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.

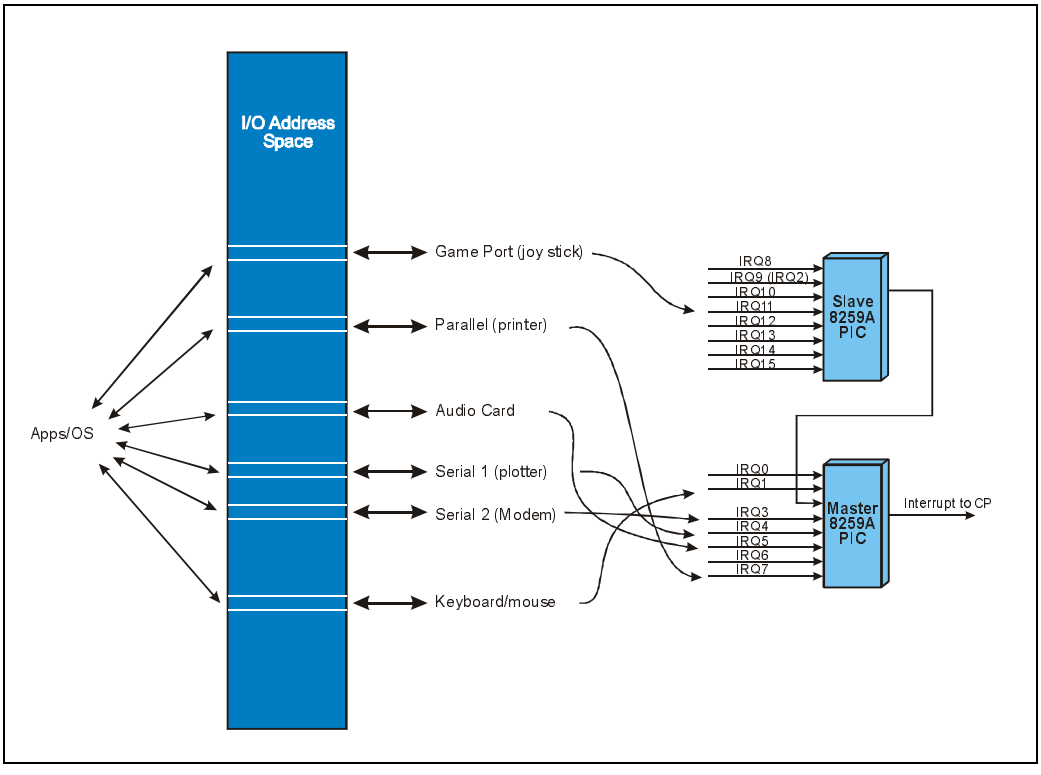
Shortcomings of the Original PC I/O Paradigm

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

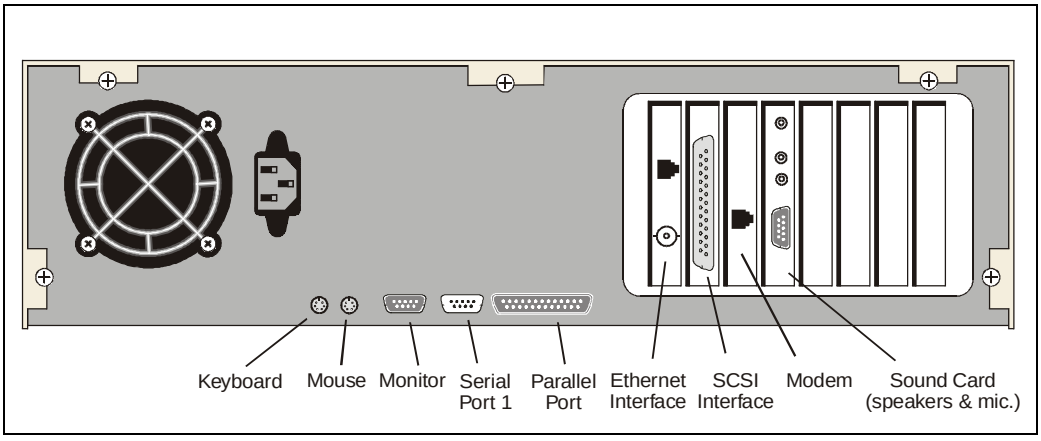
End users are faced with a variety of problems when connecting peripherals to their PCs. These concerns include:

- 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.

Cost

The cost of implementing systems and peripheral devices based on the original PC design is fairly expensive due to the high cost of the standard peripheral connectors and associated cables. Since most of the standard connectors on the PC are already used by a wide variety of peripheral devices, it may be necessary to build an expansion card to provide a way to attach your peripheral device to the system, making the solution even more costly.

The USB Paradigm

The design goals of a new peripheral standard should overcome the existing shortcomings perceived by manufacturers and users, while providing for further growth, performance, and expansion. The design goals of USB include:

- a single connector type to connect any PC peripheral
- ability to attach many peripheral devices to the same connector
- a method of easing the system resource conflicts
- hot plug support
- automatic detection and configuration of peripheral devices
- low-cost solution for both system and peripheral implementations
- enhanced performance capability
- support for attaching new peripheral designs
- support for legacy hardware and software
- low-power implementation

USB breaks away from the resource problems associated with legacy PC I/O implementations. The resource constraints related to I/O address space, IRQ lines, and DMA channels no longer exist with the USB implementation. Each device residing on the USB is assigned an address known only to the USB subsystem and does not consume any system resources. USB supports up to 127 device addresses that limit the number of USB devices supported in a single USB implementation. USB devices typically contain a number of individual registers or ports that can be indirectly accessed by USB device drivers. These registers are known as USB device endpoints.

When a transaction is sent over the USB, all devices (except low-speed devices) will see the transaction. Each transaction begins with a packet transmission that defines the type of transaction being performed along with the USB device and endpoint addresses. This addressing is managed by USB software. Other non-

USB devices and related software within the system are not impacted by these addresses. Every USB device must have an internal default address location (called endpoint zero) that is reserved for configuring the device. Via endpoint zero, USB system software reads standard descriptors from the device. These descriptors provide configuration information necessary for hardware and software initialization. In this manner, system software can detect the device type (or class information) and determine how the device is intended to be accessed.

Enhanced System Performance

The Universal Serial Bus (USB) creates a solution for attaching PC peripherals that balances performance and cost. USB supports three transmission rates:

- 1.5Mb/s
- 12Mb/s
- 480Mb/s

The 1.0 and 1.1 (1.x) versions of USB support only the 1.5 Mb/s and 12Mb/s speeds. These transmission rates were intended to support low- and medium-speed peripherals, while the 2.0 version of the USB specification defines a 480Mb/s rate that can support selected high-speed devices, and permits a larger number of low- or full-speed devices to operate on a single bus. Table 1-2 lists the types of devices that fall into these performance ranges.

Table 1-2: Applications, Relative Performance Required and Desired Attributes

Performance	Applications	Attributes
Low Speed: Interactive Devices 10-100 Kb/s	Keyboard, Mouse Stylus Game peripherals Virtual Reality peripherals	Lower cost Hot plug Ease of use Multiple peripherals
Medium Speed: Phone, Audio 500-10,000 Kb/s	ISDN PBX POTS Digital audio Scanner Printer	Lower cost Ease of use Guaranteed latency Guaranteed bandwidth Hot plug Multiple devices

Table 1-2: Applications, Relative Performance Required and Desired Attributes

Performance	Applications	Attributes
High Speed: Video, Disk, LAN 25-500 Mb/s	Mass storage Video conferencing Imaging Broadband	Low cost Hot plug High bandwidth Guaranteed bandwidth Guaranteed latency Multiple devices Ease of use

Hot Plug and Play Support

Hot Plug and Automatic configuration is crucial to satisfying end user requirements. USB can detect the attachment of a new peripheral and automatically install the relevant software needed to access the device. This process also eliminates the need to set switches and jumpers when configuring a peripheral device and eliminates the need to restart the system when the device is attached. In short, the peripheral can simply be attached by the user and be ready for immediate use.

Expandability

Hub devices provide additional ports for attaching other USB devices as illustrated in Figure 1-3 on page 22. Hubs can be stand-alone devices, or can be integrated into other USB peripherals such as printers or keyboards. Physical USB devices that contain hubs and that have one or more internal devices attached to the hub ports are called compound devices.

Legacy Hardware/Software Support

Older operating systems have no knowledge of USB, so the system designer must choose whether to provide USB support. Additionally, traditional system firmware (initialization code, boot code, and BIOS) is based on standard PC legacy hardware and must be adapted to support USB if USB boot support is desired.

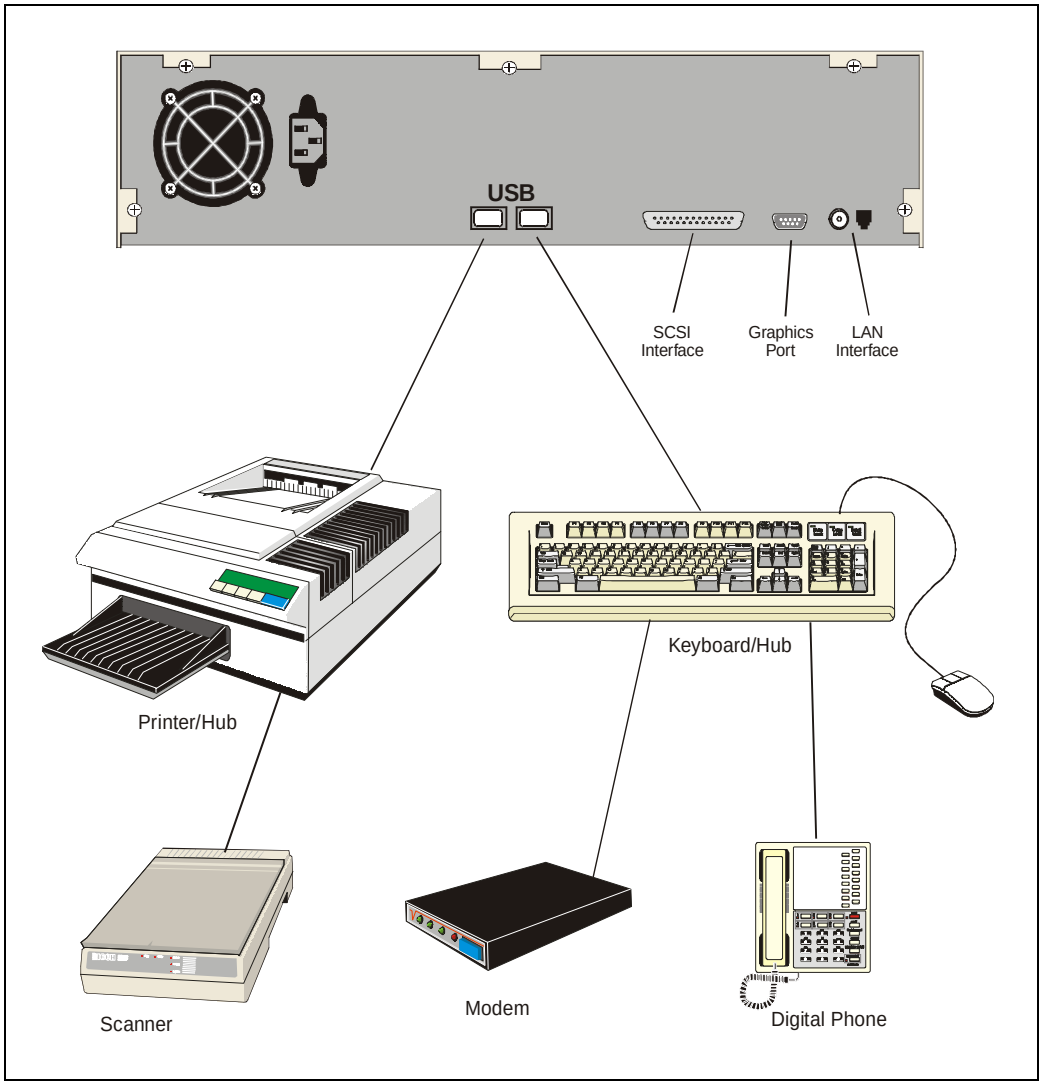
Low Cost

USB can reduce the overall cost of peripheral design and system support.

Much of this cost reduction on the peripheral side comes from the ability to connect the peripheral directly to a USB port, thereby eliminating the requirement to design an expansion card to provide a connection for the device. Another source of cost reduction for both the system and peripheral designers is the connectors and cables. The standard USB cables and connectors create a very large market for these items and competition between vendors reduces their cost. The USB serial bus implementation also reduces cost when compared to parallel bus implementations that require a much larger number of pins and traces.

The system cost savings can be realized by eliminating the cost of the wide variety of connections that must be supported for standard peripherals such as the parallel, serial, keyboard, and mouse connectors. In the short term USB has been added while the older connectors remain. Figure 1-3 on page 22 illustrates the backplane of a system as it may look sometime in the near future.

Figure 1-3: USB Device Connections



Summary of Key USB Features

Table 1-3 lists the key features that comprise the USB implementation.

Table 1-3: Key USB Features

Feature	Description
Low Cost	The USB provides a low-cost solution for attaching peripheral devices to PCs.
Hot Pluggable	Device attachment is automatically detected by the USB and software automatically configures the device for immediate use, without user intervention.
Single Connector Type	The USB defines a single connector used to attach any USB device. Additional connectors can be added with USB hubs.
127 Devices	Supports the attachment of 127 devices per USB.
Low Speed, Full Speed, and High Speed Devices	The USB 2.0 supports three device speeds: 1.5Mb/s, 12Mb/s, and 480Mb/s.
Cable Power	Peripherals can be powered directly from the cable. 5.0vdc power is available from the cable. The current available can vary from 100ma - 500ma depending on the hub port.
System Resource Requirement Eliminated	USB devices, unlike their ISA, EISA, and PCI cousins, require no memory or I/O address space and need no IRQ lines.
Error Detection and Recovery	USB transactions include error detection mechanisms that are used to ensure that data is delivered without error. In the event of errors, transactions can be retried.
Power Conservation	USB devices automatically enter a suspend state after 3ms of no bus activity. During suspend devices can consume no more than 500µa of current.

Table 1-3: Key USB Features

Feature	Description
Support for Four Types of Transfers	The USB defines four different transfer types to support different transfer characteristics required by devices. Transfer types include: bulk, isochronous, interrupt, and control transfers.
Ability to Extend Bus	USB hubs can be installed to add additional ports that permit additional devices to be attached.

How to Get the USB Specifications

The USB specifications are available from the USB web site at:

www.usb.org

This web site has the 2.0 version of the USB specification and the device class specifications, along with other information related to USB.

