requires use of a control transfer.

Figure 19-1: Hub and Port Status Change Bitmap



Getting Port Status

The hub client obtains temporary ownership of the default control pipe from the USB software driver so it can perform a GetPortStatus request to the hub. In this case the control transfer consists of:

- The Setup Stage — This setup transaction delivers 8 bytes of data that specify the GetPortStatus request.
- The Data Stage — This IN transaction is used to return port status to the hub client.
- The Status Stage — This OUT transaction verifies that the operation was successful, and that the status information is valid.

Table 19-1 defines the 8 bytes of data that are sent during the setup transaction. The first byte (Request Type) is a bitmap that is described in "Hub Request Types" on page 448. The other fields are described in the table. Note that the last column within the table specifies that the data returned are the "port status" and "change Indicators."

Table 19-1: Hub's Get Port Status Request

Request Type	Request	Value	Index	Length	Data
10100011B	GET_STATUS (00h)	Zero	Port Number	Four bytes	Port Status and Change Indicators

The following tables define the port status change indicator and current status information. The change indicators in Table 19-2 identify the various port events that may be reported, but our purpose is to note bit zero. This bit would be set indicating that a change in connection status has been detected by the hub port interface. The status change information prevents software from having to store previous status in order to detect a change.

Table 19-2: Format of Port Change Fields Returned During the GetPortStatus Request

7	6	5	4	3	2	1	0
Reserved (returns all zeros when read)			Reset Complete Change	Over-Current Indicator Change	Suspend Change (resume complete)	Port Enabled / Disabled Change	Connect Status Change

15	14	13	12	11	10	9	8
Reserved (returns all zeros when read)							

The status information in Table 19-3 specifies the current state of each field. In this case the hub client would detect bit 0 is set, indicating that a device is currently attached. Client software is also aware that bits 9 and 10 should be checked to determine the speed of the device. Note that bits 12:10 were added by the USB 2.0 specification. See "Port Change Fields" on page 459 for details regarding the definition and use of the other hub port status change indicator fields.

20 Hub Configuration

The Previous Chapter

The previous chapter discussed the configuration of USB devices that are attached to any USB port. The process is virtually the same for devices of any speed. Device descriptors and other characteristics and features that relate to configuring the device were also detailed and discussed.

This Chapter

Hub devices are configured like any other device attached to a USB port. Hub configuration differs in that it involves reporting whether or not other devices are attached to the downstream ports. This chapter reviews the hub configuration process with the focus on the issues related to extending the bus through the hub's downstream facing ports.

The Next Chapter

The next chapter introduces the concept of device classes and discusses their role within the USB. This chapter also introduces all the approved class types (at the time of this writing) and provides a more detailed summary of the audio, mass storage, monitor, and communications classes. These classes are discussed to provide the reader with a sense of the information defined for each class and the USB mechanisms that they use. A detailed discussion of device classes requires in-depth knowledge in the associated field such as telephony and audio, and is outside the scope of this book.

Configuring the Hub

Hubs must be configured like any other device, but this also involves identifying other devices that may be attached to the port. The steps taken by configuration software include:

- Reading the standard device descriptors to obtain a variety of information needed to configure the device.
- Assigning a unique address to the hub.
- Powering the ports.
- Checking the hub status change endpoint to detect port events.
- Reading status information to determine the nature of the event.
- Enabling the port to provide access to the attached device.

As indicated above, a hub must implement a status change port in addition to the default port. Figure 20-1 illustrates the required hub endpoints. The default control port provides access to the descriptors that define the type of device requiring configuration. A hub may also be implemented as part of a compound device; hence, the descriptors may describe additional functions beyond the minimum required of the hub alone.

The Default Pipe

All devices, including hubs, have a default control pipe at endpoint zero. Host software owns the default control pipe that is used to configure hubs and USB devices. This communications pipe is established during initialization so that USB devices can be accessed based on established defaults. The configuration process requires numerous default pipe accesses for configuring and controlling hub features, including: reading the device descriptors, powering hub ports, resetting ports, reading port status, and enabling ports.

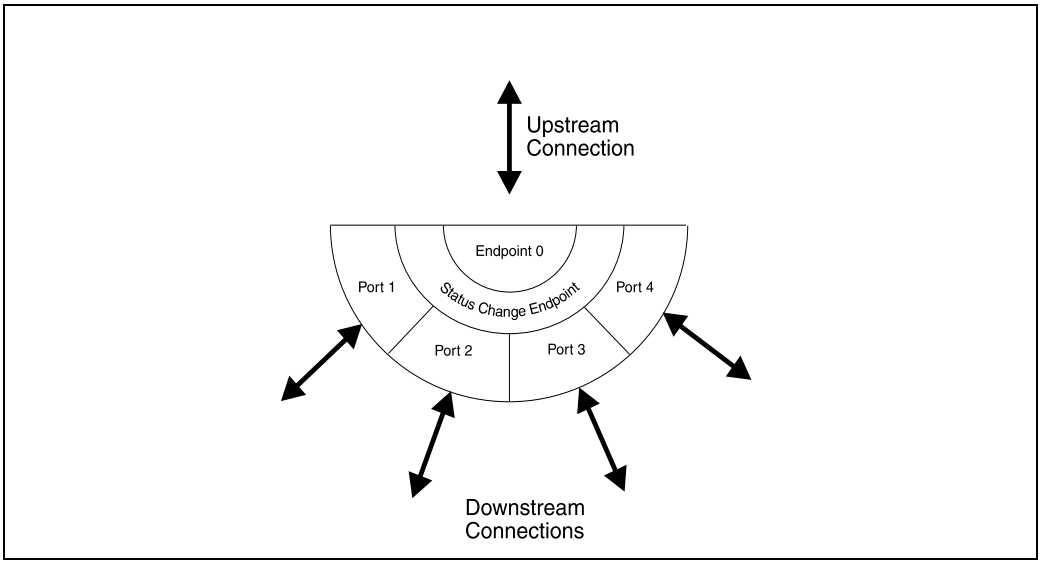
The Status Change Pipe

Hubs must implement a status change endpoint that can be polled to detect status changes that have occurred at the hub ports (e.g., device attachment and detachment). Note that the status endpoint provides status information for all hub ports. Configuration software determines the characteristics and the endpoint number of the status change register by reading the endpoint descriptor.

Reading the Hub's Descriptors

Hubs have a class specific descriptor called the hub descriptor. This descriptor contains information about the hub implementation. The hub class descriptor is read via the class specific "Get Descriptor" request.

