Universal Serial Bus

From OSDev Wiki

The Universal Serial Bus was first introduced in 1994 with the intention of replacing various specialized interfaces, and to simplify the configuration of communication devices. The communication industry did not develop as the USB-IF foresaw, but the various transfer modes that USB introduced allowed it to become one of the most popular standards in use today. Virtually every modern computer supports USB.

Contents

Introduction

Despite how attractive USB support is, the 650-page USB 2.0 specification manages to deter even some of the most driven hobbyists (especially if English isn't their primary language). Not only is the USB 2.0 specification long, but it's a prerequisite for the XHCI, EHCI, UHCI, and OHCI specifications, all of which must be implemented for full USB 3.0 support (if you only want USB 1.0, you still need UHCI and OHCI). Furthermore, the USB specification defines a plethora of terms, some used interchangeably and seemingly lazily; as a lengthy technical document, it is neither easy nor practical to flip back and forth to clarify a confusing term or concept.

What this text covers

The truth is that a software developer doesn't need to read the entire USB 2.0 specification; there are sections specific to hardware developers, for example. The information presented here attempts to summarize chapters 4, 5, and 8 through 10.

Chapter 11 is specific to hubs and is also essential for a full USB 2.0 implementation, however it is almost as long as chapters 4, 5, 8, 9, and 10 combined, and could be regarded as the documentation for a specific (albeit special) class of USB devices. Chapter 11 is covered thusly in its own wiki entry, USB Hubs. Even so, some concepts which pertain to USB hubs are briefly discussed where relevant in this article.

Ideally, the text here will establish familiarity with the terms and concepts that a hobby OS developer needs to begin implementing USB support and, if necessary, easily parse the USB specification without becoming intimidated by the amount of information. At the very least, the system programmer should keep a copy of the USB 2.0 specification for reference while working with USB-related hardware.

Fortunately, all of the necessary documentation is available for free (see Links (http://wiki.osdev.org/USB#Links)).

What this text does not cover

Please note that USB, unlike other standards like VGA or PCI, is agnostic of the hardware interface to the system bus (and, by extension, to the operating system). Such an interface is provided by one or more USB host controllers and is defined by the appropriate documentation. Therefore, one should not expect this text to discuss specifics or code samples (e.g., as one finds in the wiki entries about VGA or PCI) detailing how the operating system initiates and maintains communication with USB devices. Although such information may be found on wiki entries discussing a particular Host Controller Driver, those wiki entries assume an understanding of the concepts and terms discussed here.

Host Controllers

The **Host Controller** is the USB interface to the host computer system. In other words, the host controller is what the system software uses to communicate with USB devices.

USB 1.0 Host Controllers

> *Main article:* Universal Host Controller Interface
> *Main article:* Open Host Controller Interface

Intel brought USB 1.0 to the market with its **Universal Host Controller Interface** (**UHCI**), while Compaq, Microsoft, and National Semiconductors did the same with their **Open Host Controller Interface** (**OHCI**). Naturally, the two interfaces are incompatible, and to make things worse, VIA Technologies licensed Intel's UHCI standard, thereby ensuring that both standards survived. Typically, an on-board chip set will contain a UHCI implementation, whereas a peripheral card typically implements the OHCI standard (but this is by no means a guarantee).

USB 2.0 Host Controllers

> *Main article:* Enhanced Host Controller Interface

In designing USB 2.0, the USB-IF insisted on a single implementation. That single implementation is Intel's **Enhanced Host Controller Interface** (**EHCI**). However, even though the USB 2.0 specification requires that a USB 2.0 interface support USB 1.0 devices, this doesn't mean that the EHCI must support USB 1.0 devices, and in fact, it doesn't. Each EHCI host controller is accompanied by (usually several) UHCI and/or OHCI host controllers. When a USB 1.0 device is attached, the EHCI simply hands control over to a **companion controller**. Refer to figure 1 for a simple block diagram implementation of this behavior. Therefore, the system programmer must support all three standards in order to support USB 2.0.