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

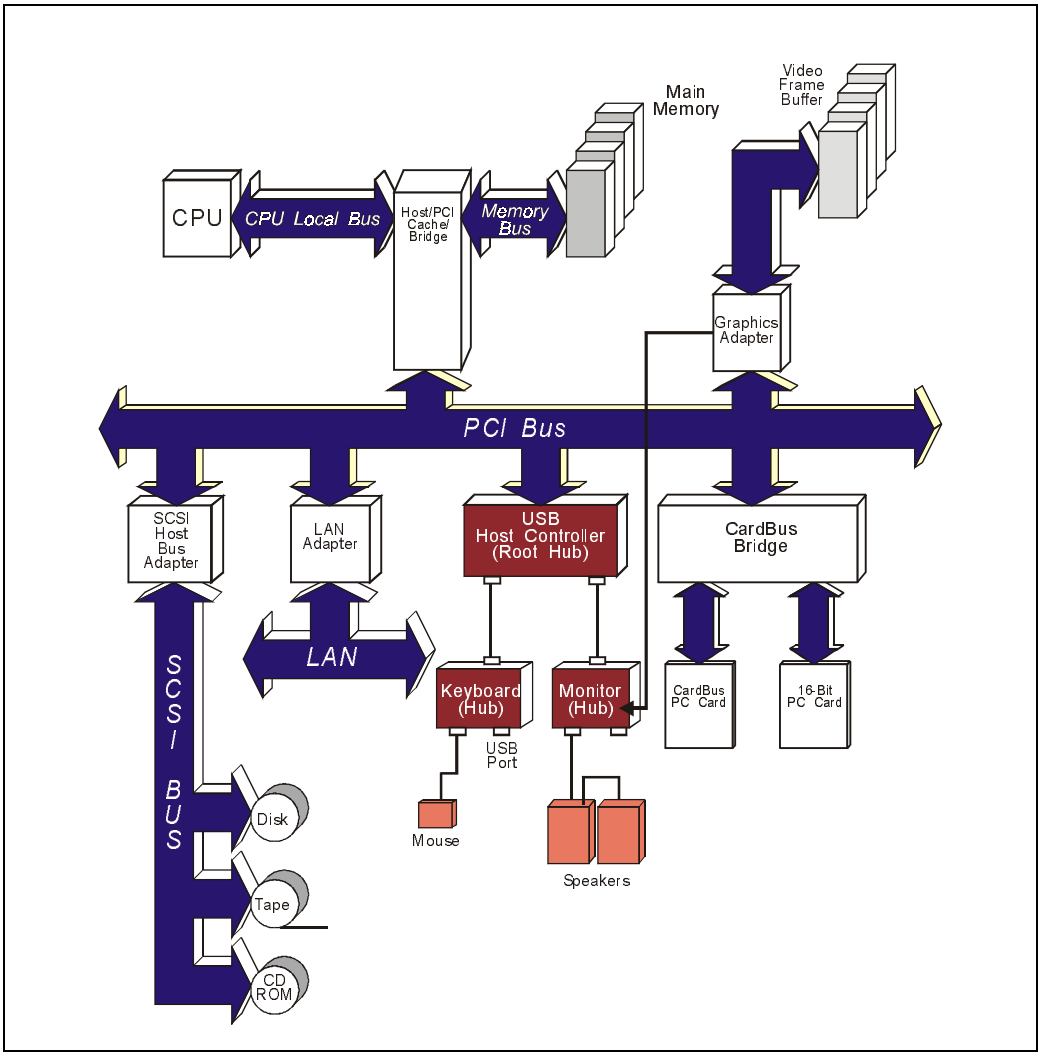
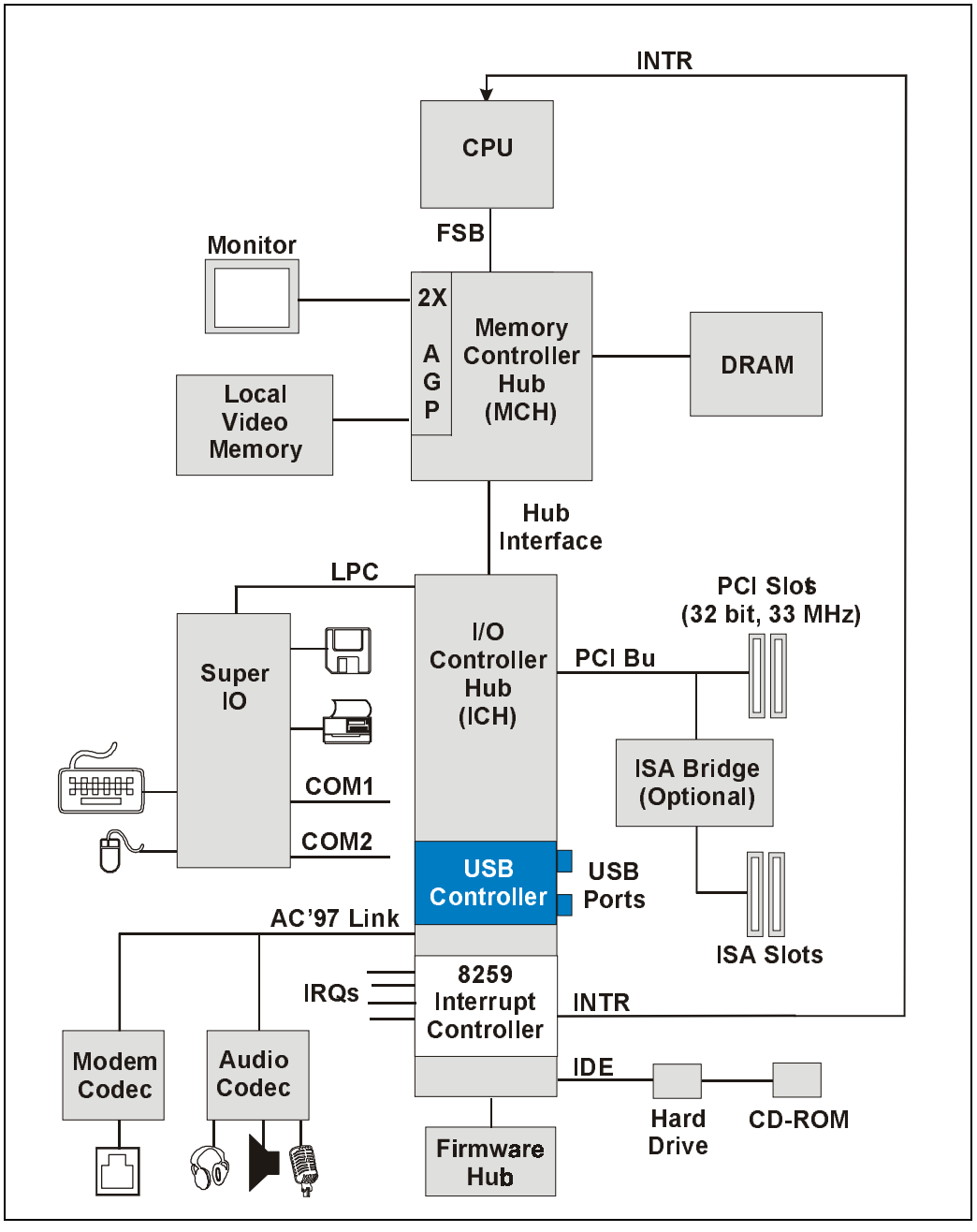


Figure 2-2: USB Controller Integrated into I/O Hub Chip



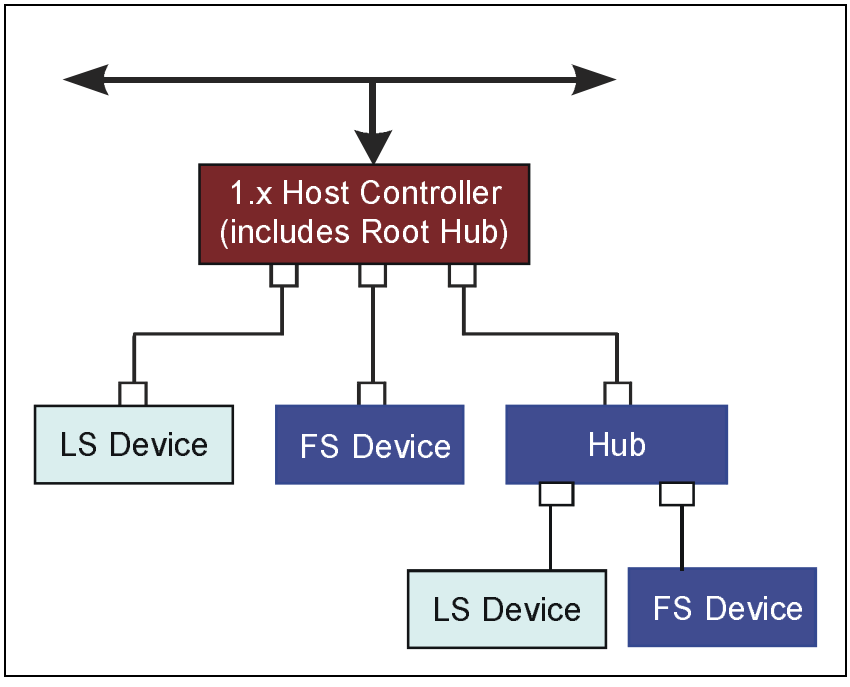
USB 1.x Systems and Devices

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

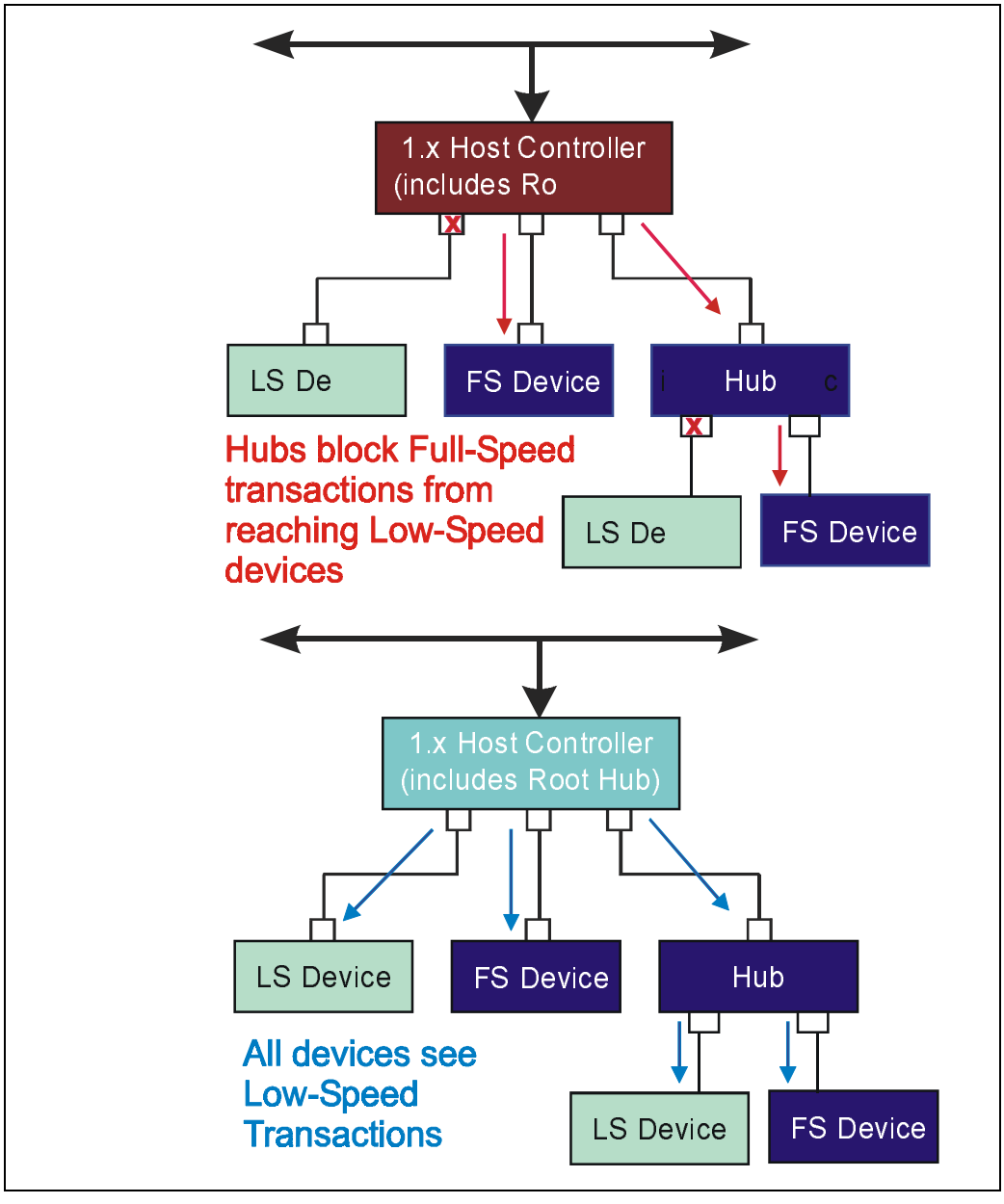
USB 1.0 and 1.1 (i.e., 1.x) systems can support only 1.5Mb/s (low speed) and 12Mb/s (full-speed) transactions as illustrated in Figure 2-3. The host delivers low- or full-speed transactions depending on the speed of the device being accessed.

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



How Transactions Are Generated

USB 1.x system implementations generate USB transactions by fetching and executing a linked list of data structures (called transfer descriptors) from memory. Each transfer descriptor defines a USB transaction that software has requested and scheduled for the purpose of accessing a USB device. For example, one transfer descriptor may specify that a USB keyboard be accessed to check whether a keystroke has occurred, while another descriptor may specify a data transfer to a printer. In this example, the keyboard is a low-speed device and the printer is accessed at full speed.

What the Descriptors Contain

Each transfer descriptor contains information that describes a transaction to be performed. The primary information includes:

- The USB device address
- The type of transaction to be performed (read or write)
- The transfer size
- Speed of the transaction
- The location of the memory data buffer (a full buffer containing data to be sent to the USB device or an empty buffer where data read from the USB device is to be placed)

With this information the USB host controller can perform the specified transaction. In the personal computer arena two host controller interfaces were designed for USB 1.x devices: the Universal Host Controller Interface (UHCI) and Open Host Controller Interface (OHCI). Each accomplishes the same jobs but in different ways, and each has its own transfer descriptor definition. See Figure C-4 on page 470 for details on the transfer descriptor definition for the UHCI host controller or Figure D-6 on page 486 for the OHCI host controller.

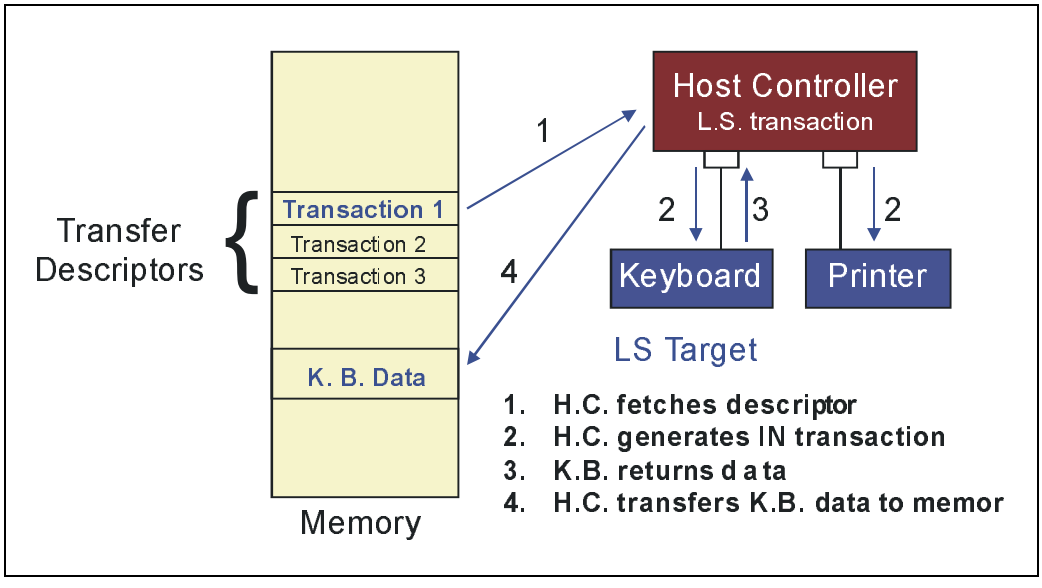
How the Transfer Descriptors Are Fetched

The linked list of descriptors is sometimes called a transaction list or frame list. During a 1ms interval (called a frame), the host fetches and executes a series of descriptors. Figure 2-5 on page 31 and Figure 2-6 on page 32 provide a conceptual view of the steps taken by the host controller when it fetches and executes transactions from the frame list. In these examples, the frame list contains transfer descriptors that access a USB keyboard and a USB printer. Note that these examples do not deal with the entire USB protocol, but rather deal only with the conceptual process of transaction generation across the USB.