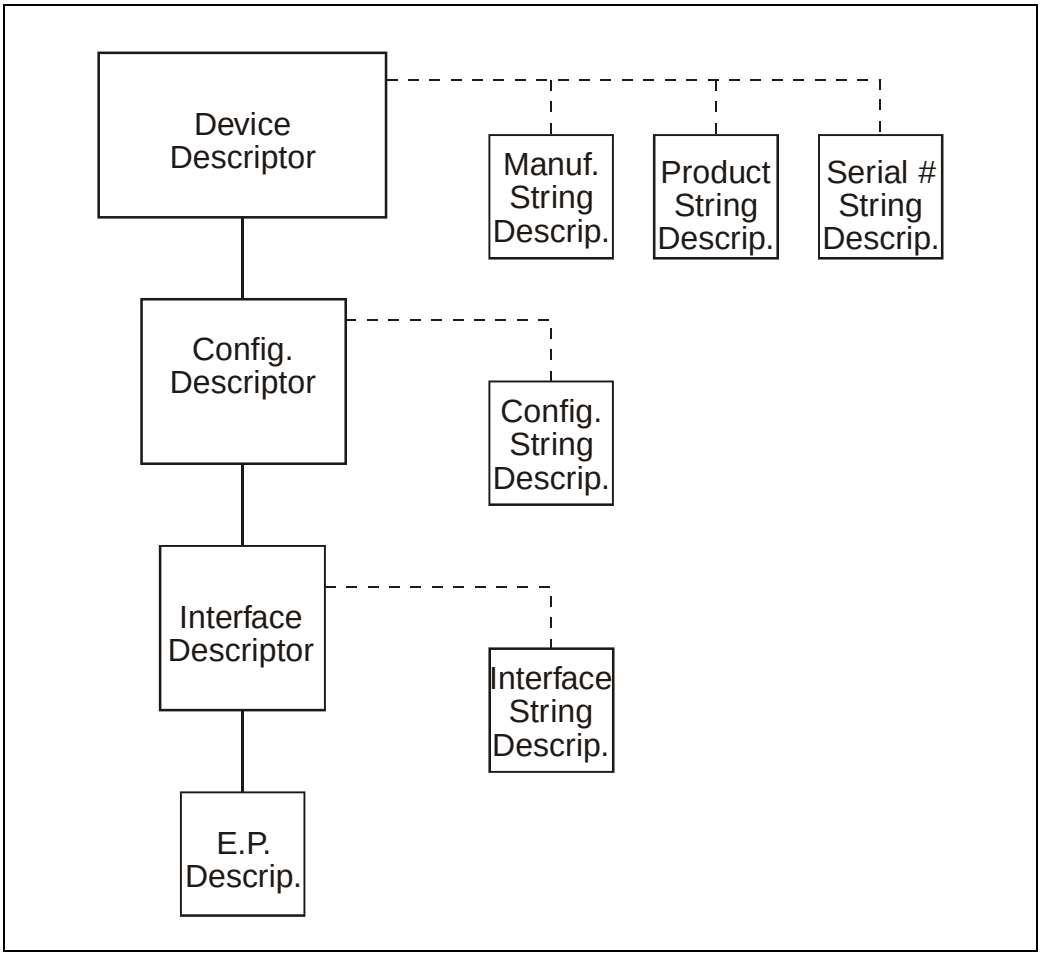
Figure 20-1: Required Hub Endpoints



Hubs, like other devices, also contain standard descriptors that must be read to determine how to configure the hub. Standard descriptors are read via the standard request, *Get Descriptor*. Hubs contains the following standard descriptors, as illustrated in Figure 20-2:

- Each USB device contains a single device descriptor that describes the number of configurations supported by the device.
- Each device contains one or more configuration descriptors that describe one or more interfaces.
- Each interface descriptor defines the number of endpoints related to the interface.
- Endpoint descriptors specify the attributes associated with a given endpoint along with information needed by host software to determine how the endpoint should be accessed.
- String descriptors are optional and consist of a UNICODE string that provides human-readable information that can be displayed.

Figure 20-2: Standard Hub Descriptors



1.x Hub Descriptors

The information provided in this section describes the hub descriptors in a 1.x compliant implementation. Most of the descriptor values discussed in this section also apply to 2.0 compliant hubs. “High-Speed Capable Hub Descriptors” on page 391 discusses and illustrates the differences between the 1.x and 2.0 descriptors.

Hub's Standard Device Descriptor

Hubs implement the standard device descriptor just like other USB devices. However, hubs contain some pre-defined descriptor fields as indicated below:

- DeviceClass field = HubClass
- DeviceSubClass field = HubSubclass
- MaxPacketSize0 field = 8 bytes

Refer to Table 20-1. The first access made by configuration software is to the device descriptor to determine the maximum payload supported by the default pipe (offset 7 within the device descriptor). In this case, the packet size is pre-defined for hubs as 8 bytes. Software may also detect the device class type by reading the device descriptor. However, if the hub is a composite device, the class field will be 0s and the interface descriptors will define the device class (one interface for the hub function and one interface for each embedded function).

Table 20-1: Hub's Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Number	Size of this descriptor in bytes.
1	DescriptorType	1	01h	DEVICE Descriptor Type.
2	USB	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e., 2.00 is 200). This field identifies the release of the USB specification that the device and its descriptors are compliant with.
4	DeviceClass	1	09	Hub Class code = 09h.
5	DeviceSubclass	1	0	Hub Subclass code (assigned by USB). These codes are qualified by the value of the DeviceClass field. If the DeviceClass field is reset to zero, this field must also be reset to zero.

21 Device Classes

The Previous Chapter

Hub devices are configured like any other device attached to a USB port. Hub configuration differs in that it involves reporting whether or not other devices are attached to the downstream ports. The previous chapter reviewed the hub configuration process with the focus on the issues related to extending the bus through the hub's downstream facing ports.

This Chapter

This chapter introduces the concept of device classes and discusses their role within the USB. This chapter introduces the class types and provides a more detailed summary of the audio, mass storage, monitor, and communications classes. These classes are discussed to provide the reader with a sense of the information defined for each class and the USB mechanisms that they use. A detailed discussion of device classes requires in-depth knowledge in the associated field such as telephony and audio, and is outside the scope of this book.

The Next Chapter

Host software consists of three types of components: the USB device drivers, the USB driver, and the host controller driver. This chapter discusses the role of each of these layers and describes the requirements of their programming interfaces.

Overview

Device classes are intended to permit a device driver design that can manipulate a set of devices that have similar attributes and services. A given class definition can further describe the individual characteristics of particular device types within the class, thereby providing the USB device driver with the information it needs to manipulate the device as required.

Device class definition relates to a functional interface used to access and control a particular class of device. For example, Figure 21-1 on page 405 illustrates a USB CD-ROM device with two interfaces: mass storage and audio. Devices that support two or more functions are called composite devices and require an individual programming interface for each function of a given class. A mass storage class driver is required to use the CD-ROM for reading files from a program disk, while the audio class device driver is required to play a music CD. Each driver accesses the collection of endpoints that constitute the status, control, and data interface(s) for the device class.

Note that the USB components that are device class specific are the functional interface and the USB device driver. The descriptors associated with a device specify several items that relate to device class definitions, including:

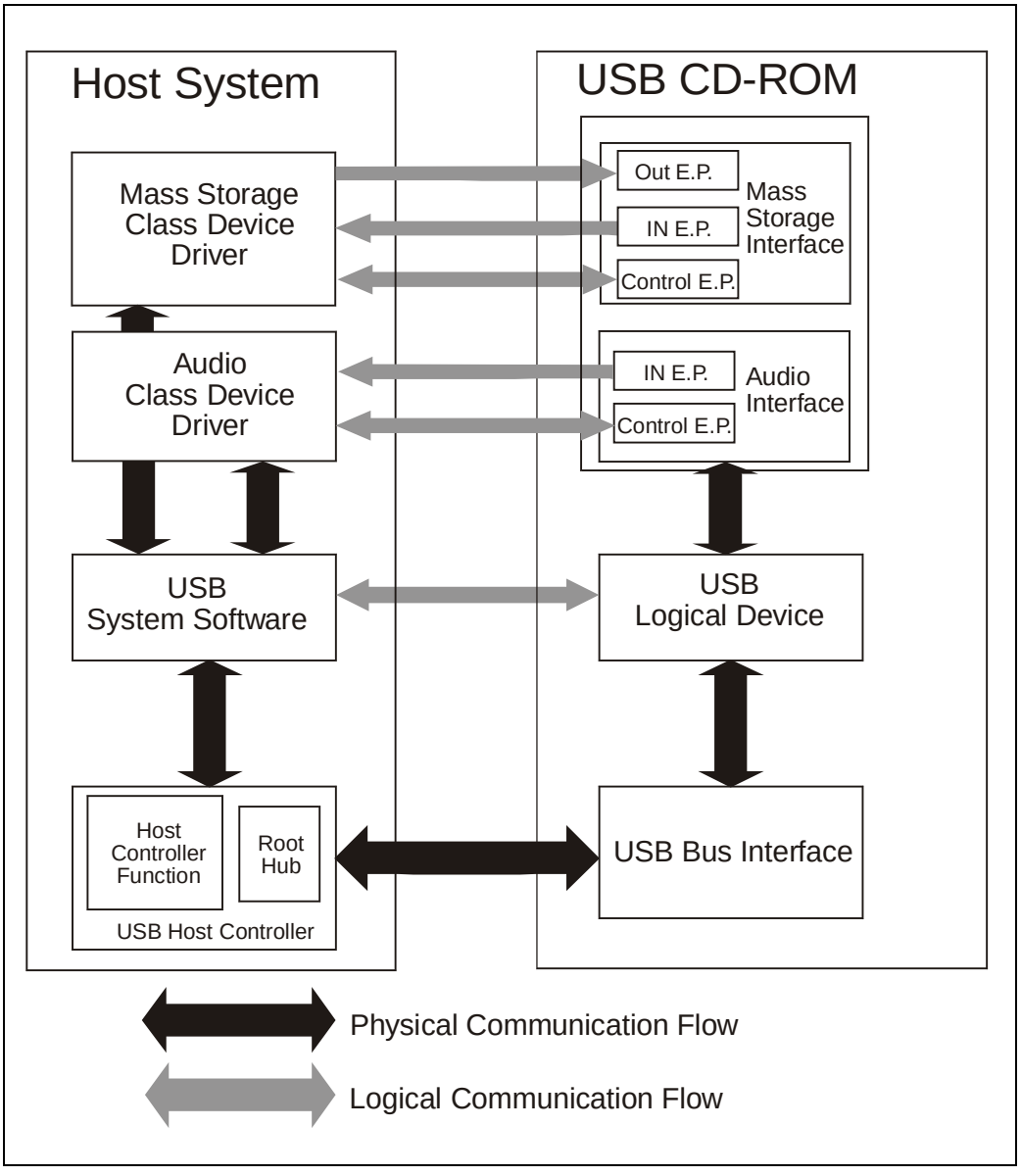
- Device Class Code field — from each functional interface descriptor.
- Sub Class Code field — the definition of this field is device class specific.
- Protocol field — this optional field may be defined for a given device class and subclass to define some element of the programming interface supported by the device.

