Note on USB 2.0 Bit Rate: This specification draft calls out a data rate of 480Mb/s. This is the target rate for which the Electrical Working Group is designing and prototyping; this rate needs to be confirmed with completed validation of prototype IC's operating on test boards.

Chapter 5
USB Data Flow Model

This chapter presents information about how data is moved across the USB. The information in this chapter affects all implementers. The information presented is at a level above the signaling and protocol definitions of the system. Consult Chapter 7 and Chapter 8 for more details about their respective parts of the USB system. This chapter provides framework information that is further expanded in Chapters 9 through 11. All implementers should read this chapter so they understand the key concepts of the USB.

5.1 Implementer Viewpoints

The USB provides communication services between a host and attached USB devices. However, the simple view an end user sees of attaching one or more USB devices to a host, as in Figure 5-1, is in fact a little more complicated to implement than is indicated by the figure. Different views of the system are required to explain specific USB requirements from the perspective of different implementers. Several important concepts and features must be supported to provide the end user with the reliable operation demanded from today's personal computers. The USB is presented in a layered fashion to ease explanation and allow implementers of particular USB products to focus on the details related to their product.

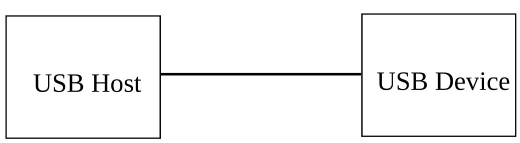


Figure 5-1. Simple USB Host/Device View

Figure 5-2 shows a deeper overview of the USB, identifying the different layers of the system that will be described in more detail in the remainder of the specification. In particular, there are four focus implementation areas:

- USB Physical Device: A piece of hardware on the end of a USB cable that performs some useful end user function.
- Client Software: Software that executes on the host, corresponding to a USB device. This client software is typically supplied with the operating system or provided along with the USB device.
- USB System Software: Software that supports the USB in a particular operating system. The USB System Software is typically supplied with the operating system, independently of particular USB devices or client software.

- USB Host Controller (Host Side Bus Interface): The hardware and software that allows USB devices to be attached to a host.

There are shared rights and responsibilities between the four USB system components. The remainder of this specification describes the details required to support robust, reliable communication flows between a function and its client.

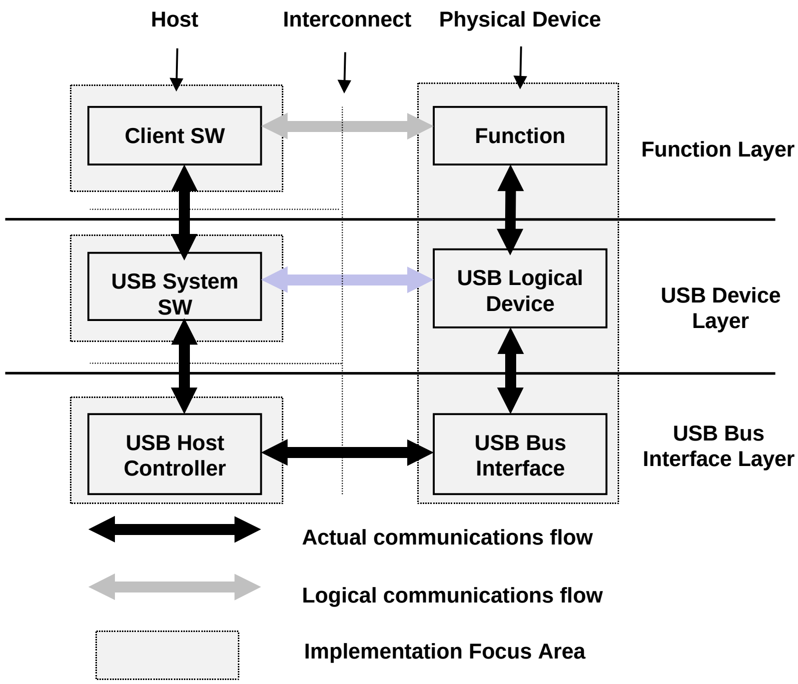


Figure 5-2. USB Implementation Areas

As shown in Figure 5-2, the simple connection of a host to a device requires interaction between a number of layers and entities. The USB Bus Interface layer provides physical/signaling/packet connectivity between the host and a device. The USB Device Layer is the view the USB System Software has for performing generic USB operations with a device. The Function Layer provides additional capabilities to the host via an appropriate matched client software layer. The USB Device and Function layers each have a view of logical communication within their layer that actually uses the USB Bus Interface Layer to accomplish data transfer.

The physical view of USB communication as described in Chapters 6, 7, and 8 is related to the logical communication view presented in Chapters 9 and 10. This chapter describes those key concepts that affect USB implementers and should be read by all before proceeding to the remainder of the specification to find those details most relevant to their product.

To describe and manage USB communication, the following concepts are important:

- Bus Topology: Section 5.2 presents the primary physical and logical components of the USB and how they interrelate.
- Communication Flow Models: Sections 5.3 through 5.8 describe how communication flows between the host and devices through the USB and defines the four USB transfer types.
- Bus Access Management: Section 5.11 describes how bus access is managed within the host to support a broad range of communication flows by USB devices.

- Special Consideration for Isochronous Transfers: Section 5.12 presents features of the USB specific to devices requiring isochronous data transfers. Device implementers for non-isochronous devices do not need to read Section 5.12.

5.2 Bus Topology

There are four main parts to USB topology:

- Host and Devices: The primary components of a USB system.
- Physical Topology: How USB elements are connected.
- Logical Topology: The roles and responsibilities of the various USB elements and how the USB appears from the perspective of the host and a device.
- Client Software-to-function Relationships: How client software and its related function interfaces on a USB device view each other.

5.2.1 USB Host

The host's logical composition is shown in Figure 5-3, and includes the following:

- USB Host Controller
- Aggregate USB System Software (USB Driver, Host Controller Driver, and host software)
- Client.

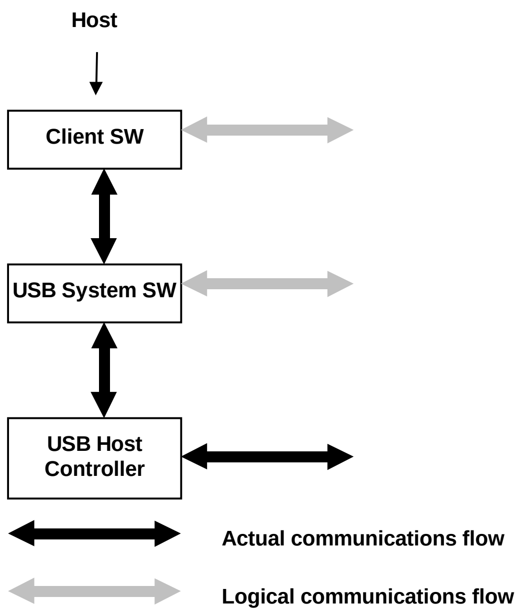


Figure 5-3. Host Composition

The USB host occupies a unique position as the coordinating entity for the USB. In addition to its special physical position, the host has specific responsibilities with regard to the USB and its attached devices. The host controls all access to the USB. A USB device gains access to the bus only by being granted access by the host. The host is also responsible for monitoring the topology of the USB.

For a complete discussion of the host and its duties, refer to Chapter 10.

5.2.2 USB Devices

A USB physical device's logical composition is shown in Figure 5-4, and includes the following:

- USB bus interface
- USB logical device
- Function.

USB physical devices provide additional functionality to the host. The types of functionality provided by USB devices vary widely. However, all USB logical devices present the same basic interface to the host. This allows the host to manage the USB-relevant aspects of different USB devices in the same manner.

To assist the host in identifying and configuring USB devices, each device carries and reports configuration-related information. Some of the information reported is common among all logical devices. Other information is specific to the functionality provided by the device. The detailed format of this information varies, depending on the device class of the device.

For a complete discussion of USB devices, refer to Chapter 9.

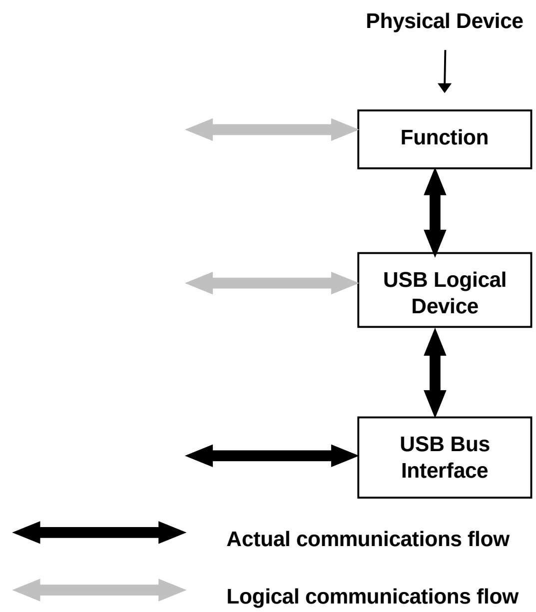


Figure 5-4. Physical Device Composition

5.2.3 Physical Bus Topology

Devices on the USB are physically connected to the host via a tiered star topology, as illustrated in Figure 5-5. USB attachment points are provided by a special class of USB device known as a hub. The additional attachment points provided by a hub are called ports. A host includes an embedded hub called the root hub. The host provides one or more attachment points via the root hub. USB devices that provide additional functionality to the host are known as functions. To prevent circular attachments, a tiered ordering is imposed on the star topology of the USB. This results in the tree-like configuration illustrated in Figure 5-5.

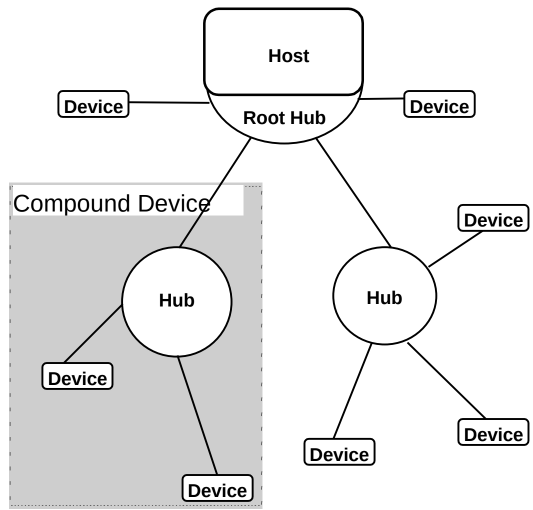


Figure 5-5. USB Physical Bus Topology

Multiple functions may be packaged together in what appears to be a single physical device. For example, a keyboard and a trackball might be combined in a single package. Inside the package, the individual functions are permanently attached to a hub and it is the internal hub that is connected to the USB. When multiple functions are combined with a hub in a single package, they are referred to as a compound device. From the host's perspective, a compound device is the same as a separate hub with multiple functions attached. Figure 5-5 also illustrates a compound device.

5.2.4 Logical Bus Topology

While devices physically attach to the USB in a tiered, star topology, the host communicates with each logical device as if it were directly connected to the root port. This creates the logical view illustrated in Figure 5-6 that corresponds to the physical topology shown in Figure 5-5. Hubs are logical devices also, but are not shown in Figure 5-6 to simplify the picture. Even though most host/logical device activities use this logical perspective, the host maintains an awareness of the physical topology to support processing the removal of hubs. When a hub is removed, all of the devices attached to the hub must be removed from the host's view of the logical topology. A more complete discussion of hubs can be found in Chapter 11.

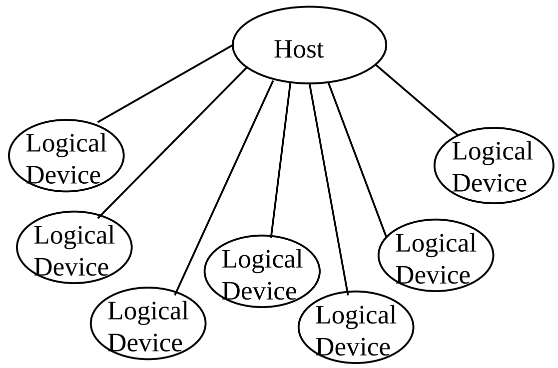


Figure 5-6. USB Logical Bus Topology

5.2.5 Client Software-to-function Relationship

Even though the physical and logical topology of the USB reflects the shared nature of the bus, client software (CSw) manipulating a USB function interface is presented with the view that it deals only with its interface(s) of interest. Client software for USB functions must use USB software programming interfaces to manipulate their functions as opposed to directly manipulating their functions via memory or I/O accesses as with other buses (e.g., PCI, EISA, PCMCIA, etc.). During operation, client software should be independent of other devices that may be connected to the USB. This allows the designer of the device and client software to focus on the hardware/software interaction design details. Figure 5-7 illustrates a device designer's perspective of the relationships of client software and USB functions with respect to the USB logical topology of Figure 5-6.

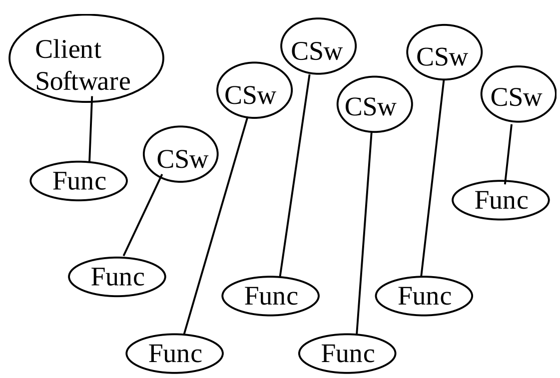


Figure 5-7. Client Software-to-function Relationships

5.3 USB Communication Flow

The USB provides a communication service between software on the host and its USB function. Functions can have different communication flow requirements for different client-to-function interactions. The USB provides better overall bus utilization by allowing the separation of the different communication flows to a USB function. Each communication flow makes use of some bus access to accomplish communication between client and function. Each communication flow is terminated at an endpoint on a device. Device endpoints are used to identify aspects of each communication flow.

Figure 5-8 shows a more detailed view of Figure 5-2. The complete definition of the actual communication flows of Figure 5-2 supports the logical device and function layer communication flows. These actual communication flows cross several interface boundaries. Chapters 6 through 8 describe the mechanical, electrical, and protocol interface definitions of the USB "wire." Chapter 9 describes the USB device programming interface that allows a USB device to be manipulated from the host side of the wire. Chapter 10 describes two host side software interfaces:

- Host Controller Driver (HCD): The software interface between the USB Host Controller and USB System Software. This interface allows a range of Host Controller implementations without requiring all host software to be dependent on any particular implementation. One USB Driver can support different Host Controllers without requiring specific knowledge of a Host Controller implementation. A Host Controller implementer provides an HCD implementation that supports the Host Controller.
- USB Driver (USBD): The interface between the USB System Software and the client software. This interface provides clients with convenient functions for manipulating USB devices.

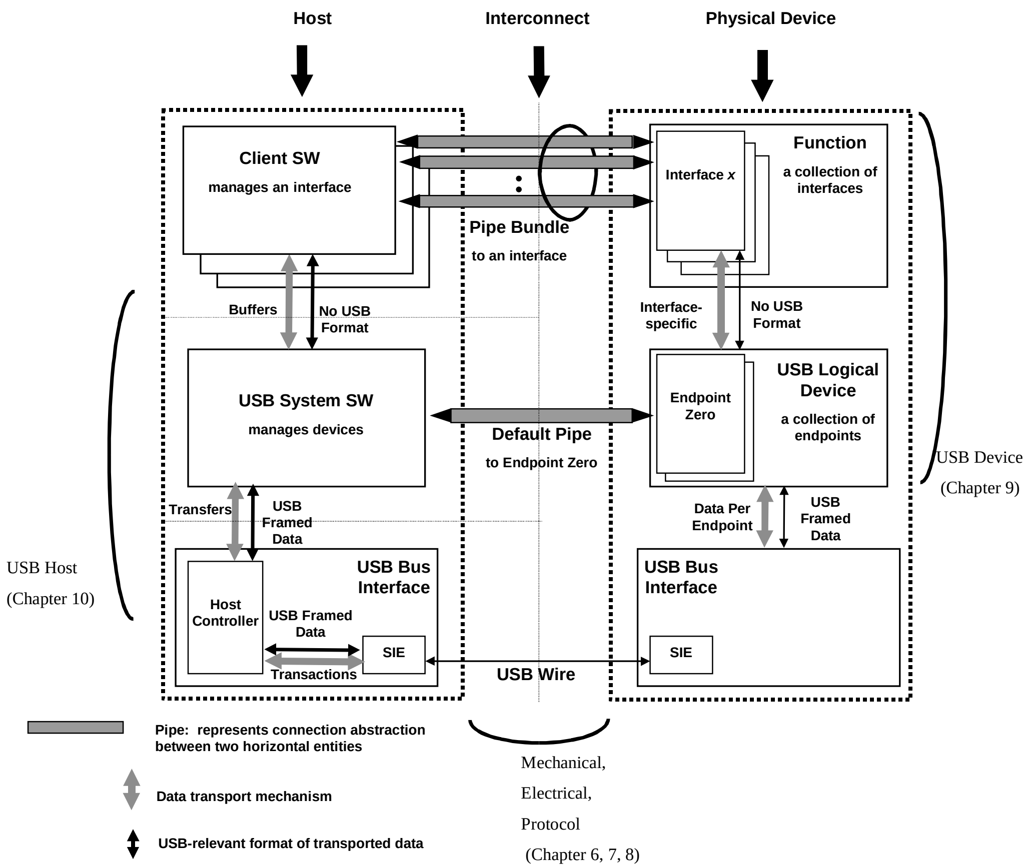


Figure 5-8. USB Host/Device Detailed View

A USB logical device appears to the USB system as a collection of endpoints. Endpoints are grouped into endpoint sets that implement an interface. Interfaces are views to the function. The USB System Software manages the device using the Default Control Pipe. Client software manages an interface using pipe bundles (associated with an endpoint set). Client software requests that data be moved across the USB between a buffer on the host and an endpoint on the USB device. The Host Controller (or USB device, depending on transfer direction) packetizes the data to move it over the USB. The Host Controller also coordinates when bus access is used to move the packet of data over the USB.

Figure 5-9 illustrates how communication flows are carried over pipes between endpoints and host side memory buffers. The following sections describe endpoints, pipes, and communication flows in more detail.