The first example depicts a keyboard being polled by software to check if a key has been pressed. The direction of data flow in USB is specified with respect to the host. Since data is being read by the host, the transaction is termed an IN transaction. Figure 2-5 illustrates the sequence of events associated with the IN transaction from the keyboard.

Previously the keyboard's USB driver has requested that the keyboard be polled periodically to determine if the user has pressed a key. The driver also supplies a memory buffer location where the keyboard data is to be returned. This request has resulted in the host software creating a transfer descriptor in memory (Transaction 1 in Figure 2-5) that describes the USB polling operation. The

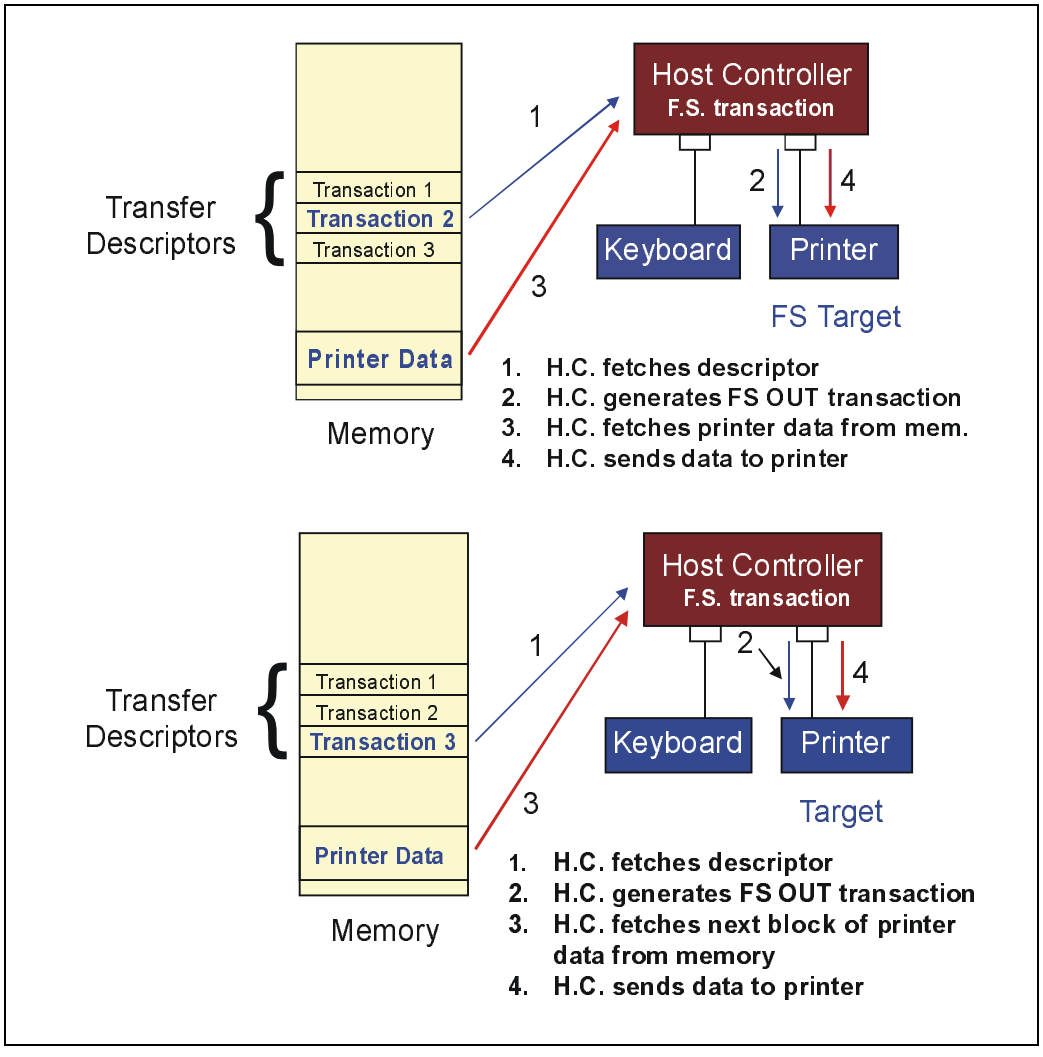
Figure 2-5: Conceptual View of Transaction Generation — Example 1



host controller fetches transaction 1 and decodes the descriptor and executes the requested IN transaction. The keyboard returns data to the host controller, which in turn places the data into the keyboard data area in memory. The pointer to the keyboard data buffer is included within the transfer descriptor. The keyboard software driver reads the keyboard data buffer to acquire the data.

The second and third descriptors define transactions that send data to the USB printer. The direction of data flow in this case is OUT from the host. Figure 2-6 on page 32 illustrates the sequence of events when executing these transactions.

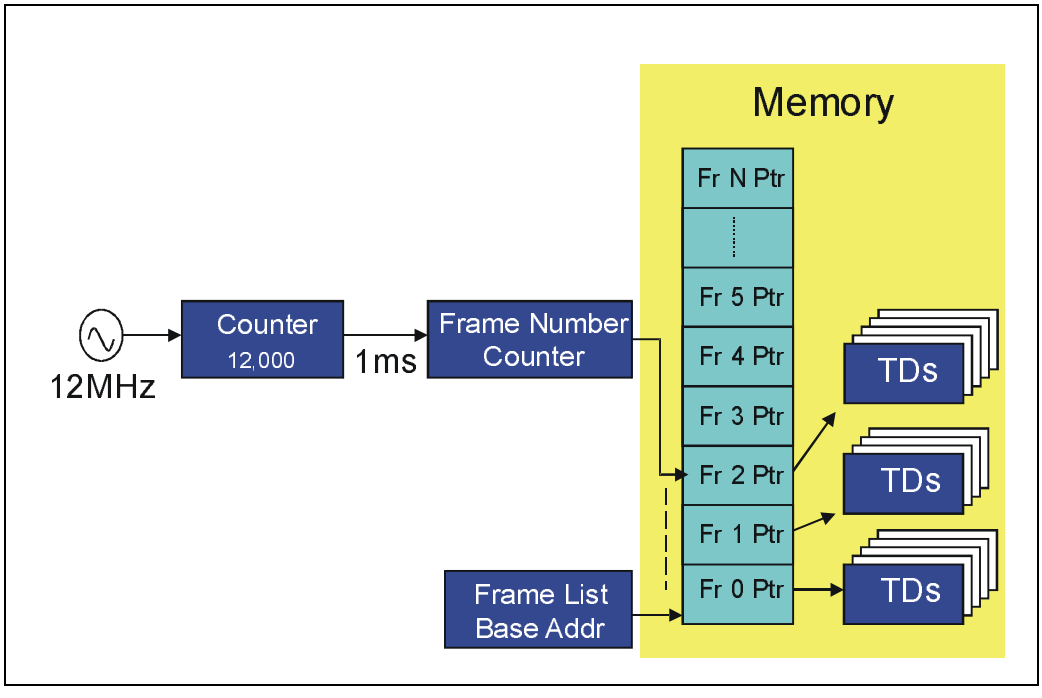
Figure 2-6: Conceptual View of Transaction Generation — Example 2



Frame Generation

Figure 2-7 illustrates how the controller fetches each frame list on 1ms intervals. Note that each 1ms frame consists of 12,000 bit times at the 12Mb/s bit rate, during which transactions are performed. A 12MHz clock increments a counter that generates a carry output when the count reaches 12,000, thereby creating a 1kHz clock (1ms periods). The carry output increments another counter containing a frame number that is used as an address location to fetch the first transfer descriptor in a linked list of descriptors. Each transfer descriptor contains a link address to the next descriptor in the list. In this way, each descriptor in the list is fetched and executed, resulting in a series of USB transactions during the current frame.

Figure 2-7: Conceptual View of 1ms Frame Generation



Sharing the Bus

The collection of devices residing on the bus must share the bus bandwidth. Figure 2-8 on page 35 depicts a single frame during which each USB device is getting a portion of the bus bandwidth. Note that some devices require bus access every frame while others may require use of the bus on a periodic basis. To avoid possible confusion, note that devices are accessed only when client software has requested data transfer to or from a given device.

This example illustrates every device being accessed in the same frame — not a likely circumstance in this case. Also, some devices require USB bandwidth every frame, thus requiring isochronous transactions (e.g., USB speakers). Other devices may require the transfer of large blocks of data but at no particular time, so their use of the bus is asynchronous in nature and does not require guaranteed bandwidth (e.g., USB printers). When an application requires large amounts of USB bandwidth every frame, little or no bandwidth may be left for devices such as printers. In such cases the transfer of data to a printer may slow or even stop temporarily, until an application performing isochronous transactions terminates.

Bandwidth Consideration Summary

The theoretical bandwidth available during each 1ms interval is 12,000 bits / ms, or 1.5KB / ms (1.5MB / s). However, overhead associated with performing transactions significantly reduces the efficiency of the bus. Consider the typical overhead associated with different types of transfers (including worst-case propagation delay):

- Isochronous transactions = 9 bytes
- Interrupt transactions = 13 bytes (FS) and 19 bytes (LS)
- Bulk transactions = 13 bytes
- Control (3 stage transfer) = 45 bytes (FS) and 63 bytes (LS)

To promote fairness during bandwidth sharing, the specification defines maximum packet sizes for various types of transfers. In general, isochronous transfers can have a maximum data payload of 1023 bytes and all others have a maximum payload of 64 bytes. Bus efficiency when performing transfers with various packet sizes is listed in Table 2-1 on page 36.

Figure 2-8: Example of USB Devices That Share Bus Bandwidth

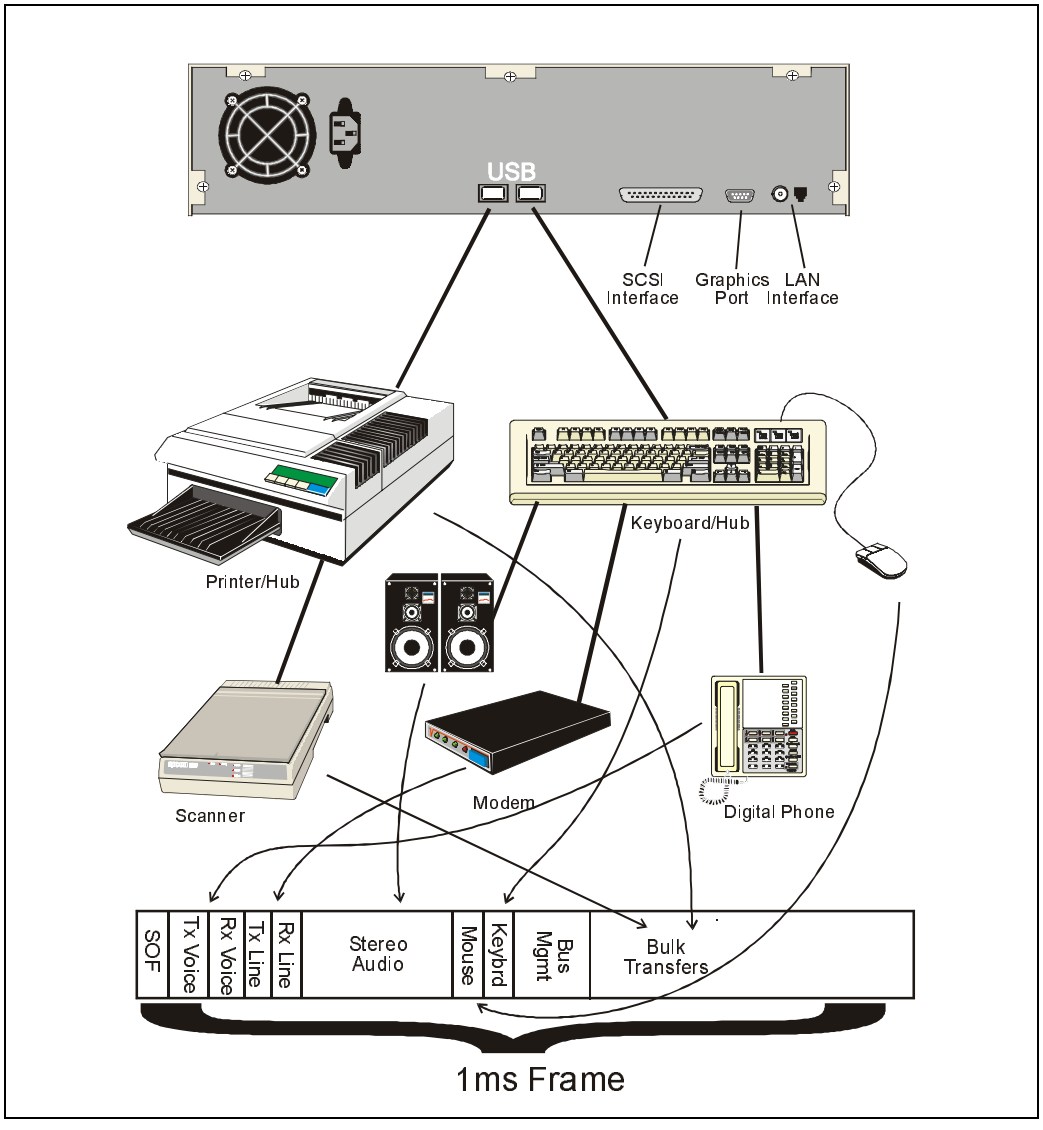


Table 2-1: Approximate Bus Efficiency of Transactions with Various Data Payloads

Transfer Type	Max. Packet Size	Efficiency
Isochronous	1023 bytes	~99%
	512 bytes	~98%
	64 bytes	~86%
Other	64 bytes	~82%
	32 bytes	~69%
	8 bytes	~36%

Another important aspect of USB performance is the available bandwidth relative to the maximum data packet size. For example, while the bus efficiency is high for an isochronous transaction with a maximum payload of 1023 bytes, this transfer takes roughly 87% of the overall bus bandwidth. In contrast, a single bulk transaction with a maximum data payload of 64 bytes takes just over 5% of the available bandwidth. Thus, when a maximum bandwidth isochronous transaction is running, the remaining bandwidth permits just two more maximum-sized bulk transfers. Now imagine a scenario such as the one illustrated in Figure 2-8 on page 35. In this example, the bandwidth available may not be sufficient to support even the isochronous devices, without regard to the other devices requiring bus bandwidth.

The USB specification permits up to 90% of the overall bandwidth to be allocated to periodic transactions (isochronous and interrupt), while control transfers have a guaranteed reservation for up to 10% of the overall bandwidth. Bulk transfers simply get the bandwidth that is left over after all of the currently scheduled transactions complete. Considering the bandwidth limitations, the number of devices that can be supported adequately by USB 1.x is much lower than might be expected.

