a continuous stream of serial data. An analogous situation occurs in the output path of the interface.

Because it requires fewer wires, serial transmission is convenient for connecting devices that are physically far away from the computer. The speed of transmission, often given as a *bit rate*, depends on the nature of the devices connected. To accommodate a range of devices, a serial interface must be able to use a range of clock speeds. The circuit in Figure 4.37 allows separate clock signals to be used for input and output operations for increased flexibility.

Because serial interfaces play a vital role in connecting I/O devices, several widely used standards have been developed. A standard circuit that includes the features of our example in Figure 4.37 is known as a Universal Asynchronous Receiver Transmitter (UART). It is intended for use with low-speed serial devices. Data transmission is performed using the asynchronous start-stop format, which we discuss in Chapter 10. To facilitate connection to communication links, a popular standard known as RS-232-C was developed. It is also described in Chapter 10.

4.7 STANDARD I/O INTERFACES

The previous sections point out that there are several alternative designs for the bus of a computer. This variety means that I/O devices fitted with an interface circuit suitable for one computer may not be usable with other computers. A different interface may have to be designed for every combination of I/O device and computer, resulting in many different interfaces. The most practical solution is to develop standard interface signals and protocols.

It is helpful at this point to understand how a computer system is put together. A typical personal computer, for example, includes a printed circuit board called the motherboard. This board houses the processor chip, the main memory, and some I/O interfaces. It also has a few connectors into which additional interfaces can be plugged.

The processor bus is the bus defined by the signals on the processor chip itself. Devices that require a very high speed connection to the processor, such as the main memory, may be connected directly to this bus. For electrical reasons, only a few devices can be connected in this manner. The motherboard usually provides another bus that can support more devices. The two buses are interconnected by a circuit, which we will call a *bridge*, that translates the signals and protocols of one bus into those of the other. Devices connected to the expansion bus appear to the processor as if they were connected directly to the processor's own bus. The only difference is that the bridge circuit introduces a small delay in data transfers between the processor and those devices.

It is not possible to define a uniform standard for the processor bus. The structure of this bus is closely tied to the architecture of the processor. It is also dependent on the electrical characteristics of the processor chip, such as its clock speed. The expansion bus is not subject to these limitations, and therefore it can use a standardized signaling scheme. A number of standards have been developed. Some have evolved by default, when a particular design became commercially successful. For example, IBM developed a bus they called ISA (Industry Standard Architecture) for their personal computer, known at the time as PC AT. The popularity of that computer led to other manufacturers producing ISA-compatible interfaces for their I/O devices, thus making ISA into a de facto standard.

Some standards have been developed through industrial cooperative efforts, even among competing companies driven by their common self-interest in having compatible products. In some cases, organizations such as the IEEE (Institute of Electrical and Electronics Engineers), ANSI (American National Standards Institute), or international bodies such as ISO (International Standards Organization) have blessed these standards and given them an official status.

In this section, we present three widely used bus standards, PCI (Peripheral Component Interconnect), SCSI (Small Computer System Interface), and USB (Universal Serial Bus). The way these standards are used in a typical computer system is illustrated in Figure 4.38. The PCI standard defines an expansion bus on the motherboard. SCSI and USB are used for connecting additional devices, both inside and outside the

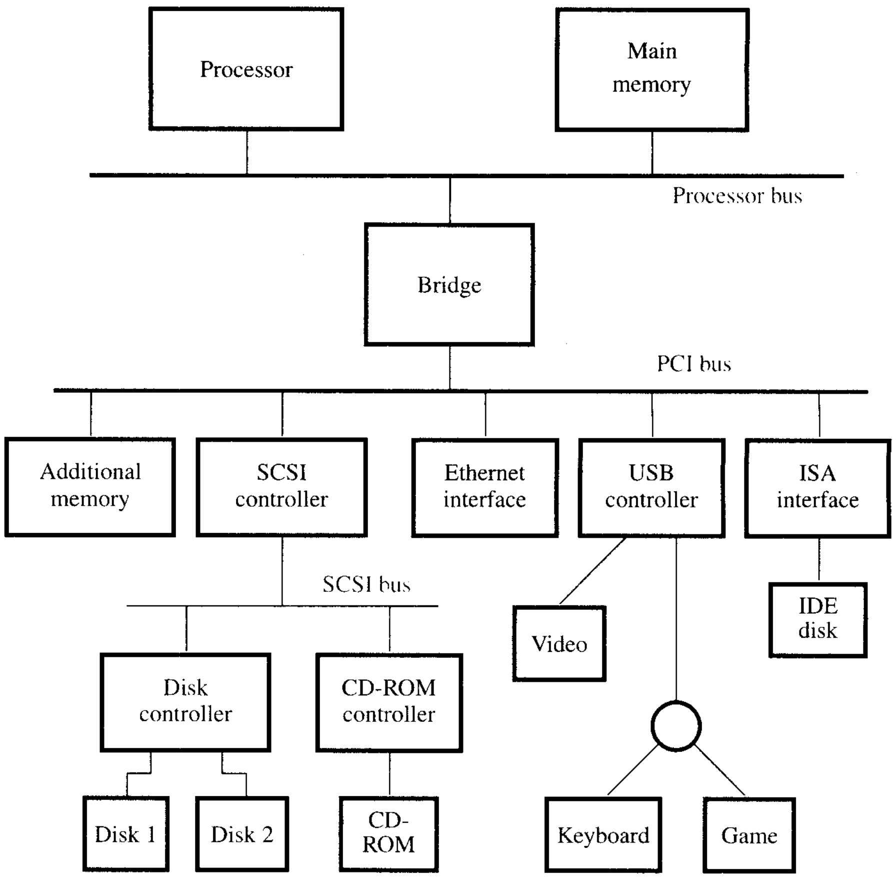


Figure 4.38 An example of a computer system using different interface standards.

computer box. The SCSI bus is a high-speed parallel bus intended for devices such as disks and video displays. The USB bus uses serial transmission to suit the needs of equipment ranging from keyboards to game controls to internet connections. The figure shows an interface circuit that enables devices compatible with the earlier ISA standard, such as the popular IDE (Integrated Device Electronics) disk, to be connected. It also shows a connection to an Ethernet. The Ethernet is a widely used local area network, providing a high-speed connection among computers in a building or a university campus.

A given computer may use more than one bus standard. A typical Pentium computer has both a PCI bus and an ISA bus, thus providing the user with a wide range of devices to choose from.

4.7.1 PERIPHERAL COMPONENT INTERCONNECT (PCI) BUS

The PCI bus [1] is a good example of a system bus that grew out of the need for standardization. It supports the functions found on a processor bus but in a standardized format that is independent of any particular processor. Devices connected to the PCI bus appear to the processor as if they were connected directly to the processor bus. They are assigned addresses in the memory address space of the processor.

The PCI follows a sequence of bus standards that were used primarily in IBM PCs. Early PCs used the 8-bit XT bus, whose signals closely mimicked those of Intel's 80*x*86 processors. Later, the 16-bit bus used on the PC AT computers became known as the ISA bus. Its extended 32-bit version is known as the EISA bus. Other buses developed in the eighties with similar capabilities are the Microchannel used in IBM PCs and the NuBus used in Macintosh computers.

The PCI was developed as a low-cost bus that is truly processor independent. Its design anticipated a rapidly growing demand for bus bandwidth to support high-speed disks and graphic and video devices, as well as the specialized needs of multiprocessor systems. As a result, the PCI is still popular as an industry standard almost a decade after it was first introduced in 1992.

An important feature that the PCI pioneered is a plug-and-play capability for connecting I/O devices. To connect a new device, the user simply connects the device interface board to the bus. The software takes care of the rest. We will discuss this feature after we describe how the PCI bus operates.