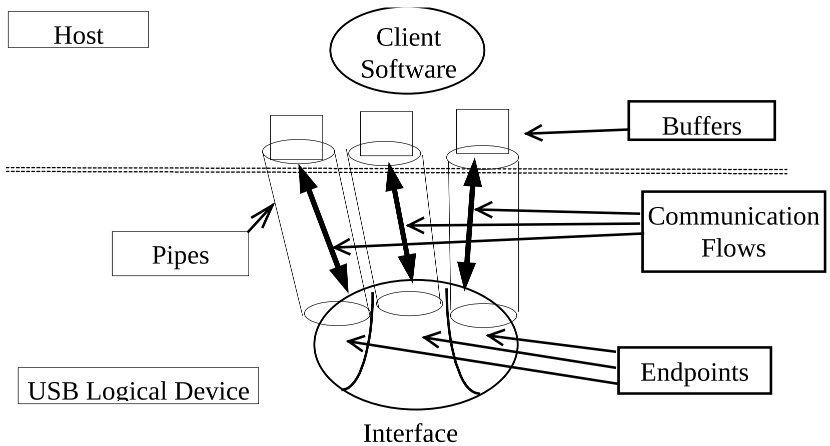


Figure 5-9. USB Communication Flow

Software on the host communicates with a logical device via a set of communication flows. The set of communication flows are selected by the device software/hardware designer(s) to efficiently match the communication requirements of the device to the transfer characteristics provided by the USB.

5.3.1 Device Endpoints

An endpoint is a uniquely identifiable portion of a USB device that is the terminus of a communication flow between the host and device. Each USB logical device is composed of a collection of independent endpoints. Each logical device has a unique address assigned by the system at device attachment time. Each endpoint on a device is given at design time a unique device-determined identifier called the endpoint number. Each endpoint has a device-determined direction of data flow. The combination of the device address, endpoint number, and direction allows each endpoint to be uniquely referenced. Each endpoint is a simplex connection that supports data flow in one direction: either input (from device to host) or output (from host to device).

An endpoint has characteristics that determine the type of transfer service required between the endpoint and the client software. Endpoints describe themselves by:

- Their bus access frequency/latency requirements
- Their bandwidth requirements
- Their endpoint number
- The error handling behavior requirements
- Maximum packet size that the endpoint is capable of sending or receiving
- The transfer type for the endpoint (refer to Section 5.4 for details)

- The direction data is transferred between the endpoint and the host.

Endpoints other than those with endpoint number zero are in an unknown state before being configured and may not be accessed by the host before being configured.

5.3.1.1 Endpoint Zero Requirements

All USB devices are required to implement a default control method that uses both the input and output endpoints with endpoint number zero. The USB System Software uses this default control method to initialize and generically manipulate the logical device (e.g., to configure the logical device) as the Default Control Pipe (see Section 5.3.2). The Default Control Pipe provides access to the device's configuration information and allows generic USB status and control access. The Default Control Pipe supports control transfers as defined in Section 5.5. The endpoints with endpoint number zero are always accessible once a device is attached, powered, and has received a bus reset.

A USB device that is capable of operating at high-speed must have a minimum level of support for operating at full-speed. When the device is attached to a hub operating in full-speed, the device must:

- Be able to reset successfully at full-speed.
- Respond successfully to standard requests: set_address, set_configuration, get_descriptor for device and configuration descriptors; and return appropriate information.

The high-speed device may or may not be able to support its intended functionality when operating at full-speed.

5.3.1.2 Non-endpoint Zero Requirements

Functions can have additional endpoints as required for their implementation. Low-speed functions are limited to two optional endpoints beyond the two required to implement the Default Control Pipe. Full-speed devices can have additional endpoints only limited by the protocol definition (i.e., a maximum of 15 additional input endpoints and 15 additional output endpoints).

Endpoints other than those for the Default Control Pipe cannot be used until the device is configured as a normal part of the device configuration process (refer to Chapter 9).

5.3.2 Pipes

A USB pipe is an association between an endpoint on a device and software on the host. Pipes represent the ability to move data between software on the host via a memory buffer and an endpoint on a device. There are two different, mutually exclusive, pipe communication modes:

- Stream: Data moving through a pipe has no USB-defined structure
- Message: Data moving through a pipe has some USB-defined structure.

The USB does not interpret the content of data it delivers through a pipe. Even though a message pipe requires that data be structured according to USB definitions, the content of the data is not interpreted by the USB.

Additionally, pipes have the following associated with them:

- A claim on USB bus access and bandwidth usage.
- A transfer type.
- The associated endpoint's characteristics, such as directionality and maximum data payload sizes. The data payload is the data that is carried in the data field of a data packet within a bus transaction (as defined in Chapter 8).

The pipe that consists of the two endpoints with endpoint number zero is called the Default Control Pipe. This pipe is always available once a device is powered and has received a bus reset. Other pipes come into existence when a USB device is configured. The Default Control Pipe is used by the USB System Software to determine device identification and configuration requirements, and to configure the device. The Default Control Pipe can also be used by device-specific software after the device is configured. The USB System Software retains "ownership" of the Default Control Pipe and mediates use of the pipe by other client software.

A software client normally requests data transfers via I/O Request Packets (IRPs) to a pipe and then either waits or is notified when they are completed. Details about IRPs are defined in an operating system-specific manner. This specification uses the term to simply refer to an identifiable request by a software client to move data between itself (on the host) and an endpoint of a device in an appropriate direction. A software client can cause a pipe to return all outstanding IRPs if it desires. The software client is notified that an IRP has completed when the bus transactions associated with it have completed either successfully or due to errors.

If there are no IRPs pending or in progress for a pipe, the pipe is idle and the Host Controller will take no action with regard to the pipe; i.e., the endpoint for such a pipe will not see any bus transactions directed to it. The only time bus activity is present for a pipe is when IRPs are pending for that pipe.

If a non-isochronous pipe encounters a condition that causes it to send a STALL to the host (refer to Chapter 8) or three bus errors are encountered on any packet of an IRP, the IRP is aborted/retired, all outstanding IRPs are also retired, and no further IRPs are accepted until the software client recovers from the condition (in an implementation-dependent way) and acknowledges the halt or error condition via a USBD call. An appropriate status informs the software client of the specific IRP result for error versus halt (refer to Chapter 10). Isochronous pipe behavior is described in Section 5.6.

An IRP may require multiple data payloads to move the client data over the bus. The data payloads for such a multiple data payload IRP are expected to be of the maximum packet size until the last data payload that contains the remainder of the overall IRP. See the description of each transfer type for more details. For such an IRP, short packets (i.e., less than maximum-sized data payloads) on input that do not completely fill an IRP data buffer can have one of two possible meanings, depending upon the expectations of a client:

- A client can expect a variable-sized amount of data in an IRP. In this case, a short packet that does not fill an IRP data buffer can be used simply as an in-band delimiter to indicate "end of unit of data." The IRP should be retired without error and the Host Controller should advance to the next IRP.
- A client can expect a specific-sized amount of data. In this case, a short packet that does not fill an IRP data buffer is an indication of an error. The IRP should be retired, the pipe should be stalled, and any pending IRPs associated with the pipe should also be retired.

Because the Host Controller must behave differently in the two cases and cannot know on its own which way to behave for a given IRP, it is possible to indicate per IRP which behavior the client desires.

An endpoint can inform the host that it is busy by responding with NAK. NAKs are not used as a retire condition for returning an IRP to a software client. Any number of NAKs can be encountered during the processing of a given IRP. A NAK response to a transaction does not constitute an error and is not counted as one of the three errors described above.

5.3.2.1 Stream Pipes

Stream pipes deliver data in the data packet portion of bus transactions with no USB-required structure on the data content. Data flows in at one end of a stream pipe and out the other end in the same order. Stream pipes are always uni-directional in their communication flow.

Data flowing through a stream pipe is expected to interact with what the USB believes is a single client. The USB System Software is not required to provide synchronization between multiple clients that may be using the same stream pipe. Data presented to a stream pipe is moved through the pipe in sequential order: first-in, first-out.

A stream pipe to a device is bound to a single device endpoint number in the appropriate direction (i.e., corresponding to an IN or OUT token as defined by the protocol layer). The device endpoint number for the opposite direction can be used for some other stream pipe to the device.

Stream pipes support bulk, isochronous, and interrupt transfer types, which are explained in later sections.

5.3.2.2 Message Pipes

Message pipes interact with the endpoint in a different manner than stream pipes. First, a request is sent to the USB device from the host. This request is followed by data transfer(s) in the appropriate direction. Finally, a Status stage follows at some later time. In order to accommodate the request/data/status paradigm, message pipes impose a structure on the communication flow that allows commands to be reliably identified and communicated. Message pipes allow communication flow in both directions, although the communication flow may be predominately one-way. The Default Control Pipe is always a message pipe.

The USB System Software ensures that multiple requests are not sent to a message pipe concurrently. A device is required to service only a single message request at a time per message pipe. Multiple software clients on the host can make requests via the Default Control Pipe, but they are sent to the device in a first-in, first-out order. A device can control the flow of information during the Data and Status stages based on its ability to respond to the host transactions (refer to Chapter 8 for more details).

A message pipe will not normally be sent the next message from the host until the current message's processing at the device has been completed. However, there are error conditions whereby a message transfer can be aborted by the host and the message pipe can be sent a new message transfer prematurely (from the device's perspective). From the perspective of the software manipulating a message pipe, an error on some part of an IRP retires the current IRP and all queued IRPs. The software client that requested the IRP is notified of the IRP completion with an appropriate error indication.

A message pipe to a device requires a single device endpoint number in both directions (IN and OUT tokens). The USB does not allow a message pipe to be associated with different endpoint numbers for each direction.

Message pipes support the control transfer type, which is explained in Section 5.5.

5.3.3 Frames and Microframes

USB establishes a 1 millisecond time base called a frame on a full-/low- speed bus and a 125µs time base called a microframe on a high-speed bus. A (micro)frame can contain several transactions. Transactions for different endpoints are allowed during a (micro)frame. Isochronous and interrupt endpoints are given opportunities to the bus every N (micro)frames. The values of N and other details about isochronous and interrupt transfers are described in Sections 5.6 and 5.7.

5.4 Transfer Types

The USB transports data through a pipe between a memory buffer associated with a software client on the host and an endpoint on the USB device. Data transported by message pipes is carried in a USB-defined structure, but the USB allows device-specific structured data to be transported within the USB-defined message data payload. The USB also defines that data moved over the bus is packetized for any pipe (stream or message), but ultimately the formatting and interpretation of the data transported in the data payload of a bus transaction is the responsibility of the client software and function using the pipe. However, the USB provides different transfer types that are optimized to more closely match the service requirements of the client software and function using the pipe. An IRP uses one or more bus transactions to move information between a software client and its function.

Each transfer type determines various characteristics of the communication flow including the following:

- Data format imposed by the USB
- Direction of communication flow
- Packet size constraints
- Bus access constraints
- Latency constraints
- Required data sequences
- Error handling.

The designers of a USB device choose the capabilities for the device's endpoints. When a pipe is established for an endpoint, most of the pipe's transfer characteristics are determined and remain fixed for the lifetime of the pipe. Transfer characteristics that can be modified are described for each transfer type.

The USB defines four transfer types:

- Control Transfers: Bursty, non-periodic, host software-initiated request/response communication, typically used for command/status operations.
- Isochronous Transfers: Periodic, continuous communication between host and device, typically used for time-relevant information. This transfer type also preserves the concept of time encapsulated in the data. This does not imply, however, that the delivery needs of such data is always time-critical.
- Interrupt Transfers: Small-data, low-frequency, bounded-latency communication.
- Bulk Transfers: Non-periodic, large-packet bursty communication, typically used for data that can use any available bandwidth and can also be delayed until bandwidth is available.

Each transfer type is described in detail in the following four major sections. The data for any IRP is carried by the data field of the data packet as described in Section 8.4.3. Chapter 8 also describes details of the protocol that are affected by use of each particular transfer type.

5.5 Control Transfers

Control transfers allow access to different parts of a device. Control transfers are intended to support configuration/command/status type communication flows between client software and its function. A control transfer is composed of a Setup bus transaction moving request information from host to function, zero or more Data transactions sending data in the direction indicated by the Setup transaction, and a Status transaction returning status information from function to host. The Status transaction returns "success" when the endpoint has successfully completed processing the requested operation. Section 8.5.2 describes the details of what packets, bus transactions, and transaction sequences are used to accomplish a control transfer. Chapter 9 describes the details of the defined USB command codes.

Each USB device is required to implement the Default Control Pipe as a message pipe. This pipe is used by the USB System Software. The Default Control Pipe provides access to the USB device's configuration, status, and control information. A function can, but is not required to, provide endpoints for additional control pipes for its own implementation needs.

The USB device framework (refer to Chapter 9) defines standard, device class, or vendor-specific requests that can be used to manipulate a device's state. Descriptors are also defined that can be used to contain different information on the device. Control transfers provide the transport mechanism to access device descriptors and make requests of a device to manipulate its behavior.

Control transfers are carried only through message pipes. Consequently, data flows using control transfers must adhere to USB data structure definitions as described in Section 5.5.1.

The USB system will make a "best effort" to support delivery of control transfers between the host and devices. A function and its client software cannot request specific bus access frequency or bandwidth for control transfers. The USB System Software may restrict the bus access and bandwidth that a device may desire for control transfers. These restrictions are defined in Section 5.5.3 and Section 5.5.4.

5.5.1 Control Transfer Data Format

The Setup packet has a USB-defined structure that accommodates the minimum set of commands required to enable communication between the host and a device. The structure definition allows vendor-specific extensions for device specific commands. The Data transactions following Setup have a USB-defined structure except when carrying vendor-specific information. The Status transaction also has a USB-defined structure. Specific control transfer Setup/Data definitions are described in Section 8.5.2 and Chapter 9.

5.5.2 Control Transfer Direction

Control transfers are supported via bi-directional communication flow over message pipes. As a consequence, when a control pipe is configured, it uses both the input and output endpoint with the specified endpoint number.

5.5.3 Control Transfer Packet Size Constraints

An endpoint for control transfers specifies the maximum data payload size that the endpoint can accept from or transmit to the bus. The USB defines the allowable maximum control data payload sizes for full-speed devices to be~~either~~ 8, 16, 32, or 64 bytes. The only allowable maximum control data payload size for high-speed devices is 64 bytes. Low-speed devices are limited to only an ~~eight-byte~~8-byte maximum data payload size. This maximum applies to the data payloads of the Data packets following a Setup; i.e., the size specified is for the data field of the packet as defined in Chapter 8, not including other information that is required by the protocol. A Setup packet is always eight bytes. A control pipe (including the Default Control Pipe) always uses its *wMaxPacketSize* value for data payloads.

An endpoint reports in its configuration information the value for its maximum data payload size. The USB does not require that data payloads transmitted be exactly the maximum size; i.e., if a data payload is less than the maximum, it does not need to be padded to the maximum size.

All Host Controllers are required to have support for 8-, 16-, 32-, and 64-byte maximum data payload sizes for full-speed control ~~endpoints and only eight-byte~~endpoints, only 8-byte maximum data payload sizes for low-speed control endpoints, and only 64-byte maximum data payload size for high-speed control endpoints. No Host Controller is required to support larger or smaller maximum data payload sizes.

In order to determine the maximum packet size for the Default Control Pipe, the USB System Software reads the device descriptor. The host will read the first eight bytes of the device descriptor. The device always responds with at least these initial bytes in a single packet. After the host reads the initial part of the device descriptor, it is guaranteed to have read this default pipe's *wMaxPacketSize* field (byte 7 of the device descriptor). It will then allow the correct size for all subsequent transactions. For all other control endpoints, the maximum data payload size is known after configuration so that the USB System Software can ensure that no data payload will be sent to the endpoint that is larger than the supported size. The host will always use a maximum data payload size of at least eight bytes.

An endpoint must always transmit data payloads with a data field less than or equal to the endpoint's *wMaxPacketSize* (refer to Chapter 9). When a control transfer involves more data than can fit in one data payload of the currently established maximum size, all data payloads are required to be maximum-sized except for the last data payload, which will contain the remaining data.

The Data stage of a control transfer from an endpoint to the host is complete when the endpoint does one of the following:

- Has transferred exactly the amount of data specified during the Setup stage

- Transfers a packet with a payload size less than *wMaxPacketSize* or transfers a zero-length packet.

When a Data stage is complete, the Host Controller advances to the Status stage instead of continuing on with another data transaction. If the Host Controller does not advance to the Status stage when the Data stage is complete, the endpoint halts the pipe as was outlined in Section 5.3.2. If a larger-than-expected data payload is received from the endpoint, the IRP for the control transfer will be aborted/retired.

The Data stage of a control transfer from the host to an endpoint is complete when all of the data has been transferred. If the endpoint receives a larger-than-expected data payload from the host, it halts the pipe.

5.5.4 Control Transfer Bus Access Constraints

Control transfers can be used by high-speed, full-speed and low-speed USB devices.

An endpoint has no way to indicate a desired bus access frequency for a control pipe. The USB balances the bus access requirements of all control pipes and the specific IRPs that are pending to provide “best effort” delivery of data between client software and functions.

The USB requires that part of each (micro-)frame be reserved to be available for use by control transfers as follows:

- If the control transfers that are attempted (in an implementation-dependent fashion) consume less than 10% of the frame time for low-speed/full-speed endpoints or less than 20% of a microframe for high-speed endpoints, the remaining time can be used to support bulk transfers (refer to Section 5.8).
- A control transfer that has been attempted and needs to be retried can be retried in the current or a future frame; i.e., it is not required to be retried in the same frame.
- If there are more control transfers than reserved time, but there is additional frame time that is not being used for isochronous or interrupt transfers, a Host Controller may move additional control transfers as they are available.
- If there are too many pending control transfers for the available frame time, control transfers are selected to be moved over the bus as appropriate.
- If there are control transfers pending for multiple endpoints, control transfers for the different endpoints are selected according to a fair access policy that is Host Controller implementation-dependent.
- A transaction of a control transfer that is frequently being retried should not be expected to consume an unfair share of the bus time.

These requirements allow control transfers between host and devices to be regularly moved over the bus with “best effort.”

The USB System Software ~~The~~can, at its discretion, vary the rate of control transfers to a particular ~~endpoint can be varied by theUSB System Software~~ endpoint. ~~at its discretion.~~ An endpoint and its client software cannot assume a specific rate of service for control transfers. A control endpoint may see zero or more transfers in a single frame. Bus time made available to a software client and its endpoint can be changed as other devices are inserted into and removed from the system or also as control transfers are requested for other device endpoints.