2.0 Systems and Devices

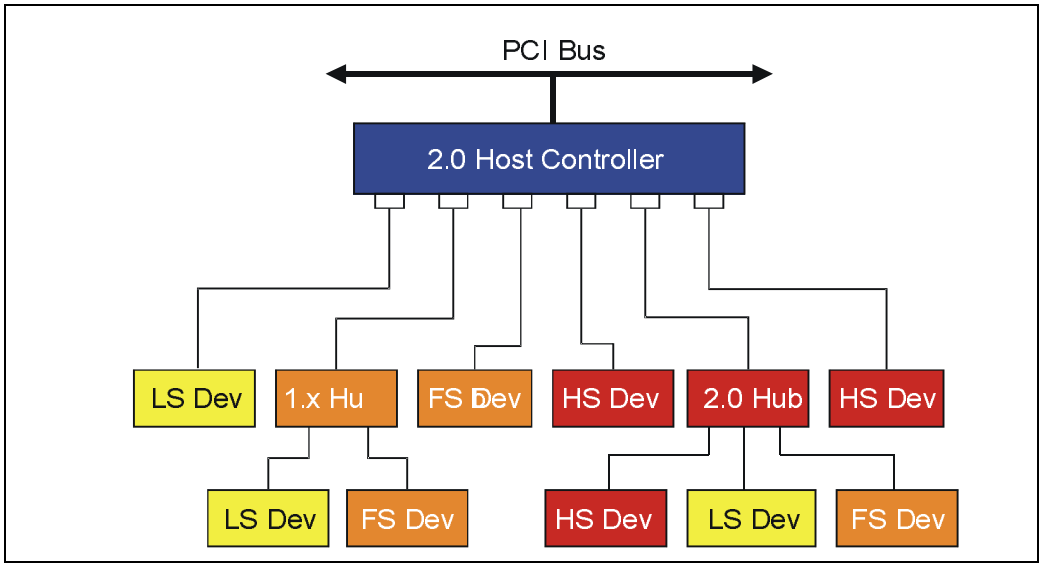
Systems based on USB 2.0 are designed to support high-speed, full-speed, and low-speed devices. This provides backward compatibility to 1.x devices, while significantly extending USB performance, and consequently, increasing the number of devices that can be supported.

USB 2.0 is backward compatible with 1.x devices and has many of the same characteristics:

- uses the same connectors
- uses FS cables for HS devices
- employs the same communications model (token / data / handshake)
- uses the same device attachment recognition
- uses the same device configuration model

The 480Mb/s transfer rate of USB 2.0 is 40 times faster than the 12Mb/s transfers of USB 1.x. The faster transfers are intended to permit a greater number of USB devices on a single bus. Additionally, both 1.x devices in HS system low-speed (LS) devices and full-speed (FS) devices are supported without significantly impacting the performance of high-speed (HS) devices. Figure 2-9 on page 37 illustrates a USB 2.0 system with devices attached to a variety of ports.

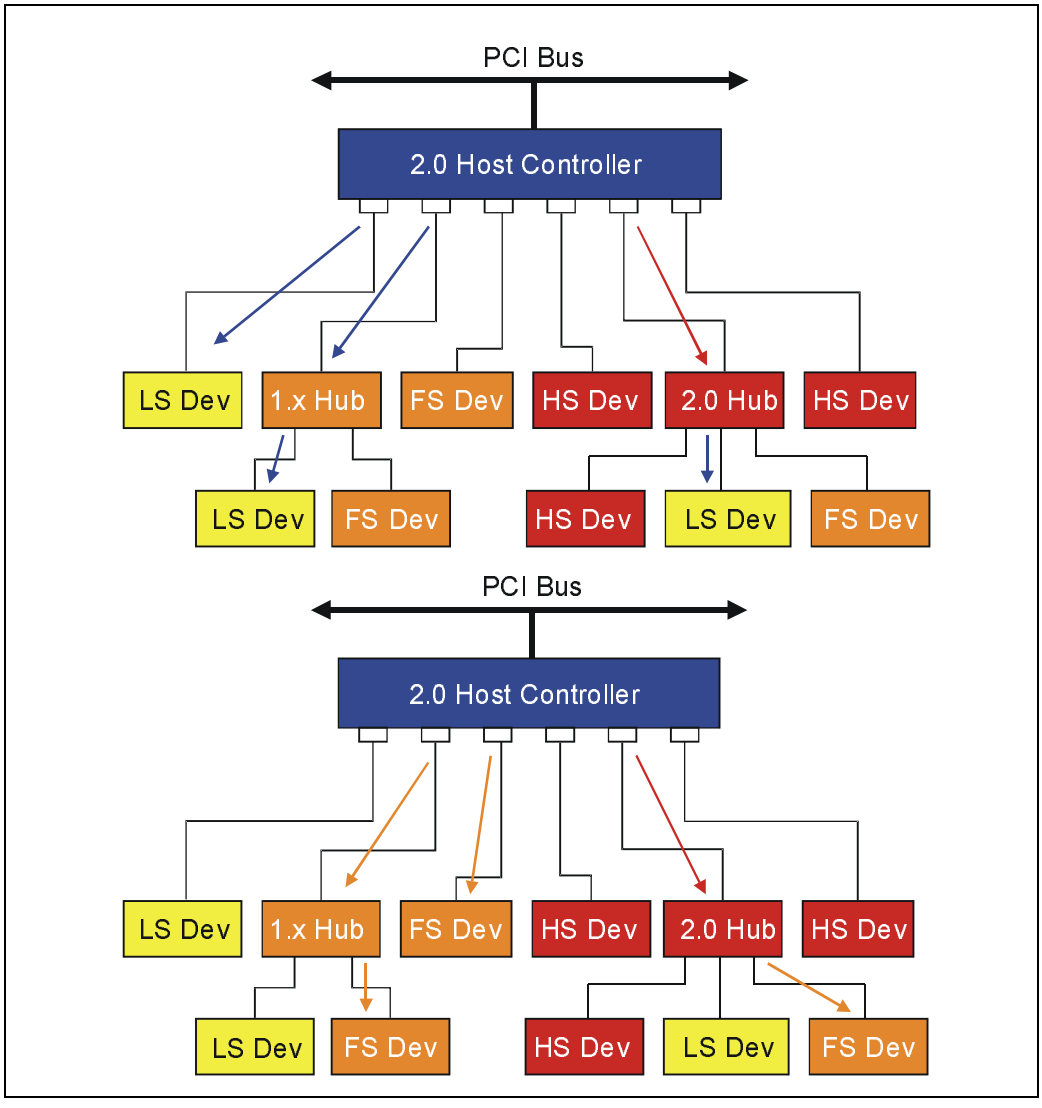
Figure 2-9: USB 2.0 System with Low-, Full-, and High-Speed Devices Attached



Low-Speed and Full-Speed Devices in a 2.0 System

Low-speed and full-speed devices may also be attached to 1.x hub ports or 2.0 HS hub ports. Figure 2-10 on page 38 illustrates the ways in which these devices may be attached to the bus.

Figure 2-10: Low- and Full-Speed Devices Attached to Ports of the Root, 1.x, and 2.0 Hubs

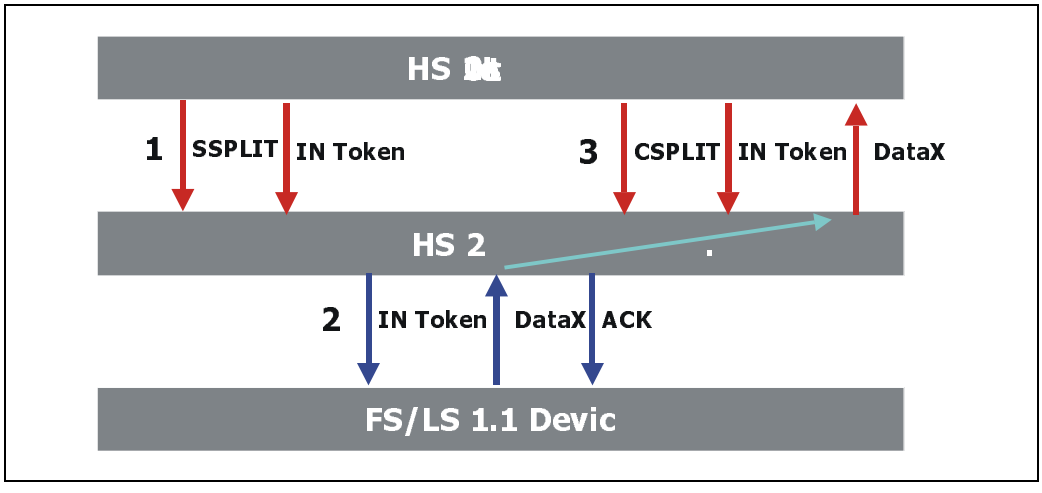


When LS/FS devices are attached to FS hubs with no HS connection between the host and the FS hub, the devices operate just as they did in 1.x systems. However, when a LS/FS device is attached to a HS port (not a root port), split transactions are used to access the device. A split transaction sequence consists of three primary steps:

1. The host delivers a HS Start Split transaction to the high-speed hub. This transaction contains the LS/FS token packet and data if the transaction is an OUT to the device.
2. The hub performs the LS/FS transaction to the device and saves completion status (data for IN transactions or handshake results for OUT transactions). During this time the host can transfer information to other devices on the bus.
3. When the host knows that the LS/FS transaction has had time to complete, it delivers a HS Complete Split transaction to the hub to obtain the LS/FS transaction results. The transaction contains the same token packet as delivered in the Start Split transaction. The hub uses the token to match the correct transaction in the event that multiple split transactions are pending completion. The hub then returns either data (IN transaction) or a handshake (OUT transaction) to verify the results of the transaction.

Figure 2-11 illustrates a split IN transaction sequence that illustrates the three stages described previously. For details regarding split transactions see “The Structure of Split Transactions” on page 290.

Figure 2-11: Split INTransaction Sequence



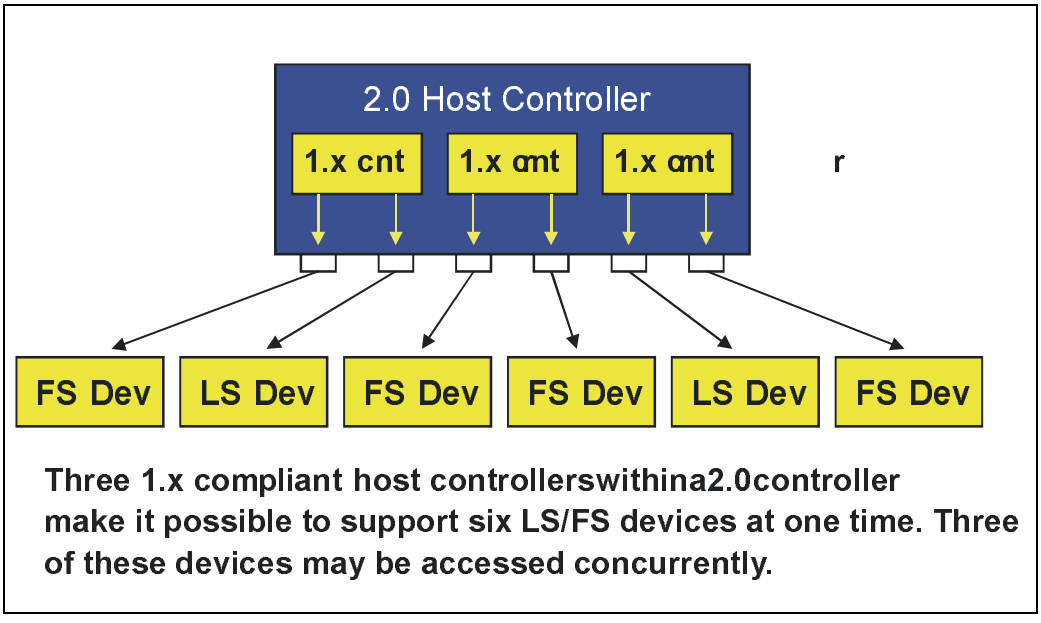
Example 2.0 Host Controller Support for LS/FS Devices

A 2.0 host controller can be implemented in a variety of ways to support LS and FS devices. The host controller must support low- and full-speed devices attached to any root hub port. Figure 2-12 illustrates a possible 2.0 host controller implementation that incorporates three 1.x controllers. Thus, any low- or full-speed device attached to a root port can be accessed via the 1.x controllers.

In this implementation the 2.0 host controller must monitor each port to detect the speed of the device connected to the root hub. If a LS or FS device is connected, the 2.0 host controller must then connect the port to one of the 1.x controllers.

An advantage of this type of solution is that each of the three controllers independently fetches and executes its own frame list. This makes it possible to have three concurrent accesses to LS / FS devices.

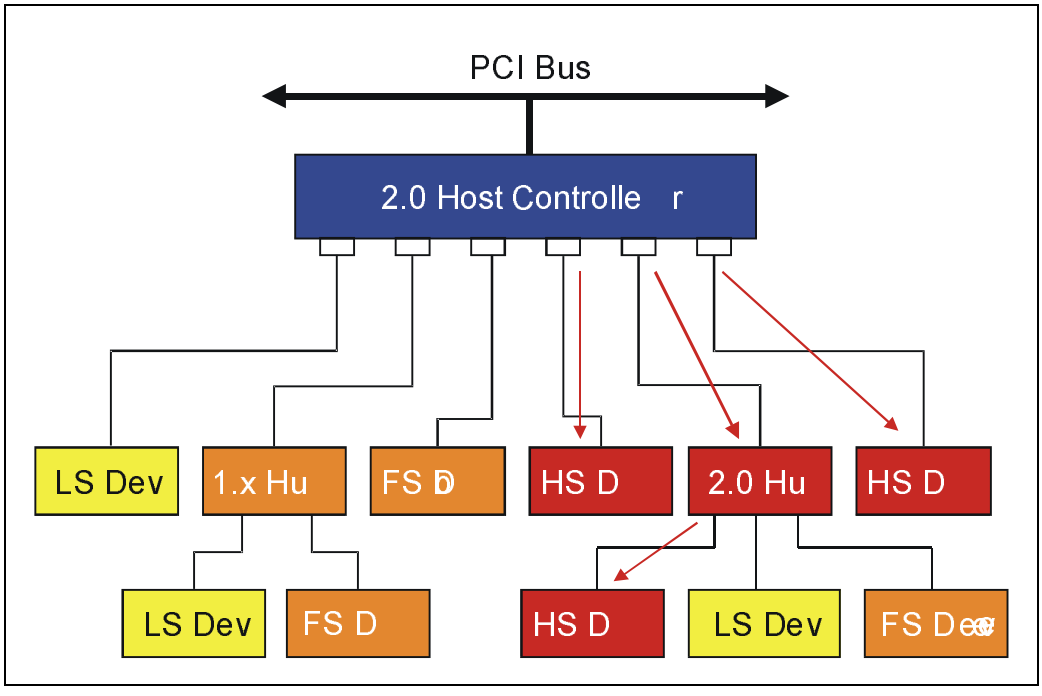
Figure 2-12: Example 2.0 Controller with Three 1.x Host Controllers Used for Low- and Full-Speed Support



High-Speed Devices in a 2.0 System

High-speed devices like low- and full-speed devices may also be attached to any 2.0 host controller port or 2.0 hub operating at high speed. Figure 2-13 illustrates a collection of high-speed devices attached to high-speed ports. Transactions generated by a high-speed (HS) host controller are repeated to all HS devices, while these transactions are blocked from reaching full- and low-speed devices.