The EHCI host controller only handles USB 1.0 devices if they are attached indirectly through a USB 2.0 hub. The specifics of handling USB 1.0 devices attached to a USB 2.0 hub are briefly discussed and illustrated in the hubs section, and in more detail in the wiki entry for USB Hubs. Note that some newer chipsets like the Intel 5-series chipsets do not have companion controllers at all and instead have internal "rate matching" hubs that all USB devices go through.

USB 3.0 Host Controllers

Main article: eXtensible Host Controller Interface

In late 2008, the USB-IF released the USB 3.0 specifications. USB 3.0 host controllers are just starting to make their way into consumer devices since NEC introduced the world's first “SuperSpeed USB 3.0 host controller” in May, 2009, techspot reports (http://www.techspot.com/news/34763-nec-introduces-worlds-first-usb-30-host-controller.html) .

Intel is currently working on a USB 3.0 host controller specification called the **eXtensible Host Controller Interface** (**xHCI**).

A Linux driver is available for reference here (http://git.kernel.org/?p=linux/kernel/git/sarah/xhci.git;a=summary) developed by Sarah Sharp at Intel.

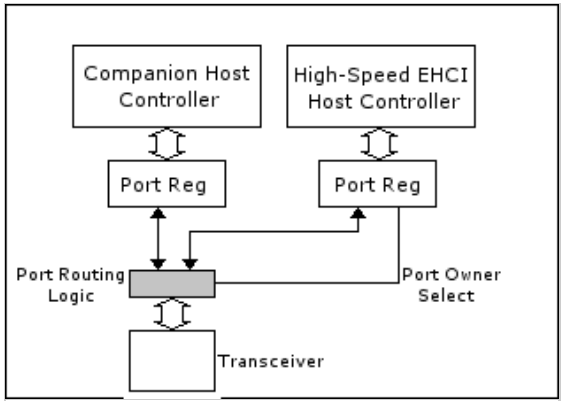


Figure 1: Block Diagram of Port Routing Behavior

In 2009, NEC introduced the µPD720200 (http://www.am.necel.com/usb/product/upd720200.html) , a USB 3.0 host controller compliant with Intel's draft xHCI specification.

On June 18th, 2010, Intel publicly released the xHCI specification.

Basic Concepts and Nomenclature

The USB is a polled bus, meaning the host controller must initiate all transfers. Do not mistake this to mean that the system software must poll the USB. The host controller takes care of polling the bus and can be programmed to issue interrupts to the OS whenever the bus needs attention.

USB System

A **USB System** consists of three discrete parts: the **USB device(s)**, the **USB interconnect**, and the **USB host**. Figure 2 illustrates a USB System.

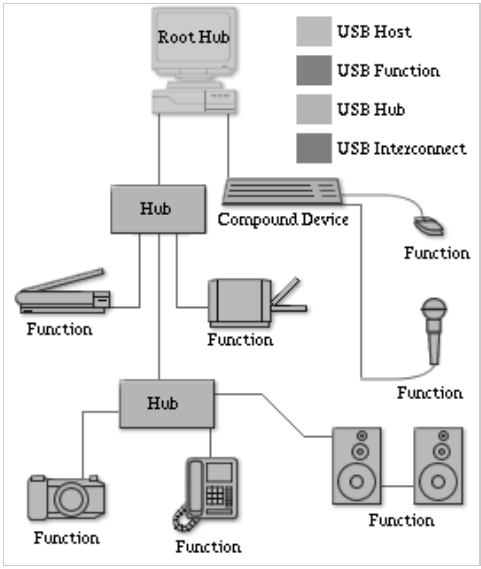


Figure 2: USB System Illustration

USB Device(s)

USB devices are classified as either a **hub** or a **function** (not to be confused with a program procedure). Hubs provide additional attachment points, whereas functions provide capabilities to the system. Some devices may implement several functions and an embedded hub in one physical package. These are called **compound devices**.