The bus frequency and (micro-)frame timing limit the maximum number of successful control transfers within a (micro-)frame for any USB system. For full/low-speed buses, the number of successful control transfers per frame is limited to less than 29 full-speed eight-byte data payloads or less than four low-speed eight-byte data payloads. For high-speed buses, the number of control transfers is limited to less than 20 high-speed 64-byte data payloads per microframe. Table 5-1 lists information about different-sized full-speed control transfers and the maximum number of transfers possible in a frame. This table was generated assuming that there is one Data stage transaction and that the Data stage has a zero-length status phase. The table illustrates the possible power of two data payloads less than or equal to the allowable maximum data payload sizes. The table does not include the overhead associated with bit stuffing.

Table 5-1. Full-speed Control Transfer Limits

	Protocol Overhead (45 bytes)		(9 SYNC bytes, 9 PID bytes, 6 Endpoint + CRC bytes, 6 CRC bytes, 8 Setup data bytes, and a 7-byte interpacket delay (EOP, etc.))			
	Data Payload	**Max Bandwidth (bytes/second)**	**Frame Bandwidth per Transfer**	**Max Transfers**	**Bytes Remaining**	**Bytes/Frame Useful Data**
	1	32000	3%	32	23	32
	2	62000	3%	31	43	62
	4	120000	3%	30	30	120
	8	224000	4%	28	16	224
	16	384000	4%	24	36	384
	32	608000	5%	19	37	608
	64	832000	7%	13	83	832
Max		1500000				1500

The 10% frame reservation for non-periodic transfers means that in a system with bus time fully allocated, all full-speed control transfers in the system contend for a nominal three control transfers per frame. Because the USB system uses control transfers for configuration purposes in addition to whatever other control transfers other client software may be requesting, a given software client and its function should not expect to be able to make use of this full bandwidth for its own control purposes. Host Controllers are also free to determine how the individual bus transactions for specific control transfers are moved over the bus within and across frames. An endpoint could see all bus transactions for a control transfer within the same frame or spread across several noncontiguous frames. A Host Controller, for various implementation reasons, may not be able to provide the theoretical maximum number of control transfers per frame.

For high-speed endpoints, the 20% microframe reservation for non-periodic transfers means that all high speed control transfers are contending for nominally eight control transfers per microframe. High-speed control transfers contend for microframe time along with split-transactions (see Section <11.TBD>> for more information about split transactions) for full- and low- speed control transfers. Both full-speed and low-speed control transfers contend for the same available frame time. However, high-speed control transfers can occur simultaneously with full- and low-speed control transfers. Low-speed control transfers simply take longer to transfer. Table 5-2 lists information about different-sized low-speed packets and the maximum number of packets possible in a frame. The table does not include the overhead associated with bit stuffing. For both speeds, because a control transfer is composed of several packets, the packets can be spread over several frames to spread the bus time required across several frames.

Table 5-2. Low-speed Control Transfer Limits

	Protocol Overhead (46 bytes)					
	Data Payload	**Max Bandwidth (Approximate)**	**Frame Bandwidth per Transfer**	**Max Transfers**	**Bytes Remaining**	**Bytes/Frame Useful Data**
	1	3000	25%	3	46	3
	2	6000	26%	3	43	6
	4	12000	27%	3	37	12
	8	24000	29%	3	25	24
Max		187500				187

Error! Reference source not found. lists information about different-sized high-speed control transfers and the maximum number of transfers possible in a microframe. This table was generated assuming that there is one Data stage transaction and that the Data stage has a zero-length status stage. The table illustrates the possible power of two data payloads less than or equal to the allowable maximum data payload sizes. The table does not include the overhead associated with bit stuffing.

Table 5-3. High-speed Control Transfer Limits

	Protocol Overhead <<Based on 480Mb/s and 80 bit bus gap, 640 bit bus turnaround, 16 bit sync, 8 bit EOP: 9 PID bytes, 6 endpoint+CRC5 bytes, 6 CRC16 bytes, 8 SETUP data bytes, 64 request bytes, 9*(2+1) sync/eop bytes, 3*10 gap bytes, 3*80 bus turn bytes>>					
	Data Payload	**Max Bandwidth (Approximate)**	**Frame Bandwidth per Transfer**	**Max Transfers**	**Bytes Remaining**	**Bytes/Frame Useful Data**
	1					
	16					
	32					
	64			20		
Max						

High-speed control OUT endpoints must support the PING flow control protocol. The details of this protocol are described in Section 8.5.1.

5.5.5 Control Transfer Data Sequences

Control transfers require that a Setup bus transaction be sent from the host to a device to describe the type of control access that the device should perform. The Setup transaction is followed by zero or more control Data transactions that carry the specific information for the requested access. Finally, a Status transaction completes the control transfer and allows the endpoint to return the status of the control transfer to the client software. After the Status transaction for a control transfer is completed, the host can advance to the next control transfer for the endpoint. As described in Section 5.5.4, each control transaction and the next control transfer will be moved over the bus at some Host Controller implementation-defined time.

The endpoint can be busy for a device-specific time during the Data and Status transactions of the control transfer. During these times when the endpoint indicates it is busy (refer to Chapter 8 and Chapter 9 for details), the host will retry the transaction at a later time.

If a Setup transaction is received by an endpoint before a previously initiated control transfer is completed, the device must abort the current transfer/operation and handle the new control Setup transaction. A Setup transaction should not normally be sent before the completion of a previous control transfer. However, if a transfer is aborted, for example, due to errors on the bus, the host can send the next Setup transaction prematurely from the endpoint's perspective.

After a halt condition is encountered or an error is detected by the host, a control endpoint is allowed to recover by accepting the next Setup PID; i.e., recovery actions via some other pipe are not required for control endpoints. For the Default Control Pipe, a device reset will ultimately be required to clear the halt or error condition if the next Setup PID is not accepted.

The USB provides robust error detection and recovery/retransmission for errors that occur during control transfers. Transmitters and receivers can remain synchronized with regard to where they are in a control transfer and recover with minimum effort. Retransmission of Data and Status packets can be detected by a receiver via data retry indicators in the packet. A transmitter can reliably determine that its corresponding receiver has successfully accepted a transmitted packet by information returned in a handshake to the packet. The protocol allows for distinguishing a retransmitted packet from its original packet except for a control Setup packet. Setup packets may be retransmitted due to a transmission error; however, Setup packets cannot indicate that a packet is an original or a retried transmission.

5.6 Isochronous Transfers

In non-USB environments, isochronous transfers have the general implication of constant-rate, error-tolerant transfers. In the USB environment, requesting an isochronous transfer type provides the requester with the following:

- Guaranteed access to USB bandwidth with bounded latency
- Guaranteed constant data rate through the pipe as long as data is provided to the pipe
- In the case of a delivery failure due to error, no retrying of the attempt to deliver the data.

While the USB isochronous transfer type is designed to support isochronous sources and destinations, it is not required that software using this transfer type actually be isochronous in order to use the transfer type. Section 5.12 presents more detail on special considerations for handling isochronous data on the USB.

5.6.1 Isochronous Transfer Data Format

The USB imposes no data content structure on communication flows for isochronous pipes.

5.6.2 Isochronous Transfer Direction

An isochronous pipe is a stream pipe and is, therefore, always uni-directional. An endpoint description identifies whether a given isochronous pipe's communication flow is into or out of the host. If a device requires bi-directional isochronous communication flow, two isochronous pipes must be used, one in each direction.

5.6.3 Isochronous Transfer Packet Size Constraints

An endpoint in a given configuration for an isochronous pipe specifies the maximum size data payload that it can transmit or receive. The USB System Software uses this information during configuration to ensure that there is sufficient bus time to accommodate this maximum data payload in each (micro-)frame. If there is sufficient bus time for the maximum data payload, the configuration is established; if not, the configuration is not established. The USB System Software does not adjust the maximum data payload size for an isochronous pipe as is the case for a control pipe. An isochronous pipe can simply either be supported or not supported in a given USB system configuration.

The USB limits the maximum data payload size to 1,023 bytes for each full-speed isochronous pipe.endpoint. High-speed endpoints are allowed up to 1024-byte data payloads. High-speed, high bandwidth endpoints are allowed up to three transactions per microframe. Table 5-1 lists information about different-sized full-speed isochronous transactions and the maximum number of transactions possible in a frame. The table does not include the overhead associated with bit stuffing.

Table 5-1. Full-Speed Isochronous Transaction Limits

	Protocol Overhead (9 bytes)	(2 SYNC bytes, 2 PID bytes, 2 Endpoint + CRC bytes, 2 CRC bytes, and a 1-byte interpacket delay)				
	Data Payload	**Max Bandwidth**	**Frame Bandwidth per Transfer**	**Max Transfers**	**Bytes Remaining**	**Bytes/Frame Useful Data**
	1	150000	1%	150	0	150
	2	272000	1%	136	4	272
	4	460000	1%	115	5	460
	8	704000	1%	88	4	704
	16	960000	2%	60	0	960
	32	1152000	3%	36	24	1152
	64	1280000	5%	20	40	1280
	128	1280000	9%	10	130	1280
	256	1280000	18%	5	175	1280
	512	1024000	35%	2	458	1024
	1023	1023000	69%	1	468	1023
Max		1500000				1500

Error! Reference source not found. lists information about different-sized high-speed isochronous transactions and the maximum number of transactions possible in a microframe. The table does not include the overhead associated with bit stuffing.

Table 5-3. Isochronous Transaction Limits for High-speed Endpoints

	Protocol Overhead		<<Based on 480Mb/s and 80 bit bus gap, 640 bit bus turnaround, 16 bit sync, 8 bit EOP, 80% of 7500 byte microframe: 2 PID bytes, 2 endpoint+CRC5 bytes, 2 CRC16 bytes, 1 data byte, 2*(2+1) sync/eop bytes, 2*10 gap bytes, 0 bus turn bytes>>			
	Data Payload	**Max Bandwidth**	**Frame Bandwidth per Transfer**	**Max Transfers**	**Bytes Remaining**	**Bytes/Frame Useful Data**
	1			182		
Max						

Any given transaction for an isochronous pipe need not be exactly the maximum size specified for the endpoint. The size of a data payload is determined by the transmitter (client software or function) and can vary as required from transaction to transaction. The USB ensures that whatever size is presented to the Host Controller is delivered on the bus. The actual size of a data payload is determined by the data transmitter and may be less than the prenegotiated maximum size. Bus errors can change the actual packet size seen by the receiver. However, these errors can be detected by either CRC on the data or by knowledge the receiver has about the expected size for any transaction.

A high-speed device must have its default configuration specify any iosochronous endpoints as requiring a zero data payload size via *wMaxPacketSize*. Alternate configurations (or interface settings) may specify non-zero data payload sizes. If the isochronous endpoints have a large data payload size, it is recommended that additional alternate configurations or interface settings be used to specify a range of data payload sizes. This increases the chances that the device can be used sucessfully in combination with other USB devices.

5.6.4 Isochronous Transfer Bus Access Constraints

Isochronous transfers can be used by only~~by~~ full-speed and high-speed devices.

The USB requires that no more than 90% of any frame be allocated for periodic (isochronous and interrupt) transfers for full-speed endpoints. High-speed endpoints can allocate only 80% of a microframe for periodic transfers.

An isochronous endpoint must specify its required bus access period. High-speed endpoints can specify a desired period between 1x125µs to (2^N)x125µs, where N is less than or equal to 16<<TBD>>. This allows high-speed isochronous transfers to have rates slower than one transaction per microframe. However, an isochronous endpoint must be prepared to handle poll rates higher than the one specified. It must return a zero-length packet whenever data is requested at a different interval than the specified interval. This allows a high-speed endpoint to move up to 3072 bytes per microframe (or 192Mb/s). A high-speed endpoint that requires more than 1024 bytes per microframe must specify a period of 1 (e.g. a *bInterval* value of 0). See Section 5.9 for more information about the details of multiple transactions per microframe for high bandwidth high-speed endpoints.

~~An endpoint for an isochronous pipe does not include information about bus access frequency. All isochronous pipes normally move exactly one data packet each frame (i.e., every 1ms).~~ Errors on the bus or delays in operating system scheduling of client software can result in no packet being transferred for a ~~frame.~~ (micro-)frame. An error indication should be returned as status to the client software in such a case. A device can also detect this situation by tracking SOF tokens and noticing ~~two SOF tokens without an intervening data packet for an isochronous endpoint.~~a disturbance in the specified bus access period pattern.

The bus frequency and frame timing limit the maximum number of successful isochronous transactions within a (micro-)frame for any USB system to less than 151 full-speed one-byte data payloads and less than 182 high-speed one-byte data payloads. A Host Controller, for various implementation reasons, may not be able to provide the theoretical maximum number of isochronous transactions per (micro-)frame.

5.6.5 Isochronous Transfer Data Sequences

Isochronous transfers do not support data retransmission in response to errors on the bus. A receiver can determine that a transmission error occurred. The low-level USB protocol does not allow handshakes to be returned to the transmitter of an isochronous pipe. Normally, handshakes would be returned to tell the transmitter whether a packet was successfully received or not. For isochronous transfers, timeliness is more important than correctness/retransmission, and given the low error rates expected on the bus, the protocol is optimized by assuming transfers normally succeed. Isochronous receivers can determine whether they missed data during a (micro-)frame. Also, a receiver can determine how much data was lost. Section 5.12 describes these USB mechanisms in more detail.

An endpoint for isochronous transfers never halts because there is no handshake to report a halt condition. Errors are reported as status associated with the IRP for an isochronous transfer, but the isochronous pipe is not halted in an error case. If an error is detected, the host continues to process the data associated with the next (micro-)frame of the transfer. ~~Limited~~Only limited error detection is possible because the protocol for isochronous transactions does not allow per-transaction handshakes.

5.7 Interrupt Transfers

The interrupt transfer type is designed to support those devices that need to send or receive small amounts of data infrequently, but with bounded service periods. Requesting a pipe with an interrupt transfer type provides the requester with the following:

- Guaranteed maximum service period for the pipe
- Retry of transfer attempts at the next period, in the case of occasional delivery failure due to error on the bus.

5.7.1 Interrupt Transfer Data Format

The USB imposes no data content structure on communication flows for interrupt pipes.

5.7.2 Interrupt Transfer Direction

An interrupt pipe is a stream pipe and is therefore always uni-directional. An endpoint description identifies whether a given interrupt pipe's communication flow is into or out of the host.

5.7.3 Interrupt Transfer Packet Size Constraints

An endpoint for an interrupt pipe specifies the maximum size data payload that it will transmit or receive. The maximum allowable interrupt data payload size is 64 bytes or less for full-speed. High-speed endpoints are allowed maximum data payload sizes up to 1024 bytes. A high speed, high bandwidth endpoint specifies whether it requires two or three transactions per microframe. Low-~~Low~~ speed devices are limited to eight bytes or less maximum data payload size. This maximum applies to the data payloads of the data packets; i.e., the size specified is for the data field of the packet as defined in Chapter 8, not including other protocol-required information. The USB does not require that data packets be exactly the maximum size; i.e., if a data packet is less than the maximum, it does not need to be padded to the maximum size.

All Host Controllers are required to have support for up to 64-byte maximum data payload sizes for full-speed interrupt ~~endpoints and~~endpoints, eight bytes or less maximum data payload sizes for low-speed interrupt endpoints, and three 1024 byte data payloads or less for high-speed interrupt endpoints. See Section 5.9 for more information about the details of multiple transactions per microframe for high bandwidth high-speed endpoints. No Host Controller is required to support larger maximum data payload sizes.

The USB System Software determines the maximum data payload size that will be used for a interrupt pipe during device configuration. This size remains constant for the lifetime of a device's configuration. The USB System Software uses the maximum data payload size determined during configuration to ensure that there is sufficient bus time to accommodate this maximum data payload in its assigned period. If there is sufficient bus time, the pipe is established; if not, the pipe is not established.~~The USB System Software does not adjust the bus time made available to an interrupt pipe as is the case for a control pipe. An interrupt pipe can simply either be supported or not supported in a given USB system configuration.~~ However, the actual size of a data payload is still determined by the data transmitter and may be less than the maximum size.

An endpoint must always transmit data payloads with a data field less than or equal to the endpoint's *wMaxPacketSize* value. A device can move data via an interrupt pipe that is larger than *wMaxPacketSize*. A software client can accept this data via an IRP for the interrupt transfer that requires multiple bus transactions without requiring an IRP-complete notification per transaction. This can be achieved by specifying a buffer that can hold the desired data size. The size of the buffer is a multiple of *wMaxPacketSize* with some remainder. The endpoint must transfer each transaction except the last as *w*Max*PacketSize* and the last transaction is the remainder. The multiple data transactions are moved over the bus at the period established for the pipe.

When an interrupt transfer involves more data than can fit in one data payload of the currently established maximum size, all data payloads are required to be maximum-sized except for the last data payload, which will contain the remaining data. An interrupt transfer is complete when the endpoint does one of the following:

- Has transferred exactly the amount of data expected
- Transfers a packet with a payload size less than *wMaxPacketSize* or transfers a zero-length packet.

When an interrupt transfer is complete, the Host Controller retires the current IRP and advances to the next IRP. If a data payload is received that is larger than expected, the interrupt IRP will be aborted/retired and the pipe will stall future IRPs until the condition is corrected and acknowledged.

A high-speed device must have its default configuration specify any interrupt endpoints as requiring a zero data payload size via *wMaxPacketSize*. Alternate configurations (or interface settings) may specify non-zero data payload sizes. If the interrupt endpoints have a large data payload size, it is recommended that additional alternate configurations or interface settings be used to specify a range of data payload sizes. This increases the chances that the device can be used sucessfully in combination with other USB devices.

5.7.4 Interrupt Transfer Bus Access Constraints

Interrupt transfers can be used by ~~full-speed and low-speed~~full-speed, low-speed, and high-speed devices. High-speed endpoints must not require more than 80% of a microframe for periodic transfers.

The USB requires that no more than 90% of any frame be allocated for periodic (isochronous and interrupt) full/low-speed transfers.