Figure 2-13: Example of High-Speed Devices Attached to 2.0 Root Hub and High-Speed Hub



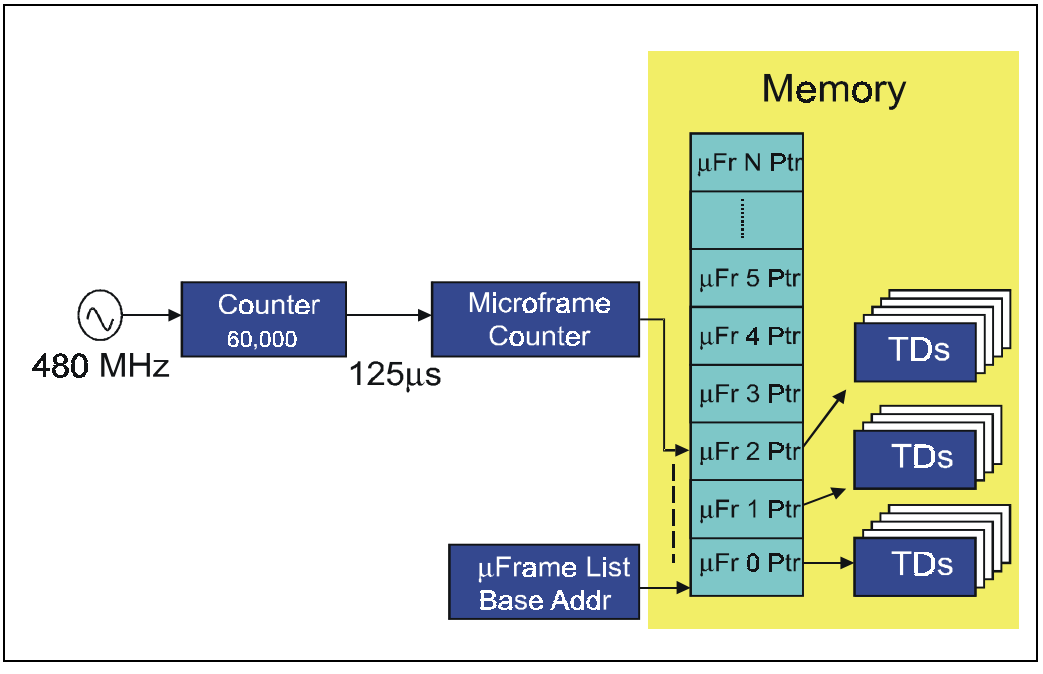
High-Speed Devices Attached to 1.x Ports

High-speed devices that connect to full-speed ports must operate correctly at full speed. Note that even though the device must operate at full speed it may be limited to supporting only accesses via endpoint zero (e.g., reading its descriptors). Thus, a HS device is not required to have full functionality at full speed.

High-Speed Transactions and Microframe Generation

The method of generating accesses to high-speed transactions is conceptually the same as for 1.x systems except transactions are scheduled and performed during 125µs intervals called microframes. Figure 2-14 illustrates the generation of microframes and how the host controller fetches transfer descriptors. An oscillator running at 480MHz increments a counter that produces a carry output after 60,000 clocks (at 125µs intervals). This carry output advances the microframe count, that in conjunction with the µframe base address register, selects a memory pointer that contains the memory address of the first transfer descriptor in the microframe list.

Figure 2-14: Conceptual View of Host Controller Generation of Microframes



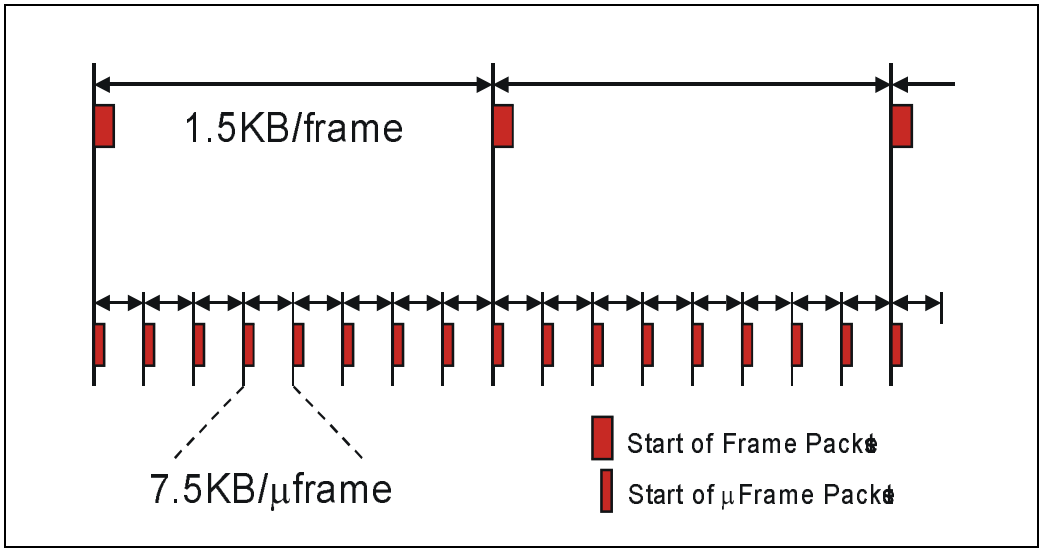
High-Speed Bandwidth Summary

The theoretical bandwidth available during each 125µs interval is 60,000 bits, or 7.5KB per 125µs interval, or 60KB/ms (60MB/s). See Figure 2-15 on page 43. Since high-speed transactions use the same packets, overhead is similar when considering packets alone. However, the propagation delay in the high-speed

environment represents a much larger number of bit times due to the much higher frequency. For example, compare the overhead associated with different types of transfers at high speed (below) with the full-speed overhead listed on page 34:

- Isochronous transactions = 38 bytes
- Interrupt transactions = 55 bytes
- Bulk transactions = 55 bytes
- Control (3 stage transfer) = 173 bytes

Figure 2-15: Bandwidth Comparison Between 12MHz Frames and 480MHz Microframes



The specification redefines the maximum packet size for some types of transfers. Table 2-2 lists the maximum packet sizes and the resulting bus efficiencies.

Table 2-2: Approximate Bus Efficiencies of Transactions with Various Data Payloads

Transfer Type	Max. Packet Size	Efficiency
Isochronous	1024 bytes	~96.4%
Interrupt	1024 bytes	~95.9%
Bulk	512 bytes	~90.3%
Control	64 bytes	~27.0%

Bus efficiency is lower in the high-speed environment compared with full speed. More importantly, the amount of data that can be transferred in a given unit of time is much greater with high-speed transactions. Thus, available bandwidth relative to the maximum data payloads is much greater in the high-speed environment. For example, an isochronous transaction with a maximum payload of 1024 bytes consumes only 13.6% of the available bus bandwidth, compared with 87% in the full-speed environment. This makes it feasible to support a far greater number of USB devices on a single bus.

The USB specification permits up to 80% of the overall high-speed bandwidth to be allocated to periodic transactions (isochronous and interrupt), and control transfers have a guaranteed reservation for up to 20% of the overall bandwidth. Bulk transfers get the bandwidth left over after all of the currently scheduled transactions complete.

The Players

Figure 2-16 on page 45 illustrates the hardware and software elements involved in the USB system. All USB transactions are initiated by USB software. These accesses are typically originated by a USB device driver that wants to communicate with its device. The USB driver provides the interface between USB device driver and the USB host controller. This software is responsible for translating client requests into transactions that send information either to or from a target USB device.

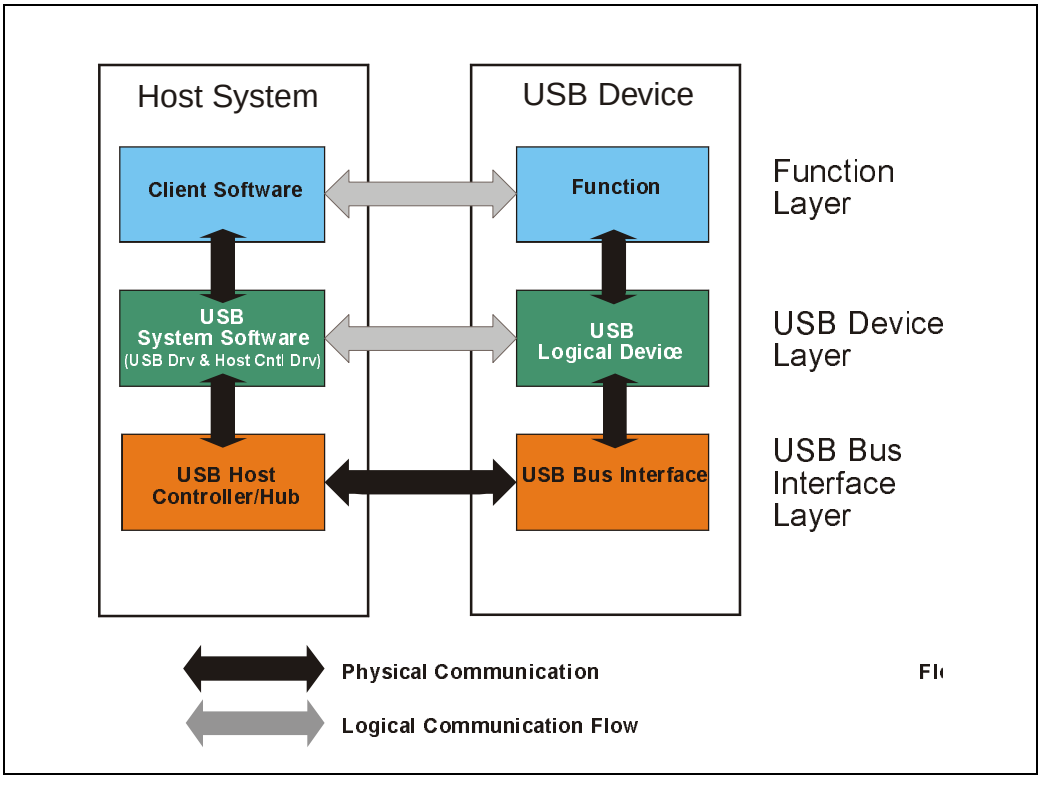
The primary hardware and software elements associated with a USB solution includes:

- USB Hardware
    - USB Host Controller/Root Hub
    - USB Hubs
    - USB Devices

- USB Software
    - USB Device Drivers
    - USB Driver
    - Host Controller Driver

The following sections describe the role of each component involved in USB transfers. Refer to Figure 2-16 on page 45 during the following discussions. More detail regarding the role of each hardware and software component can be found in subsequent chapters.

Figure 2-16: Communication Flow in a USB System



USB Client Drivers

USB device drivers (or client drivers) issue requests to the USB bus driver via I/O Request Packets (IRPs). These IRPs initiate a given transfer to or from a target USB device. For example, a USB keyboard driver must initiate an interrupt transfer by establishing an IRP and supplying a memory buffer into which data will be returned from the USB keyboard. Note that the client driver has no knowledge of the USB serial transfer mechanisms.

USB Bus Driver

The USB bus driver knows the characteristics of the USB target device and how to communicate with the device via USB. The USB characteristics are detected by the USB driver when it parses the device descriptors during device configuration. For example, some devices require a specific amount of throughput during each frame, while others may only require periodic access every *nth* frame.

When an IRP is received from a USB client driver, the USB driver organizes the request into individual transactions that will be executed during a series of bus intervals called frames (to low- and full-speed devices) and microframes (for high-speed devices). The USB driver sets up the transactions based on its knowledge of the USB device requirements, the needs of the client driver, and the limitations/capabilities of the USB.

Depending on the operating environment, the USB driver may be shipped with the operating system or added as an extension via a loadable device driver.

USB Host Controller Driver

The USB host controller driver (HCD) schedules transactions to be broadcast over the USB. Transactions are scheduled by software (host controller driver) via a series of transaction lists. Each list consists of pending transactions targeted for one or more of the USB devices attached to the bus and defines the sequence of transactions to be performed during each frame or microframe. The USB host controller fetches and executes a new list every 1ms, or 125µs. Note that a single block transfer requested by a USB client may be performed as a series of transactions that are scheduled and executed during consecutive (µ)frames. The actual scheduling depends on a variety of factors including device speed, type of transaction, transfer requirements specified by the device, and the transaction traffic on the USB bus.

The USB host controller initiates transactions via its root hub or hubs. Each frame begins with a start of frame (SOF) packet and is followed by the serial broadcast of all transactions contained within the current list. For example, if one of the requested transactions is a request to transfer data to a USB printer, the host controller would obtain the data to be sent from a memory buffer supplied by the client software and transmit the data over the USB. The hub portion of the controller converts the requested transactions into the low-level protocols required by the USB.

USB Host Controller/Root Hub

All communication on USB originates at the host under software control. The host hardware consists the USB host controller, which initiates transactions over the USB system, and the root hub, which provides attachment points (or ports) for USB devices. Three USB host controller designs have been developed:

- Universal Host Controller Interface (UHCI) -- 1.x
- Open Host Controller Interface (OHCI) -- 1.x
- Enhanced Host Controller (EHCI) -- 2.0

Each of these controllers perform the same basic job although in slightly different ways. Appendix C and D discuss the operation of the 1.x host controllers. The EHCI specification was under non-disclosure during the writing of this book. Check MindShare's web site at www.mindshare.com for a white paper on the EHCI implementation that will be developed once the specification is released.

The Host Controller

The host controller is responsible for generating the transactions that have been scheduled by the host software. The host controller software driver builds a linked list of data structures in memory that defines the transactions that are scheduled to be performed during a given frame. These data structures, called transfer descriptors, contain all of the information the host controller needs to generate the transactions. This information includes:

- USB Device Address
- Type of Transfer
- Direction of Transfer
- Address of Device Driver's Memory Buffer

The host controller performs writes to a target device by reading data from a memory buffer (supplied by the USB device driver) that is to be delivered to the target device. The host controller performs a parallel to serial conversion on the data, creates the USB transaction, and forwards it to the root hub to send over the bus.

If a read transfer is required, the host controller builds the read transaction and sends it to the root hub. The hub transmits the read transaction over the USB. The target device recognizes that it is being addressed and that data is being requested. The device then transmits data back to the root hub, which forwards

the data on to the host controller. The host controller performs the serial to parallel conversion on the data and transfers the data to the device driver's memory buffer.

Note that the USB root hub and target devices perform error checks during a transaction. Errors detected are recognized by the root hub, forwarded to the host controller to be logged and reported to the host software.