The device class code fields can be used by host software to locate the appropriate device driver needed to access a device's functional interface. All other standard descriptor information is related to USB specific information that the host software can interpret without knowledge of the device class definitions.

Note also that some device class specifications define additional descriptors that host software has no particular knowledge of. These descriptors are intended for the class-specific device drivers to detect device attributes and characteristics required to access the device.

In summary, class definitions help establish a common grouping of devices that a common class driver could accommodate and allow a device to describe its capabilities to the host and USB class device driver. Class codes provide a mechanism for USB host software to identify the appropriate device driver that is designed to manipulate a given USB device's functional interface. The individual class specifications describe specific characteristics and attributes that devices within the class may support and define the control mechanisms used by a USB class device driver to access and manipulate its function.

Figure 21-1: CD-ROM Supporting Mass Storage and Audio Interfaces



Device Classes

Device class documents can be found on the USB Implementers Forum web site (www.usb.org). The device classes that were approved at the time of this writing include:

- Audio Class — Devices that are the source or sink of real-time audio information. This class is defined in four separate documents.
  - Audio Device Document 1.0
  - Audio Data Formats 1.0
  - Audio Terminal Types 1.0
  - USB MIDI (music instrument device interface) Devices 1.0
- Communications Device Class — Devices that attach to a telephone line (not local area networks). This class is defined in two documents:
  - Class Definitions for Communication Devices 1.1
  - Communications Device Class 1.0
- Content Security — This class defines transport mechanisms, descriptors, and USB requests to support a method for protecting the distribution of digital content via USB. The digital content being protected is usually copyrighted information. This class is defined in three documents:
  - Device Class Definition for Content Security Devices 1.0
  - Content Security Method 1 - Basic Authentication Protocol 1.0
  - Content Security Method 2 - USB Digital Transmission Content Protection Implementation 1.0
- Human Interface Device Class (HID) — Devices manipulated by end-users, and is defined in three documents:
  - Human Interface Devices 1.1
  - HID Usage Tables 1.1
  - HID Point of Sale Usage Tables 1.01
- Image Device Class — Devices that deal with still image capture.
  - Still Image Capture Device Definition 1.0 document.
- IrDA Class — This class defines an interface for infrared transceivers and is defined by the:
  - IrDA Bridge Device Definition 1.0 Document.
- Mass Storage Device Class — Devices used to store large amounts of information (for example, floppy drives, hard drives, and tape drives). This class is defined by four documents:
  - Mass Storage Overview 1.1
  - Mass Storage Bulk Only 1.0
  - Mass Storage Control/Bulk/Interrupt (CBI) Specification 1.0
  - Mass Storage UFI Command Specification 1.0

- Monitor Class — Defined to control monitor configuration and is specified in a single document.
  - Monitor Device Document 1.0.
- Physical Interface Device Class (PID) — Devices that provide tactile feedback to operator. Examples include: Joystick with variable resistance for simulating increased stick forces and turbulence. Split off from HID class. This class is specified in the:
  - Device Class Definition for PID 1.0 document.
- Power Device Class — Devices that provide power to system or to peripherals. Example devices include: Uninterruptable power supplies and smart batteries. Can be either stand alone device or integrated into the interface. The related document is the:
  - Power Device Class Document 1.0.
- Printer Device Class — Defines the descriptors, endpoints, and requests for printers. This class is specified by a single document.
  - Printer Device Class Document 1.1

Another important class document that relates to all device classes is the "Universal Serial Bus Common Class Specification." This document is intended as a guideline for developing device class specifications to promote compliant implementations of generic device drivers. To this end, the document describes the basic requirements for all USB classes and related specifications. It also describes common characteristic, attributes, and services used by many of the classes.

The following sections introduce the major features of the audio, device classes discussed previously.

Audio Device Class

The audio class specification defines standardized audio transport mechanisms used to propagate and control digital audio. A major focus of the audio class is synchronization of the audio data stream to ensure no distortion of the sound.

Each audio function has its own device interface that is used to access and control the function. Audio devices are those devices that interact with USB-compliant audio data streams. Audio devices are grouped into subclasses as listed below:

- 8-bit Pulse Code Modulated (PCM) Audio Data
- 16-bit PCM Audio Data
- 16-bit Dolby Surround Data

22 Overview of USB Host Software

The Previous Chapter

The previous chapter introduced the concept of device classes and discussed their role within the USB. The chapter introduced the class types and provided a more detailed summary of the audio, mass storage, monitor, and communications classes. These classes were discussed to provide the reader with a sense of the information defined for each class and the USB mechanisms that they use. A detailed discussion of device classes requires in-depth knowledge in the associated field such as telephony and audio, and is outside the scope of this book.

This Chapter

Host software consists of three types of components: the USB Device Drivers, the USB Driver, and the host controller driver. This chapter discusses the role of each of these layers and describes the requirements of their programming interface.

USB Software

Host software provides the interface between USB device drivers (or client drivers) and the devices that they must communicate with. USB device drivers are unaware of the USB implementation. That is, they have no knowledge of the characteristics, capabilities, or limitations of the USB nor of the USB device. Consequently, USB host software must accept transfer requests made by USB device drivers and perform the requested transfers based on the requirements of the USB. The following general capabilities are supported by a USB system.

- USB Interface Control
- Configuration Services
- Bus and Device Management
- Power Control

- Device Data Access
- Event Notification
- Collection of Status and Activity Statistics
- Error Detection and Handling

USB software is based on the device framework established by the USB specification. This framework describes the logical view that each software element has of the USB devices. The relationships between the USB software layers and their view of USB devices are reviewed below. Refer to Figure 22-1 on page 423.

Function Layer

Transactions performed over the USB are initiated by USB device drivers (client drivers). During configuration, the hub client accesses the bus, and when accessing other USB devices, the client drivers may be class-specific or vendor-specific. No matter which USB client driver wishes to access a given USB device, it must use host software to request its I/O transfer be performed over USB (via an I/O request packet, or IRP). These clients only have knowledge of the device interface (consisting of a collection of endpoints) that they wish to manipulate. Therefore a USB client driver's visibility to USB is limited to:

- the interface within their device
- class-specific descriptors that have been pre-defined to help them determine specific characteristics of their interface
- the mechanisms provided by host software to access and control their function

Device Layer

Client initiated transfers must be performed according to the characteristics of the USB and the capabilities of the target USB device. USB host software exists to support USB clients by providing services that they can use to initiate transfers and control their devices.