The bus frequency and (micro-)frame timing limit the maximum number of successful interrupt transactions within a (micro-)frame for any USB system to less than 108 full-speed one-byte data payloads or 14 low-speed one-byte data payloads or to less than 62 high-speed one-byte data payloads. A Host Controller, for various implementation reasons, may not be able to provide the above maximum number of interrupt transactions per (micro-)frame.

Table 5-4 lists information about different ~~sized full-speed~~high-speed interrupt transactions and the maximum number of transactions possible in a frame. Table 5-5 lists similar information for full-speed interrupt transactions. Table 5-6 lists similar information for low-speed interrupt transactions. The tables do not include the overhead associated with bit stuffing.

Table 5-4. High-speed Interrupt Transaction Limits

	Protocol Overhead			<<Based on 480Mb/s and 80 bit bus gap, 640 bit bus turnaround, 16 bit sync, 8 bit EOP, 80% of 7500 byte microframe: 3 PID bytes, 2 endpoint+CRC5 bytes, 2 CRC16 bytes, 1 data byte, 3*(2+1) sync/eop bytes, 0 gap bytes, 80 bus turn bytes>>		
	Data Payload	**Max Bandwidth**	**Frame Bandwidth per Transfer**	**Max Transfers**	**Bytes Remaining**	**Bytes/Frame Useful Data**
	1			62		
Max						

Table 5-5. Full-speed Interrupt Transaction Limits

	Protocol Overhead (13 bytes)		(3 SYNC bytes, 3 PID bytes, 2 Endpoint + CRC bytes, 2 CRC bytes, and a 3-byte interpacket delay)			
	Data Payload	**Max Bandwidth**	**Frame Bandwidth per Transfer**	**Max Transfers**	**Bytes Remaining**	**Bytes/Frame Useful Data**
	1	107000	1%	107	2	107
	2	200000	1%	100	0	200
	4	352000	1%	88	4	352
	8	568000	1%	71	9	568
	16	816000	2%	51	21	816
	32	1056000	3%	33	15	1056
	64	1216000	5%	19	37	1216
Max		1500000				1500

An endpoint for an interrupt pipe specifies its desired bus access period. A full-speed endpoint can specify a desired period from 1ms to 255ms. Low-speed endpoints are limited to specifying only 10ms to 255ms. High-speed endpoints can specify a desired period from 1x125µs to (2^N)x125µs, where N is less than or equal to 16. The USB System Software will use this information during configuration to determine a period that can be sustained. The period provided by the system may be shorter than that desired by the device up to the shortest period defined by the USB ~~(1ms).~~(125µs microframe or 1ms frame). The client software and device can depend only on the fact that the host will ensure that the time duration between two transaction attempts with the endpoint will be no longer than the desired period. Note that errors on the bus can prevent an interrupt transaction from being successfully delivered over the bus and consequently exceed the desired period. Also, the endpoint is only polled when the software client has an IRP for an interrupt transfer pending. If the bus time for performing an interrupt transfer arrives and there is no IRP pending, the endpoint will not be given an opportunity to transfer data at that time. Once an IRP is available, its data will be transferred at the next allocated period.

Table 5-6. Low-speed Interrupt Transaction Limits

	Protocol Overhead (13 bytes)					
	Data Payload	Max Bandwidth (Approximate)	Frame Bandwidth per Transfer	Max Transfers	Bytes Remaining	Bytes/Frame Useful Data
	1	13000	7%	13	5	13
	2	24000	8%	12	7	24
	4	44000	9%	11	0	44
	8	64000	11%	8	19	64
Max		187500				187

Interrupt transfers are moved over the USB by accessing an interrupt endpoint every period. For input interrupt endpoints, the host has no way to determine whether an endpoint will source an interrupt without accessing the endpoint and requesting an interrupt transfer. If the endpoint has no interrupt data to transmit when accessed by the host, it responds with NAK. An endpoint should only provide interrupt data when it has an interrupt pending to avoid having a software client erroneously notified of IRP complete. A zero-length data payload is a valid transfer and may be useful for some implementations.

5.7.5 Interrupt Transfer Data Sequences

Interrupt transactions may use either alternating data toggle bits, such that the bits are toggled only upon successful transfer completion, or a continuously toggling of data toggle bits. The host in any case must assume that the device is obeying full handshake/retry rules as defined in Chapter 8. A device may choose to always toggle DATA0/DATA1 PIDs so that it can ignore handshakes from the host. However, in this case, the client software can miss some data packets when an error occurs, because the Host Controller interprets the next packet as a retry of a missed packet.

If a halt condition is detected on an interrupt pipe due to transmission errors or a STALL handshake being returned from the endpoint, all pending IRPs are retired. Removal of the halt condition is achieved via software intervention through a separate control pipe. This recovery will reset the data toggle bit to DATA0 for the endpoint on both the host and the device. Interrupt transactions are retried due to errors detected on the bus that affect a given transfer.

5.8 Bulk Transfers

The bulk transfer type is designed to support devices that need to communicate relatively large amounts of data at highly variable times where the transfer can use any available bandwidth. Requesting a pipe with a bulk transfer type provides the requester with the following:

- Access to the USB on a bandwidth-available basis
- Retry of transfers, in the case of occasional delivery failure due to errors on the bus
- Guaranteed delivery of data, but no guarantee of bandwidth or latency.

Bulk transfers occur only on a bandwidth-available basis. For a USB with large amounts of free bandwidth, bulk transfers may happen relatively quickly; for a USB with little bandwidth available, bulk transfers may trickle out over a relatively long period of time.

5.8.1 Bulk Transfer Data Format

The USB imposes no data content structure on communication flows for bulk pipes.

5.8.2 Bulk Transfer Direction

A bulk pipe is a stream pipe and, therefore, always has communication flowing either into or out of the host for a given pipe. If a device requires bi-directional bulk communication flow, two bulk pipes must be used, one in each direction.

5.8.3 Bulk Transfer Packet Size Constraints

An endpoint for bulk transfers specifies the maximum data payload size that the endpoint can accept from or transmit to the bus. The USB defines the allowable maximum bulk data payload sizes to be only 8, 16, 32, or 64 bytes for full-speed endpoints and 512 bytes for high-speed endpoints. This maximum applies to the data payloads of the data packets; i.e.; the size specified is for the data field of the packet as defined in Chapter 8, not including other protocol-required information.

A bulk endpoint is designed to support a maximum data payload size. A bulk endpoint reports in its configuration information the value for its maximum data payload size. The USB does not require that data payloads transmitted be exactly the maximum size; i.e., if a data payload is less than the maximum, it does not need to be padded to the maximum size.

All Host Controllers are required to have support for 8-, 16-, 32-, and 64-byte maximum packet sizes for full-speed bulk endpoints and 512 bytes for high-speed bulk endpoints. No Host Controller is required to support larger or smaller maximum packet sizes.

During configuration, the USB System Software reads the endpoint's maximum data payload size and ensures that no data payload will be sent to the endpoint that is larger than the supported size.

An endpoint must always transmit data payloads with a data field less than or equal to the endpoint's reported *wMaxPacketSize* value. When a bulk IRP involves more data than can fit in one maximum-sized data payload, all data payloads are required to be maximum size except for the last data payload, which will contain the remaining data. A bulk transfer is complete when the endpoint does one of the following:

- Has transferred exactly the amount of data expected
- Transfers a packet with a payload size less than *wMaxPacketSize* or transfers a zero-length packet.

When a bulk transfer is complete, the Host Controller retires the current IRP and advances to the next IRP. If a data payload is received that is larger than expected, all pending bulk IRPs for that endpoint will be aborted/retired.

5.8.4 Bulk Transfer Bus Access Constraints

Bulk transfers can be used by only~~by~~ full-speed and high-speed devices.

An endpoint has no way to indicate a desired bus access frequency for a bulk pipe. The USB balances the bus access requirements of all bulk pipes and the specific IRPs that are pending to provide "good effort" delivery of data between client software and functions. Moving control transfers over the bus has priority over moving bulk transfers.

There is no time guaranteed to be available for bulk transfers as there is for control transfers. Bulk transfers are moved over the bus only on a bandwidth-available basis. If there is bus time that is not being used for other purposes, bulk transfers will be moved over the bus. If there are bulk transfers pending for multiple endpoints, bulk transfers for the different endpoints are selected according to a fair access policy that is Host Controller implementation-dependent.

All bulk transfers pending in a system contend for the same available bus time. Because of this, the bus time made available for bulk transfers to a particular endpoint can be varied by the USB System Software at its discretion. An endpoint and its client software cannot assume a specific rate of service for bulk transfers. Bus time made available to a software client and its endpoint can be changed as other devices are inserted into and removed from the system or also as bulk transfers are requested for other device endpoints. Client software cannot assume ordering between bulk and control transfers; i.e., in some situations, bulk transfers can be delivered ahead of control transfers.

The bus frequency and (micro-)frame timing limit the maximum number of successful bulk transactions within a (micro-)frame for any USB system to less than 72 full-speed eight-byte data payloads or less than 13 high-speed 512-byte data payloads. Table 5-1 lists information about different-sized bulk transactions and the maximum number of transactions possible in a frame. The table does not include the overhead associated with bit stuffing.

Table 5-1. Full-speed Bulk Transaction Limits

	Protocol Overhead (13 bytes)		(3 SYNC bytes, 3 PID bytes, 2 Endpoint + CRC bytes, 2 CRC bytes, and a 3-byte interpacket delay)			
	Data Payload	**Max Bandwidth (bytes/second)**	**Frame Bandwidth per Transfer**	**Max Transfers**	**Bytes Remaining**	**Bytes/Frame Useful Data**
	1	107000	1%	107	2	107
	2	200000	1%	100	0	200
	4	352000	1%	88	4	352
	8	568000	1%	71	9	568
	16	816000	2%	51	21	816
	32	1056000	3%	33	15	1056
	64	1216000	5%	19	37	1216
Max		1500000				1500

Table 5-3. High-speed Interrupt Transaction Limits

Protocol Overhead	<<Based on 480Mb/s and 80 bit bus gap, 640 bit bus turnaround, 16 bit sync, 8 bit EOP. 3 PID bytes, 2 endpoint+CRC5 bytes, 2 CRC16 bytes, 512 data bytes, 3*(2+1) sync/eop bytes, 0 gap bytes, 80 bus turn bytes>>

	Data Payload	**Max Bandwidth (bytes/second)**	**Frame Bandwidth per Transfer**	**Max Transfers**	**Bytes Remaining**	**Bytes/Frame Useful Data**
	512			13		
Max						

Host Controllers are free to determine how the individual bus transactions for specific bulk transfers are moved over the bus within and across (micro)frames. An endpoint could see all bus transactions for a bulk transfer within the same (micro)frame or spread across several (micro)frames. A Host Controller, for various implementation reasons, may not be able to provide the above maximum number of transactions per (micro)frame.

High-speed bulk OUT endpoints must support the PING flow control protocol. The details of this protocol are described in Section 8.5.1.

5.8.5 Bulk Transfer Data Sequences

Bulk transactions use data toggle bits that are toggled only upon successful transaction completion to preserve synchronization between transmitter and receiver when transactions are retried due to errors. Bulk transactions are initialized to DATA0 when the endpoint is configured by an appropriate control transfer. The host will also start the first bulk transaction with DATA0. If a halt condition is detected on an bulk pipe due to transmission errors or a STALL handshake being returned from the endpoint, all pending IRPs are retired. Removal of the halt condition is achieved via software intervention through a separate control pipe. This recovery will reset the data toggle bit to DATA0 for the endpoint on both the host and the device.

Bulk transactions are retried due to errors detected on the bus that affect a given transaction.

5.9 High-Speed, High Bandwidth Endpoints

USB 2.0 supports individual high-speed interrupt or isochronous endpoints that require data rates up to 192 Mb/s (i.e., 3072 data bytes per microframe). This is done by allowing multiple high-speed transactions in a single microframe.

Table 5-4. *wMaxPacketSize* Field of Endpoint Descriptor

Bits	15..14	13..12	11..0
Field	Reserved	Number of transactions per microframe	Maximum size of data payload in bytes

A high-speed interrupt or isochronous endpoint indicates that it requires more than 1024 bytes per microframe by bits 13..12 of the *wMaxPacketSize* field of the endpoint descriptor being non-zero. The lower 12 bits of *wMaxPacketSize* indicate the size of the data payload for each individual transaction while bits 13..12 indicate the maximum number of required transactions possible. See Section 9.6.6 for restrictions on the allowed combinations of values for bits 13..12 and bits 11..0.

NOTE: This representation means that endpoints requesting two transactions per microframe will specify a total data payload size in the microframe that is a multiple of two bytes. Also endpoints requesting three transactions per microframe will specify a total data payload size that is a multiple of three bytes. In any case, any number of bytes can actually be transferred in a microframe.

The host controller must issue an appropriate number of high-speed transactions per microframe. Errors in the host or on bus can result in the host controller issuing fewer transactions than requested for the endpoint. The first transaction(s) must have a data payload(s) as specified by the lower 12 bits of *wMaxPacketSize* if enough data is available, while the last transaction has any remaining data less than or equal to the maximum size specified. The host controller may issue transactions for the same endpoint one immediately after the other (as required for the actual data provided) or may issue transactions for other endpoints in between the transactions for a high bandwidth endpoint.

5.9.1 High Bandwidth Interrupt Endpoints

For interrupt transactions, if the endpoint NAKs a transaction during a microframe, the host controller will not issue further transactions for that endpoint until the next period.

If the endpoint times-out a transaction, the host controller will retry the transaction. The endpoint specifies the maximum number of desired transactions per microframe. If the maximum number of transactions per microframe has not been reached, the host controller will immediately retry the transaction during the current microframe. If the maximum number of transactions per microframe has been reached, the host controller will retry the transaction at the next period for the endpoint.

A host controller is allowed to issue less than a maximum number of transactions to an endpoint per microframe if more than a single memory buffer is required for the transactions within the microframe.

Normal DATA0/DATA1 data toggle sequencing is used for each interrupt transaction during a microframe.

5.9.2 High Bandwidth Isochronous Endpoints

For isochronous transactions, if an IN endpoint provides less than a maximum data payload as specified by its endpoint descriptor, the host must not issue further transactions for that endpoint for that microframe.

For an isochronous OUT endpoint, the host controller must issue the number of transactions as required for the actual data provided, not exceeding the maximum number specified by the endpoint descriptor. The transactions issued must adhere to the maximum payload sizes as specified in the endpoint descriptor.

No retries are ever done for isochronous endpoints.

High bandwidth isochronous endpoints (IN and OUT) must support data PID sequencing. Data PID sequencing provides the required support for the data receiver to detect one or more lost/damaged packets per microframe.

Data PID sequencing for a high-speed, high bandwidth isochronous IN endpoint uses a repeating sequence of DATA2, DATA1, DATA0 PIDs for the data packet of each transaction in a microframe. If there is only a single transaction in the microframe, only a DATA0 data packet PID is used. If there are two transactions per microframe, DATA1 is used for the first transaction data packet and DATA0 is used for the 2nd transaction data packet. If there are three transactions per microframe, DATA2 is used for the first transaction data packet, DATA1 is used for the second and DATA0 is used for the third. In all cases, the data PID sequence starts over again the next microframe.

An endpoint must respond to an IN token for the first transaction with a DATA2 when it requires three transactions of data to be moved. It must respond with a DATA1 for the first transaction when it requires two transactions and with a DATA0 when it requires only a single transaction. After the first transaction, the endpoint follows the data PID sequence described above.

The host knows the maximum number of allowed transactions per microframe for the IN endpoint. The host expects the response to the first transaction to specify (via the data packet PID) how many transactions are required by the endpoint for this microframe. If the host doesn't receive an error-free, appropriate response to any transaction, the host must not issue any further transactions to the endpoint for that microframe. When the host receives a DATA0 data packet from the endpoint, it must not issue any further transactions to the endpoint for that microframe.

Data PID sequencing for a high-speed, high bandwidth isochronous OUT endpoint uses a different sequence than that used for an IN endpoint. The host must issue a DATA0 data packet when there is a single transaction. The host must issue an MDATA for the first transaction and a DATA1 for the second transaction when there are two transactions per microframe. The host must issue two MDATA transactions and a DATA2 for the third transaction when there are three transactions per microframe. These sequences allow the endpoint to detect if there was a lost/damaged transaction during a microframe.

If the wrong OUT transactions are detected by the endpoint, all of the data transferred during the microframe must be treated as if it had encountered an error. Note that for the three transactions per microframe case with a missing MDATA transaction, USB provides no way for the endpoint to determine which of the two MDATA transactions was lost. There may be application specific methods to more precisely determine which data was lost, but USB provides no method to do so at the bus level.

5.10 Split Transactions

Host Controllers and Hubs support one additional transaction type called split transactions. This transaction type allows full- and low-speed devices to be attached to hubs operating at high speed. These transactions involve only Host controllers and Hubs and are not visible to devices. High speed split transactions for interrupt and isochronous transfers must be allocated from the 80% periodic portion of a microframe. More information on split transactions can be found in Chapter 8 and Chapter 11.

5.11 Bus Access for Transfers

Accomplishing any data transfer between the host and a USB device requires some use of the USB bandwidth. Supporting a wide variety of isochronous and asynchronous devices requires that each device's transfer requirements are accommodated. The process of assigning bus bandwidth to devices is called transfer management. There are several entities on the host that coordinate the information flowing over the USB: client software, the USB Driver (USBD), and the Host Controller Driver (HCD). Implementers of these entities need to know the key concepts related to bus access:

- Transfer Management: The entities and the objects that support communication flow over the USB.
- Transaction Tracking: The USB mechanisms that are used to track transactions as they move through the USB system.
- Bus Time: The time it takes to move a packet of information over the bus.
- Device/Software Buffer Size: The space required to support a bus transaction.
- Bus Bandwidth Reclamation: Conditions where bandwidth that was allocated to other transfers but was not used and can now be possibly reused by control and bulk transfers.

The previous sections focused on how client software relates to a function and what the logical flows are over a pipe between the two entities. This section focuses on the different parts of the host and how they must interact to support moving data over the USB. This information may also be of interest to device implementers so they understand aspects of what the host is doing when a client requests a transfer and how that transfer is presented to the device.