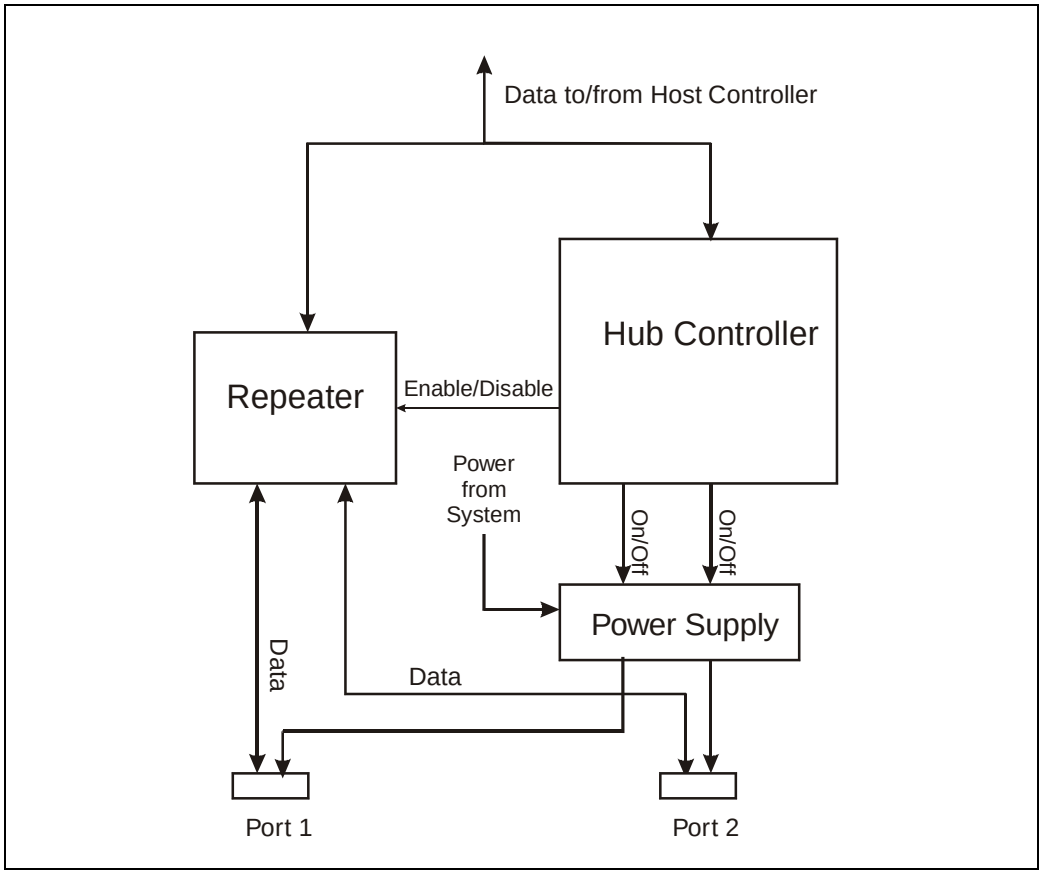
The Root Hub

Transactions generated by the host controller are forwarded to the root hub to be transmitted to the USB. Consequently, every USB transaction originates at the root hub. The root hub provides the connection points for USB devices and performs the following key operations:

- controls power to its USB ports
- enables and disables ports
- recognizes devices attached to each port
- sets and reports status events associated with each port (when polled by host software)

The root hub consists of a hub controller and repeater as illustrated in Figure 2-17 on page 49. The hub controller responds to accesses made to the hub itself, for example, requests by the host software to apply or remove power to a port. The repeater forwards transactions to and from the USB and the host controller.

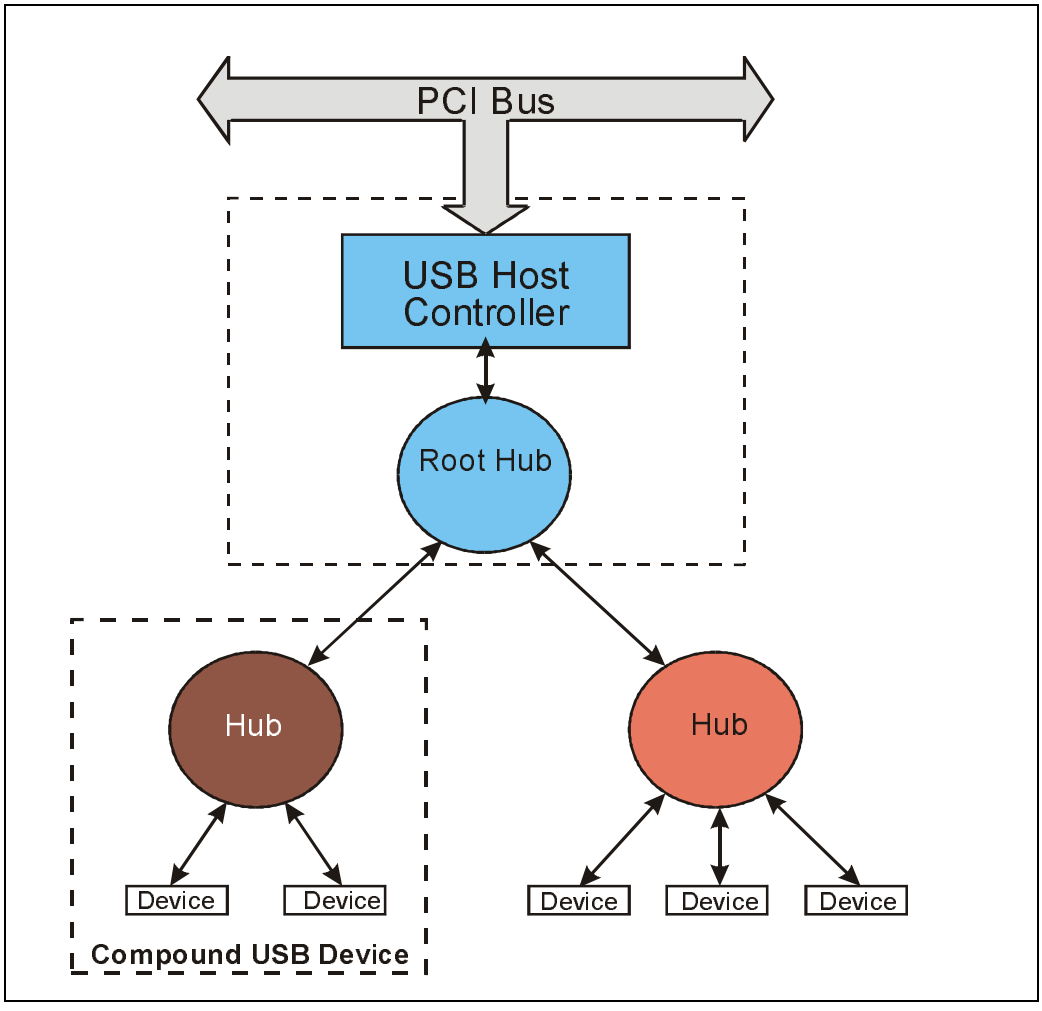
Figure 2-17: Block Diagram of Major Root Hub Functions



USB Hubs

In addition to the root hub, USB systems support additional hubs that provide one or more USB ports for attaching other USB devices. USB hubs may be integrated into devices such as keyboards or monitors (called compound devices), or implemented as stand-alone devices as illustrated in Figure 2-18. Furthermore, hubs may be bus powered (i.e. derive power for themselves and all attached devices from the USB bus) or may be self-powered. Bus-powered hubs are limited by the amount of power available from the bus and can therefore support a maximum of four USB ports. Chapter 4 discusses USB power issues.

Figure 2-18: USB Hub Types



Hubs contain two major functional elements:

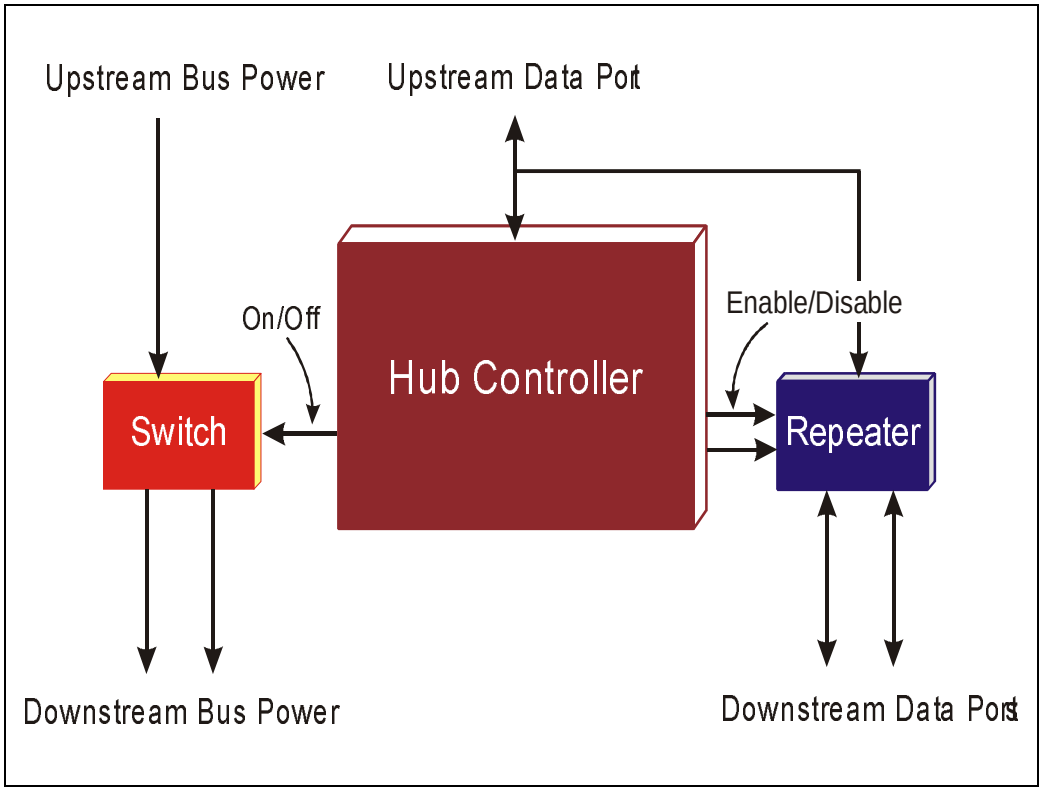
- hub controller
- repeater

Figure 2-19 on page 51 illustrates these functions.

Hub Controller

The hub controller contains a USB interface, or serial interface engine (SIE). It also contains the descriptors that software reads to identify the device as a hub. The hub controller gathers hub and port status information also read by the USB host software to detect the connection and removal of devices and to determine other status information. The controller also receives commands from host software to control various aspects of the hub's operation (e.g., powering and enabling the ports).

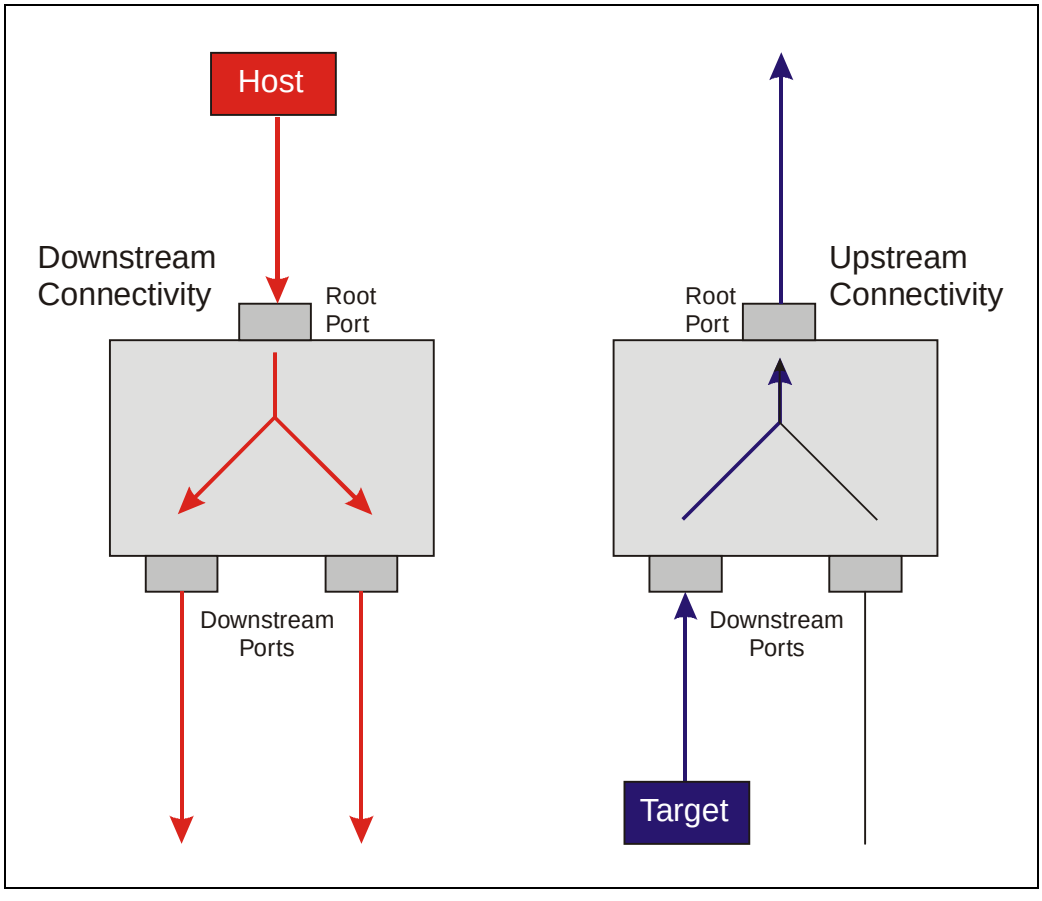
Figure 2-19: Primary Hub Functions



Hub Repeater

Refer to Figure 2-20. Bus traffic arriving at the hub must be forwarded on in either the upstream (toward the host) or downstream (away from the host) direction. Transmissions originating at the host will arrive on the hub's root port and must be forwarded to all enabled ports. When a target device responds to a host-initiated transaction, it must transmit a response upstream, which the hub must forward from the downstream port to the root port.

Figure 2-20: Hub Repeater Performing Downstream and Upstream Connectivity



Hub's Role in Configuration

Hubs also play a pivotal role in the hot attachment/detachment (automatic detection and configuration during runtime) of USB devices. Hubs must recognize that a device has been attached or detached and report the event when host software polls the hub.

USB Devices

USB devices contain descriptors that specify a given device's attributes and characteristics. This information specifies to host software a variety of features and capabilities that are needed to configure the device and to locate the USB client software driver. The USB device driver may also use device descriptors to determine additional information needed to access the device in the proper fashion. This mechanism is referred to as the Device Framework and must be understood by software in order to configure and access the device correctly. See the section entitled "Device Framework" on page 63 for a more complete discussion. As mentioned previously, USB devices can be implemented either as high-speed, full-speed or low-speed devices.

High-Speed Devices

High-speed devices see only high-speed transactions. Low- and full-speed devices are accessed via high-speed split transactions delivered to high-speed hubs. The high-speed hubs translate the split transactions into low- or full-speed transactions and deliver them to the target devices.

Full-Speed Devices

Full-speed devices see all transactions broadcast over the USB and can be implemented as full-feature devices. These devices accept and send serial data at the maximum 12Mb/s rate.

Low-Speed Devices

Low-speed devices are limited in not only throughput (1.5Mb/s) but feature support. Furthermore, low-speed devices only see USB transactions that follow a preamble packet. Low-speed hub ports remain disabled during full-speed transactions, preventing full-speed bus traffic from being sent over low-speed cables. Preamble packets specify that the following transaction will be broadcast at low speed. Hubs enable their low-speed ports after detecting a preamble packet, permitting low-speed devices to see the low-speed bus activity.

USB Communications Model

Unlike devices that reside on other common bus structures, USB devices do not directly consume system resources. That is, USB devices are not mapped into memory or I/O address space, nor do they use IRQ lines or DMA channels. Furthermore, all transactions originate from the host system. The only system resources required by a USB system are the memory locations used by USB system software and the memory and/or I/O address space and IRQ line used by the USB host controller. This eliminates much of the difficulty encountered with standard peripheral implementations that require a considerable amount of I/O space and a large number of interrupt lines.

Communications Flow

Figure 2-21 on page 56 illustrates the basic communication flow and the system resources used by USB systems. The USB client initiates a transfer when it calls USB system software and requests a transfer. USB client drivers supply a memory buffer used to store data when transferring data to or from the USB device. Each transfer between a given register (or endpoint) within a USB device and the client driver occurs via a communication pipe that USB system software establishes during device configuration. USB system software splits the client's request into individual transactions that are consistent with the bus bandwidth requirements of the device and the USB protocol mechanisms.

The requests are passed to the USB host controller driver, which in turn schedules the transaction to be performed over the USB. The host controller performs the transaction based on the contents of a transfer descriptor that is built by the HCD. The HCD knows all the information necessary to perform the required transaction via the USB. The key information contained within a transfer descriptor includes:

- Address of the target USB device
- Speed of the target device
- Type of transfer to be performed
- Size of the data packet
- Location of the client's memory buffer

The host controller may have registers that are mapped into the processor's I/O or memory address space. These registers control the operation of the host controller and must be loaded with values by the HCD to ensure desired operation. For example, a register is loaded with an address pointer that specifies the

memory location where the transfer descriptors reside.