Functions

All functions understand the USB protocol, respond to standard operations (e.g, configuration or reset), and describe capabilities to the USB host.

There are three speed classes of functions:

- **High-speed** functions operate at up to 480 Mb/s.
- **Full-speed** functions operate at up to 12 Mb/s.
- **Low-speed** functions operate at up to 1.5 Mb/s.

The original USB specification defined low- and full-speed devices, while USB 2.0 added high-speed devices. USB 3.0 will add a fourth transfer speed of up to 5 Gb/s, called SuperSpeed.

Hubs

In a high-speed system, a high-speed hub plays a special role. Since the high-speed hub establishes a high-speed transfer rate with the host, it must isolate any full- or low-speed signaling from both the host and any attached high-speed devices.

To better understand, consider that the EHCI controller is accompanied by one or more companion controllers, as illustrated in figure 1 above. When a full- or low-speed device is attached directly to the root hub, the EHCI controller can relinquish ownership of that specific port to a companion controller as seen in figure 3. However, if a high-speed hub is connected to a port, as in Figure 4, then the EHCI controller must retain ownership of the port because it is a high-speed device. Now suppose other high-speed devices are attached to the high-speed hub in figure 4; obviously the EHCI controller retains control as in figure 5.

But what happens when a full- or low-speed device is connected to the high-speed hub in figure 5? If the EHCI controller were to relinquish ownership of the port, the high-speed devices will no longer be able to operate at high-speed, if at all, as in figure 6. Instead, the host controller and the hub support a special type of transaction called a split transaction. A **split transaction** involves only the host controller and a high-speed hub; it is transparent to any devices. This scheme of using split-transaction to support low- and full-speed devices on a high-speed hub is illustrated in figure 7.

Note that some newer chipsets like the Intel 5-series chipsets do not have companion controllers at all and instead have internal "rate matching" hubs that all USB devices go through.

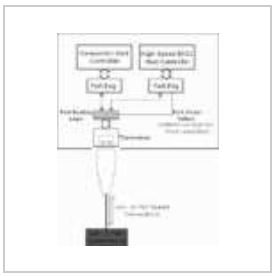


Figure 3: Low- or Full-speed device connected to a high-speed capable USB port

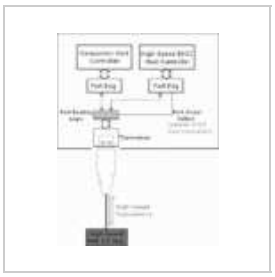


Figure 4: High-speed hub connected to a high-speed capable USB port

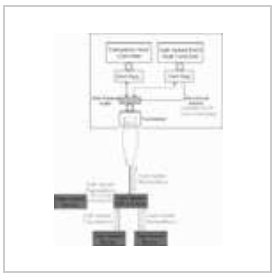


Figure 5: High-speed devices connected to a high-speed hub which is connected to a high-speed USB port

Figure 6: Incorrect illustration of Low- and Full-speed devices on a high-speed bus

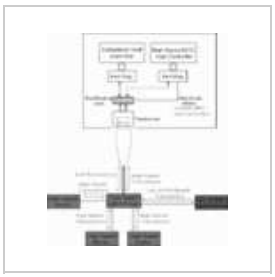


Figure 7: Correct illustration of split transactions allowing Low- and Full-speed devices on a high-speed bus

USB Interconnect

The **USB interconnect** provides a connection from the USB device(s) to the USB host. Physically, the USB interconnect is a tiered star topology. A maximum of seven tiers are allowed, and the root hub occupies the first tier. Since compound devices contain an embedded hub, a compound device cannot be attached in tier 7. Figure 8 illustrates a USB topology (taken from Figure 4-1 of the USB 2.0 specifications).