5.11.1 Transfer Management

Transfer management involves several entities that operate on different objects in order to move transactions over the bus:

- Client Software: Consumes/generates function-specific data to/from a function endpoint via calls and callbacks requesting IRPs with the USBD interface.
- USB Driver (USBD): Converts data in client IRPs to/from device endpoint via calls/callbacks with the appropriate HCD. A single client IRP may involve one or more transfers.
- Host Controller Driver (HCD): Converts IRPs to/from transactions (as required by a Host Controller implementation) and organizes them for manipulation by the Host Controller. Interactions between the HCD and its hardware is implementation-dependent and is outside the scope of the USB Specification.
- Host Controller: Takes transactions and generates bus activity via packets to move function-specific data across the bus for each transaction.

Figure 5-10 shows how the entities are organized as information flows between client software and the USB. The objects of primary interest to each entity are shown at the interfaces between entities.

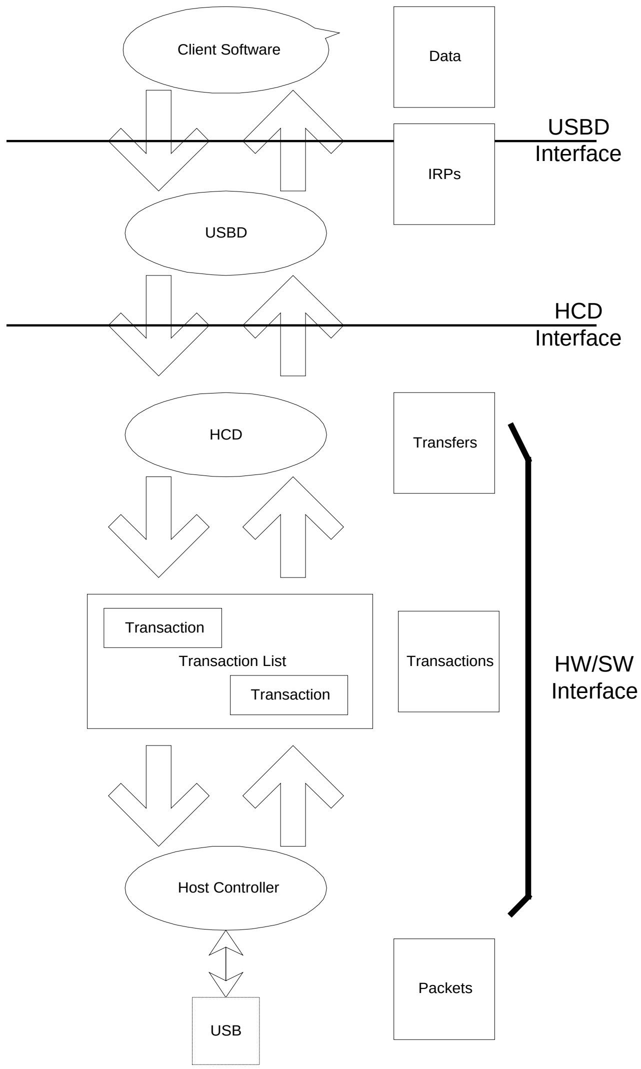


Figure 5-10. USB Information Conversion From Client Software to Bus

5.11.1.1 Client Software

Client software determines what transfers need to be made with a function. It uses appropriate operating system-specific interfaces to request IRPs. Client software is aware only of the set of pipes (i.e., the interface) it needs to manipulate its function. The client is aware of and adheres to all bus access and bandwidth constraints as described previously for each transfer type. The requests made by the client software are presented via the USBD interface.

Some clients may manipulate USB functions via other device class interfaces defined by the operating system and may themselves not make direct USBD calls. However, there is always some lowest level client that makes USBD calls to pass IRPs to the USBD. All IRPs presented are required to adhere to the prenegotiated bandwidth constraints set when the pipe was established. If a function is moved from a non-USB environment to the USB, the driver that would have directly manipulated the function hardware via memory or I/O accesses is the lowest client software in the USB environment that now interacts with the USBD to manipulate the driver's USB function.

After client software has requested a transfer of its function and the request has been serviced, the client software receives notification of the completion status of the IRP. If the transfer involved function-to-host data transfer, the client software can access the data in the data buffer associated with the completed IRP.

The USBD interface is defined in Chapter 10.

5.11.1.2 USB Driver

The Universal Serial Bus Driver (USBD) is involved in mediating bus access at two general times:

- While a device is attached to the bus during configuration
- During normal transfers.

When a device is attached and configured, the USBD is involved to ensure that the desired device configuration can be accommodated on the bus. The USBD receives configuration requests from the configuring software that describe the desired device configuration: endpoint(s), transfer type(s), transfer period(s), data size(s), etc. The USBD either accepts or rejects a configuration request based on bandwidth availability and the ability to accommodate that request type on the bus. If it accepts the request, the USBD creates a pipe for the requester of the desired type and with appropriate constraints as defined for the transfer type. Bandwidth allocation for periodic endpoints does not have to be made when the device is configured and, once made, an bandwidth allocation can be released without changing the device configuration.

The configuration aspects of the USBD are typically operating system-specific and heavily leverage the configuration features of the operating system to avoid defining additional (redundant) interfaces.

Once a device is configured, the software client can request IRPs to move data between it and its function endpoints.

5.11.1.3 Host Controller Driver

The Host Controller Driver (HCD) is responsible for tracking the IRPs in progress and ensuring that USB bandwidth and frame time maximums are never exceeded. When IRPs are made for a pipe, the HCD adds them to the transaction list. When an IRP is complete, the HCD notifies the requesting software client of the completion status for the IRP. If the IRP involved data transfer from the function to the software client, the data was placed in the client-indicated data buffer.

IRPs are defined in an operating system-dependent manner.

5.11.1.4 Transaction List

The transaction list is a Host Controller implementation-dependent description of the current outstanding set of bus transactions that need to be run on the bus. Only the HCD and its Host Controller have access to the specific representation. Each description contains transaction descriptions in which parameters, such as data size in bytes, the device address and endpoint number, and the memory area to which data is to be sent or received, are identified.

A transaction list and the interface between the HCD and its Host Controller is typically represented in an implementation-dependent fashion and is not defined explicitly as part of the USB Specification.

5.11.1.5 Host Controller

The Host Controller has access to the transaction list and translates it into bus activity. In addition, the Host Controller provides a reporting mechanism whereby the status of a transaction (done, pending, halted, etc.) can be obtained. The Host Controller converts transactions into appropriate implementation-dependent activities that result in USB packets moving over the bus topology rooted in the root hub.

The Host Controller ensures that the bus access rules defined by the protocol are obeyed, such as inter-packet timings, timeouts, babble, etc. The HCD interface provides a way for the Host Controller to participate in deciding whether a new pipe is allowed access to the bus. This is done because Host Controller implementations can have restrictions/constraints on the minimum inter-transaction times they may support for combinations of bus transactions.

The interface between the transaction list and the Host Controller is hidden within an HCD and Host Controller implementation.

5.11.2 Transaction Tracking

A USB function sees data flowing across the bus in packets as described in Chapter 8. The Host Controller uses some implementation-dependent representation to track what packets to transfer to/from what endpoints at what time or in what order. Most client software does not want to deal with packetized communication flows because this involves a degree of complexity and interconnect dependency that limits the implementation. The USB System Software (USBD and HCD) provides support for matching data movement requirements of a client to packets on the bus. The Host Controller hardware and software uses IRPs to track information about one or more transactions that combine to deliver a transfer of information between the client software and the function. Figure 5-11 summarizes how transactions are organized into IRPs for the four transfer types. Detailed protocol information for each transfer type can be found in Chapter 8. More information about client software views of IRPs can be found in Chapter 10 and in the operating system specific-information for a particular operating system.

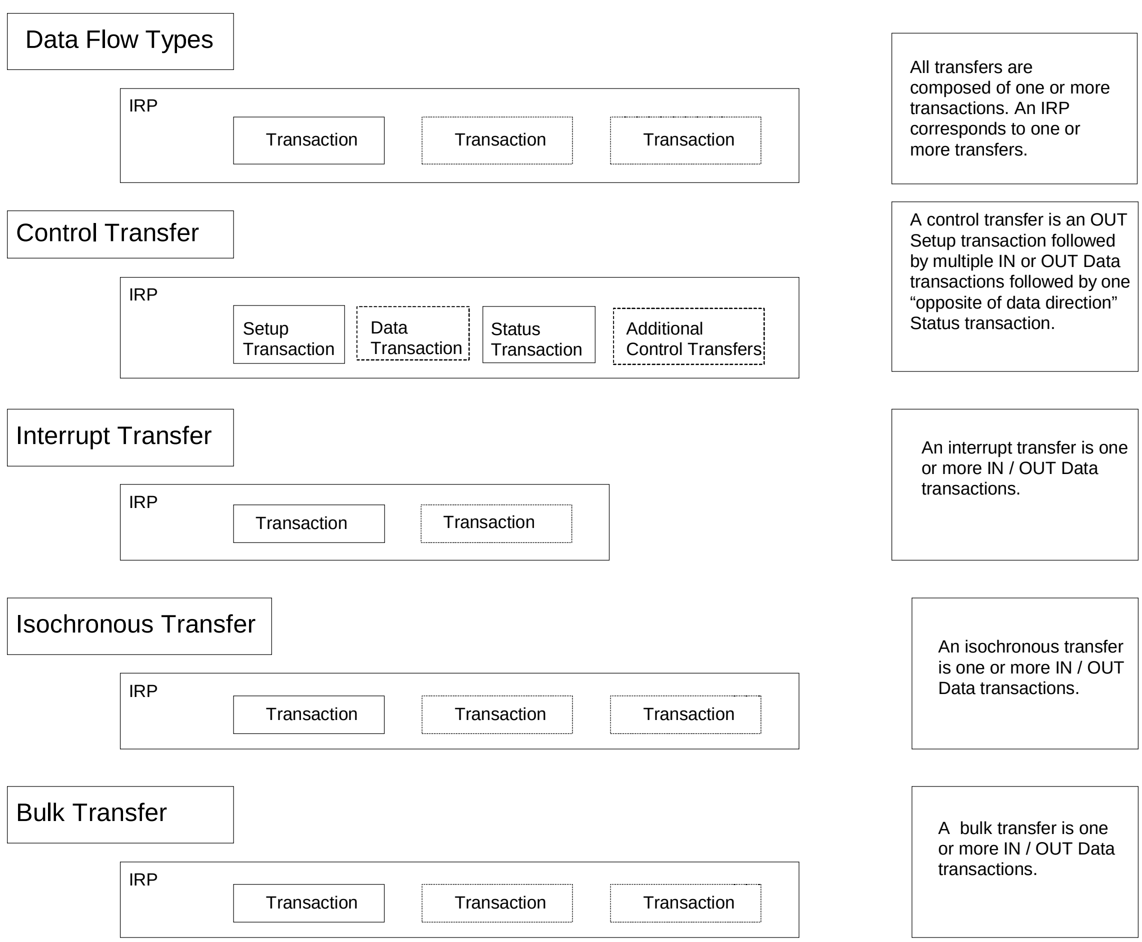


Figure 5-11. Transfers for Communication Flows

Even though IRPs track the bus transactions that need to occur to move a specific data flow over the USB, Host Controllers are free to choose how the particular bus transactions are moved over the bus subject to the USB-defined constraints (e.g., exactly one transaction per frame for isochronous transfers). In any case, an endpoint will see transactions in the order they appear within an IRP unless errors occur. For example, Figure 5-12 shows two IRPs, one each for two pipes where each IRP contains three transactions. For any transfer type, a Host Controller is free to move the first transaction of the first IRP followed by the first transaction of the second IRP somewhere in Frame 1, while moving the second transaction of each IRP in opposite order somewhere in Frame 2. If these are isochronous transfer types, that is the only degree of freedom a Host Controller has. If these are control or bulk transfers, a Host Controller could further move more or less transactions from either IRP within either frame. Functions cannot depend on seeing transactions within an IRP back-to-back within a frame nor should they depend on not seeing transactions back-to-back within a (micro)frame.

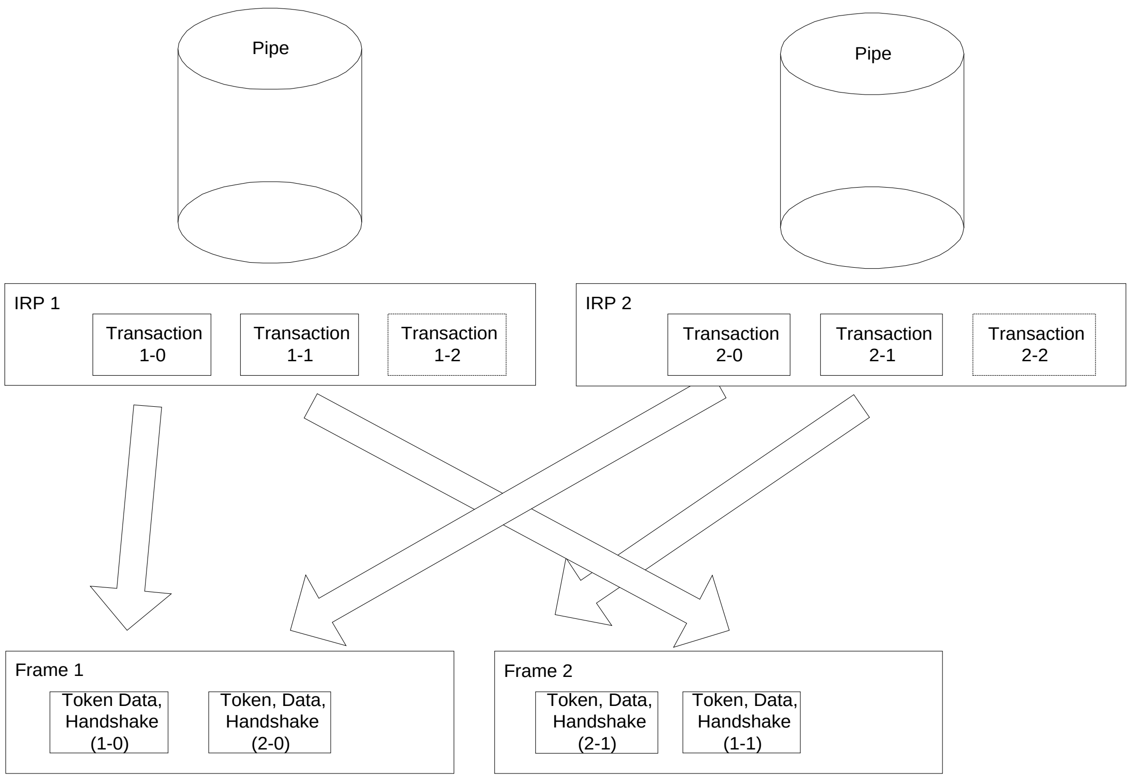


Figure 5-12. Arrangement of IRPs to ~~Transactions/Frames~~Transactions/(Micro)frames

5.11.3 Calculating Bus Transaction Times

When the USB System Software allows a new pipe to be created for the bus, it must calculate how much bus time is required for a given transaction. That bus time is based on the maximum packet size information reported for an endpoint, the protocol overhead for the specific transaction type request, the overhead due to signaling imposed bit stuffing, inter-packet timings required by the protocol, inter-transaction timings, etc. These calculations are required to ensure that the time available in a frame is not exceeded. The equations used to determine transaction bus time are:

KEY:

Data_bc	The byte count of data payload
Host_Delay	The time required for the host to prepare for or recover from the transmission; Host Controller implementation-specific
Floor()	The integer portion of argument
Hub_LS_Setup	The time provided by the Host Controller for hubs to enable low-speed ports; measured as the delay from the end of the PRE PID to the start of the low-speed SYNC; minimum of four full-speed bit times
BitStuffTime	Function that calculates theoretical additional time required due to bit stuffing in signaling; worst case is (1.1667*8*Data_bc)

```
High-speed (Input)

    Non-Isochronous Transfer (Handshake Included)
    =

    Isochronous Transfer (No Handshake)
    =

High-speed (Output)

    Non-Isochronous Transfer (Handshake Included)
    =

    Isochronous Transfer (No Handshake)
    =

Full-speed (Input)

    Non-Isochronous Transfer (Handshake Included)
    = 9107 + (83.54 * Floor(3.167 + BitStuffTime(Data_bc))) + Host_Delay

    Isochronous Transfer (No Handshake)
    = 7268 + (83.54 * Floor(3.167 + BitStuffTime(Data_bc))) + Host_Delay

Full-speed (Output)

    Non-Isochronous Transfer (Handshake Included)
          = 9107 + (83.54 * Floor(3.167 + BitStuffTime(Data_bc))) + Host_Delay

    Isochronous Transfer (No Handshake)
          = 6265 + (83.54 * Floor(3.167 + BitStuffTime(Data_bc))) + Host_Delay
```

```
Low-speed (Input)
          = 64060 + (2 * Hub_LS_Setup) +
                (676.67 * Floor(3.167 + BitStuffTime(Data_bc))) + Host_Delay

Low-speed (Output)
          = 64107 + (2 * Hub_LS_Setup) +
                (667.0 * Floor(3.167 + BitStuffTime(Data_bc))) + Host_Delay
```

The bus times in the above equations are in nanoseconds and take into account propagation delays due to the distance the device is from the host. These are typical equations that can be used to calculate bus time; however, different implementations may choose to use coarser approximations of these times.

The actual bus time taken for a given transaction will almost always be less than that calculated because bit stuffing overhead is data-dependent. Worst case bit stuffing is calculated as 1.1667 (7/6) times the raw time (i.e., the BitStuffTime function multiplies the Data_bc by 8*1.1667 in the equations). This means that there will almost always be time unused on the bus (subject to data pattern specifics) after all regularly scheduled transactions have completed. The bus time made available due to less bit stuffing can be reused as discussed in Section 5.11.5.

The Host_Delay term in the equations is Host Controller- and system-dependent and allows for additional time a Host Controller may require due to delays in gaining access to memory or other implementation dependencies. This term is incorporated into an implementation of these equations by using the transfer management functions provided by the HCD interface. These equations are typically implemented by a combination of USBD and HCD software working in cooperation. The results of these calculations are used to determine whether a transfer or pipe creation can be supported in a given USB configuration.

5.11.4 Calculating Buffer Sizes in Functions and Software

Client software and functions both need to provide buffer space for pending data transactions awaiting their turn on the bus. For non-isochronous pipes, this buffer space needs to be just large enough to hold the next data packet. If more than one transaction request is pending for a given endpoint, the buffering for each transaction must be supplied. Methods to calculate the precise absolute minimum buffering a function may require because of specific interactions defined between its client software and the function are outside the scope of the USB Specification.