Data Transfer

In today's computers, most memory transfers involve a burst of data rather than just one word. The reason is that modern processors include a cache memory (see Figure 1.6). Data are transferred between the cache and the main memory in bursts of several words each, as we will explain in Chapter 5. The words involved in such a transfer are stored at successive memory locations. When the processor (actually the cache controller) specifies an address and requests a read operation from the main memory, the memory responds by sending a sequence of data words starting at that address. Similarly, during a write operation, the processor sends a memory address followed by a sequence of data words, to be written in successive memory locations starting at that address. The PCI is designed primarily to support this mode of operation. A read or a write operation involving a single word is simply treated as a burst of length one.

The bus supports three independent address spaces: memory, I/O, and configuration. The first two are self-explanatory. The I/O address space is intended for use with processors, such as Pentium, that have a separate I/O address space. However, as noted in Chapter 3, the system designer may choose to use memory-mapped I/O even when a separate I/O address space is available. In fact, this is the approach recommended by the PCI standard for wider compatibility. The configuration space is intended to give the PCI its plug-and-play capability, as we will explain shortly. A 4-bit command that accompanies the address identifies which of the three spaces is being used in a given data transfer operation.

Figure 4.38 shows the main memory of the computer connected directly to the processor bus. An alternative arrangement that is used often with the PCI bus is shown in Figure 4.39. The PCI bridge provides a separate physical connection for the main memory. For electrical reasons, the bus may be further divided into segments connected via bridges. However, regardless of which bus segment a device is connected to, it may still be mapped into the processor's memory address space.

The signaling convention on the PCI bus is similar to the one used in Figure 4.25. In that figure, we assumed that the master maintains the address information on the bus until data transfer is completed. But, this is not necessary. The address is needed only long enough for the slave to be selected. The slave can store the address in its internal buffer. Thus, the address is needed on the bus for one clock cycle only, freeing the address lines to be used for sending data in subsequent clock cycles. The result is

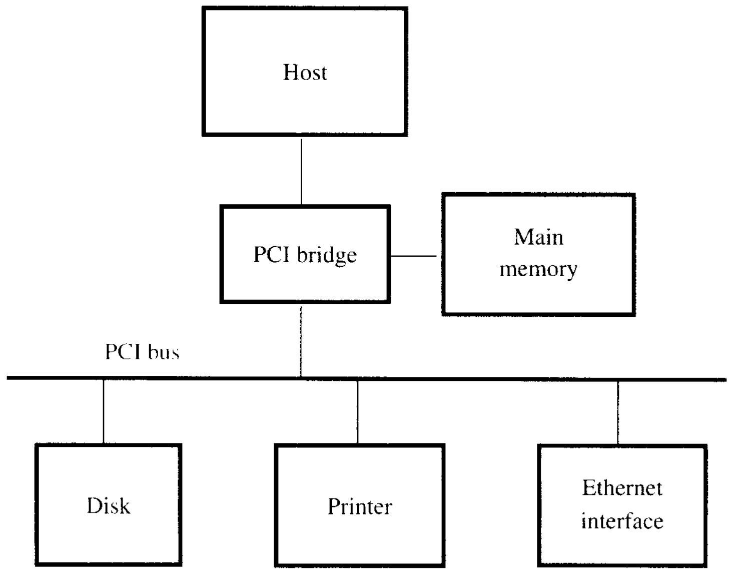


Figure 4.39 Use of a PCI bus in a computer system.

Table 4.3 Data transfer signals on the PCI bus.

Name	Function
CLK	A 33-MHz or 66-MHz clock.
FRAME#	Sent by the initiator to indicate the duration of a transaction.
AD	32 address/data lines, which may be optionally increased to 64.
C/BE#	4 command/byte-enable lines (8 for a 64-bit bus).
IRDY#, TRDY#	Initiator-ready and Target-ready signals.
DEVSEL#	A response from the device indicating that it has recognized its address and is ready for a data transfer transaction.
IDSEL#	Initialization Device Select.

a significant cost reduction because the number of wires on a bus is an important cost factor. This approach is used in the PCI bus.

At any given time, one device is the bus master. It has the right to initiate data transfers by issuing read and write commands. A master is called an *initiator* in PCI terminology. This is either a processor or a DMA controller. The addressed device that responds to read and write commands is called a *target*.

To understand the operation of the bus and its various features, we will examine a typical bus transaction. The main bus signals used for transferring data are listed in Table 4.3. Signals whose name ends with the symbol # are asserted when in the low-voltage state. The main difference between the PCI protocol and Figure 4.25 is that in addition to a Target-ready signal, PCI also uses an Initiator-ready signal, IRDY#. The latter is needed to support burst transfers.

Consider a bus transaction in which the processor reads four 32-bit words from the memory. In this case, the initiator is the processor and the target is the memory. A complete transfer operation on the bus, involving an address and a burst of data, is called a *transaction*. Individual word transfers within a transaction are called *phases*. The sequence of events on the bus is shown in Figure 4.40. A clock signal provides the timing reference used to coordinate different phases of a transaction. All signal transitions are triggered by the rising edge of the clock. As in the case of Figure 4.25, we show the signals changing later in the clock cycle to indicate the delays they encounter.

In clock cycle 1, the processor asserts FRAME# to indicate the beginning of a transaction. At the same time, it sends the address on the AD lines and a command on the C/BE# lines. In this case, the command will indicate that a read operation is requested and that the memory address space is being used.

Clock cycle 2 is used to turn the AD bus lines around. The processor removes the address and disconnects its drivers from the AD lines. The selected target enables its drivers on the AD lines, and fetches the requested data to be placed on the bus during clock cycle 3. It asserts DEVSEL# and maintains it in the asserted state until the end of the transaction.

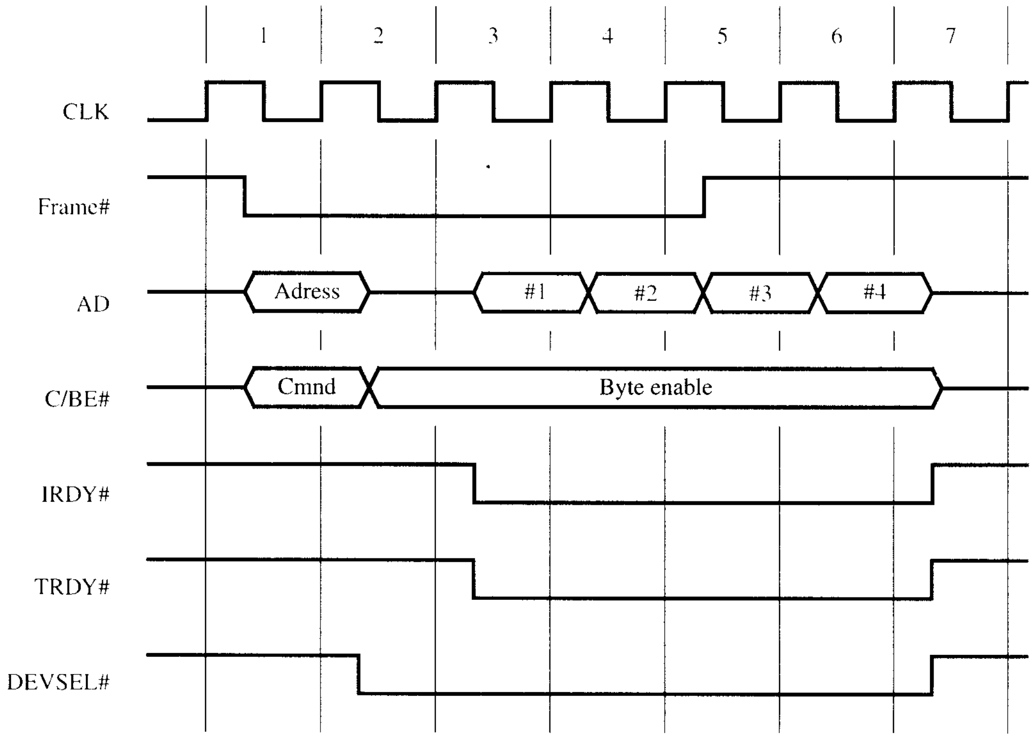


Figure 4.40 A read operation on the PCI bus.