The host controller fetches the transfer descriptors that have been built by the host controller driver. Each descriptor defines a given transaction that must be performed to satisfy a client's transfer request. The host controller generates the USB transaction that is specified by each transfer descriptor. Each transaction results in data being transferred either from the client buffer to the USB device or from the device to the buffer depending on the direction of the transfer. When the entire transfer has completed, USB system software notifies the client driver.

Transfers, IRPs, Frames, and Packets

Figure 2-23 on page 59 illustrates the mechanisms used during the USB communication process and the relationships that exist between each layer of the USB system. Transfers are initiated by the client driver when it issues a transfer request to the USB driver. Ultimately, the transaction is performed via the low-level packetized transactions over the USB. The following sections discuss each layer involved in completing a USB transfer.

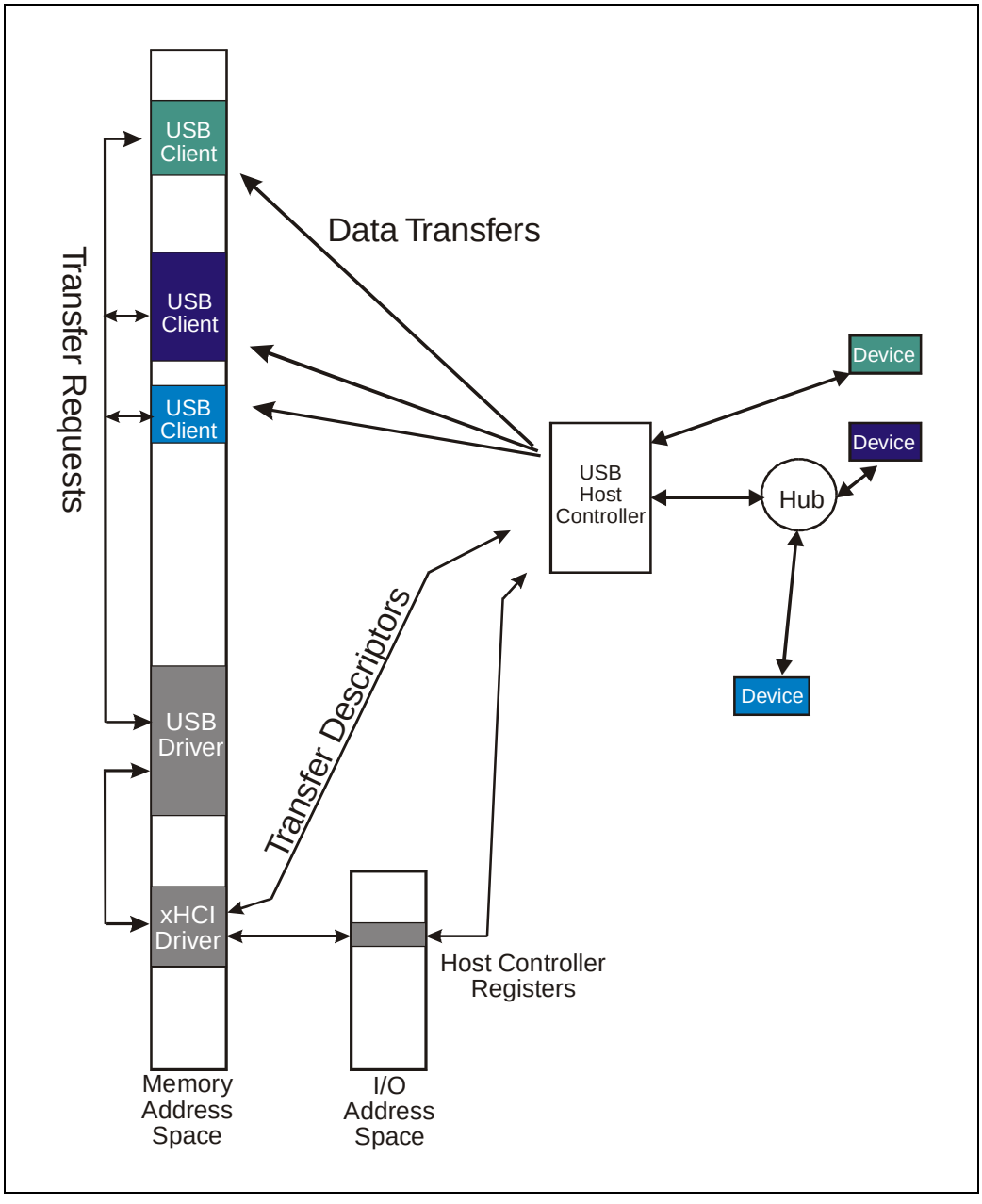
Transfers

Each USB function is designed with a collection of registers, or endpoints, used by the client driver when accessing its function. Each endpoint has particular transfer characteristics that it supports. For example, when transferring information to a speaker, the data transfer must continue at a constant data rate to prevent distortion of the audio. Other endpoints may have different characteristics and thus require a different transfer type. The transfer types supported by USB include:

- Isochronous Transfers
- Bulk Transfers
- Interrupt Transfers
- Control Transfers

Client drivers understand the nature of the transfer related to each endpoint associated with its function, as does the USB driver. This information is determined by reading descriptors from the device. Chapter 6 describes the unique characteristics of each transfer type.

Figure 2-21: The Communications Model



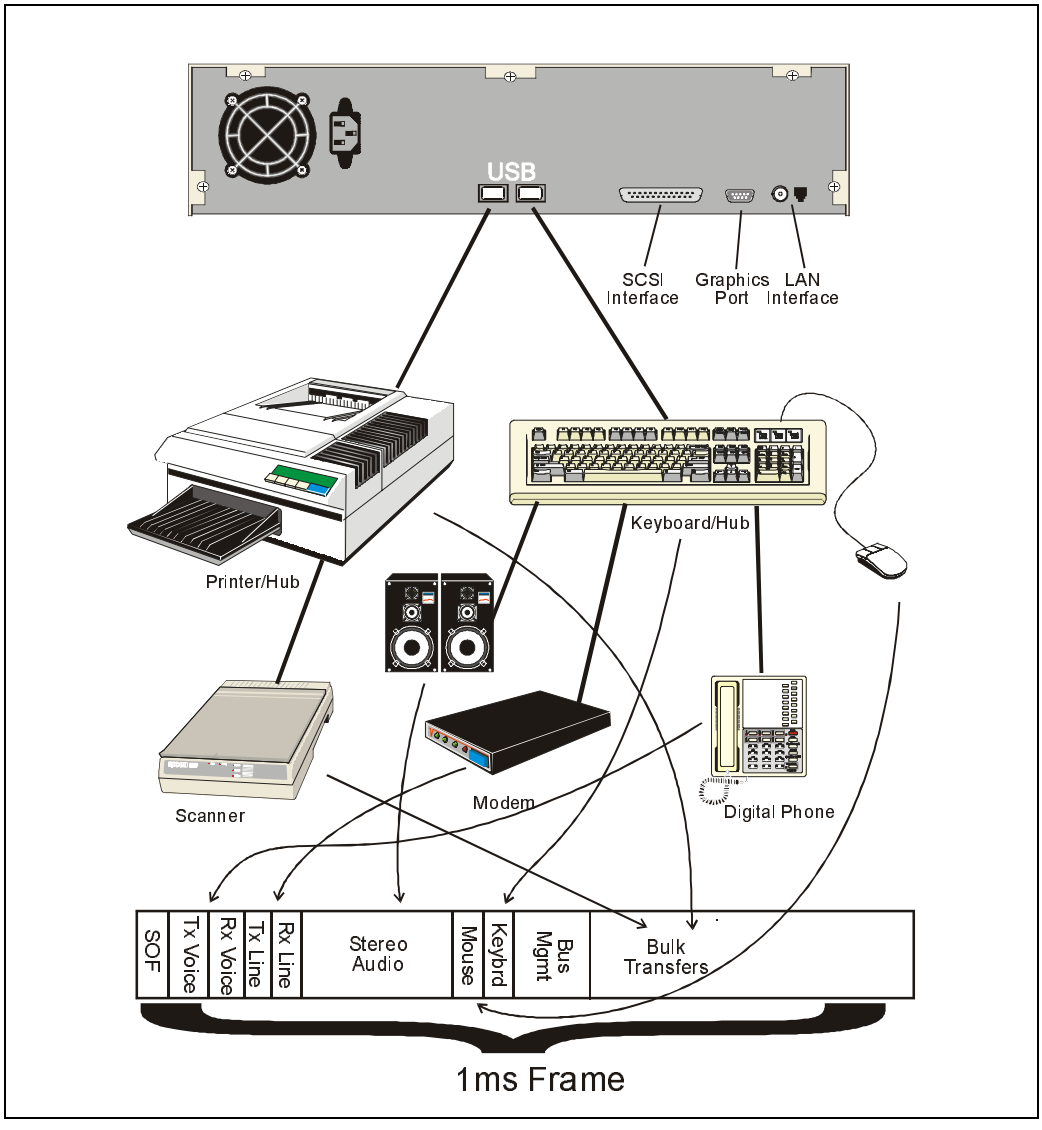
The USB Driver, IRPs, and Frames

When a client driver wishes to perform a transfer to or from a given endpoint, it calls the USB driver to initiate the transfer. The requested transfer is called an I/O Request Packet (IRP). Some transfers consist of a large block of data. Since USB is a shared bus (i.e., many devices use the same bus at the same time), a single device cannot typically perform an entire block transfer across USB at one time. Rather, a transfer is typically split up and performed in segments (called transactions) over a longer period of time. This ensures that a portion of the USB bandwidth can be allocated for the other USB devices residing on the bus.

USB communication is based on transferring data at regular (1ms) intervals called frames. Each USB device requires that a portion of the USB bandwidth be allocated during these 1ms frames. Bandwidth allocation depends on the required throughput of the device (as specified by device descriptors) and the available USB bandwidth not used by other USB devices. As each USB device is attached and configured, system software parses its device descriptors to determine the amount of bus bandwidth it requires. Software checks the remaining bandwidth and if the device's requirements can be satisfied, it is configured. If the bandwidth required by the device is not available, due to bus bandwidth already allocated to other devices previously attached, the device will not be configured and the user will be notified.

Figure 2-22 on page 58 illustrates a community of devices attached to the USB and the variety of potential transactions that could be performed during a single 1ms frame. This is a contrived example to illustrate the shared nature of the USB frame. Not every USB device will necessarily transfer data during each frame. For example, host software will poll the keyboard every *nth* frame to check for keystrokes. Devices are allocated a portion of the overall bus bandwidth that they require during each frame. This will likely result in large bulk transfers, such as print jobs, being split over a fairly large number of 1ms frames. The actual number of frames required depends on the transfer capability of the printer's USB interface, specified limitations placed on bulk transfers, and the amount of bus bandwidth being used by other devices currently installed on the USB.

Figure 2-22: USB Devices Performing Transfers During Frame

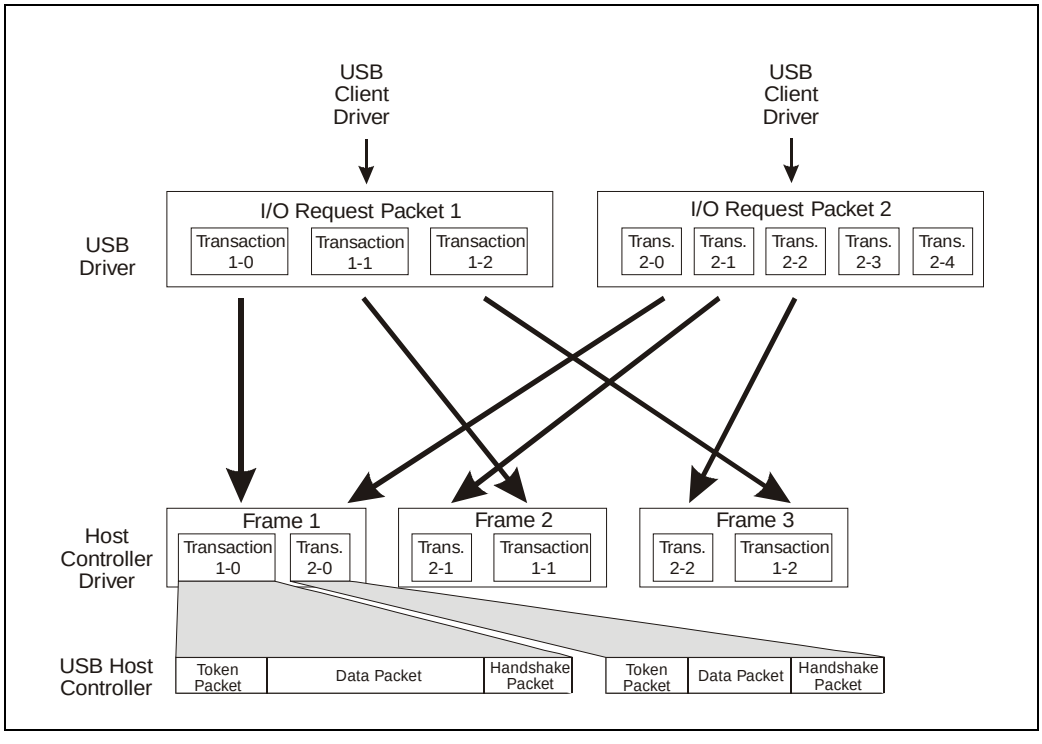


The Host Controller Driver and Transactions

The host controller driver receives the packet requests from the USB driver and schedules them to be performed during a series of frames. The scheduling order is based on an algorithm defined by the host controller driver. The algorithm is based on USB transfer capabilities and limitations (to be discussed in subsequent chapters).

Scheduling is performed by building a series of data structures (called transfer descriptors) that define each sequential transaction to be performed over the USB. The host controller reads and interprets these transfer descriptors and executes the USB transaction described.

Figure 2-23: Relationship Between IRPs, Transfers, Frames, and Packets



The Host Controller and Packets

The host controller and root hub generates transactions over the USB. Transactions consist of a series of packets that typically include token packets, data packets, and handshake packets. Refer to Chapter 7 for details regarding transactions and packets.

Device Framework (how devices present themselves to software)

USB has been designed to promote class device driver implementations. A set of devices that have similar attributes and services are defined as belonging to a given class of device. These common groupings of devices have a common class driver that can accommodate all devices within the class.

Device Descriptors

A device describes itself to host software via a number of standard descriptors, illustrated in Figure 2-24 on page 61. These descriptors include:

- Device Descriptor — Each device has a single device descriptor containing information about the default communications pipe that is used to configure the device, along with general information about the device. The device descriptor also identifies the number of possible configurations (one or more) that a device supports.
- Configuration Descriptor — A device has a configuration descriptor for each configuration that it supports. For example, a high-power device may also support a low-power mode, resulting in a configuration descriptor for each power mode. The configuration descriptor includes general information about the configuration and defines the number of interfaces for the device when used in this configuration.
- Interface Descriptor — A given configuration may have one or more interfaces that it supports. An example of a multiple interface device could be a CD-ROM, in which case three device drivers may be used to access the different functional devices: one device driver for the device's mass storage interface (for storing files), one for the audio device (for playing music CDs), and one for the video image driver (for displaying images).

  Interface descriptors provide general information about this interface. They also indicate the class of device supported by this particular interface and

specify the number of endpoint descriptors used when communicating with this interface.