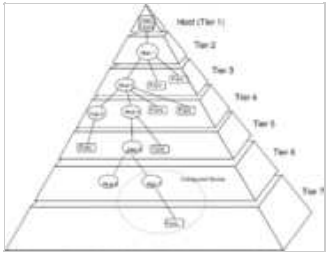


Figure 8: USB Topology

USB Host

A USB system contains only one **USB host**. The host interfaces with the USB interconnect via a host controller. The host includes an embedded hub called the **root hub** which provides one or more **attachment points**, or **ports**.

USB Communication Flow

Figure 9 illustrates the concepts of USB communication flow and is taken from Figure 5-10 of the USB 2.0 Specifications.

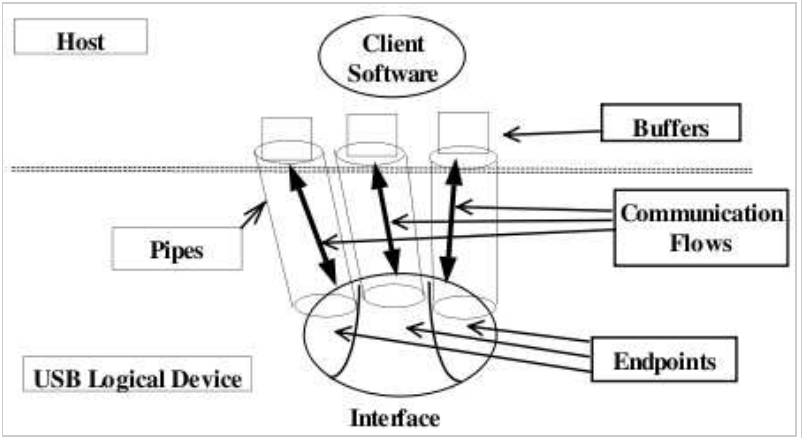


Figure 9: Illustration of USB Communication Flow

Device Endpoints and Endpoint Numbers

Each USB device contains a collection of endpoints. Every endpoint has the following characteristics:

- Bus access frequency/latency requirement
- Bandwidth requirement
- A unique device-determined identifier called the endpoint number
- Error handling behavior requirements
- Maximum packet size the endpoint can send or receive
- The transfer type of the endpoint
- Device-determined direction of data transfer:
  - **Input**: from the device to the host
  - **Output**: from the host to the device

As an example, consider an “all-in-one” printer/scanner device. Such a device may implement an endpoint number for printing functionality, and a separate endpoint number for scanning functionality.

Although endpoints have a specific direction, two endpoints may have the same endpoint number but opposing data transfer directions. All functions implement two such endpoints with the endpoint number 0. Only endpoints with the endpoint number 0 may be accessed as soon as the device is powered and has received a bus reset; all other endpoints are in an undefined state until the device is configured.

Besides the two required endpoints, functions may implement additional endpoints as necessary, with the following limitations:

- Low-speed functions may implement up to two additional endpoints.
- Full- and high-speed devices may implement up to 15 additional input endpoints and 15 additional output endpoints. This is a physical limitation of the USB protocol and is discussed under Endpoint_Field.

Endpoint Zero

All USB devices implement input and output endpoints with an endpoint number of 0. These endpoints are collectively known as the **default control pipe**. Endpoints with an endpoint number 0 are special in that they are accessible whenever the device is attached, powered and has received a bus reset.

In the interest of backwards compatibility, all high-speed functions must support these endpoints even when connected to a hub operating at full-speed. This means that high-speed devices must be able to reset at full-speed, as well as respond successfully to standard requests at full-speed. The high-speed device is not, however, required to support its intended functionality at full-speed. This allows USB 1.0 systems to identify a USB 2.0 device and alert the user if the device cannot function properly at full-speed

Pipes

A pipe associates software on the host (specifically, a buffer on the host) with an endpoint on a device.

There are two kinds of pipe communication modes:

- **Stream pipes** impose no structure on the data being transferred. Stream pipes are always **uni-directional** in their communication flow.
- **Message pipes** impose some structure on the data being transfered. Message pipes are **bi-directional**, however data

may predominantly transfer in one direction.