The Host Controller is expected to be able to support an unlimited number of transactions pending for the bus subject to available system memory for buffer and descriptor space, etc. Host Controllers are allowed to limit how many frames into the future they allow a transaction to be requested.

For isochronous pipes, Section 5.12.4 describes details affecting host side and device side buffering requirements. In general, buffers need to be provided to hold approximately twice the amount of data that can be transferred in 1ms for full-speed endpoints or 125µs for high-speed endpoints..

5.11.5 Bus Bandwidth Reclamation

The USB bandwidth and bus access are granted based on a calculation of ~~worst case~~worst-case bus transmission time and required latencies. However, due to the constraints placed on different transfer types and the fact that the bit stuffing bus time contribution is calculated as a constant but is data-dependent, there will frequently be bus time remaining in each (micro)frame time versus what the ~~frame~~ (micro)frame transmission time was calculated to be. In order to support the most efficient use of the bus bandwidth, control and bulk transfers are candidates to be moved over the bus as bus time becomes available. Exactly how a Host Controller supports this is implementation-dependent. A Host Controller can take into account the transfer types of pending IRPs and implementation-~~specific~~ specific knowledge of remaining frame time to reuse reclaimed bandwidth.

5.12 Special Considerations for Isochronous Transfers

Support for isochronous data movement between the host and a device is one of the system capabilities supported by the USB. Delivering isochronous data reliably over the USB requires careful attention to detail. The responsibility for reliable delivery is shared by several USB entities:

- The device/function
- The bus
- The Host Controller
- One or more software agents.

Because time is a key part of an isochronous transfer, it is important for USB designers to understand how time is dealt with within the USB by these different entities.

The examples in this section describe USB for an example involving full-speed endpoints. The general example details are also appropriate for high-speed endpoints when corresponding changes are made; for example, frame replaced with microframe, 1ms replaced with 125µs, rate adjustments made between full-speed and high-speed, etc.

All isochronous devices must report their capabilities in the form of device-specific descriptors. The capabilities should also be provided in a form that the potential customer can use to decide whether the device offers a solution to his problem(s). The specific capabilities of a device can justify price differences.

In any communication system, the transmitter and receiver must be synchronized enough to deliver data robustly. In an asynchronous communication system, data can be delivered robustly by allowing the transmitter to detect that the receiver has not received a data item correctly and simply retrying transmission of the data.

In an isochronous communication system, the transmitter and receiver must remain time- and data-synchronized to deliver data robustly. The USB does not support transmission retry of isochronous data so that minimal bandwidth can be allocated to isochronous transfers and time synchronization is not lost due to a retry delay. However, it is critical that a USB isochronous transmitter/receiver pair still remain synchronized both in normal data transmission cases and in cases where errors occur on the bus.

In many systems that deal with isochronous data, a single global clock is used to which all entities in the system synchronize. An example of such a system is the PSTN (Public Switched Telephone Network). Given that a broad variety of devices with different natural frequencies may be attached to the USB, no single clock can provide all the features required to satisfy the synchronization requirements of all devices and software while still supporting the cost targets of mass-market PC products. The USB defines a clock model that allows a broad range of devices to coexist on the bus and have reasonable cost implementations.

This section presents options or features that can be used by isochronous endpoints to minimize behavior differences between a non-USB implemented function and a USB version of the function. An example is included to illustrate the similarities and differences between the non-USB and USB versions of a function.

The remainder of the section presents the following key concepts:

- USB Clock Model: What clocks are present in a USB system that have impact on isochronous data transfers
- USB Frame Clock-to-function Clock Synchronization Options: How the USB frame clock can relate to a function clock
- SOF Tracking: Responsibilities and opportunities of isochronous endpoints with respect to the SOF token and USB frames
- Data Prebuffering: Requirements for accumulating data before generation, transmission, and consumption
- Error Handling: Isochronous-specific details for error handling

- Buffering for Rate Matching: Equations that can be used to calculate buffer space required for isochronous endpoints.

5.12.1 Example Non-USB Isochronous Application

The example used is a reasonably generalized example. Other simpler or more complex cases are possible and the relevant USB features identified can be used or not as appropriate.

The example consists of an 8kHz mono microphone connected through a mixer driver that sends the input data stream to 44kHz stereo speakers. The mixer expects the data to be received and transmitted at some sample rate and encoding. A rate matcher driver on input and output converts the sample rate and encoding from the natural rate and encoding of the device to the rate and encoding expected by the mixer. Figure 5-13 illustrates this example.

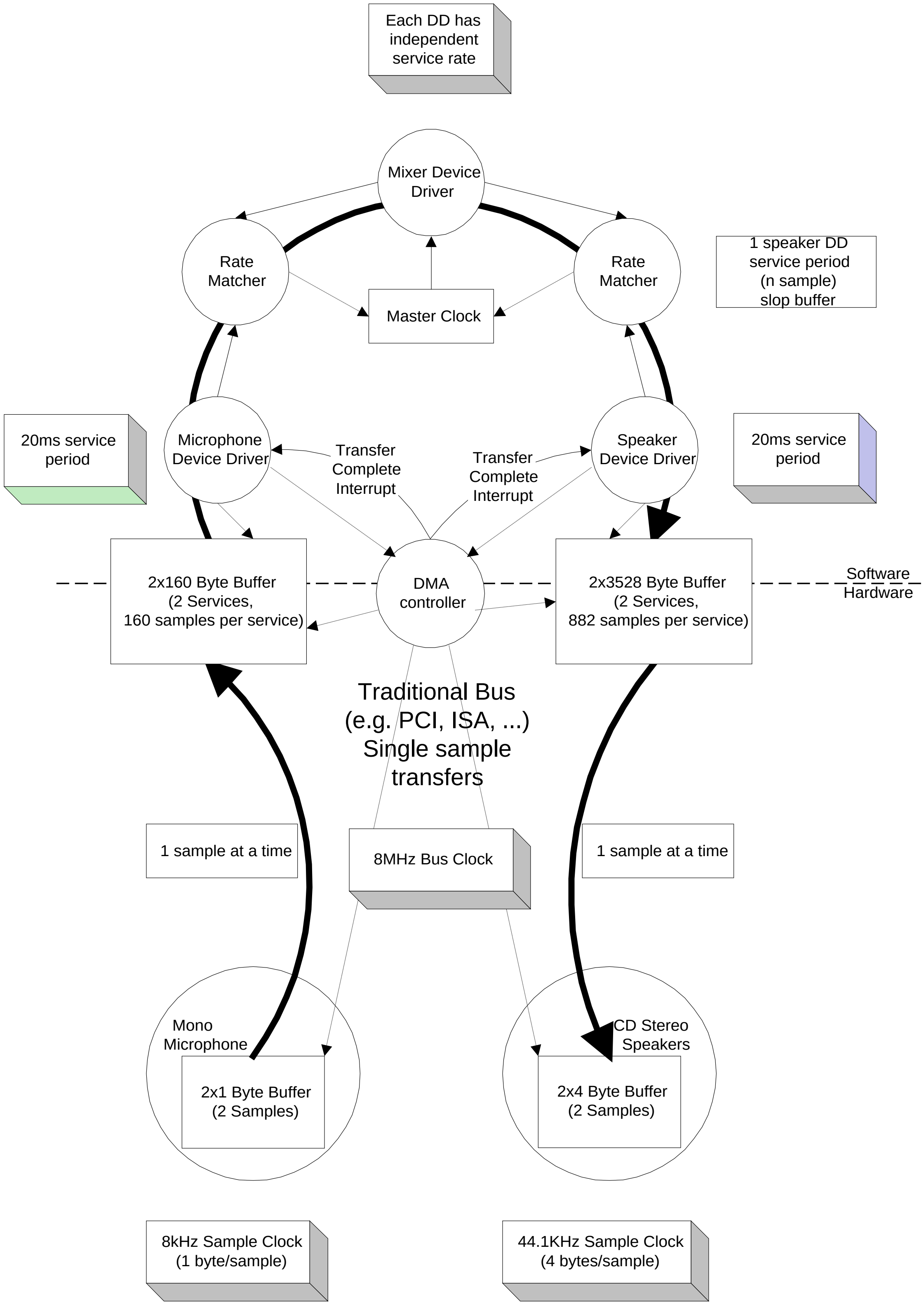


Figure 5-13. Non-USB Isochronous Example

A master clock (which can be provided by software driven from the real time clock) in the PC is used to awaken the mixer to ask the input source for input data and to provide output data to the output sink. In this example, assume it awakens every 20ms. The microphone and speakers each have their own sample clocks that are unsynchronized with respect to each other or the master mixer clock. The microphone produces data at its natural rate (one-byte samples, 8,000 times a second) and the speakers consume data at their natural rate (four-byte samples, 44,100 times a second). The three clocks in the system can drift and jitter with respect to each other. Each rate matcher may also be running at a different natural rate than either the mixer driver, the input source/driver, or output sink/driver.

The rate matchers also monitor the long-term data rate of their device compared to the master mixer clock and interpolate an additional sample or merge two samples to adjust the data rate of their device to the data rate of the mixer. This adjustment may be required every couple of seconds, but typically occurs infrequently. The rate matchers provide some additional buffering to carry through a rate match.

Note: Some other application might not be able to tolerate sample adjustment and would need some other means of accommodating master clock-to-device clock drift or else would require some means of synchronizing the clocks to ensure that no drift could occur.

The mixer always expects to receive exactly a service period of data (20ms service period) from its input device and produce exactly a service period of data for its output device. The mixer can be delayed up to less than a service period if data or space is not available from its input/output device. The mixer assumes that such delays do not accumulate.

The input and output devices and their drivers expect to be able to put/get data in response to a hardware interrupt from the DMA controller when their transducer has processed one service period of data. They expect to get/put exactly one service period of data. The input device produces 160 bytes (ten samples) every service period of 20ms. The output device consumes 3,528 bytes (882 samples) every 20ms service period. The DMA controller can move a single sample between the device and the host buffer at a rate much faster than the sample rate of either device.

The input and output device drivers provide two service periods of system buffering. One buffer is always being processed by the DMA controller. The other buffer is guaranteed to be ready before the current buffer is exhausted. When the current buffer is emptied, the hardware interrupt awakens the device driver and it calls the rate matcher to give it the buffer. The device driver requests a new IRP with the buffer before the current buffer is exhausted.

The devices can provide two samples of data buffering to ensure that they always have a sample to process for the next sample period while the system is reacting to the previous/next sample.

The service periods of the drivers are chosen to survive interrupt latency variabilities that may be present in the operating system environment. Different operating system environments will require different service periods for reliable operation. The service periods are also selected to place a minimum interrupt load on the system, because there may be other software in the system that requires processing time.

5.12.2 USB Clock Model

Time is present in the USB system via clocks. In fact, there are multiple clocks in a USB system that must be understood:

- Sample Clock: This clock determines the natural data rate of samples moving between client software on the host and the function. This clock does not need to be different between non-USB and USB implementations.
- Bus Clock: This clock runs at a 1.000ms period (1kHz frequency) on full-speed segments and 125.000 µs (8kHz frequency) on high-speed segments of the bus and is indicated by the rate of SOF packets on the bus. This clock is somewhat equivalent to the 8MHz clock in the non-USB example. In the USB case, the bus clock is often a lower-frequency clock than the sample clock, whereas the bus clock is almost always a higher-frequency clock than the sample clock in anon- non-USB case.
- Service Clock: This clock is determined by the rate at which client software runs to service IRPs that may have accumulated between executions. This clock also can be the same in the USB and non-USB cases.

In most existing operating systems, it is not possible to support a broad range of isochronous communication flows if each device driver must be interrupted for each sample for fast sample rates. Therefore, multiple samples, if not multiple packets, will be processed by client software and then given to the Host Controller to sequence over the bus according to the prenegotiated bus access requirements. Figure 5-14 presents an example for a reasonable USB clock environment equivalent to the non-USB example in Figure 5-13.

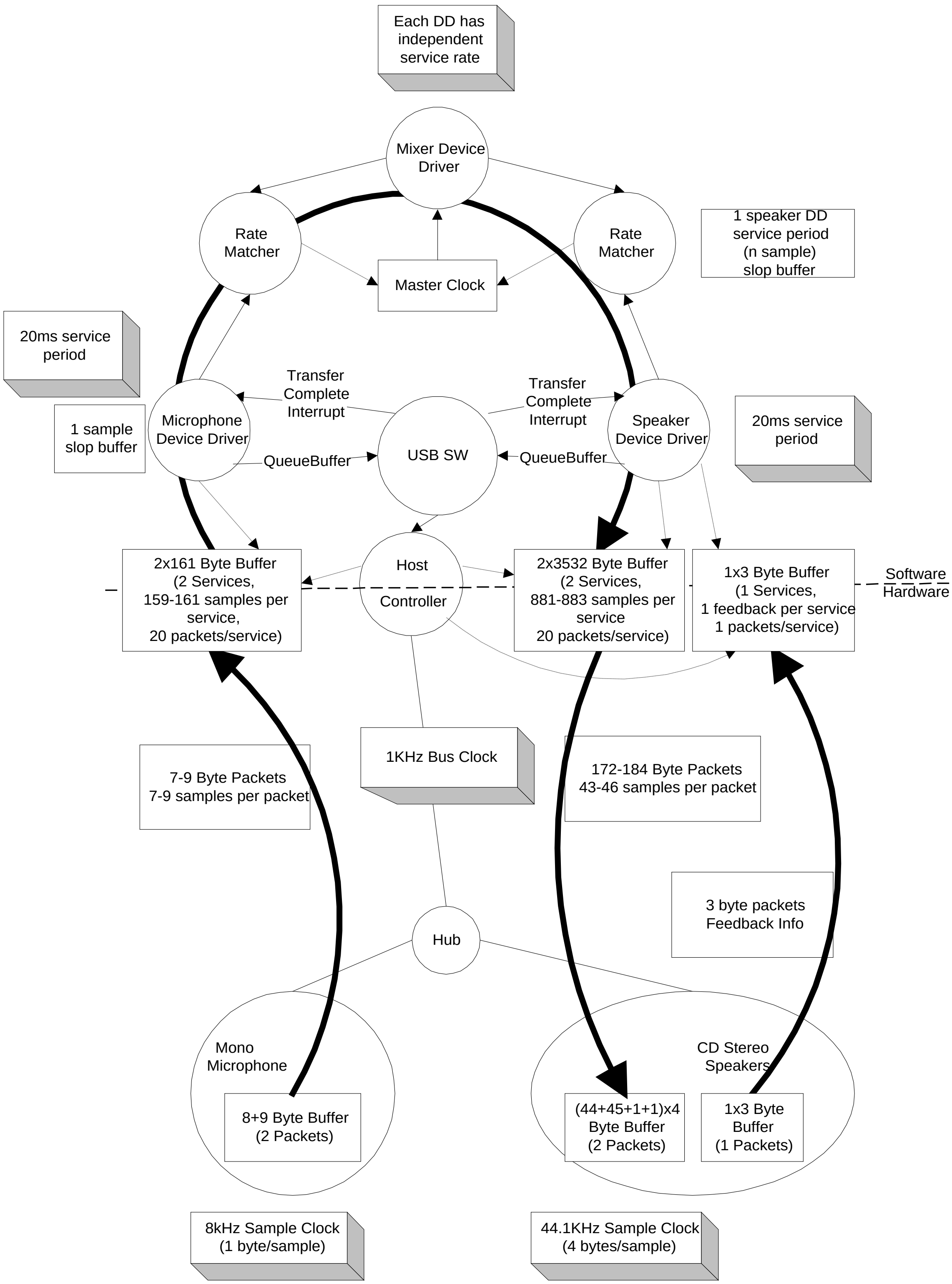


Figure 5-14. USB Full-Speed Isochronous Application

Figure 5-14 shows a typical round trip path of information from a microphone as an input device to a speaker as an output device. The clocks, packets, and buffering involved are also shown. Figure 5-14 will be explored in more detail in the following sections.

The focus of this example is to identify the differences introduced by the USB compared to the previous non-USB example. The differences are in the areas of buffering, synchronization given the existence of a USB bus clock, and delay. The client software above the device drivers can be unaffected in most cases.

5.12.3 Clock Synchronization

In order for isochronous data to be manipulated reliably, the three clocks identified above must be synchronized in some fashion. If the clocks are not synchronized, several clock-to-clock attributes can be present that can be undesirable:

- Clock Drift: Two clocks that are nominally running at the same rate can, in fact, have implementation differences that result in one clock running faster or slower than the other over long periods of time. If uncorrected, this variation of one clock compared to the other can lead to having too much or too little data when data is expected to always be present at the time required.
- Clock Jitter: A clock may vary its frequency over time due to changes in temperature, etc. This may also alter when data is actually delivered compared to when it is expected to be delivered.
- Clock-to-clock Phase Differences: If two clocks are not phase locked, different amounts of data may be available at different points in time as the beat frequency of the clocks cycle out over time. This can lead to quantization/sampling related artifacts.

The bus clock provides a central clock with which USB hardware devices and software can synchronize to one degree or another. However, the software will, in general, not be able to phase- or frequency-lock precisely to the bus clock given the current support for “real time-like” operating system scheduling support in most PC operating systems. Software running in the host can, however, know that data moved over the USB is packetized. For isochronous transfer types, a single packet of data is moved exactly once per ~~frame and the frame~~(micro-)frame and the (micro-)frame clock is reasonably precise. Providing the software with this information allows it to adjust the amount of data it processes to the actual frame time that has passed.

Note: For high-speed high-bandwidth endpoints, the data exchanged in the two or three transactions per microframe is still considered to belong to the same “single packet”. The large amount of data perpacket is split into two or three transactions only for bus efficiency reasons.

5.12.4 Isochronous Devices

The USB includes a framework for isochronous devices that defines synchronization types, how isochronous endpoints provide data rate feedback, and how they can be connected together. Isochronous devices include sampled analog devices (for example, audio and telephony devices) and synchronous data devices. Synchronization type classifies an endpoint according to its capability to synchronize its data rate to the data rate of the endpoint to which it is connected. Feedback is provided by indicating accurately what the required data rate is, relative to the SOF frequency. The ability to make connections depends on the quality of connection that is required, the endpoint synchronization type, and the capabilities of the host application that is making the connection. Additional device class-specific information may be required, depending on the application.

Note: the term “data” is used very generally, and may refer to data that represents sampled analog information (like audio), or it may be more abstract information. “Data rate” refers to the rate at which analog information is sampled, or the rate at which data is clocked.