The C/BE# lines, which were used to send a bus command in clock cycle 1, are used for a different purpose during the rest of the transaction. Each of these four lines is associated with one byte on the AD lines. The initiator sets one or more of the C/BE# lines to indicate which byte lines are to be used for transferring data. Assuming that the target is capable of transferring 32 bits at a time, all four C/BE# lines are asserted.

During clock cycle 3, the initiator asserts the initiator ready signal, IRDY#, to indicate that it is ready to receive data. If the target has data ready to send at this time, it asserts target ready, TRDY#, and sends a word of data. The initiator loads the data into its input buffer at the end of the clock cycle. The target sends three more words of data in clock cycles 4 to 6.

The initiator uses the FRAME# signal to indicate the duration of the burst. It negates this signal during the second last word of the transfer. Since it wishes to read four words, the initiator negates FRAME# during clock cycle 5, the cycle in which it receives the third word. After sending the fourth word in clock cycle 6, the target disconnects its drivers and negates DEVSEL# at the beginning of clock cycle 7.

Figure 4.41 gives an example of a more general input transaction. It shows how the IRDY# and TRDY# signals can be used by the initiator and target, respectively, to indicate a pause in the middle of a transaction. The read operation starts the same way as in Figure 4.40, and the first two words are transferred. The target sends the third

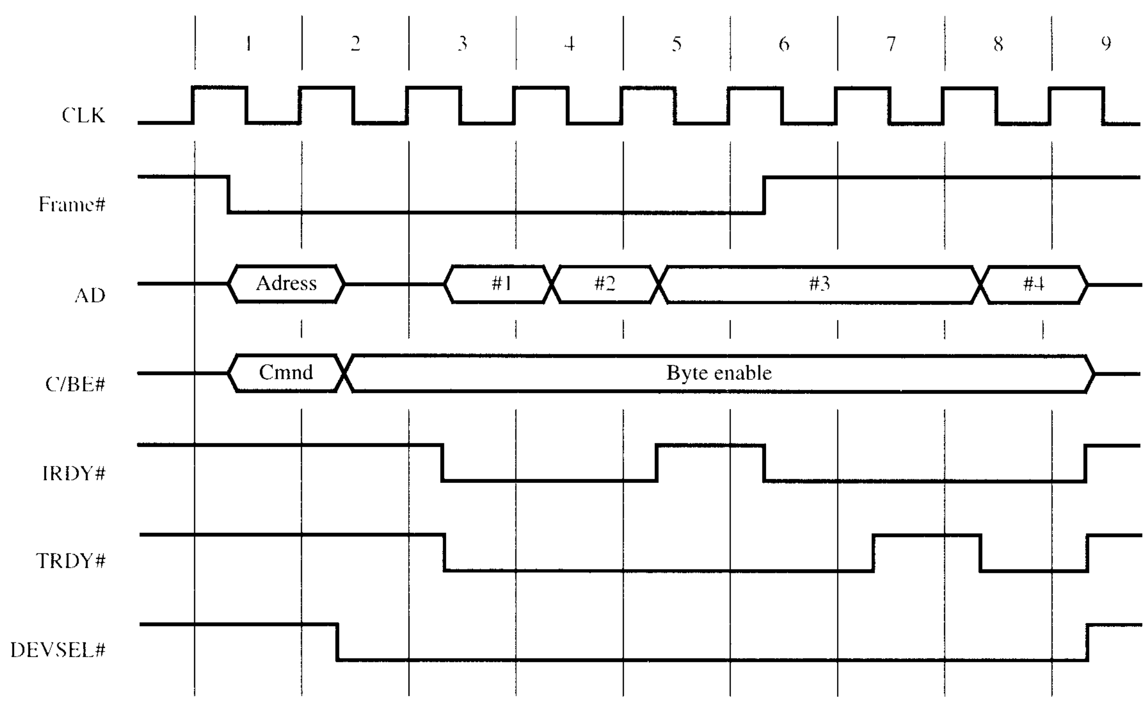


Figure 4.41 A read operation showing the role of IRDY#/TRDY#.

word in cycle 5. However, we assume that the initiator is not able to receive it. Hence, it negates IRDY#. In response, the target maintains the third data word on the AD lines until IRDY# is asserted again. In cycle 6, the initiator asserts IRDY# and loads the data into its input buffer at the end of the clock cycle. At this point, we assume that the target is not ready to transfer the fourth word immediately; hence, it negates TRDY# at the beginning of cycle 7. In cycle 8, it sends the fourth word and asserts TRDY#. Since Frame# was negated with the third data word, the transaction ends after the fourth word has been transferred.

Device Configuration

When an I/O device is connected to a computer, several actions are needed to configure both the device and the software that communicates with it. A typical device interface card for an ISA bus, for example, has a number of jumpers or switches that have to be set by the user to select certain options. Once the device is connected, the software needs to know the address of the device. It may also need to know relevant device characteristics, such as the speed of the transmission link, whether parity bits are used, and so on.

The PCI simplifies this process by incorporating in each I/O device interface a small configuration ROM memory that stores information about that device. The configuration ROMs of all devices are accessible in the configuration address space. The PCI initialization software reads these ROMs whenever the system is powered up or reset. In each case, it determines whether the device is a printer, a keyboard, an Ethernet interface, or a disk controller. It can further learn about various device options and characteristics.

Devices are assigned addresses during the initialization process. This means that during the bus configuration operation, devices cannot be accessed based on their address, as they have not yet been assigned one. Hence, the configuration address space uses a different mechanism. Each device has an input signal called Initialization Device Select, IDSEL#. During a configuration operation, it is this signal, rather than the address applied to the AD inputs of the device, that causes the device to be selected. The motherboard in which device connectors are plugged typically has the IDSEL# pin of each device connected to one of the upper 21 address lines, AD11 to AD31. Hence, a device can be selected for a configuration operation by issuing a configuration command and an address in which the corresponding AD line is set to 1 and the remaining 20 lines set to 0. The lower address lines, AD10 to AD00, are used to specify the type of operation and to access the contents of the device configuration ROM. This arrangement limits the number of I/O devices to 21.

The configuration software scans all 21 locations in the configuration address space to identify which devices are present. Each device may request an address in the I/O space or in the memory space. The device is then assigned an address by writing that address into the appropriate device register. The configuration software also sets such parameters as the device interrupt priority. The PCI bus has four interrupt-request lines. By writing into a device configuration register, the software instructs the device as to which of these lines it can use to request an interrupt. If a device requires initialization, the initialization code is stored in a ROM in the device interface. (This is a different ROM from that used in the configuration process.) The PCI software reads this code and executes it to perform the required initialization.

This process relieves the user from having to be involved in the configuration process. The user simply plugs in the interface board and turns on the power. The software does the rest. The device is ready to use.

The PCI bus has gained great popularity in the PC world. It is also used in many other computers, such as SUNs, to benefit from the wide range of I/O devices for which a PCI interface is available. In the case of some processors, such as the Compaq Alpha, the PCI-processor bridge circuit is built on the processor chip itself, further simplifying system design and packaging.

Electrical Characteristics

The PCI bus has been defined for operation with either a 5- or 3.3-V power supply. The motherboard may be designed to operate with either signaling system. Connectors on expansion boards are designed to ensure that they can be plugged only in a compatible motherboard.