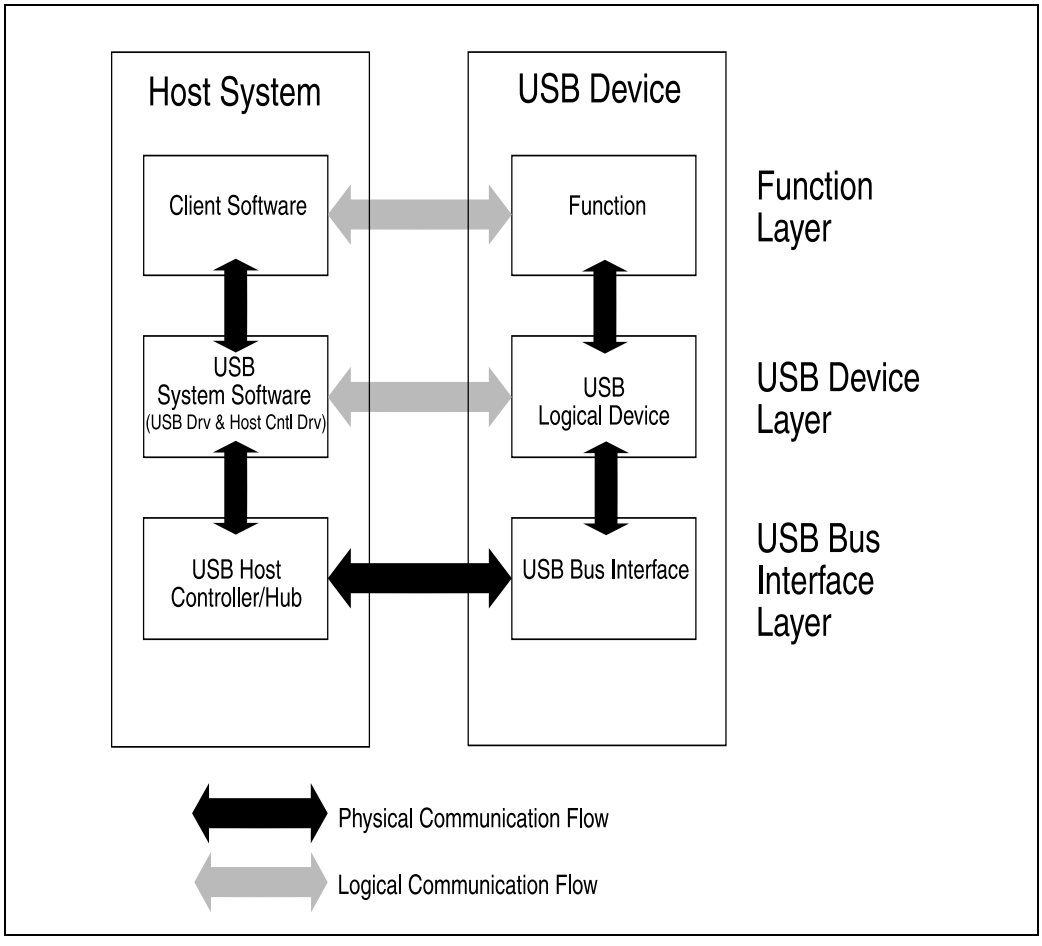
Host software also ensures that the bus can support all devices attached to the USB. The host software views each device through its standard device descriptors. These descriptors provide the necessary information to determine which driver will use this device and how much of the bus bandwidth is required. With this knowledge, host software can establish the communications pipes that the clients will later use when they access their interface.

Host software must also forward IRPs to the host controller driver (HCD), which has specific knowledge of the host controller design. The HCD passes the IRPs on to the host controller in a form that it understands.

Interface Layer

This layer is represented by the host controller (including the root hub), the USB cable(s), and the device's USB interface. These components actually transmit control and data information to/from the USB devices.

Figure 22-1: Device Framework — Software's View of Hardware



The Software Components

The three software components that compose the host USB software solution are illustrated in Figure 22-2 on page 425. The primary functions associated with each layer include the following (note, however, that the exact division of responsibility is not precisely defined by the specification):

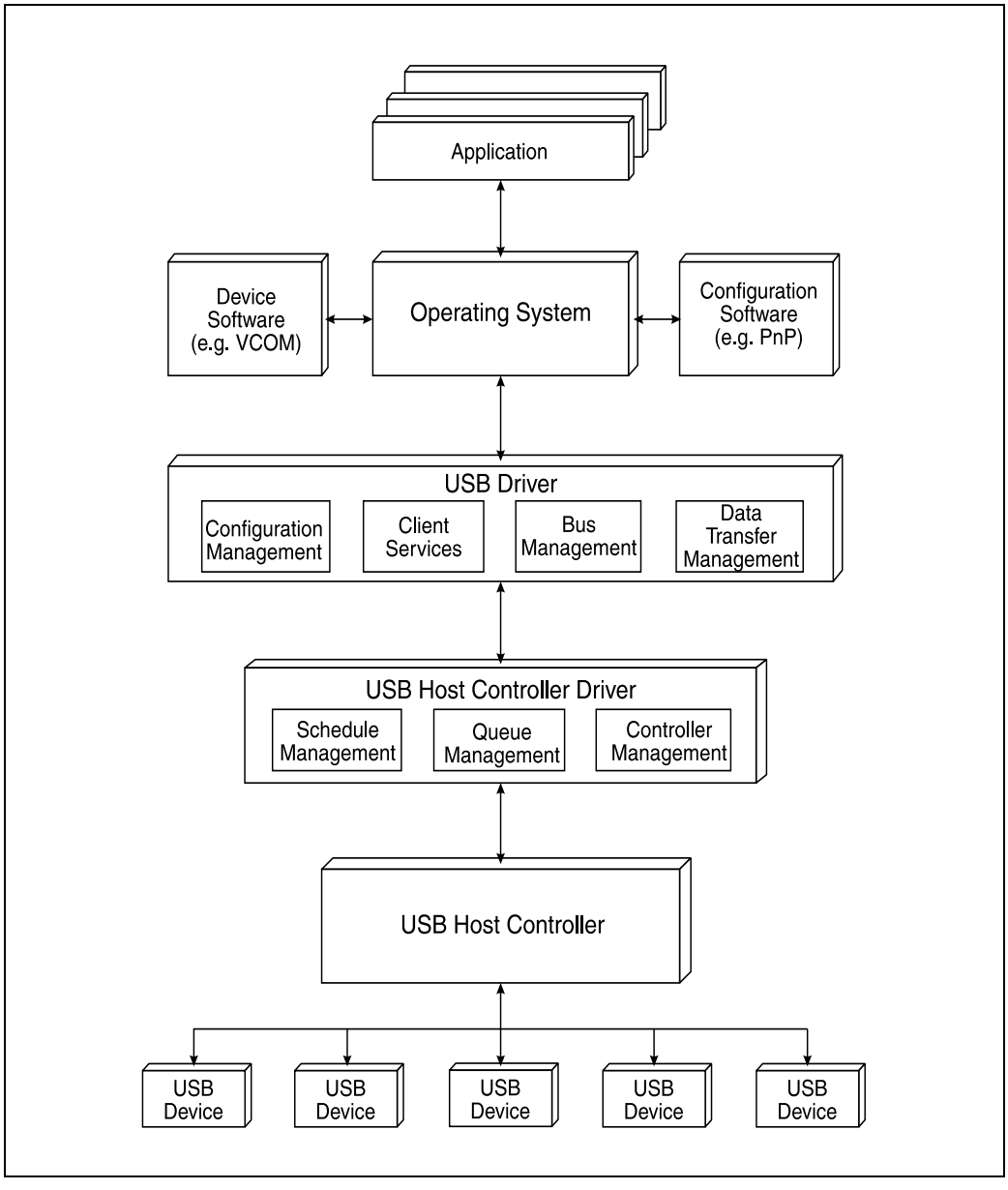
USB Client Drivers — client drivers are the software entities that control a given USB functional device. Client drivers must exist for each type (class) of function attached to the USB. These drivers are unaware of the details associated with the USB transfer mechanisms and must rely on USB host software to manage their transfer requests based on the capabilities and limitations of the USB. The intended implementation of client drivers is based on device class definitions. Client drivers view the USB devices as a collection of endpoints that can be accessed to control and communicate with their function.

USB Driver (USBD) — The USB driver has knowledge of the devices requirements (via device descriptors), as well as knowledge of the USB's capabilities. With this knowledge the USBD must divide IRPs into USB- and device-sized chunks. The USBD also supports USB device configuration by ensuring that the USB resources required by each device can be supported. It also establishes communications pipes for each endpoint detected during configuration, but only if the necessary USB bandwidth is available. The USBD provides a programming interface called USBDI (USB driver interface), giving client drivers a way to request transfers be performed to or from their USB function. A variety of client services are provided by the USB driver to assist the USB client in controlling and accessing its function.

USB Host Controller Driver (HCD) — two implementations of USB Host Controllers have been defined: the Open Host Controller and Universal Host Controller. Consequently, two Host Controller Drivers must be implemented if each controller is to be supported by host software. The Host Controller Driver provides the low level support for the USB by converting IRPs into individual transactions to be performed via the USB. The programming interface between the USB Driver and the Host Controller Driver is implementation specific and is not addressed by the USB specification.

Note that the USB specification does not define the exact division of duties that the USBD and HCD are responsible for. Subsequent chapters define the basic requirements of their programming interfaces; however, the exact implementation of these interfaces is operating system dependent.