Pipes also have the following attributes:

- A claim on bus access and bandwidth usage
- A transfer type
- The associated endpoint's characteristics

Data flow in one pipe is independent of data flow in any other pipe. Most pipes are available after a device has been configured, however the **default control pipe** always exists after a USB device is powered and has received a bus reset.

Default Control Pipe

The **default control pipe** is a special type of message pipe that is always accessible once a device is powered and has received a bus reset. Thus, the default control pipe provides a means to identify and configure devices so that additional endpoints, if any, are made available.

The information required to completely identify a device is associated with the default control pipe; such information falls into the following categories:

- **Standard** information is common among all USB devices.
- **Class** information depends on the class of the USB device, as identified by the standard information.
- **USB Vendor** information is free for use by the hardware vendor.

Basics of USB Transfers

Most USB transactions consist of three packets:

- A **token packet** indicates the type and direction of the transaction, the device address, and an endpoint number.
- Depending on the direction of the transaction, either the host or the function sends a **data packet** (which may simply indicate that there is no data to send).
- The receiving device responds with a **handshake packet** to indicate if the transfer was successful.

USB supports four basic types of data transfer which take place via pipes. A single pipe supports only (and exactly) one transfer type for any given device configuration. That is, a function may provide a means to change the transfer type of a device-implemented endpoint number.

Briefly, the four basic transfer types are:

- **Control Transfers** provide lossless transmissions and are used to configure a device. Thus, all USB devices must support control transfers at least via the default control pipe.
- **Bulk Data Transfers** provide lossless, sequential transmissions and are typically used to transfer large amounts of data.
- **Interrupt Data Transfers** provide reliable, limited-latency transmissions typically needed by human input devices such as a mouse or a joystick.
- **Isochronous Data Transfers**, also called **Streaming Real-time Transfers**, negotiate a required bandwidth and latency when initializing the transfer. This transfer type is predominantly used for such applications as streaming audio. Since data-delivery rate is considered more important than data integrity for this type of transfer, it does not provide any type of error checking or correction mechanism.

Control Transfers

Control transfers support configuration/command/status type communication flow. The host initiates a control transfer with a SETUP bus transaction to the function, which establishes details of the intended data transfer such as whether the host wishes to send or receive data. Next, zero or more DATA transactions take place in the appropriate direction. Finally, a STATUS transaction from the function to the host indicates whether the transfer was successful.

Clearly, control transfers adhere to a USB-defined structure, so it should come as no surprise that control transfers may only be carried out via messages pipes.

Neither a function nor the host are guaranteed any specific latency or bandwidth for control transfers.

Maximum Data Payload Size

An endpoint used for a control transfer specifies the **maximum data payload size** that it can accept or transmit to the bus. The allowable maximum data payload sizes depend on the speed of the device:

- High-speed device endpoints may only select a maximum data payload size of 64 bytes.
- Full-speed device endpoints may select a maximum data payload size of 8, 16, 32, or 64 bytes.
- Low-speed device endpoints may only select a maximum data payload size of 8 bytes.

A control transfer always uses it's maximum data payload size for data payloads unless the data payload is less than the maximum data payload size. That is, if an endpoint has a maximum data payload size of 64 bytes, and a control transfer intends to transmit 100 bytes, the first data payload must contain 64 bytes and no less. The remaining 36 bytes are transferred in the second payload and need not be padded to 64 bytes. When the host receives a data payload less than the maximum data payload, the host may consider the transfer complete.

A SETUP transaction's data payload is always 8 bytes and thus receivable by the endpoint of any USB device. Consequently, the host may query the appropriate descriptor from a newly-attached full-speed device during configuration in order to determine the maximum data payload size for any endpoint; the host can then adhere to that maximum for any future transmissions.

Transmission Errors

Too Much Data

When transferring from host to device, if the host sends more data than negotiated during the SETUP transaction (i.e., the device receives more data than it expects; specifically, the host does not advance to the STATUS stage when the device expects), the device endpoint halts the pipe.