4.7.2 SCSI BUS

The acronym SCSI stands for Small Computer System Interface. It refers to a standard bus defined by the American National Standards Institute (ANSI) under the designation X3.131 [2]. In the original specifications of the standard, devices such as disks are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates up to 5 megabytes/s.

The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years. SCSI-2 and SCSI-3 have been defined, and each has several options. A SCSI bus may have eight data lines, in which case it is called a narrow bus and transfers data one byte at a time. Alternatively, a wide SCSI bus has 16 data lines and transfers data 16 bits at a time. There are also several options for the electrical signaling scheme used. The bus may use single-ended transmission (SE), where each signal uses one wire, with a common ground return for all signals. In another option, differential signaling is used, where a separate return wire is provided for each signal. In this case, two voltage levels are possible. Earlier versions use 5 V (TTL levels) and are known as High Voltage Differential (HVD). More recently, a 3.3 V version has been introduced and is known as Low Voltage Differential (LVD).

Because of these various options, the SCSI connector may have 50, 68, or 80 pins. The maximum transfer rate in commercial devices that are currently available varies from 5 megabytes/s to 160 megabytes/s. The most recent version of the standard is intended to support transfer rates up to 320 megabytes/s, and 640 megabytes/s is anticipated a little later. The maximum transfer rate on a given bus is often a function of the length of the cable and the number of devices connected, with higher rates for a shorter cable and fewer devices. To achieve the top data transfer rate, the bus length is typically limited to 1.6 m for SE signaling and 12 m for LVD signaling. However, manufacturers often provide special bus expanders to connect devices that are farther away. The maximum capacity of the bus is 8 devices for a narrow bus and 16 devices for a wide bus.

Devices connected to the SCSI bus are not part of the address space of the processor in the same way as devices connected to the processor bus. The SCSI bus is connected to the processor bus through a SCSI controller, as shown in Figure 4.38. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa. A packet may contain a block of data, commands from the processor to the device, or status information about the device.

To illustrate the operation of the SCSI bus, let us consider how it may be used with a disk drive. Communication with a disk drive differs substantially from communication with the main memory. As described in Chapter 5, data are stored on a disk in blocks called *sectors*, where each sector may contain several hundred bytes. These data may not necessarily be stored in contiguous sectors. Some sectors may already contain previously stored data; others may be defective and must be skipped. Hence, a read or write request may result in accessing several disk sectors that are not necessarily contiguous. Because of the constraints of the mechanical motion of the disk, there is a long delay, on the order of several milliseconds, before reaching the first sector to or from which data are to be transferred. Then, a burst of data are transferred at high speed. Another delay may ensue, followed by a burst of data. A single read or write request may involve several such bursts. The SCSI protocol is designed to facilitate this mode of operation.

A controller connected to a SCSI bus is one of two types — an *initiator* or a *target*. An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. Clearly, the controller on the processor side, such as the SCSI controller in Figure 4.38, must be able to operate as an initiator. The disk controller operates as a target. It carries out the commands it receives from the initiator. The initiator establishes a *logical connection* with the intended target.

Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data. While a particular connection is suspended, other devices can use the bus to transfer information. This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.

Data transfers on the SCSI bus are always controlled by the target controller. To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it. Then the controller starts a data transfer operation to receive a command from the initiator.

Let us examine a complete disk read operation as an example. In this discussion, even though we refer to the initiator controller as taking certain actions, it should be clear that it performs these actions after receiving appropriate commands from the processor. Assume that the processor wishes to read a block of data from a disk drive and that these data are stored in two disk sectors that are not contiguous. The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

1. The SCSI controller, acting as an initiator, contends for control of the bus.
2. When the initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
3. The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.
4. The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
5. The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control of the bus. After it wins arbitration, it reselects the initiator controller, thus restoring the suspended connection.
6. The target transfers the contents of the data buffer to the initiator and then suspends the connection again. Data are transferred either 8 or 16 bits in parallel, depending on the width of the bus.
7. The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator, as before. At the end of this transfer, the logical connection between the two controllers is terminated.
8. As the initiator controller receives the data, it stores them into the main memory using the DMA approach.
9. The SCSI controller sends an interrupt to the processor to inform it that the requested operation has been completed.

This scenario shows that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. In this context, a "higher level" means that the messages refer to operations that may require several steps to complete, depending on the device. Neither the processor nor the SCSI controller need be aware of the details of operation of the particular device involved in a data transfer. In the preceding example, the processor need not be involved in the disk seek operations.

The SCSI bus standard defines a wide range of control messages that can be exchanged between the controllers to handle different types of I/O devices. Messages are also defined to deal with various error or failure conditions that might arise during device operation or data transfer.

Bus Signals

We now describe the operation of the SCSI bus from the hardware point of view. The bus signals are summarized in Table 4.4. For simplicity we show the signals for a narrow bus (8 data lines). Note that all signal names are preceded by a minus sign. This indicates that the signals are active, or that a data line is equal to 1, when they are in the low-voltage state. The bus has no address lines. Instead, the data lines are used to identify the bus controllers involved during the selection or reselection process and during bus arbitration. For a narrow bus, there are eight possible controllers, numbered 0 through 7, and each is associated with the data line that has the same number. A wide bus accommodates up to 16 controllers. A controller places its own address or the address of another controller on the bus by activating the corresponding data line. Thus, it is possible to have more than one address on the bus at the same time, as in the arbitration process we describe next. Once a connection is established between two

Table 4.4 The SCSI bus signals

Category	Name	Function
Data	−DB(0) to −DB(7)	Data lines: Carry one byte of information during the information transfer phase and identify device during arbitration, selection and reselection phases
	−DB(P)	Parity bit for the data bus
Phase	−BSY	Busy: Asserted when the bus is not free
	−SEL	Selection: Asserted during selection and reselection
Information type	−C/D	Control/Data: Asserted during transfer of control information (command, status or message)
	−MSG	Message: indicates that the information being transferred is a message
Handshake	−REQ	Request: Asserted by a target to request a data transfer cycle
	−ACK	Acknowledge: Asserted by the initiator when it has completed a data transfer operation
Direction of transfer	−I/O	Input/Output: Asserted to indicate an input operation (relative to the initiator)
Other	−ATN	Attention: Asserted by an initiator when it wishes to send a message to a target
	−RST	Reset: Causes all device controls to disconnect from the bus and assume their start-up state

controllers, there is no further need for addressing, and the data lines are used to carry data.

The main phases involved in the operation of the SCSI bus are arbitration, selection, information transfer, and reselection. We now examine each of these phases.

Arbitration

The bus is free when the −BSY signal is in the inactive (high-voltage) state. Any controller can request the use of the bus while it is in this state. Since two or more controllers may generate such a request at the same time, an arbitration scheme must be implemented. A controller requests the bus by asserting the −BSY signal and by asserting its associated data line to identify itself. The SCSI bus uses a simple distributed arbitration scheme. It is illustrated by the example in Figure 4.42, in which controllers 2 and 6 request the use of the bus simultaneously.

Each controller on the bus is assigned a fixed priority, with controller 7 having the highest priority. When −BSY becomes active, all controllers that are requesting the bus examine the data lines and determine whether a higher-priority device is requesting the bus at the same time. The controller using the highest-numbered line realizes that it has won the arbitration process. All other controllers disconnect from the bus and wait for −BSY to become inactive again.

In Figure 4.42, we have assumed that controller 6 is an initiator that wishes to establish a connection to controller 5. After winning arbitration, controller 6 proceeds to the selection phase, in which it identifies the target.