The following information is required in order to determine how to connect isochronous endpoints:

- Synchronization type:
  - Asynchronous: Unsynchronized, although sinks provide data rate feedback
  - Synchronous: Synchronized to the USB's SOF
  - Adaptive: Synchronized using feedback or feedforward data rate information
- Available data rates
- Available data formats.

Synchronization type and data rate information are needed to determine if an exact data rate match exists between source and sink, or if an acceptable conversion process exists that would allow the source to be connected to the sink. It is the responsibility of the application to determine whether the connection can be supported within available processing resources and other constraints (like delay). Specific USB device classes define how to describe synchronization type and data rate information.

Data format matching and conversion is also required for a connection, but it is not a unique requirement for isochronous connections. Details about format conversion can be found in other documents related to specific formats.

5.12.4.1 Synchronization Type

Three distinct synchronization types are defined. Table 5-5 presents an overview of endpoint synchronization characteristics for both source and sink endpoints. The types are presented in order of increasing capability.

Table 5-5. Synchronization Characteristics

	Source	**Sink**
Asynchronous	Free running F_s Provides implicit feedforward (data stream)	Free running F_s Provides explicit feedback (interrupt pipe)
Synchronous	F_s locked to SOF Uses implicit feedback (SOF)	F_s locked to SOF Uses implicit feedback (SOF)
Adaptive	F_s locked to sink Uses explicit feedback (control pipe)	F_s locked to data flow Uses implicit feedforward (data stream)

5.12.4.1.1 Asynchronous

Asynchronous endpoints cannot synchronize to SOF or any other clock in the USB domain. They source or sink an isochronous data stream at either a fixed data rate (single-frequency endpoints), a limited number of data rates (32kHz, 44.1kHz, 48kHz, …), or a continuously programmable data rate. If the data rate is programmable, it is set during initialization of the isochronous endpoint. Asynchronous devices must report their programming capabilities in the class-specific endpoint descriptor as described in their device class specification. The data rate is locked to a clock external to the USB or to a free-running internal clock. These devices place the burden of data rate matching elsewhere in the USB environment. Asynchronous source endpoints carry their data rate information implicitly in the number of samples they produce per frame. Asynchronous sink endpoints must provide explicit feedback information to an adaptive driver (refer to Section 5.12.4.2).

An example of an asynchronous source is a CD-audio player that provides its data based on an internal clock or resonator. Another example is a Digital Audio Broadcast (DAB) receiver or a Digital Satellite Receiver (DSR). Here too, the sample rate is fixed at the broadcasting side and is beyond USB control.

Asynchronous sink endpoints could be low-cost speakers, running off of their internal sample clock.

~~Another case arises when there are two or more devices present on the USB that need to have mastership control over SOF generation in order to operate as synchronous devices. This could happen if there were two telephony devices, each locked to a different external clock. One telephony device could be digitally connected to a Private Branch Exchange (PBX) that is not synchronized to the ISDN. The other device could be connected directly to the ISDN. Each device will source or sink data to/from the network side at an externally driven rate. Because only one of the devices can take mastership over the SOF, the other will sink or source data at a rate that is asynchronous to the SOF. This example indicates that every device capable of SOF mastership may be forced to operate as an asynchronous device.~~

5.12.4.1.2 Synchronous

Synchronous endpoints can have their clock system (their notion of time) controlled externally through SOF synchronization. These endpoints must~~be doing one of the following~~:

- ~~Slaving~~Slave their sample clock to the 1ms SOF tick (by means of a programmable PLL). For high-speed endpoints, the presence of the microframe SOF can be used for tighter frame clock tracking.

~~Controlling the rate of USB SOF generation so that their data rate becomes automatically locked to SOF. In case these endpoints are not granted SOF mastership, they must degenerate to the asynchronous mode of operation (refer to the asynchronous example).~~

Synchronous endpoints may source or sink isochronous data streams at either a fixed data rate (single-frequency endpoints), a limited number of data rates (32kHz, 44.1kHz, 48kHz, …), or a continuously programmable data rate. If programmable, the operating data rate is set during initialization of the isochronous endpoint. The number of samples or data units generated in a series of USB frames is deterministic and periodic. Synchronous devices must report their programming capabilities in the class-specific endpoint descriptor as described in their device class specification.

An example of a synchronous source is a digital microphone that synthesizes its sample clock from SOF and produces a fixed number of audio samples every USB frame. ~~Another possibility is a 64kb/s bit-stream from an ISDN "modem." If the USB SOF generation is locked to the PSTN clock (perhaps through the same ISDN device), the data generation will also be locked to SOF and the~~ Likewise, a synchronous sink derives its sample clock from SOF and consumes a fixed number of samples every USB frame.

~~endpoint will produce a stable 64kb/s data stream, referenced to the SOF time notion.~~

5.12.4.1.3 Adaptive

Adaptive endpoints are the most capable endpoints possible. They are able to source or sink data at any rate within their operating range. Adaptive source endpoints produce data at a rate that is controlled by the data sink. The sink provides feedback (refer to Section 5.12.4.2) to the source, which allows the source to know the desired data rate of the sink. Adaptive endpoints can communicate with all types of sink endpoints. For adaptive sink endpoints, the data rate information is embedded in the data stream. The average number of samples received during a certain averaging time determines the instantaneous data rate. If this number changes during operation, the data rate is adjusted accordingly.

The data rate operating range may center around one rate (e.g., 8kHz), select between several programmable or auto-detecting data rates (32kHz, 44.1kHz, 48kHz, …), or may be within one or more ranges (e.g., 5kHz to 12kHz or 44kHz to 49kHz). Adaptive devices must report their programming capabilities in the class-specific endpoint descriptor as described in their device class specification

An example of an adaptive source is a CD player that contains a fully adaptive sample rate converter (SRC) so that the output sample frequency no longer needs to be 44.1kHz but can be anything within the operating range of the SRC. Adaptive sinks include such endpoints as high-end digital speakers, headsets, etc.

5.12.4.2 Feedback

An asynchronous sink ~~provides feedback to an adaptive source~~must provide explixit feedback to the host by indicating accurately what its desired data rate (F_f) is, relative to the USB (micro)SOF frequency. This allows the host to continuously adjust the number of samples sent to the sink so that neither underflow or overflow of the data buffer occurs. Likewise, an adaptive source must receive explicit feedback from the host so that it can accurately generate the number of samples required by the host.

~~SOF frequency. The required data rate is accurate~~To generate the desired data rate F_f, the device must measure its actual sampling rate F_s, referenced to the USB notion of time, i.e. the USB (micro)SOF frequency. This specification requires the data rate F_f to be resolved to better than one sample per second (1Hz) in order to allow a high-quality source rate to be created and to tolerate delays and errors in the feedback ~~loop.~~

loop. To achieve this ~~The Ff value consists of a fractional part, in order to get the required resolution with 1kHz frames, and an integer part, which gives the minimum number of samples per frame. Ten bits are required to resolve one sample within a 1kHz frame frequency (1000 / 2^10 = 0.98). This is a ten-bit fraction, represented in unsigned fixed binary point 0.10 format. The integer part needs ten bits (2^10~~ accuracy, the measurement time T_{meas} must be at least 1 second. Therefore:

$$T_{meas} = 2^K$$

where T_{meas} is now expressed in USB (micro)frames and K=10 for full-speed devices (1 ms frames) and K=13 for high-speed devices (125 µs microframes).However, in most devices, the actual sampling rate F_s is derived from a master clock F_m through a binary divider. Therefore:

$$F_m = F_s * 2^P$$

where P is a positive integer (including 0 if no higher-frequency master clock is available) The measurement time T_{meas} can now be decreased by measuring F_m instead of F_s and:

$$T_{meas} = \frac{2^K}{2^P} = 2^{(K-P)}$$

In this way, a new estimate for F_f becomes available every $2^{(K-P)}$ (micro)frames. P is practically bound to be in the range [0,K] because there is no point in using a clock slower than F_s (P=0), and no point in trying to update F_f more than once per (micro)frame (P=K).A sink can determine F_f by counting cycles of the master clock F_m for a period of $2^{(K-P)}$(micro)frames. The counter is read into F_f and reset every $2^{(K-P)}$(micro)frames. As long as no clock cycles are skipped, the count will be accurate over the long term.

Each (micro)frame, an adaptive source adds F_f to any remaining fractional sample count from the previous (micro)frame, sources the number of samples in the integer part of the sum, and retains the fractional sample count for the next (micro)frame. The source can look at the behavior of F_f over many (micro)frames to determine an even more accurate rate, if it needs to.

F_f is expressed in number of samples per (micro)frame. The F_f value consists of an integer part that represents the (integer) number of samples per (micro)frame and a fractional part that represents the 'fraction' of a sample that would be needed to match the sampling frequency F_s to a resolution of 1 Hz or better. The fractional part requires at least K bits to represent the 'fraction' of a sample to a resolution of 1 Hz or better. The integer part must have enough bits to represent the maximum number of samples that can ever occur in a single (micro)frame. Assuming that the minimum sample size is one byte, then this number is limited to 1,023 for full-speed endpoints. Ten bits are therefore sufficient to encode this value. For high-speed endpoints this number is limited to 3*1,024=3,072 and twelve bits are needed.

= 1024) to encode up to 1,023 one-byte samples per frame. The ten-bit integer is represented in unsigned fixed binary point 10.0 format. The combined F_f value can be coded in unsigned fixed binary point 10.10 format, which fits into three bytes (24 bits). BecauseIn summary, for full-speed endpoints the F_f value shall be encoded in an unsigned 10.10 (K=10) format which fits into three bytes.Because the maximum integer value is fixed to 1,023, the 10.10 number will be left-justified in the 24 bits, so that it has a 10.14 format. Only the first ten bits behind the binary point are required. The lower four bits may be optionally used to extend the precision of F_f, otherwise, they shall be reported as zero. The bit and byte ordering follows the definitions of other multi-byte fields contained in Chapter 8.

For high-speed endpoints the F_f value shall be encoded in an unsigned 12.13 (K=13) format which fits into four bytes. The value shall be aligned into these four bytes so that the binary point is located between the second and the third byte so that it has a 16.16 format. The most significant four bits shall be reported zero. Only the first 13 bits Each frame, the adaptive source adds F_f to any remaining fractional sample count from the previous frame, sources the number of samples in the integer part of the sum, and retains the fractional sample count for the next frame. The source can look at the behavior of F_f over many frames to determine an even more accurate rate, if it needs to.

The sink can determine F_f by counting cycles of a clock with a frequency of F_s * 2^P for a period of behind the binary point are required. The lower three bits may be optionally used to extend the precision of F_f, otherwise, they shall be reported as zero.

2^(10-P) frames, where P is an integer. P is practically bound to be in the range [0,10] because there is no point in using a clock slower than Fs, and no point in trying to update more than once a frame. The counter is read into F_f and reset every 2^(10-P) frames. As long as no clock cycles are skipped, the count will be accurate over the long term. An endpoint needs to implement only the number ofcounter bits that it effectively requires for its maximum F_f.

A digital telephony endpoint, for example, will usually derive its 8kHz F_s by dividing down the 64kHz clock (P=3) which it uses to serialize the data stream. The 64kHz clock phase can also give an additional one bit of accuracy, effectively giving P=4. This would give F_f updates every 2^(10-4) = 64 frames. A 13-bit counter would be required to obtain F_f, with three bits for eight samples per frame, and ten bits for the fractional part. The 13 bits would provide a 3.10 field within the 10.14 F_f value, with the remaining bits set to zero.

The choice of P is endpoint-specific. Use the following guidelines when choosing P:

- P shouldmust be in the range [1,9]. [0,K].
- Larger values of P are preferred, because they reduce the size of the frame counter and increase the rate at which F_f is updated. More frequent updates result in a tighter control of the source data rate, which reduces the buffer space required to handle F_f changes.
- P should be less than ten$K$ so that F_f is averaged across at least two frames in order to reduce SOF jitter effects.

- P should not be zero in order to keep the deviation in the number of samples sourced to less than 1 in the event of a lost F_f value.

Isochronous transfers are used to read F_f from the feedback register. The desired reporting rate for the feedback should be 2^(10-P) frames. $2^{(K-P)}$ frames. F_f will be reported at most once per update period. There is nothing to be gained by reporting the same F_f value more than once per update period. The endpoint may choose to report F_f only if the updated value has changed from the previous F_f value.

It is possible that the source will deliver one too many or one too few samples over a long period, due to errors or accumulated inaccuracies in measuring F_f. The sink must have sufficient buffer capability to accommodate this. When the sink recognizes this condition, it should adjust the reported F_f value to correct it. This may also be necessary to compensate for relative clock drifts. The implementation of this correction process is endpoint-specific and is not specified.

An adaptive source may obtain the sink data rate information from an adaptive sink that is locked to the same clock as the sink, as would be the case for a two-way speech connection. In this case, the feedback pipe is not needed.

5.12.4.3 Connectivity

In order to fully describe the source-to-sink connectivity process, an interconnect model is presented. The model indicates the different components involved and how they interact to establish the connection.

The model provides for multi-source/multi-sink situations. Figure 5-15 illustrates a typical situation (highly condensed and incomplete). A physical device is connected to the host application software through different hardware and software layers as described in the USB Specification. At the client interface level, a virtual device is presented to the application. From the application standpoint, only virtual devices exist. It is up to the device driver and client software to decide what the exact relation is between physical and virtual device.

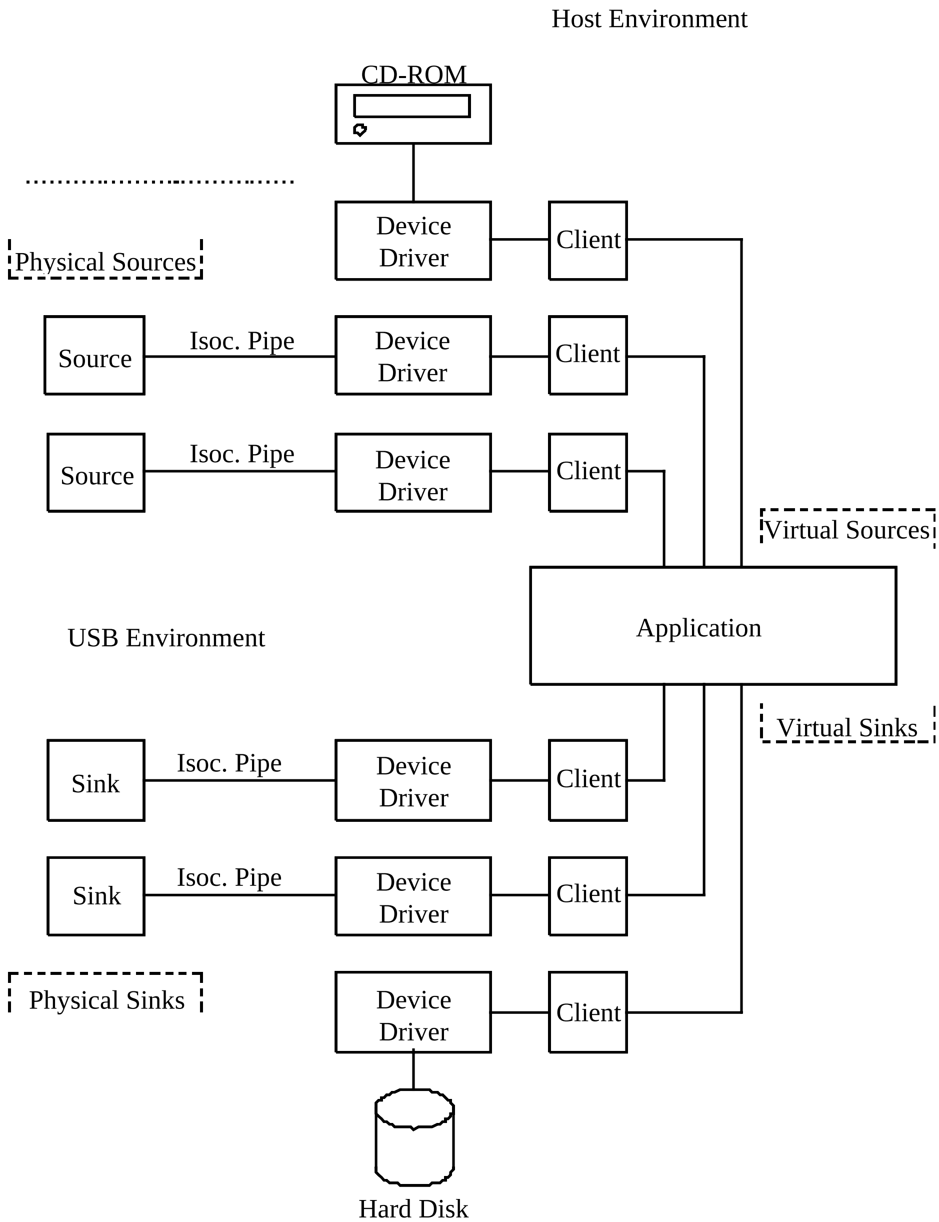


Figure 5-15. Example Source/Sink Connectivity

Device manufacturers (or operating system vendors) must provide the necessary device driver software and client interface software to convert their device from the physical implementation to a USB-compliant software implementation (the virtual device). As stated before, depending on the capabilities built into this software, the virtual device can exhibit different synchronization behavior from the physical device. However, the synchronization classification applies equally to both physical and virtual devices. All physical devices belong to one of the three possible synchronization types. Therefore, the capabilities that have to be built into the device driver and/or client software are the same as the capabilities of a physical device. The word "application" must be replaced by "device driver/client software." In the case of a physical source to virtual source connection, "virtual source device" must be replaced by "physical source device" and "virtual sink device" must be replaced by "virtual source device." In the case of a virtual sink to physical sink connection, "virtual source device" must be replaced by "virtual sink device" and "virtual sink device" must be replaced by "physical sink device."

Placing the rate adaptation (RA) functionality into the device driver/client software layer has the distinct advantage of isolating all applications, relieving the device from the specifics and problems associated with rate adaptation. Applications that would otherwise be multi-rate degenerate to simpler mono-rate systems.