When transferring from device to host, if the device sends more data than negotiated during the SETUP transaction (i.e., the host receives an extra data payload, or the final data payload is larger than it should be), the host considers it an error and aborts the transfer.

Bus Errors

In the event of a bus error or anomaly, an endpoint may receive a SETUP packet in the middle of a control transfer. In such a case, the endpoint must abort the current transfer and handle the unexpected SETUP packet. This behavior should be completely transparent to the host; the host should neither expect nor take advantage of this behavior.

Halt Conditions

A control endpoint may recover from a halt condition upon receiving a SETUP packet. If the endpoint does not recover from a SETUP packet, it may need to be recovered via a different pipe. If an endpoint with the endpoint number 0 does not recover with a SETUP packet, the host should issue a device reset.

Bulk Data Transfers

A pipe with a bulk transfer type provides:

- Access to the USB on a bandwidth-available basis
- Retry of transfers that encounter the occasional delivery failure
- Guaranteed data integrity, but no guaranteed bandwidth

The host controller gives bulk data transfers low priority; they are generally only processed when bandwidth is available, however software may not assume that a control transfer will be processed before a bulk transfer. If multiple bulk transfers are pending, the host controller may begin moving bulk transfers over the bus according to an implementation-dependent policy. The system software may vary the bus time made available for a bulk transfer to a specific endpoint.

The USB does not impose any structure on the data content of a bulk transfer; thus, bulk transfers are carried via stream pipes.

Maximum Data Payload Size

An endpoint used for a bulk data transfer specifies the **maximum data payload size** that it can accept or transmit to the bus. The allowable maximum data payload sizes depend on the speed of the device:

- High-sped device endpoints may only select a maximum data payload size of 512 bytes.
- Full-speed device endpoints may select a maximum data payload size of 8, 16, 32, or 64 bytes.
- Low-speed devices may not implement bulk endpoints.

Like control transfers, a bulk transfer endpoint must transmit data payloads of the maximum data payload size for that endpoint with the exception of the last data payload in a particular transfer. The last data payload need not (and should not) be padded out to the maximum data payload size.

The bulk transfer is considered complete when the endpoint has transferred exactly as much data as expected, the endpoint transfers a packet with a data payload size less than the endpoint's maximum data payload size, or the endpoint transfers a zero-length packet.

Transmission Errors

If a data payload is transferred that is larger than expected, the transfer should be aborted along with any pending bulk transfers through the same pipe.

Bulk data transfers employ data toggle bits to both detect errors and provide the necessary synchronization to recover from an error. If a halt condition is detected, any remaining bulk transfers should be retired. The halt condition is resolved by means of a separate control pipe.

Interrupt Data Transfers

Interrupt data transfers guarantee a maximum service time for any data transfer. In the even of a transmission failure, data is retransmitted at the next period. Thus, an interrupt data transfer is ideal for devices that do not send data often, but when they do, they require timely transmission as well as data integrity; most human input devices have these requirements.

Interrupt data transfers are carried out by a stream pipe and thus do not need to adhere to any USB data structure.

Maximum Data Payload Size

An endpoint used for a interrupt data transfer specifies the **maximum data payload size** that it can accept or transmit to the bus. The allowable maximum data payload sizes depend on the speed of the device:

- High-speed device endpoints may select a maximum data payload size of up to 1024 bytes.
- Full-speed device endpoints may select a maximum data payload size of up to 64 bytes.
- Low-speed device endpoints may select a maximum data payload size of up to 8 bytes.

Additionally, a high-speed, high-bandwidth endpoint may specify that it requires two or three transactions per microframe. High-speed, high-bandwidth endpoints, frames, and microframes will be discussed later.

Notice that the maximum data payload size for interrupt data transfers allows for more granularity than control or bulk data transfers. That is, an interrupt data transfer endpoint for a high-speed device may be any integer from 0 to 1024. The maximum data payload size for an interrupt transfer endpoint remains constant during the lifetime of the device's configuration.