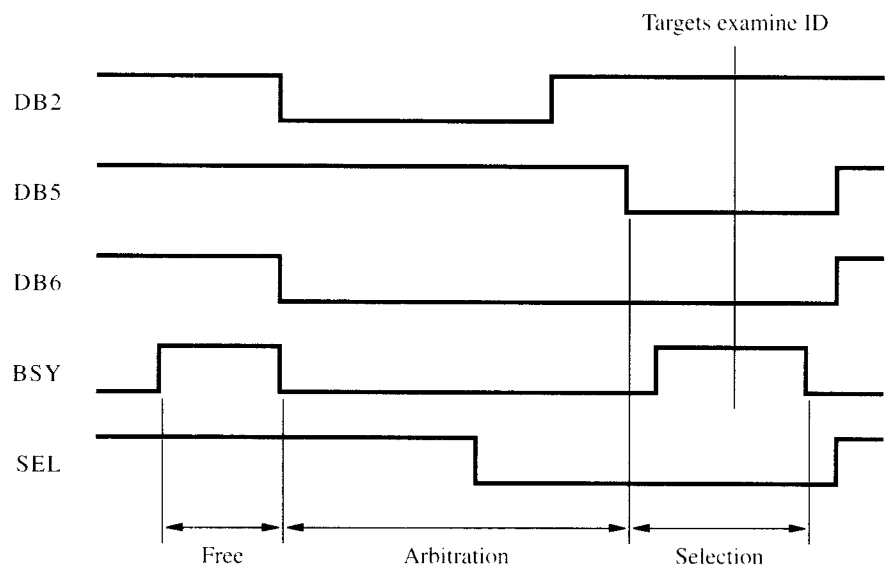


Figure 4.42 Arbitration and selection on the SCSI bus. Device 6 wins arbitration and selects device 2.

Selection

Having won arbitration, controller 6 continues to assert −BSY and −DB6 (its address). It indicates that it wishes to select controller 5 by asserting the −SEL and then the −DB5 lines. Any other controller that may have been involved in the arbitration phase, such as controller 2 in the figure, must stop driving the data lines once the −SEL line becomes active, if it has not already done so. After placing the address of the target controller on the bus, the initiator releases the −BSY line.

The selected target controller responds by asserting −BSY. This informs the initiator that the connection it is requesting has been established, so that it may remove the address information from the data lines. The selection process is now complete, and the target controller (controller 5) is asserting −BSY. From this point on, controller 5 has control of the bus, as required for the information transfer phase.

Information Transfer

The information transferred between two controllers may consist of commands from the initiator to the target, status responses from the target to the initiator, or data being transferred to or from the I/O device. Handshake signaling is used to control information transfers in the same manner as described in Section 4.5.2, with the target controller taking the role of the bus master. The −REQ and −ACK signals replace the Master-ready and Slave-ready signals in Figures 4.26 and 4.27. The target asserts −I/O during an input operation (target to initiator), and it asserts −C/D to indicate that the information being transferred is either a command or a status response rather than data.

We should point out that high-speed versions of the SCSI bus use a technique known as double-edge clocking or Double Transitions (DT). In Figures 4.26 and 4.27, each data transfer requires a high-to-low transition followed by a low-to-high transition on the two handshake signals. Double-edge clocking means that data are transferred on both the rising and falling edges of these signals, thus doubling the transfer rate.

At the end of the transfer, the target controller releases the −BSY signal, thus freeing the bus for use by other devices. Later, it may reestablish the connection to the initiator controller when it is ready to transfer more data. This is done in the reselection operation described next.

Reselection

When a logical connection is suspended and the target is ready to restore it, the target must first gain control of the bus. It starts an arbitration cycle, and after winning arbitration, it selects the initiator controller in exactly the same manner as described above. But with the roles of the target and initiator reversed, the initiator is now asserting −BSY. Before data transfer can begin, the initiator must hand control over to the target. This is achieved by having the target controller assert −BSY after selecting the initiator. Meanwhile, the initiator waits for a short period after being selected to make sure that the target has asserted −BSY, and then it releases the −BSY line. The connection between the two controllers has now been reestablished, with the target in control of the bus as required for data transfer to proceed.

The bus signaling scheme described above provides the mechanisms needed for two controllers to establish a logical connection and exchange messages. The connection may be suspended and reestablished at any time. The SCSI standard defines the structure and contents of various types of packets that the controllers exchange to handle different situations. The initiator uses these packets to send the commands it receives from the processor to the target. The target responds with status information, and data transfer operations. The latter are controlled by the target, because it is the target that knows when data are available, when to suspend and reestablish connections, etc.

Additional information on the SCSI bus and various SCSI products is available on the web from the standards committee [2].

4.7.3 UNIVERSAL SERIAL BUS (USB)

The synergy between computers and communications is at the heart of today's information technology revolution. A modern computer system is likely to involve a wide variety of devices such as keyboards, microphones, cameras, speakers, and display devices. Most computers also have a wired or wireless connection to the Internet. A key requirement in such an environment is the availability of a simple, low-cost mechanism to connect these devices to the computer, and an important recent development in this regard is the introduction of the Universal Serial Bus (USB) [3]. This is an industry standard developed through a collaborative effort of several computer and communications companies, including Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, Nortel Networks, and Philips.

The USB supports two speeds of operation, called low-speed (1.5 megabits/s) and full-speed (12 megabits/s). The most recent revision of the bus specification (USB 2.0) introduced a third speed of operation, called high-speed (480 megabits/s). The USB is quickly gaining acceptance in the market place, and with the addition of the high-speed capability it may well become the interconnection method of choice for most computer devices.

The USB has been designed to meet several key objectives:

- Provide a simple, low-cost, and easy to use interconnection system that overcomes the difficulties due to the limited number of I/O ports available on a computer
- Accommodate a wide range of data transfer characteristics for I/O devices, including telephone and Internet connections
- Enhance user convenience through a "plug-and-play" mode of operation

We will elaborate on these objectives before discussing the technical details of the USB.

Port Limitation

The parallel and serial ports described in Section 4.6 provide a general-purpose point of connection through which a variety of low- to medium-speed devices can be connected to a computer. For practical reasons, only a few such ports are provided in a typical computer. To add new ports, a user must open the computer box to gain access to the internal expansion bus and install a new interface card. The user may also need to know how to configure the device and the software. An objective of the USB is to make it possible to add many devices to a computer system at any time, without opening the computer box.

Device Characteristics

The kinds of devices that may be connected to a computer cover a wide range of functionality. The speed, volume, and timing constraints associated with data transfers to and from such devices vary significantly.

In the case of a keyboard, one byte of data is generated every time a key is pressed, which may happen at any time. These data should be transferred to the computer promptly. Since the event of pressing a key is not synchronized to any other event in a computer system, the data generated by the keyboard are called *asynchronous*. Furthermore, the rate at which the data are generated is quite low. It is limited by the speed of the human operator to about 100 bytes per second, which is less than 1000 bits per second.

A variety of simple devices that may be attached to a computer generate data of a similar nature — low speed and asynchronous. Computer mice and the controls and manipulators used with video games are good examples.

Let us consider a different source of data. Many computers have a microphone either externally attached or built in. The sound picked up by the microphone produces an analog electrical signal, which must be converted into a digital form before it can be handled by the computer. This is accomplished by sampling the analog signal periodically. For each sample, an analog-to-digital (A/D) converter generates an n-bit number representing the magnitude of the sample. The number of bits, n, is selected based on the desired precision with which to represent each sample. Later, when these data are sent to a speaker, a digital-to-analog (D/A) converter is used to restore the original analog signal from the digital format.

The sampling process yields a continuous stream of digitized samples that arrive at regular intervals, synchronized with the sampling clock. Such a data stream is called *isochronous*, meaning that successive events are separated by equal periods of time.