Figure 22-2: Software Layers



Appendix A: Standard Device Requests

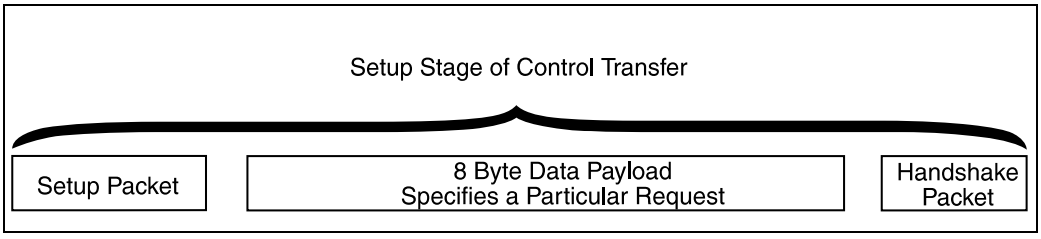
Overview

All USB devices must respond to a variety of requests called “standard” requests. These requests are used for configuring a device, for controlling the state of its USB interface, and with other miscellaneous features. Device requests are issued by the host using the control transfer mechanism. Prior to configuration, a device responds to its default address of zero. This permits configuration software to request the contents of any device’s descriptors (from endpoint zero) using device address zero during configuration.

Additionally, devices may also support class-specific requests. Except for hub devices, these class requests are defined by the device-class specifications and are not included within the main portion of the specification. Similarly, a device may support vendor-specific requests that pertain to a given vendor’s implementation.

Control transfers, used to transmit device requests, consist minimally of a setup stage and a status stage, but may also include a data stage, depending on the type of request being performed. The setup stage consists of setup transactions as illustrated in Figure A-1. The eight byte data payload of a setup transaction defines the type of request being issued by the host. This appendix discusses only the standard device requests. Refer to the particular device class chapter for information related to class-specific requests.

Figure A-1: Format of Setup Transaction that Specifies the Device Request Being Performed



Standard Device Requests

When a control transfer is initiated, the setup stage of the transaction specifies the particular request to be performed by the device. The format of the setup data is shown in Table A-1, while Table A-2 on page 438 defines the contents of the setup data for each of the standard request types.

Note that in Figure A-1 bits 5 and 6 of offset zero are 00b indicating that the specific request is a standard request type. The second byte (the request field) defines which standard request is to be performed, while the definitions of the other fields are dependent upon the request specified. For example, the “Clear Feature” request defines the “value” field as the feature selector. Figure A-3 lists the specific features that the request applies to. The following sections describe each of the standard requests.

Note that the last column in Table A-2, labeled “Data,” indicates whether the request requires a data stage during the control transfer. For example, the “Get Configuration” request uses the data stage to transfer the configuration value. Many requests, however, can be performed without a data stage.

Table A-1: Format of Data Payload during Setup Transactions

Offset	Field	Size	Value	Description
0	Request-Type	1	Bit-map	Characteristics of Request D7 Data xfer direction 0 = Host to device 1 = Device to host D6:5 Type (h) 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4:0 Recipient (h) 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4-31 = Reserved
1	Request	1	Value	Specific Request.
2	Value	2	Value	Word-sized field that varies according to request.
4	Index	2	Index or Offset	Word-sized field that varies according to request. Typically used to pass an index or offset.
6	Length	2	Count	Number of bytes to transfer if there is a data stage required for this transfer.

If the request contains fields with illegal values or values not supported, the endpoint to which the request is directed will automatically enter the stalled state. Host software must clear the stall condition using the “Clear Stall” request. A control endpoint must continue accepting the setup transaction, even if it is stalled. If the default control endpoint fails to respond to a setup transaction, the device must be reset to clear the condition.

Table A-2: Standard Device Requests

Request-Type	Request	value (2 bytes)	index (2 bytes)	length (2 bytes)	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE (01h)	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION (08h)	Zero	Zero	One	Configura-tion Value
10000000B	GET_DESCRIPTOR (06h)	Descriptor Type and Descriptor Index	Zero or Language ID	Descrip-tor Length	Descriptor
100000001B	GET_INTERFACE (10h)	Zero	Interface	One	Alternate Interface
100000000B 100000001B 100000010B	GET_STATUS (00h)	Zero	Zero Interface Endpoint	Two	Device Interface, or Endpoint Status
00000000B	SET_ADDRESS (05h)	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION (09h)	Configura-tion Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR (07h)	Descriptor Type and Descriptor Index	Zero or Language ID	Descrip-tor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE (03h)	Feature Selector	Zero Interface Endpoint	Zero	None
00000001B	SET_INTERFACE (11h)	Alternate Setting	Interface	Zero	None
10000010B	SYNC_FRAME (12h)	Zero	Endpoint	Two	Frame Number

Set/Clear Feature

The “Set” and “Clear Feature” requests provide a method of enabling and disabling a set of features defined by the feature selector value. Two features are defined for the standard device requests as shown inTable A-3.

Table A-3: Feature Selectors

Feature Selector	Recipient	Value
DEVICE_REMOTE_WAKEUP	device	1
ENDPOINT_STALL	endpoint	0

Device Remote Wakeup

Some devices may be designed to wake the system in the event of a global suspend or to wake a hub port that has been selectively suspended. (See Chapter 9 for details regarding suspend.) The “Set Feature” request, with device remote wakeup selected, enables a device to signal wakeup to the hub. The “Clear Device Remote Wakeup” request prevents a device from signaling remote wakeup to the hub. Whether a device’s ability to signal remote wakeup is currently enabled or disabled is reported to software via the “Get Status” request.

Endpoint Stall

Software has the ability to stall a given endpoint or to clear a stall condition. The “Set” and “Clear Endpoint Stall” requests define which endpoint within the device is being targeted via the “index” field of the setup transaction. A stall bit is defined for each endpoint that indicates whether the endpoint is currently stalled or not. The stall bit is read in conjunction with the “Get Status” request.

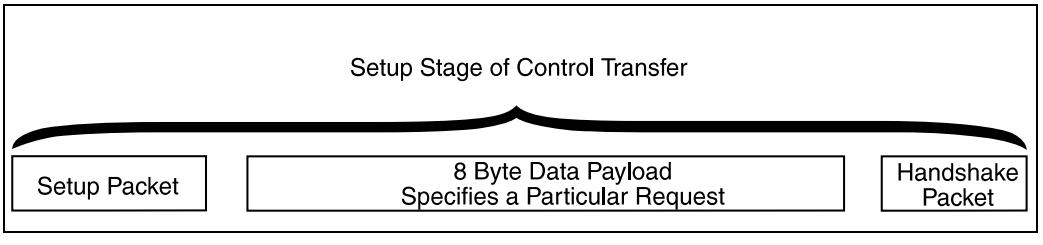
Appendix B: Hub Requests

Overview

Hubs must respond to a variety of USB device requests or commands. Standard requests are used for configuring the hub, for controlling the state of its USB interface, and other miscellaneous features. Hubs must also support class-specific requests that are used to control specific hub and port features. All requests are issued by the host using the control transfer mechanism. Prior to configuration, a device responds to its default address of zero. This permits configuration software to request the contents of any device's descriptors (from endpoint zero) using device address zero during configuration.

Control transfers, used to transmit device requests, consist minimally of a setup stage and a status stage, but may also include a data stage, depending on the type of request being performed. The setup stage consists of setup transactions as illustrated in Figure B-1. The eight byte data payload of a setup transaction defines the type of request being issued by the host.

Figure B-1: Format of Setup Transaction That Specifies the Device Request Being Performed



Hub Request Types

The eight byte data packet that defines the type of request being made is illustrated in Table B-1. Byte zero of the data packet contains a bitmap that defines:

- Direction of data transfer
- Type of request
- Recipient of request

Byte zero of the data packet consists of a bit-mapped value that identifies the packet type for a standard request. That is, if bits 5 and 6 are both zero, then the request is one of the standard requests listed in Table B-2, and a value of 01b specifies that the request is hub-specific. The hub-specific requests are listed in Table B-4.

Table B-1: Format of Setup Transaction Data Phase

Offset	Field	Size	Value	Description
0	Request-Type	1	Bitmap	Characteristics of Request D7 Data xfer direction 0 = Host to device 1 = Device to host D6:5 Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4:0 Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4-31 = Reserved
1	Request	1	Value	Specific Request.
2	Value	2	Value	Word-sized field that varies according to request.