Note: the model is not limited to only USB devices. For example, a CD-ROM drive containing 44.1kHz audio can appear as either an asynchronous, synchronous, or adaptive source. Asynchronous operation means that the CD-ROM fills its buffer at the rate that it reads data from the disk, and the driver empties the buffer according to its USB service interval. Synchronous operation means that the driver uses the USB service interval (e.g., 10ms) and nominal sample rate of the data (44.1kHz) to determine to put out 441 samples every USB service interval. Adaptive operation would build in a sample rate converter to match the CD-ROM output rate to different sink sampling rates.

Using this reference model, it is possible to define what operations are necessary to establish connections between various sources and sinks. Furthermore, the model indicates at what level these operations must or can take place. First there is the stage where physical devices are mapped onto virtual devices and vice versa. This is accomplished by the driver and/or client software. Depending on the capabilities included in this software, a physical device can be transformed into a virtual device of an entirely different synchronization type. The second stage is the application that uses the virtual devices. Placing rate matching capabilities at the driver/client level of the software stack relieves applications communicating with virtual devices from the burden of performing rate matching for every device that is attached to them. Once the virtual device characteristics are decided, the actual device characteristics are not any more interesting than the actual physical device characteristics of another driver.

As an example, consider a mixer application that connects at the source side to different sources, each running at their own frequencies and clocks. Before mixing can take place, all streams must be converted to a common frequency and locked to a common clock reference. This action can be performed in the physical-to-virtual mapping layer or it can be handled by the application itself for each source device independently. Similar actions must be performed at the sink side. If the application sends the mixed data stream out to different sink devices, it can either do the rate matching for each device itself or it can rely on the driver/client software to do that, if possible.

Table 5-6 indicates at the intersections what actions the application must perform to connect a source endpoint to a sink endpoint.

Table 5-6. Connection Requirements

	Source Endpoint		
Sink Endpoint	**Asynchronous**	**Synchronous**	**Adaptive**
Asynchronous	Async Source/Sink RA See Note 1.	Async SOF/Sink RA See Note 2.	Data + Feedback Feedthrough See Note 3.
Synchronous	Async Source/SOF RA See Note 4.	Sync RA See Note 5.	Data Feedthrough + Application Feedback See Note 6.
Adaptive	Data Feedthrough See Note 7.	Data Feedthrough See Note 8.	Data Feedthrough See Note 9.

Notes:

1. Asynchronous RA in the application. Fs_i is determined by the source, using the feedforward information embedded in the data stream. Fs_o is determined by the sink, based on feedback information from the sink. If nominally $Fs_i = Fs_o$, the process degenerates to a feedthrough connection if slips/stuffs due to lack of synchronization are tolerable. Such slips/stuffs will cause audible degradation in audio applications.
2. Asynchronous RA in the application. Fs_i is determined by the source but locked to SOF. Fs_o is determined by the sink, based on feedback information from the sink. If nominally $Fs_i = Fs_o$, the process degenerates to a feedthrough connection if slips/stuffs due to lack of synchronization are tolerable. Such slips/stuffs will cause audible degradation in audio applications.
3. If Fs_o falls within the locking range of the adaptive source, a feedthrough connection can be established. $Fs_i = Fs_o$ and both are determined by the asynchronous sink, based on feedback information from the sink. If Fs_o falls outside the locking range of the adaptive source, the adaptive source is switched to synchronous mode and Note 2 applies.
4. Asynchronous RA in the application. Fs_i is determined by the source. Fs_o is determined by the sink and locked to SOF. If nominally $Fs_i = Fs_o$, the process degenerates to a feedthrough connection if slips/stuffs due to lack of synchronization are tolerable. Such slips/stuffs will cause audible degradation in audio applications.
5. Synchronous RA in the application. Fs_i is determined by the source and locked to SOF. Fs_o is determined by the sink and locked to SOF. If $Fs_i = Fs_o$, the process degenerates to a loss-free feedthrough connection.
6. The application will provide feedback to synchronize the source to SOF. The adaptive source appears to be a synchronous endpoint and Note 5 applies.
7. If Fs_i falls within the locking range of the adaptive sink, a feedthrough connection can be established. $Fs_i = Fs_o$ and both are determined by and locked to the source.
   If Fs_i falls outside the locking range of the adaptive sink, synchronous RA is done in the host to provide an Fs_o that is within the locking range of the adaptive sink.
8. If Fs_i falls within the locking range of the adaptive sink, a feedthrough connection can be established. $Fs_o = Fs_i$ and both are determined by the source and locked to SOF.
   If Fs_i falls outside the locking range of the adaptive sink, synchronous RA is done in the host to provide an Fs_o that is within the locking range of the adaptive sink.
9. The application will use feedback control to set Fs_o of the adaptive source when the connection is set up. The adaptive source operates as an asynchronous source in the absence of ongoing feedback information and Note 7 applies.

In cases where RA is needed but not available, the rate adaptation process could be mimicked by sample dropping/stuffing. The connection could then still be made, possibly with a warning about poor quality; otherwise, the connection cannot be made.

5.12.4.3.1 Audio Connectivity

When the above is applied to audio data streams, the RA process is replaced by sample rate conversion, which is a specialized form of rate adaptation. Instead of error control, some form of sample interpolation is used to match incoming and outgoing sample rates. Depending on the interpolation techniques used, the audio quality (distortion, signal to noise ratio, etc.) of the conversion can vary significantly. In general, higher quality requires more processing power.

5.12.4.3.2 Synchronous Data Connectivity

For the synchronous data case, RA is used. Occasional slips/stuffs may be acceptable to many applications that implement some form of error control. Error control includes error detection and discard, error detection and retransmit, or forward error correction. The rate of slips/stuffs will depend on the clock mismatch between the source and sink, and may be the dominant error source of the channel. If the error control is sufficient, then the connection can still be made.

5.12.5 Data Prebuffering

The USB requires that devices prebuffer data before processing/transmission to allow the host more flexibility in managing when each pipe’s transaction is moved over the bus from ~~frame to frame.~~(micro)frame to (micro)frame.

For transfers from function to host, the endpoint must accumulate samples during (micro)frame X until it receives the SOF token for (micro)frame X+1. It “latches” the data from (micro)frame X into its packet buffer and is now ready to send the packet containing those samples during (micro)frame X+1. When it will send that data during the (micro)frame is determined solely by the Host Controller and can vary from ~~frame to~~ (micro)frame to (micro)frame.

~~frame.~~

For transfers from host to function, the endpoint will accept a packet from the host sometime during (micro)frame Y. When it receives the SOF for (micro)frame Y+1, it can then start processing the data received in (micro)frame Y.

This approach allows an endpoint to use the SOF token as a stable clock with very little jitter and/or drift when the Host Controller moves the packet over the bus. This approach also allows the Host Controller to vary within a (micro)frame precisely when the packet is actually moved over the bus. This prebuffering introduces some additional delay between when a sample is available at an endpoint and when it moves over the bus compared to an environment where the bus access is at exactly the same time offset from SOF from ~~frame to frame.~~(micro)frame to (micro)frame.

Figure 5-16 shows the time sequence for a function-to-host transfer (IN process). Data D_0 is accumulated during (micro)frame F_i at time T_i, and transmitted to the host during (micro)frame F_{i+1}. Similarly, for a host-to-function transfer (OUT process), data D_0 is received by the endpoint during (micro)frame F_{i+1} and processed during (micro)frame F_{i+2}.

Time:	T_i	T_{i+1}	T_{i+2}	T_{i+3}	------	------	T_m	T_{m+1}	------
(Micro)Frame:	F_i	F_{i+1}	F_{i+2}	M_{i+3}	------	------	F_m	F_{m+1}	------
Data on Bus:	------	D_0	D_1	D_2	------	------	D_0	D_1	------
OUT Process:	------	------	D_0	D_1	------	------	------	D_0	------
IN Process:	D_0	D_1	------	------	------	D_0	------	------	------

Figure 5-16. Data Prebuffering

5.12.6 SOF Tracking

Functions supporting isochronous pipes must receive and comprehend the SOF token to support prebuffering as previously described. Given that SOFs can be corrupted, a device must be prepared to recover from a corrupted SOF. These requirements limit isochronous transfers to full-speed and high-speed devices only, because low-speed devices do not see SOFs on the bus. Also, because SOF packets can be damaged in transmission, devices that support isochronous transfers need to be able to synthesize the existence of an SOF that they may not see due to a bus error.

Isochronous transfers require the appropriate data to be transmitted in the corresponding (micro)frame. The USB requires that when an isochronous transfer is presented to the Host Controller, it identifies the (micro)frame number for the first (micro)frame. The Host Controller must not transmit the first transaction before the indicated (micro)frame number. Each subsequent transaction in the IRP must be transmitted in succeeding ~~frames.~~(micro)frames (except for high-speed high-bandwidth transfers where up to three transactions may occur in the same microframe). If there are no transactions pending for the current (micro)frame, then the Host Controller must not transmit anything for an isochronous pipe. If the indicated (micro)frame number has passed, the Host Controller must skip (i.e., not transmit) all transactions until the one corresponding to the current (micro)frame is reached.

5.12.7 Error Handling

Isochronous transfers provide no data packet retries (i.e., no handshakes are returned to a transmitter by a receiver) so that timeliness of data delivery is not perturbed. However, it is still important for the agents responsible for data transport to know when an error occurs and how the error affects the communication flow. In particular, for a sequence of data packets (A, B, C, D), the USB allows sufficient information such that a missing packet (A, _, C, D) can be detected and will not unknowingly be turned into an incorrect data or time sequence (A, C, D or A, _, B, C, D). The protocol provides four mechanisms that support this: ~~exactly one packet per frame,~~a strictly defined periodicity for the transmission of packets and data PID sequencing mechanisms for high-speed high-bandwidth endpoints, SOF, CRC, and bus transaction timeout.

- Isochronous transfers require ~~exactly one data transaction every frame~~periodic occurrence of data transactions for normal operation. The period must be an exact power of two (micro)frames. The USB does not dictate what data is transmitted in each frame. The data transmitter/source determines specifically what data to provide. This regular ~~data-per-frame~~periodic data delivery provides a framework that is fundamental to detecting missing data errors. For high-speed high-bandwidth endpoints, data PID sequencing allows the detection of missing or damaged transactions during a microframe. Any phase of a transaction can be damaged during transmission on the bus. ~~Chapter 8~~Chapter 8 describes how each error case affects the protocol.

- Because every (micro)frame is preceded by an SOF and a receiver can see SOFs on the bus, a receiver can determine that its expected transaction for that (micro)frame did not occur between two SOFs. Additionally, because even an SOF can be damaged, a device must be able to reconstruct the existence of a missed SOF as described in Section 5.12.6.

- A data packet may be corrupted on the bus; therefore, CRC protection allows a receiver to determine that the data packet it received was corrupted.

- The protocol defines the details that allow a receiver to determine via bus transaction timeout that it is not going to receive its data packet after it has successfully seen its token packet.

Once a receiver has determined that a data packet was not received, it may need to know the size of the data that was missed in order to recover from the error with regard to its functional behavior. If the communication flow is always the same data size per (micro)frame, then the size is always a known constant. However, in some cases the data size can vary from (micro)frame to ~~frame to frame.~~(micro)frame.

In this case, the receiver and transmitter have an implementation-dependent mechanism to determine the size of the lost packet.

In summary, whether a transaction is actually moved successfully over the bus or not, the transmitter and receiver always advance their data/buffer streams ~~one transaction per frame~~as indicated by the bus access period to keep data-per-time synchronization. The detailed mechanisms described above allow detection, tracking, and reporting of damaged transactions so that a function or its client software can react to the damage in a function-appropriate fashion. The details of that function- or application-specific reaction are outside the scope of the USB Specification.

5.12.8 Buffering for Rate Matching

Given that there are multiple clocks that affect isochronous communication flows in the USB, buffering is required to rate match the communication flow across the USB. There must be buffer space available both in the device per endpoint and on the host side on behalf of the client software. These buffers provide space for data to accumulate until it is time for a transfer to move over the USB. Given the natural data rates of the device, the maximum size of the data packets that move over the bus can also be calculated.

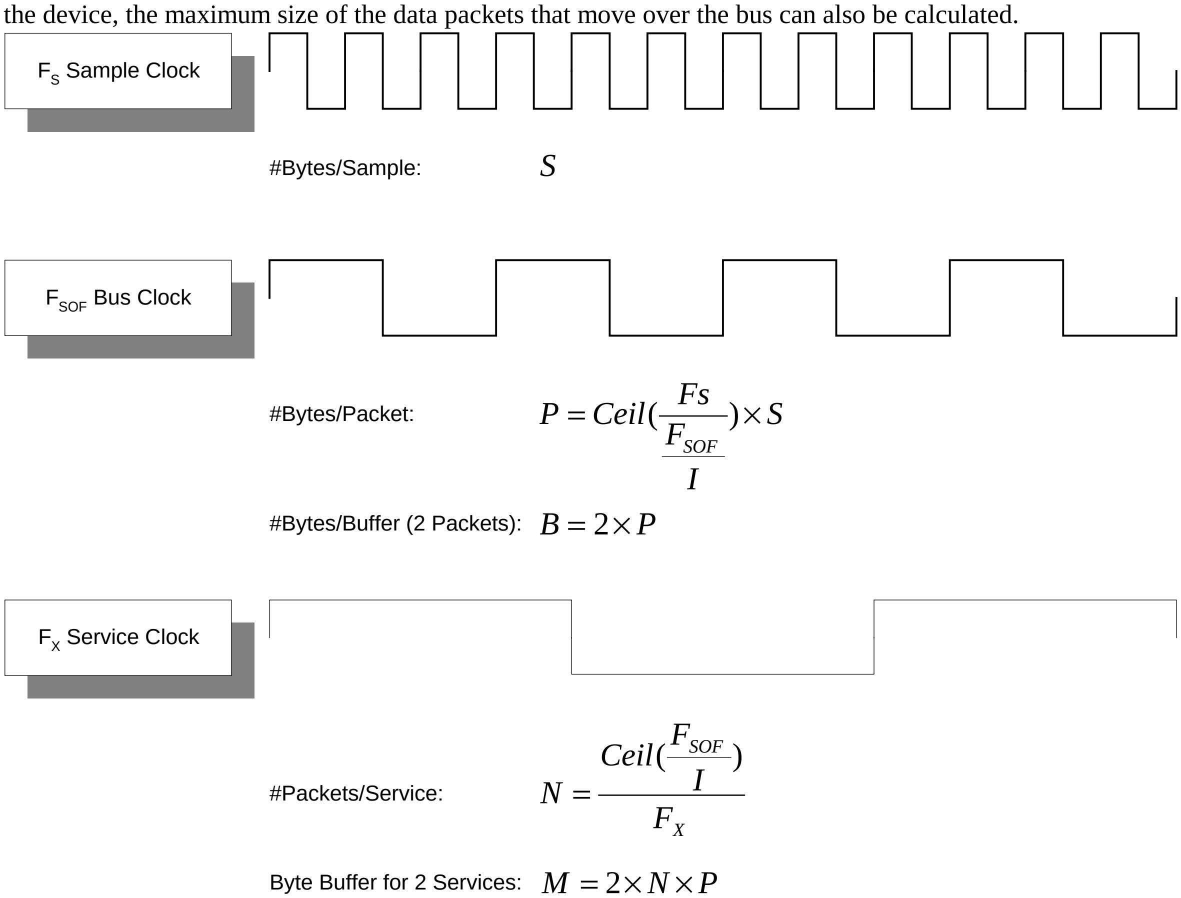
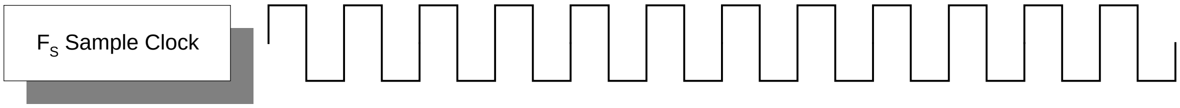
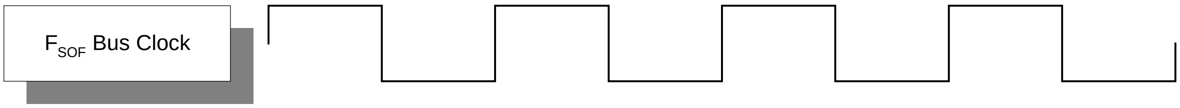


Figure 5-17 shows the equations used to determine buffer size on the device and host and maximum packet size that must be requested to support a desired data rate. These equations ~~allow a device and client software design time-determined~~are a function of the service clock rate ~~(variable X),~~(F_X), bus clock rate (F_{SOF}), sample clock rate ~~(variable C),~~(F_S), bus access period (I), and sample size ~~(variable S). The USB allows only one transaction per bus clock.~~ (S). These equations should provide design information for selecting the appropriate packet size that an endpoint will report in its characteristic information and the appropriate buffer requirements for the device/endpoint and its client software. Figure 5-14 shows actual buffer, packet, and clock values for a typical full-speed isochronous example.

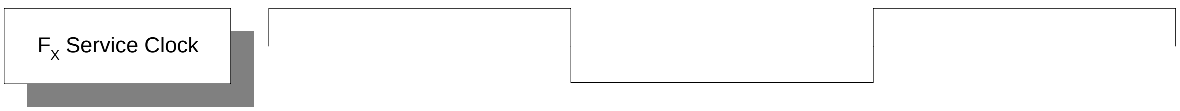


#Bytes/Sample: S



#Bytes/Packet: $P = Ceil\left(\frac{Fs}{\frac{F_{SOF}}{I}}\right) \times S$

#Bytes/Buffer (2 Packets): $B = 2 \times P$



#Packets/Service: $N = \frac{Ceil\left(\frac{F_{SOF}}{I}\right)}{F_X}$

Byte Buffer for 2 Services: $M = 2 \times N \times P$

Figure 5-17. Packet and Buffer Size Formulas for Rate-Matched Isochronous Transfers

The USB data model assumes that devices have some natural sample size and rate. The USB supports the transmission of packets that are multiples of sample size to make error recovery handling easier when isochronous transactions are damaged on the bus. If a device has no natural sample size or if its samples are larger than a packet, it should describe its sample size as being one byte. If a sample is split across a data packet, the error recovery can be harder when an arbitrary transaction is lost. In some cases, data synchronization can be lost unless the receiver knows in what (micro-)frame number each partial sample is transmitted. Furthermore, if the number of samples can vary due to clock correction (e.g., for a non-derived device clock), it may be difficult or inefficient to know when a partial sample is transmitted. Therefore, the USB does not split samples across packets.