A signal must be sampled quickly enough to track its highest-frequency components. In general, if the sampling rate is s samples per second, the maximum frequency component that will be captured by the sampling process is $s/2$. For example, human speech can be captured adequately with a sampling rate of 8 kHz, which will record sound signals having frequencies up to 4 kHz. For a higher-quality sound, as needed in a music system, higher sampling rates are used. A standard rate for digital sound is 44.1 kHz. Each sample is represented by 4 bytes of data to accommodate the wide range in sound volume (dynamic range) that is necessary for high-quality sound reproduction. This yields a data rate of about 1.4 megabits/s.

An important requirement in dealing with sampled voice or music is to maintain precise timing in the sampling and replay processes. A high degree of jitter (variability in sample timing) is unacceptable. Hence, the data transfer mechanism between a computer and a music system must maintain consistent delays from one sample to the next. Otherwise, complex buffering and retiming circuitry would be needed. On the other hand, occasional errors or missed samples can be tolerated. They either go unnoticed by the listener or they may cause an unobtrusive click. No sophisticated mechanisms are needed to ensure perfectly correct data delivery.

Data transfers for images and video have similar requirements, but at much higher data transfer bandwidth. The term bandwidth refers to the total data transfer capacity of a communications channel, measured in a suitable unit such as bits or bytes per second. To maintain the picture quality of commercial television, an image should be represented by about 160 kilobytes and transmitted 30 times per second, for a total bandwidth of 44 megabits/s. Higher-quality images, as in HDTV (High Definition TV), require higher rates.

Large storage devices such as hard disks and CD-ROMs present different requirements. These devices are part of the computer's memory hierarchy, as will be discussed in Chapter 5. Their connection to the computer must provide a data transfer bandwidth of at least 40 or 50 megabits/s. Delays on the order of a millisecond are introduced by the disk mechanism. Hence, a small additional delay introduced while transferring data to or from the computer is not important, and jitter is not an issue.

Plug-and-Play

As computers become part of everyday life, their existence should become increasingly transparent. For example, when operating a home theater system, which includes at least one computer, the user should not find it necessary to turn the computer off or to restart the system to connect or disconnect a device.

The *plug-and-play* feature means that a new device, such as an additional speaker, can be connected at any time while the system is operating. The system should detect the existence of this new device automatically, identify the appropriate device-driver software and any other facilities needed to service that device, and establish the appropriate addresses and logical connections to enable them to communicate.

The plug-and-play requirement has many implications at all levels in the system, from the hardware to the operating system and the applications software. One of the primary objectives of the design of the USB has been to provide a plug-and-play capability.

USB Architecture

The discussion above points to the need for an interconnection system that combines low cost, flexibility, and high data-transfer bandwidth. Also, I/O devices may be located at some distance from the computer to which they are connected. The requirement for high bandwidth would normally suggest a wide bus that carries 8, 16, or more bits in parallel. However, a large number of wires increases cost and complexity and is inconvenient to the user. Also, it is difficult to design a wide bus that carries data for a long distance because of the data skew problem discussed in Section 4.5.2. The amount of skew increases with distance and limits the data rate that can be used.

A serial transmission format has been chosen for the USB because a serial bus satisfies the low-cost and flexibility requirements. Clock and data information are encoded together and transmitted as a single signal. Hence, there are no limitations on clock frequency or distance arising from data skew. Therefore, it is possible to provide a high data transfer bandwidth by using a high clock frequency. As pointed out earlier, the USB offers three bit rates, ranging from 1.5 to 480 megabits/s, to suit the needs of different I/O devices.

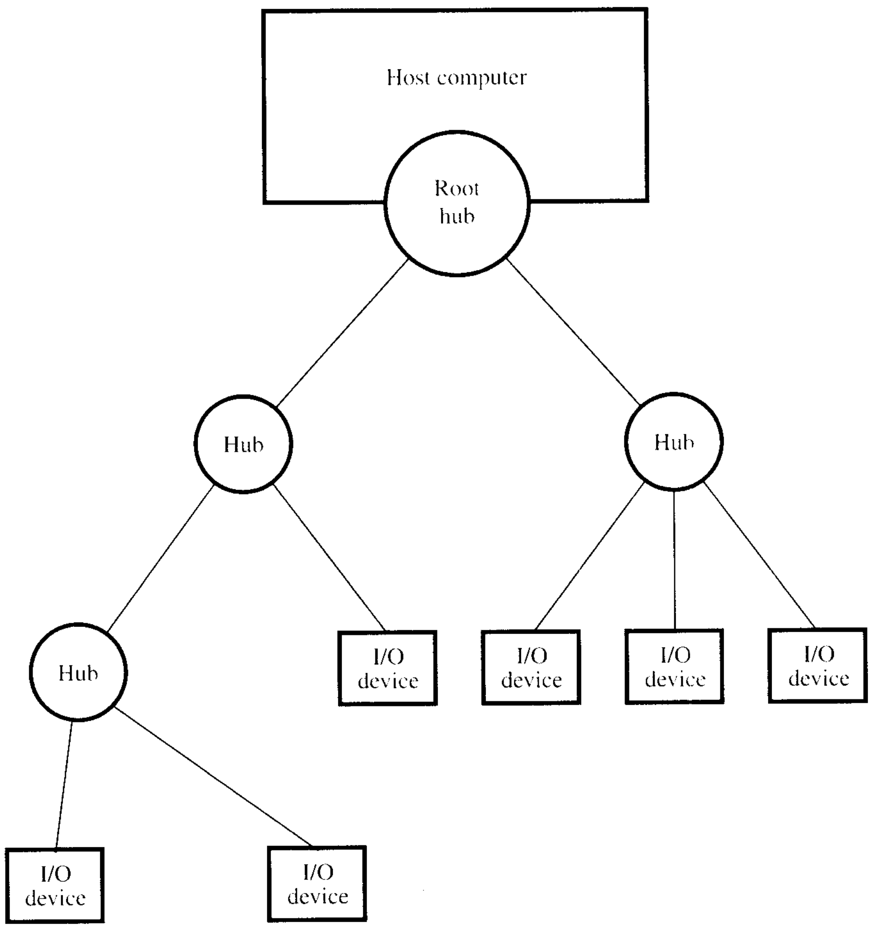


Figure 4.43 Universal Serial Bus tree structure.

To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure shown in Figure 4.43. Each node of the tree has a device called a *hub*, which acts as an intermediate control point between the host and the I/O devices. At the root of the tree, a *root hub* connects the entire tree to the host computer. The leaves of the tree are the I/O devices being served (for example, keyboard, Internet connection, speaker, or digital TV), which are called *functions* in USB terminology. For consistency with the rest of the discussion in the book, we will refer to these devices as I/O devices.

The tree structure enables many devices to be connected while using only simple point-to-point serial links. Each hub has a number of ports where devices may be connected, including other hubs. In normal operation, a hub copies a message that it receives from its upstream connection to all its downstream ports. As a result, a message sent by the host computer is broadcast to all I/O devices, but only the addressed device will respond to that message. In this respect, the USB functions in the same way as the bus in Figure 4.1. However, unlike the bus in Figure 4.1, a message from an I/O device is sent only upstream towards the root of the tree and is not seen by other devices. Hence, the USB enables the host to communicate with the I/O devices, but it does not enable these devices to communicate with each other.

Note how the tree structure helps meet the USB's design objectives. The tree makes it possible to connect a large number of devices to a computer through a few ports (the root hub). At the same time, each I/O device is connected through a serial point-to-point connection. This is an important consideration in facilitating the plug-and-play feature, as we will see shortly. Also, because of electrical transmission considerations, serial data transmission on such a connection is much easier than parallel transmission on buses of the form represented in Figure 4.1. Much higher data rates and longer cables can be used.

The USB operates strictly on the basis of polling. A device may send a message only in response to a poll message from the host. Hence, upstream messages do not encounter conflicts or interfere with each other, as no two devices can send messages at the same time. This restriction allows hubs to be simple, low-cost devices.