Table B-1: Format of Setup Transaction Data Phase

Offset	Field	Size	Value	Description
4	Index	2	Index or Offset	Word-sized field that varies according to request. Typically used to pass an index or offset.
6	Length	2	Count	Number of bytes to transfer if there is a data stage required for this transfer.

Standard Requests and Hub Response

Hubs must support standard device requests like any other USB device. Table B-2 lists the standard requests and the hub's responses to these requests.

Table B-2: Hub's Response to Standard Device Requests

Request	Request Field Value	Hub Response
CLEAR_FEATURE	1	Clears the selected feature within the device
GET_CONFIGURATION	8	Returns the configuration value used to configure the device.
GET_DESCRIPTOR	6	Returns the selected descriptor(s).
GET_INTERFACE	10	Optional (hubs only required to support one interface).
GET_STATUS	0	Returns status information regarding the state of the device.
SET_ADDRESS	5	Used to assign a unique address to the device.
SET_CONFIGURATION	9	Used to configure a device by assigning the configuration value of the selected configuration descriptor.

Table B-2: Hub's Response to Standard Device Requests

Request	Request Field Value	Hub Response
SET_DESCRIPTOR	7	Optional (used to update or modify a selected descriptor).
SET_FEATURE	3	Sets the selected feature associated with the device.
SET_INTERFACE	11	Optional (hubs only required to support one interface).
SYNCH_FRAME	12	Optional (hubs are not required to have isochronous endpoints).

Hub Class Requests

Hubs must also support specific class requests. When the request type field (bits 5:4) in Table B-1 is set to 01b the request is interpreted as being class specific. The hub class requests are listed in Table B-3.

Table B-3: Hub Class Request Codes

Request	Value
GET_STATUS	0
CLEAR_FEATURE	1
Reserved (GET_STATE in 1.x)	2
SET_FEATURE	3
reserved	4-5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
CLEAR_TT_BUFFER	**8**
RESET_TT	**9**

Table B-3: Hub Class Request Codes

Request	Value
GET_TT_STATE	**10**
STOP_TT	**11**

The format and definition of each hub class request is shown in Table B-4. Only a hub device supports these specific requests. Each of the requests is detailed in the following sections.

Table B-4: Hub Class-Specific Requests

Request-Type	Request	Value	Index	Length	Data
00100000B	CLEAR_FEATURE (01)	Feature Selector	Zero	Zero	None
00100011B	CLEAR_FEATURE (01)	Feature Selector	Port	Zero	None
00100011B	CLEAR_TT_BUFFER (08)	Dev_Addr, EP Number	TT_port	Zero	None
10100011B	RESERVED (2.0) Get_Bus-State (1.x) (02)	Zero	Port	One	Per Port Bus State
10100000B	GET_DESCRIPTOR (06)	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
10100000B	GET_STATUS (00)	Zero	Zero	Four	Hub Status and Change Indicators
10100011B	GET_STATUS (00)	Zero	Port	Four	Port Status and Change Indicators
00100011B	RESET_TT (09)	Zero	Port	Zero	None

Appendix C: Universal Host Controller

Overview

The Universal Host Controller (UHC) and the Universal Host Controller Driver (UHCD) are responsible for scheduling and executing IRPs forwarded from the USB driver. The UHC also integrates the root hub function that is compliant with the USB hub definition. The root hub integrated into the UHC has two USB ports. The following sections describe the mechanism used by the UHC to schedule and generate transactions via the USB.

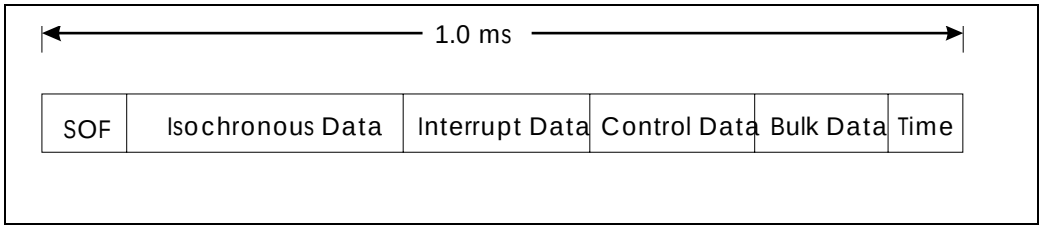
The UHC is integrated into the Intel PIIX3 PCI ISA Expansion Bus Bridge and later chips. It is implemented as a PCI master and is capable of performing transactions to and from memory to fetch and update data structures built by the UHCD.

Universal Host Controller Transaction Scheduling

The sequence of transactions scheduled and performed during each 1ms frame is illustrated in Figure C-1. Note that the periodic transfers are scheduled first (isochronous and interrupt), followed by the control and bulk transfers. The periodic transfers can take up to 90% of the bus bandwidth and control transfers are guaranteed at least 10% of the bandwidth.

The UHCD schedules transactions by building a series of transfer descriptors that are linked to form the collection of transactions that are to be performed during a given frame. This is known as the frame list and is located in system memory.

Figure C-1: Universal Host Controller Transfer Scheduling



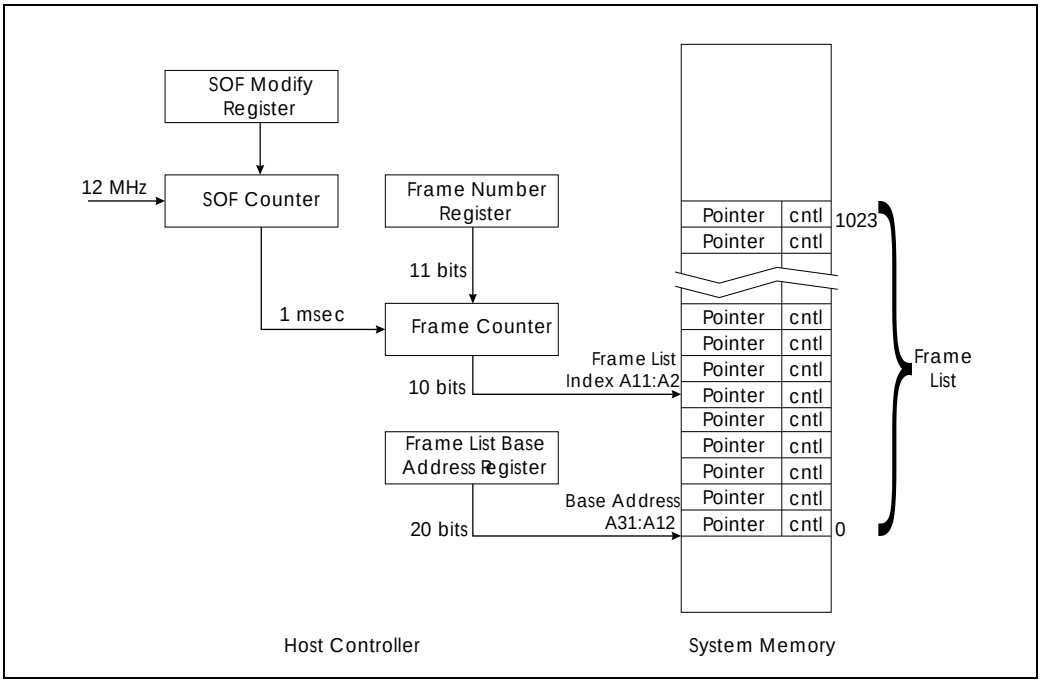
Universal Host Controller Frame List Access

Figure C-2 illustrates the mechanism used by the UHC to access the frame list from memory. The components involved are:

- Start of Frame (SOF) Counter — This counter decrements with each 12MHz clock cycle. When the counter expires, the frame counter is incremented and the next frame begins. This clock is also the source of USB bit timing for transmissions initiated by the root hub.
- The SOF Modify Register — This register can be used to adjust the number of bit times contained in each frame. This changes the interval at which frames are started. The modify register supports the master client feature that allows a single client driver to adjust SOF timing to permit the USB frame rate to synchronize to its isochronous transfer rate. The default value results in 1ms frame generation.
- Frame Counter — The frame counter increments at each frame time to select the next sequential entry in the frame list. Each entry contains a pointer to the first transfer descriptor.
- Frame Number Register — The UHCD programs the start frame number into this register to define the initial entry point within the frame list. This value is loaded into the frame counter and is incremented by the SOF Counter.
- Frame List Base Address Register — The UHCD identifies the base address of the 4KB frame list.

The frame list is an array of up to 1024 entries corresponding to a particular frame. Each entry contains a pointer to a linked list of data structures that contain the information needed by the host controller to build a transaction that will be forwarded to the root hub for transmission over the USB. The UHC reads and interprets each transfer descriptor and generates the transaction described for each descriptor.