- Endpoint Descriptors — A device interface contains one or more endpoint descriptors, each of which defines a point of communication (e.g., a data register). The endpoint descriptor contains information, such as the transfer type supported by the endpoint (i.e., isochronous, bulk, interrupt, or control), and the maximum transfer rate supported.
- String Descriptors — String descriptors can be defined for the overall device, for a given configuration, and/or for each interface definition. These string descriptors describe the configuration and interfaces in unicode that can be displayed and read by the user.
- Class-Specific Descriptors — Some device classes require descriptors beyond the standard descriptors defined by the USB specification. These descriptors are defined by the relevant device class specification (not shown).

Figure 2-24: Standard Descriptors

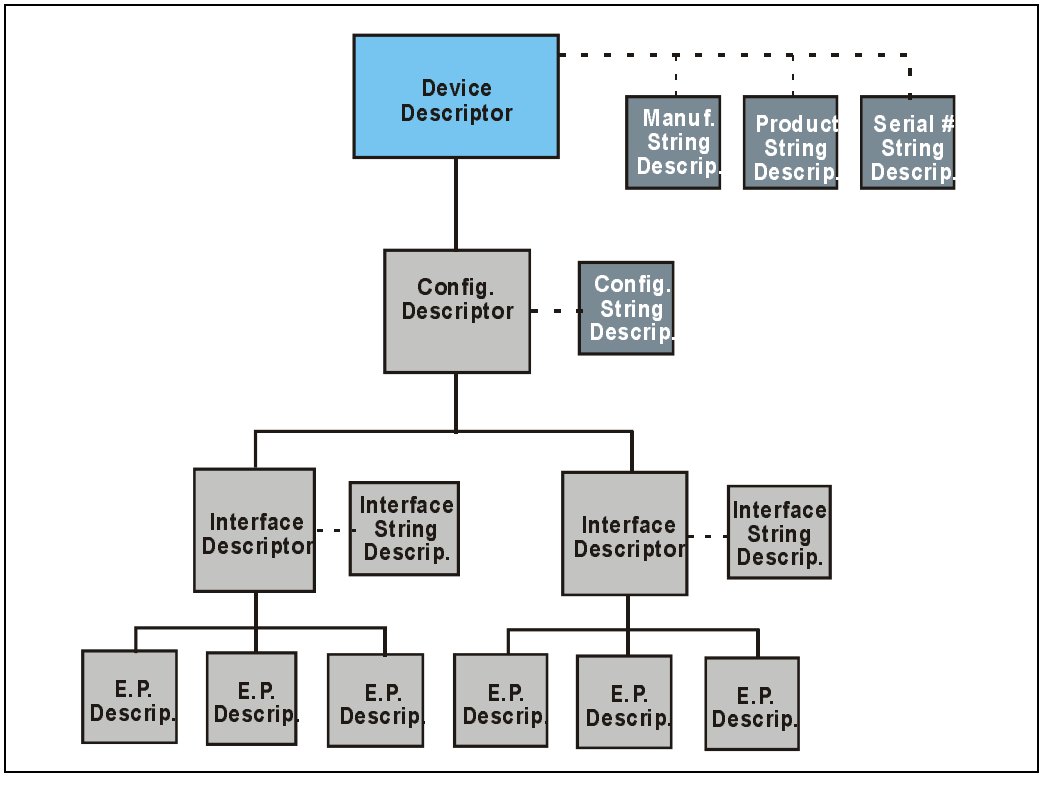


Figure 2-25: Standard Descriptors with Two Configurations

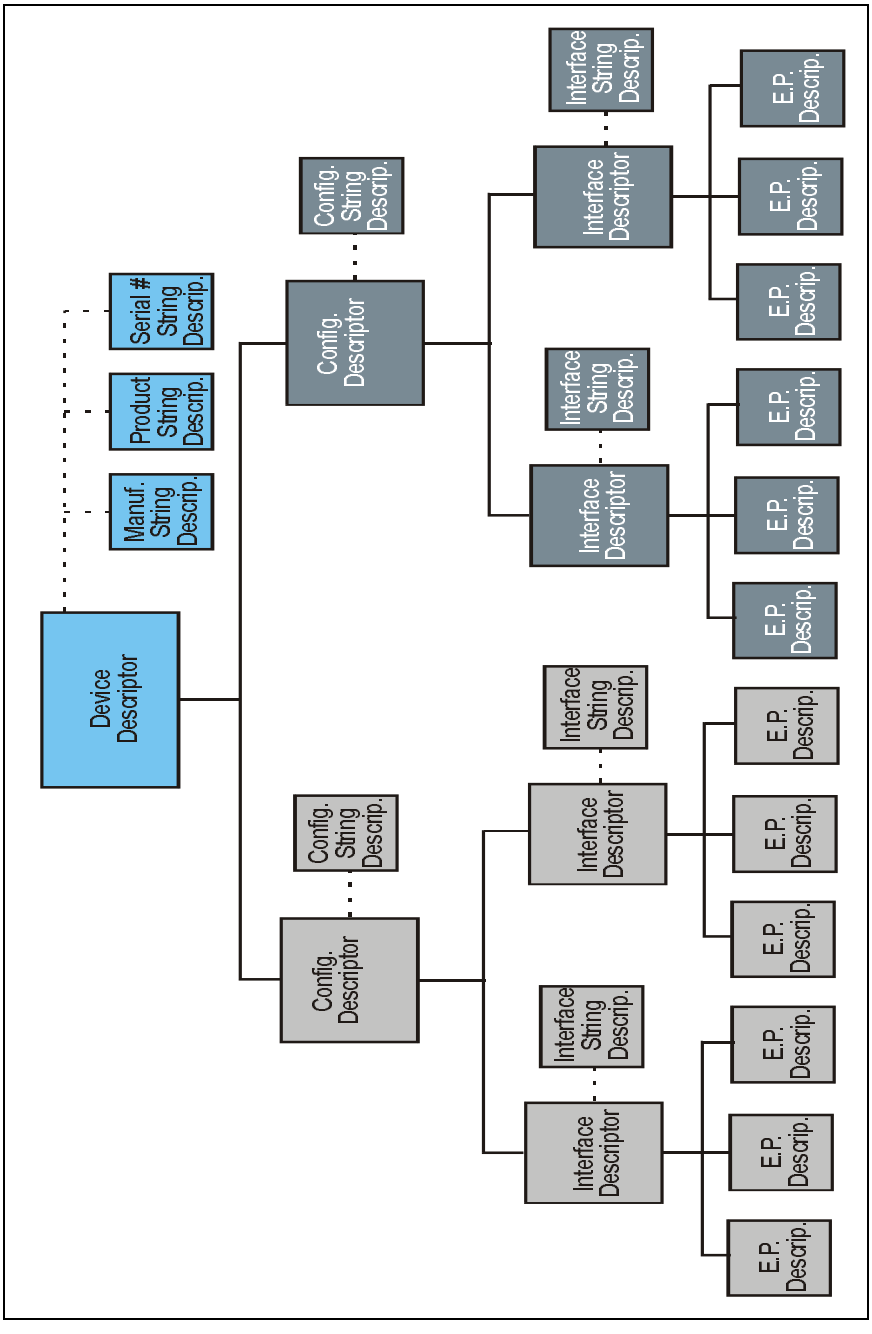


Figure 2-25 on page 62 illustrates another set of descriptors. In this example, two separate configurations are defined, each of which includes two interface descriptors. This illustration like the previous does not show any class-specific descriptors that may be required by some device classes.

Device Framework

The device framework provides three logical layers that describe the relationship between the host and device hardware and software. Figure 2-26 on page 64 illustrates these layers and the relationship between the host and a given USB device. The layered approach helps explain the relationships between the different pieces of host software and the responsibilities each has in the USB system. The separate layers are provided to promote understanding of the USB communication mechanisms and are discussed in the following sections.

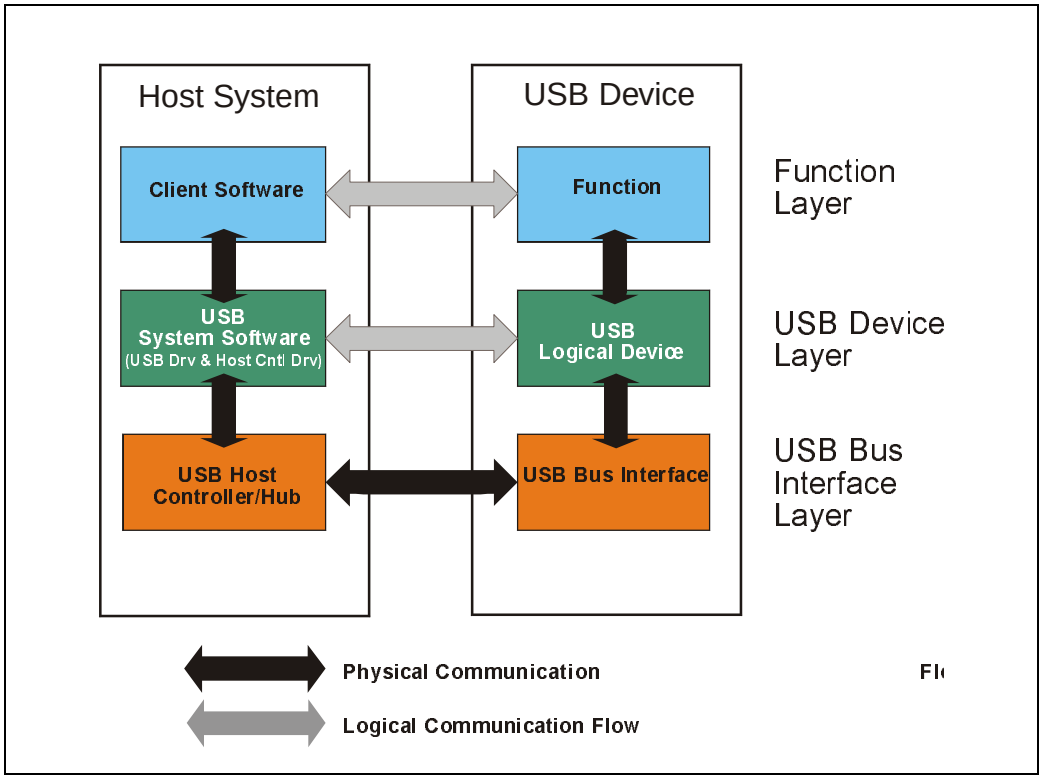
USB Bus Interface Layer

The USB bus interface layer provides the low-level transfer of data over the USB cables. This layer consists of the:

- physical connection
- electrical signaling environment
- packet transfer mechanisms

This layer represents the actual transfer of data across the USB cable between the host system and the USB devices. The host side consists of the USB host controller and root hub, while the USB side consists of the USB interface within the device. Details related to the transfer of data across the USB cable are covered in later chapters.

Figure 2-26: Device Framework — Software's View of Hardware



USB Device Layer

The USB device layer represents the portion of USB that comprehends the actual USB communication mechanism and the nature of the transfers required by a USB functional device. This layer consists of USB system software on the host side and a logical view of the USB device on the device side. USB system software views a logical device as a collection of endpoints that compose a given functional interface.

USB system software provides the services needed to interface client software with its USB function. USB system software has specific knowledge of the USB transfer mechanisms and must allocate bus bandwidth for the community of USB devices. The logical USB device represents the collection of endpoints through which a client communicates with its function. USB system software

views these endpoints via the standard descriptors, which are parsed by the USB system software to obtain the transfer characteristics of a given device. These characteristics in conjunction with system software's knowledge of the USB transfer mechanisms permit bus bandwidth to be reserved for each functional device as it's configured.

USB system software performs a variety of key functions including:

- Device attachment / detachment detection
- Device configuration
- Bandwidth allocation
- Managing control flow between client and device
- Managing data flow between client and device
- Collecting status and transaction statistics
- Transaction scheduling
- Controlling the electrical interface (e.g., limited cable power management)

Note that one set of USB system software exists in the system to manage accesses to all USB devices attached to the USB bus. USB system software consists of the following entities:

- USB Driver (USBD) — provides interface and services for client software drivers, allocates bus bandwidth, and manages configuration process.
- USB Host Controller Driver — controls operation of the host controller, schedules transactions, and monitors completion status of transactions.

A brief description of the primary jobs that each performs is also provided. A more comprehensive description of these software layers are provided in Chapter 22, entitled "Overview of USB Host Software," on page 421.

Function Layer

This layer represents the relationship between client software and a given device's functional interface. Each interface consists of a particular class of device that a matched class driver is designed to manipulate. USB client software cannot access their function directly as is typically done in other environments (e.g., ISA, PCI, and PCMCIA), since they are not mapped directly into memory and I/O address space. Instead, USB device drivers must use the USBD programming interface to access their devices.

USB clients view their USB devices as consisting of a given interface, which they know how to manipulate. USB system software must report the interface type and other device characteristics to USB clients.

USB Peripheral Connection

As stated in the previous chapters, USB provides a single type of connector for attaching peripherals to a system. USB 2.0 also supports three different speeds of USB devices:

- low-speed devices — 1.5Mb (megabits) / second
- full-speed devices — 12Mb / second
- high-speed devices — 480Mb / second

All USB devices attach via a USB hub that provides one or more ports. Figure 2-27 on page 68 illustrates a variety of devices attached to USB ports provided by the system. Hub ports may support only full- and low-speed or may support all three speeds as illustrated in Figure 2-27. A device's speed is detected when it is attached to the hub port. (Refer to Chapter 5, entitled "LS / FS Signaling Environment," on page 93, and Chapter 11, entitled "The High-Speed Signaling Environment," on page 217 for details).

Some devices such as keyboards and mice typically operate at low speed, while other devices such as digital telephones must operate at either full or high speed. However, several connection issues can exist depending on the device speed and the hub port capability as listed below:

- full-speed hub ports (1.x hubs) — support for LS and FS devices only
- high-speed hub ports (2.0 hubs) — support for LS, FS, and HS devices

Due to related EMI differences at the different transmission rates, the cables used for low-speed versus full- / high-speed devices are subject to different electrical characteristics. See Chapter 3 for details regarding the electrical characteristics of the cables.

Full-Speed Hubs

Hubs based on the 1.0 and 1.1 versions of the specification can support only low- and full-speed devices. These hubs block all full-speed traffic from reaching low-speed devices attached to its ports. low-speed transactions targeting these devices will always be preceded by a preamble packet that serves as a command to l.x hubs to enable their low-speed ports. This ensures that low-speed devices see only the low-speed transactions.

Any high-speed capable devices attached to a 1.x hub must operate in full-speed mode. Note that the minimum requirement is that a high-speed device must permit access to its descriptors at full-speed, but may not function beyond that basic capability.

High-Speed Hubs

Each USB high-speed capable port must support the attachment and operation of high-, full- and low-speed devices. The HS hub interface detects the speed of the device and makes the necessary adjustments to operate at the required speed.

High-Speed Devices

The hub repeats high-speed packets to all ports that have high-speed devices attached. The high-speed devices decode the high-speed packets to determine if they are being targeted by the host.

Low- and Full-Speed Devices

When a low- or full-speed device is attached to a high-speed hub port, the hub checks for high-speed split transactions that are targeted for one of the low- or full-speed devices attached to its ports. When this occurs the hub translates the high-speed split transaction into the required low- or full-speed transaction and delivers it to the target device.

Topology

USB employs a tiered star topology where hubs provide attachment points for USB devices. The host controller contains the root hub, which is the origin of all USB ports in the system. As illustrated in Figure 2-27, three tiers are created by the three hubs: the root hub, a 2.0 hub, and a 1.x hub. Note that devices of any speed may be connected to any of the hub ports regardless of their speed.

Figure 2-27: USB's Tiered Star Topology