The mode of operation described above is observed for all devices operating at either low speed or full speed. However, one exception has been necessitated by the introduction of high-speed operation in USB version 2.0. Consider the situation in Figure 4.44. Hub A is connected to the root hub by a high-speed link. This hub serves

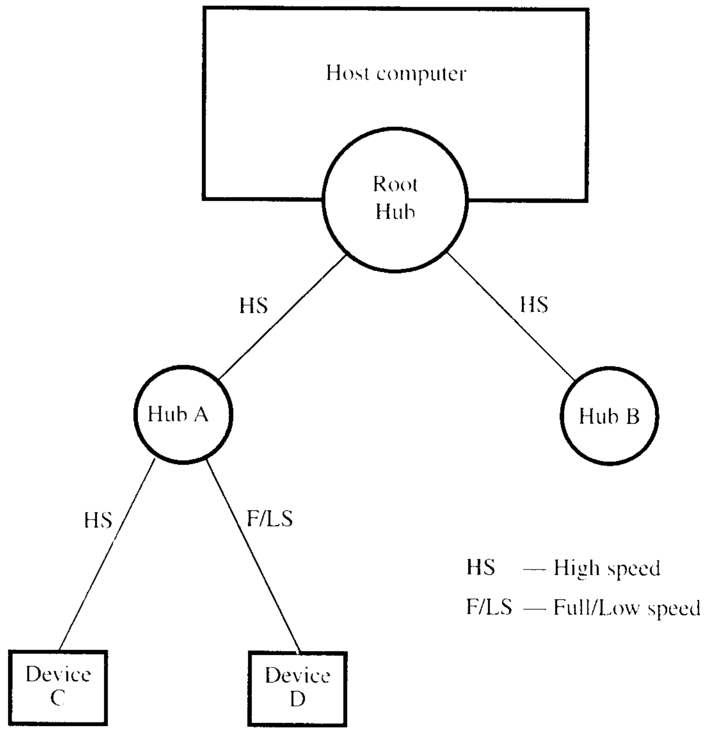


Figure 4.44 Split bus operation.

one high-speed device, C, and one low-speed device, D. Normally, a message to device D would be sent at low speed from the root hub. At 1.5 megabits/s, even a short message takes several tens of microseconds. For the duration of this message, no other data transfers can take place, thus reducing the effectiveness of the high-speed links and introducing unacceptable delays for high-speed devices. To mitigate this problem, the USB protocol requires that a message transmitted on a high-speed link is always transmitted at high speed, even when the ultimate receiver is a low-speed device. Hence, a message intended for device D is sent at high speed from the root hub to hub A, then forwarded at low speed to device D. The latter transfer will take a long time, during which high-speed traffic to other nodes is allowed to continue. For example, the root hub may exchange several messages with device C while the low-speed message is being sent from hub A to device D. During this period, the bus is said to be split between high-speed and low-speed traffic. The message to device D is preceded and followed by special commands to hub A to start and end the split-traffic mode of operation, respectively.

The USB standard specifies the hardware details of USB interconnections as well as the organization and requirements of the host software. The purpose of the USB software is to provide bidirectional communication links between application software and I/O devices. These links are called *pipes*. Any data entering at one end of a pipe is delivered at the other end. Issues such as addressing, timing, or error detection and recovery are handled by the USB protocols.

We mentioned in Section 4.2.6 that the software that transfers data to or from a given I/O device is called the device driver for that device. The device drivers depend on the characteristics of the devices they support. Hence, a more precise description of the USB pipe is that it connects an I/O device to its device driver. It is established when a device is connected and assigned a unique address by the USB software. Once established, data may flow through the pipe at any time.

We will now examine how devices are addressed on the USB. Then we will discuss the various ways in which data transfer can take place.

Addressing

In earlier discussions of input and output operations, we explained that I/O devices are normally identified by assigning them a unique memory address. In fact, a device usually has several addressable locations to enable the software to send and receive control and status information and to transfer data.

When a USB is connected to a host computer, its root hub is attached to the processor bus, where it appears as a single device. The host software communicates with individual devices attached to the USB by sending packets of information, which the root hub forwards to the appropriate device in the USB tree.

Each device on the USB, whether it is a hub or an I/O device, is assigned a 7-bit address. This address is local to the USB tree and is not related in any way to the addresses used on the processor bus. A hub may have any number of devices or other hubs connected to it, and addresses are assigned arbitrarily. When a device is first connected to a hub, or when it is powered on, it has the address 0. The hardware of the hub to which this device is connected is capable of detecting that the device has been connected, and it records this fact as part of its own status information. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected. When the host is informed that a new device has been connected, it uses a sequence of commands to send a reset signal on the corresponding hub port, read information from the device about its capabilities, send configuration information to the device, and assign the device a unique USB address. Once this sequence is completed the device begins normal operation and responds only to the new address.

The initial connection procedure just described is a key feature that helps give the USB its plug-and-play capability. The host software is in complete control of the procedure. It is able to sense that a device has been connected, to read information about the device, which is typically stored in a small read-only memory in the device hardware, to send commands that will configure the device by enabling or disabling certain features or capabilities, and finally to assign a unique USB address to the device. The only action required from the user is to plug the device into a hub port and to turn on its power switch

When a device is powered off, a similar procedure is followed. The corresponding hub reports this fact to the USB system software, which in turn updates its tables. Of course, if the device that has been disconnected is itself a hub, all devices connected through that hub must also be recorded as disconnected. The USB software must maintain a complete picture of the bus topology and the connected devices at all times.

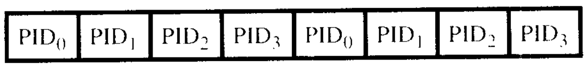
Locations in the device to or from which data transfer can take place, such as status, control, and data registers, are called *endpoints*. They are identified by a 4-bit number. Actually, each 4-bit value identifies a pair of endpoints, one for input and one for output. Thus, a device may have up to 16 input/output pairs of endpoints. A USB pipe, which is a bidirectional data transfer channel, is connected to one such pair. The pipe connected to endpoints number 0 exists all the time, including immediately after a device is powered on or reset. This is the control pipe that the USB software uses in the power-on procedure. As part of that procedure, other pipes using other endpoint pairs may be established, depending on the needs and complexity of the device. The 4-bit endpoint number is part of the addressing information sent by the host, as we will see shortly.

USB Protocols

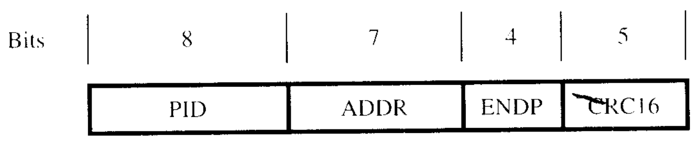
All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information. There are many types of packets that perform a variety of control functions. We illustrate the operation of the USB by giving a few examples of the key packet types and show how they are used.

The information transferred on the USB can be divided into two broad categories: control and data. Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error. Data packets carry information that is delivered to a device. For example, input and output data are transferred inside data packets.

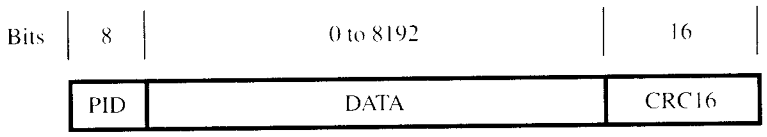
A packet consists of one or more fields containing different kinds of information. The first field of any packet is called the packet identifier, PID, which identifies the type of that packet. There are four bits of information in this field, but they are transmitted twice. The first time they are sent with their true values, and the second time with each



(a) Packet identifier field



(b) Token packet, IN or OUT



(c) Data packet

Figure 4.45 USB packet formats.

bit complemented, as shown in Figure 4.45*a*. This enables the receiving device to verify that the PID byte has been received correctly.