Figure C-2: Frame List Access



UHC Transfer Scheduling Mechanism

Figure C-3 illustrates the frame list and the linked list of transfer descriptors that define the transactions to be performed by the UHC. This illustration presumes that many different devices are attached to the USB and that all transfer types are being used by these devices. The order in which the transfer descriptors are linked determines the order that each transaction will be transmitted over the USB. Note that the frame list entry points to transfer descriptors defined for isochronous transfer endpoints. The non-isochronous transfers are queued to permit retries in the event of a failed transaction, whereas isochronous transactions cannot be retried.

Each queue head (QH) and associated transfer descriptor (TD) list is associated with a given transfer type. The first queue is allocated for interrupt transfers, followed by the control transfer queue and finally bulk transfer queues. When all scheduled transactions have completed, the host controller can reclaim the

remaining bus bandwidth by performing additional control and bulk transfers.

Bus Bandwidth Reclamation

Bus bandwidth reclamation can be implemented such that when all scheduled transactions have completed, the UHC can use the remaining frame time to perform additional transactions. This requires that the last QH in the list points back to the beginning of the control and bulk queues. Each QH also has a termination bit that, when set, terminates the transfer sequence, and no reclamation occurs.

The UHC tracks frame timing to monitor the amount of frame time left for scheduling additional transactions. A sample point (called PreSOF point) permits the UHC to determine if there is sufficient time to start the next transaction before the end of packet. The PreSOF point can be selected for 32- or 64-byte packets under software control. If the packet cannot complete in the remaining time, it is not performed.

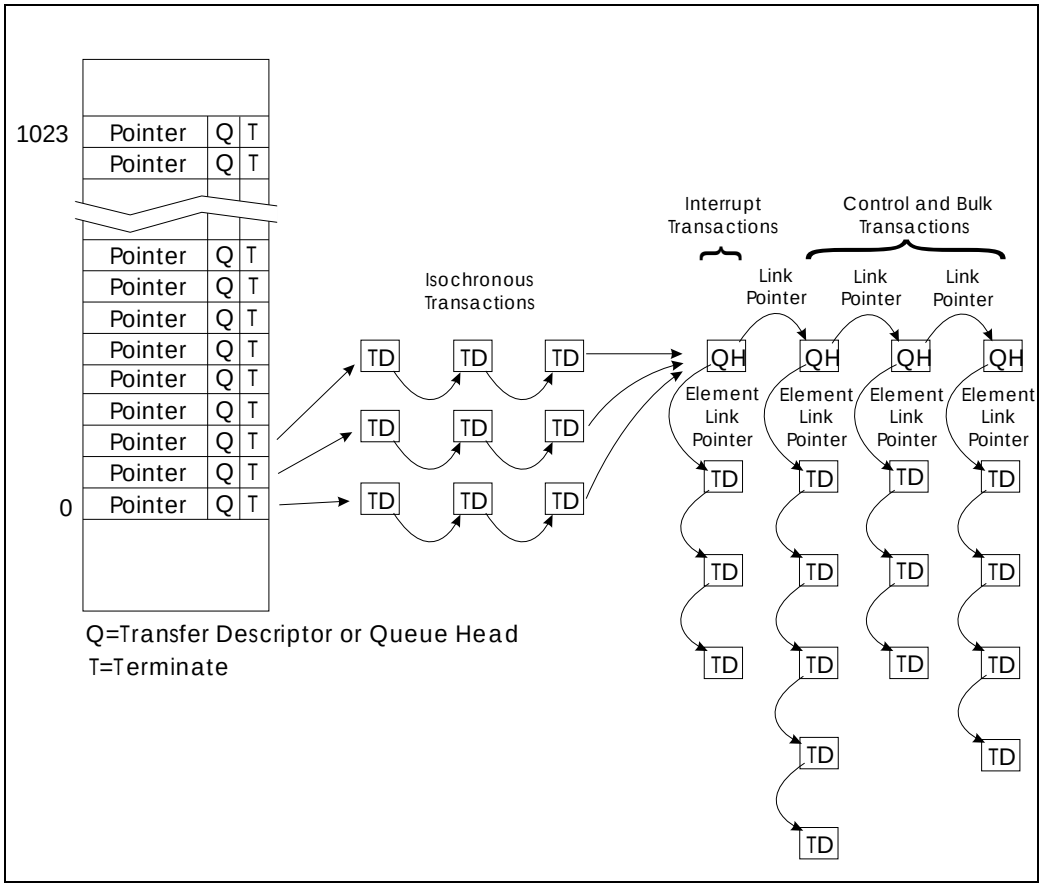
Transfer Descriptors

This section defines the contents of the transfer descriptors and the queue heads. Figure C-4 illustrates the contents of a transfer descriptor. In general, transfer descriptors contain the information needed by the UHC to generate a transaction and report status, including:

- Transfer Type (isochronous and other)
- Type of Token Packet (IN, OUT, SETUP)
- Direction of Transfer
- Size of Data Packet
- Data Toggle Bit
- Memory Buffer Location
- Completion Status

The transfer descriptor consists of four double words (DW0 - DW3). Each DW within the transfer descriptor is defined in tables Table C-1 - Table C-4.

Figure C-3: Transfer Mechanism and Execution Order



Appendix D: Open Host Controller

Overview

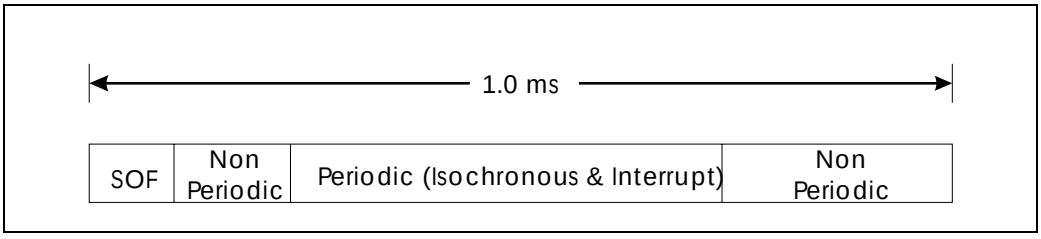
The Open Host Controller (OHC) and the Open Host Controller Driver (OHCD) are responsible for scheduling and executing IRPs forwarded from the USB driver. The OHC also integrates the root hub function that is compliant with the USB hub definition. The root hub integrated into the OHC has two USB ports. The following sections describe the mechanisms used by the OHC and OHCD to schedule and generate transactions via the USB.

Open Host Controller Transfer Scheduling

Figure D-1 on page 478 illustrates the sequence of transfers performed by the Open Host Controller. Note that a reservation can be made at the beginning of the frame (for non-periodic transfers) to support the 10% bandwidth guarantee for control transfers and to ensure that some non-periodic transfers (control and bulk) get performed during each frame. Next, the periodic transfers (interrupt and isochronous) are performed and can take up to 90% of the bus bandwidth; if time remains, then additional non-periodic transfers can be scheduled.

The OHCD schedules transactions by building a series of transfer descriptors that are linked to form the collection of transactions to be performed during a given frame. This is known as the frame list and is located in system memory.

Figure D-1: USB Transfer Scheduling

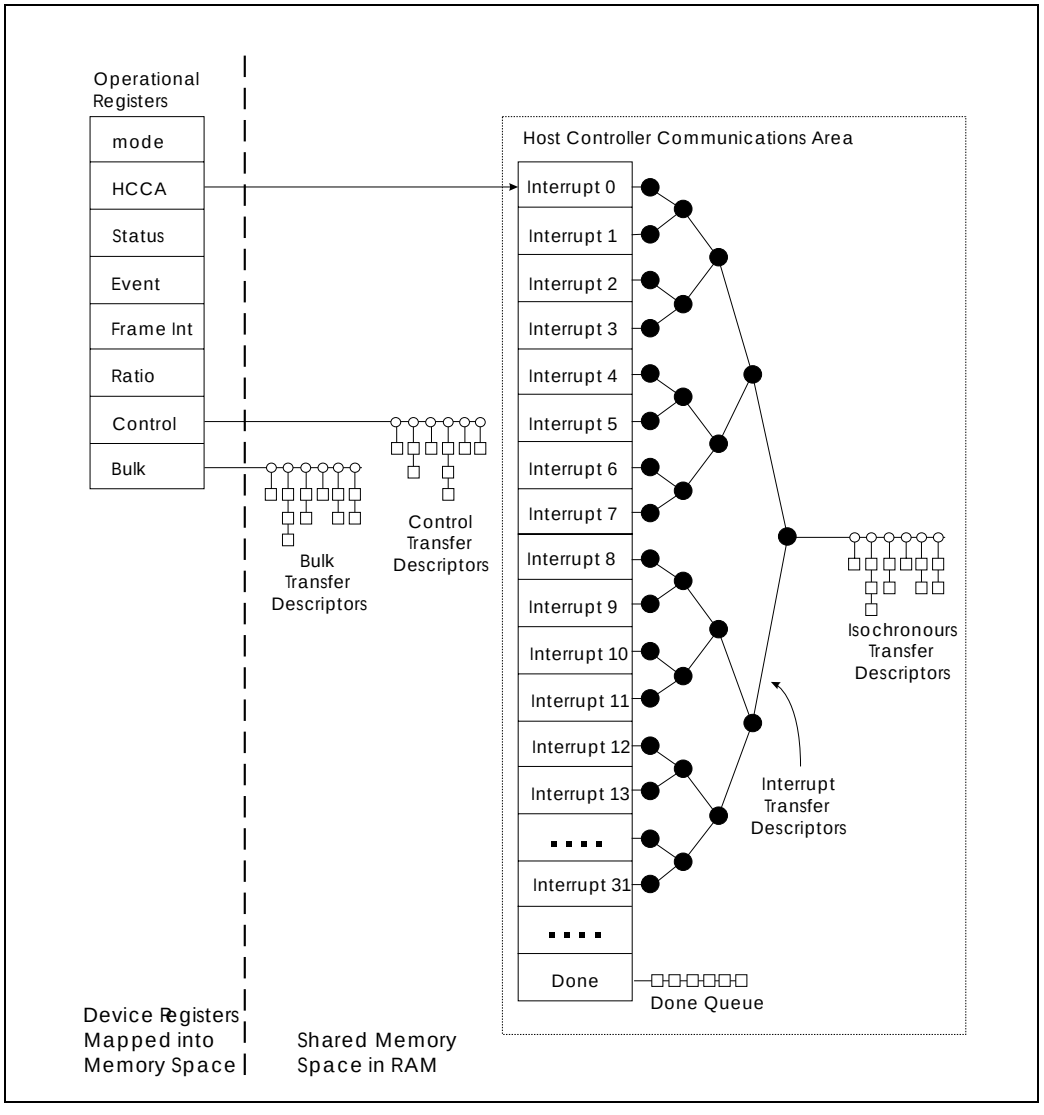


The Open Host Controller Transfer Mechanism

Figure D-2 on page 479 illustrates the mechanism used to generate transactions during each consecutive frame. The OHCD builds descriptors and places them into an area of memory called the host controller communications area (HCCA). These descriptors include Endpoint Descriptors (EDs) and Transfer Descriptors (TDs). The OHCD assigns an ED to each endpoint in the system. The ED contains the address and endpoint number, thus providing information the controller needs to communicate with the endpoint. Each ED is represented as a circle in Figure D-2. A queue of TDs is linked to each ED that represents the transactions that are pending completion for that endpoint.

EDs of a giver transfer type are linked together and pointed to by a register within the controller. Notice that there is a register that points to each of the non-periodic transfer types (control EDs and bulk EDs), and a separate register (HCCA register) that points to the linked list of periodic transfers (interrupt and isochronous) that points to one of 32 entries points for processing interrupts and isochronous transfers.

Figure D-2: The Transfer Scheduling Mechanism



The OHC traverses the ED list in the order that is illustrated in Figure D-1. The controller begins by accessing the control and bulk descriptors where it left off at the end of the previous transfer. After a predetermined interval that is programmed into the controller, it discontinues non-periodic transfers and pro-

ceeds to the periodic transfers via the HCCA register. Note that interrupts are scheduled differently from the other transfer types. The HCCA register increments at the end of each frame to point at the next linked list of interrupt TDs, which are scheduled for completion during the current frame. The end of the interrupt list always links to the beginning of the isochronous EDs. Once the isochronous transfers have completed, the controller is free to continue processing control and bulk transfers where it left off, if sufficient time remains within the current frame.

The OHC links transfer descriptors to the "Done" queue after it has completed successfully or if an error has occurred when performing the TD. The OHCD can then check the done queue for completion status information.

The ED and TD List Structure

All transfers are scheduled using the standard TD list structure as illustrated in Figure D-3. The head pointer is the OHC register for control and bulk transfers. A different interrupt head pointer is selected for each frame as the HCCA register value increments. The interrupt head pointer is located in the HCCA memory area. Isochronous EDs simply link from the last interrupt ED in the current interrupt list. TDs are enqueued to each ED as IRPs are requested by USB device drivers. An endpoint that is currently idle will not have any TDs enqueued.

Interrupt and Isochronous Transfer Processing

Processing the interrupt and isochronous ED list begins with the interrupt head pointer for the current frame. The list is traversed sequentially, until one transaction associated with the first TD has been performed for each endpoint in the list.

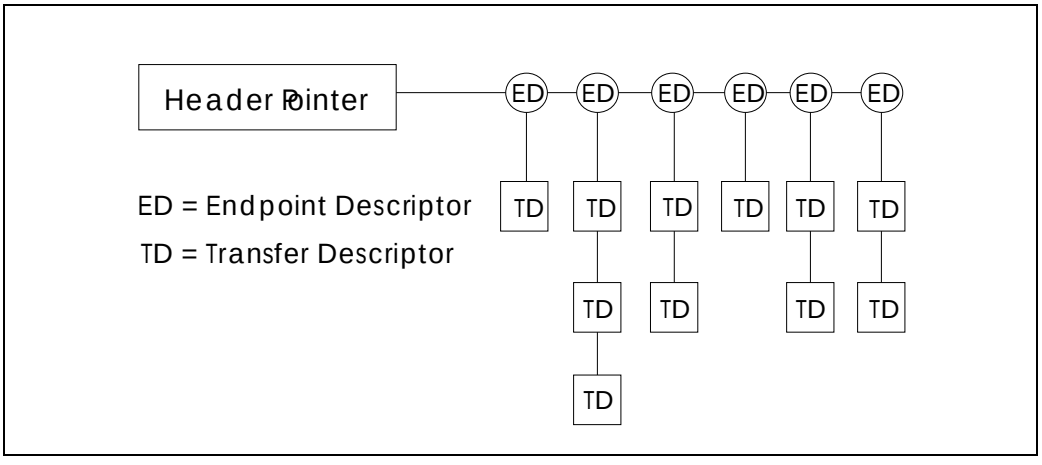
Control and Bulk Transfer Processing

At the beginning of a frame the control and bulk queues are processed until the "remaining" field of the HcFmRemaining register is less than or equal to the "start" field of the HcPeriodicStart register. More control transfers are performed than bulk transfers based on a ratio of control to bulk transfers specified within the "ControlBulkServiceRatio" field of the HcControl register. The controller performs these transfers based on a round robin sequence tied to the ratio (e.g., three control transfer followed by one bulk transfer). If sufficient time remains within the frame after completing the periodic transfers, the controller returns to the control and bulk lists, and continues processing where it left off.

The Done Queue

When the OHC completes a transfer, the TD is linked to the Done queue and written back to the HCCA. In this way, the OHCD can search the Done queue to determine which transactions have been de-queued by the controller and what their completion status is.

Figure D-3: Transfer Queues



Interrupt Transfer Scheduling

Figure D-4 illustrates conceptually how the interrupt EDs are linked to ensure that interrupt transactions occur during the specified polling interval. The HCCA pointer register specifies the base address of the HCCA area in memory where a list of 32 interrupt head pointers reside. The five least significant bits of the frame number are used as an index into the interrupt head pointer list. Each frame a different sequence of interrupt transaction are performed based on the ED links from the selected interrupt head pointer.

The interrupt EDs are linked such that they appear in the list for every nth frame that represents its polling interval. For example, interrupts with a polling interval of 32ms would be linked into only one interrupt head pointer. Since a given head pointer is accessed once every 32 frames the interrupt transaction occurs once every 32ms. At the other extreme, a interrupt endpoint with a polling interval of 1ms would be linked into every interrupt head pointer list. In this way, interrupt transactions can be scheduled at intervals of 1ms, 2ms, 4ms, 8ms, 16ms, or 32ms.

D

I