The four PID bits identify one of 16 different packet types. Some control packets, such as ACK (Acknowledge), consist only of the PID byte. Control packets used for controlling data transfer operations are called token packets. They have the format shown in Figure 4.45*b*. A token packet starts with the PID field, using one of two PID values to distinguish between an IN packet and an OUT packet, which control input and output transfers, respectively. The PID field is followed by the 7-bit address of a device and the 4-bit endpoint number within that device. The packet ends with 5 bits for error checking, using a method called cyclic redundancy check (CRC). The CRC bits are computed based on the contents of the address and endpoint fields. By performing an inverse computation, the receiving device can determine whether the packet has been received correctly.

Data packets, which carry input and output data, have the format shown in Figure 4.45*c*. The packet identifier field is followed by up to 8192 bits of data, then 16 error-checking bits. Three different PID patterns are used to identify data packets, so that data packets may be numbered 0, 1, or 2. Note that data packets do not carry a

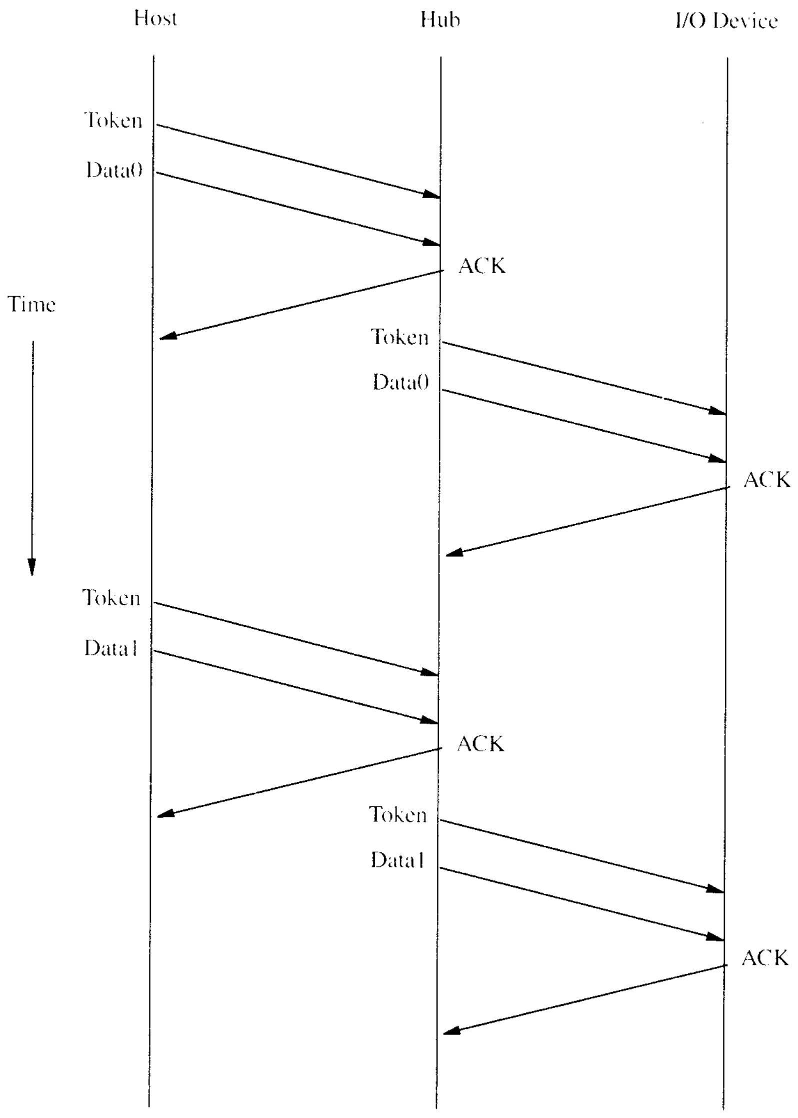


Figure 4.46 An output transfer.

device address or an endpoint number. This information is included in the IN or OUT token packet that initiates the transfer.

Consider an output device connected to a USB hub, which in turn is connected to a host computer. An example of an output operation is shown in Figure 4.46. The host computer sends a token packet of type OUT to the hub, followed by a data packet containing the output data. The PID field of the data packet identifies it as data packet number 0. The hub verifies that the transmission has been error free by checking the error control bits, then sends an acknowledgment packet (ACK) back to the host. The hub forwards the token and data packets downstream. All I/O devices receive this sequence of packets, but only the device that recognizes its address in the token packet accepts the data in the packet that follows. After verifying that transmission has been error free, it sends an ACK packet to the hub.

Successive data packets on a full-speed or low-speed pipe carry the numbers 0 and 1, alternately. This simplifies recovery from transmission errors. If a token, data, or acknowledgment packet is lost as a result of a transmission error, the sender resends the entire sequence. By checking the data packet number in the PID field, the receiver can detect and discard duplicate packets. High-speed data packets are sequentially numbered 0, 1, 2, 0, and so on.

Input operations follow a similar procedure. The host sends a token packet of type IN containing the device address. In effect, this packet is a poll asking the device to send any input data it may have. The device responds by sending a data packet followed by an ACK. If it has no data ready, it responds by sending a negative acknowledgment (NAK) instead.

In earlier discussion, we pointed out that a bus that has a mix of full/low-speed links and high-speed links uses the split-traffic mode of operation in order not to delay messages on high-speed links. In such cases, an IN or an OUT packet intended for a full- or low-speed device is preceded by a special control packet that starts the split-traffic mode.

This discussion should give the reader an idea about the data transfer protocols used on the USB. There are many different ways in which such transactions take place and many protocol rules governing the behavior of the devices involved. A detailed description of these protocols can be found in the USB specification document [3].

Isochronous Traffic on USB

One of the key objectives of the USB is to support the transfer of isochronous data, such as sampled voice, in a simple manner. Devices that generate or receive isochronous data require a time reference to control the sampling process. To provide this reference, transmission over the USB is divided into *frames* of equal length. A frame is 1 ms long for low- and full-speed data. The root hub generates a Start Of Frame control packet (SOF) precisely once every 1 ms to mark the beginning of a new frame.

The arrival of an SOF packet at any device constitutes a regular clock signal that the device can use for its own purposes. To assist devices that may need longer periods of time, the SOF packet carries an 11-bit frame number, as shown in Figure 4.47*a*. Following each SOF packet, the host carries out input and output transfers for isochronous devices. This means that each device will have an opportunity for an input or output transfer once every 1 ms.

The main requirement for isochronous traffic is consistent timing. An occasional error can be tolerated. Hence, there is no need to retransmit packets that are lost or to send acknowledgments. Figure 4.47*b* shows the first two transmissions following SOF. A control packet carrying device address 3 is followed by data for that device. This may be input or output data, depending on whether the control packet is an IN or OUT control packet. There is no acknowledgment packet. The next transmission sequence is for device 7.

[illegible]
read. Why is this important?

4.2 Write a program that displays the contents of 10 bytes of the main memory in hexadecimal format on a video display. Use either the assembler instructions of a processor of your choice or pseudo-instructions. Start at location LOC in the memory, and use two hex characters per byte. The contents of successive bytes should be separated by a space.

4.3 The address bus of a computer has 16 address lines, A_{15-0}. If the address assigned to one device is $7CA4_{16}$ and the address decoder for that device ignores lines A_8 and A_9, what are all the addresses to which this device will respond?

4.4 What is the difference between a subroutine and an interrupt-service routine?

4.5 The discussion in this chapter assumed that interrupts are not acknowledged until the current machine instruction completes execution. Consider the possibility of suspending operation of the processor in the middle of executing an instruction in order to acknowledge an interrupt. Discuss the difficulties that may arise.