Like control and bulk data transfers, an interrupt transfer endpoint must transmit data payloads of the maximum data payload size for that endpoint with the exception of the last data payload in a particular transfer. The last data payload need not (and should not) be padded out to the maximum data payload size.

The interrupt transfer is considered complete when the endpoint has transferred exactly as much data as expected, the endpoint transfers a packet with a data payload size less than the endpoint's maximum data payload size, or the endpoint transfers a zero-length packet.

Transmission Errors

If a data payload is transferred that is larger than expected, the transfer should be aborted and the pipe stalls any future interrupt transfers until the error is acknowledged and corrected.

Interrupt data transfers may use one of two data toggle bit schemes to ensure successful data transmission. Devices that require higher through-put may choose to toggle every transmission rather than perform a handshake with the host. This method is more susceptible to errors than the alternative method of toggling bits upon successful transaction (after a handshake).

If a halt condition is detected, any pending interrupt transfers should be retired. The halt condition is resolved via a separate control pipe.

Isochronous Data Transfers

Isochronous data transfers are similar to interrupt transfers in that they guarantee a maximum service time for any transfer, but isochronous data transfers do not ensure data integrity. When data is ready to be transmitted to or from an isochronous endpoint, the data is always transferred at a constant rate.

The data being transmitted via an isochronous pipe need not have any specific structure, therefore isochronous pipes are stream pipes.

Maximum Data Payload Size

An endpoint used for a isochronous data transfer specifies the **maximum data payload size** that it can accept or transmit to the bus. The allowable maximum data payload sizes depend on the speed of the device:

- High-speed device endpoints may select a maximum data payload size of up to 1024 bytes.
- Full-speed device endpoints may select a maximum data payload size of up to 1023 bytes.
- Low-speed devices may not implement isochronous endpoints.

Like interrupt endpoints, isochronous endpoints may specify a maximum data payload size with byte granularity. Also like interrupt endpoints, high-speed, high-bandwidth isochronous endpoints may specify if they require two or three transactions per microframe.

Unlike any other transfer types, isochronous transfers may transmit any amount of data up to the maximum data payload size during any transaction.

Transmission Errors

Isochronous transfers are meant for devices where data transmission rate is more important than data integrity. For that reason, isochronous transfers do not allow handshakes and thus cannot stall. It is still important that the agent of an isochronous transfer know if an error occurred, and possibly how much data was lost. The USB protocol provides several mechanisms for detecting data transmission errors in an isochronous transfer, these mechanisms will be discussed later. Determining the amount of data lost is implementation-dependent. It is up to the software on the host or firmware on the function to implement any sort of data corruption detection/correction.

Advanced USB Concepts

The topics in this section build upon the topics previously discussed. The information in this section provides some useful lower-level details about USB systems.

Distribution of Bus Access Time

Frames and Microframes

To ensure synchronization between the host and the functions, the USB divides bus time into fixed-length segments. For low- or full-speed buses, the USB divides the bus time into 1 millisecond units, called **frames**. For a high-speed bus, the USB divides the bus time into 125 microsecond units, called **microframes**.

Note that frames and microframes do not coexist on one bus; low- and full-speed buses used frames, but in developing a high-speed bus, a shorter frame was necessary because the significantly higher signaling bit rate is more sensitive to smaller shifts in synchronization between the host and the function.

Frames and microframes are mostly a physical-layer detail and should not be confused with any of the previous concepts. Frames and microframes do not correspond to any packet or transaction; in fact, several transactions usually take place during one (micro)frame. The host controller issues a **start-of-frame** (**SOF**) packet at the beginning of every (micro)frame. The remainder of the (micro)frame is available for the host controller to carry out transactions. A transaction may not take place if it cannot be completed in the same (micro)frame (because otherwise the next SOF packet would interrupt the transaction).

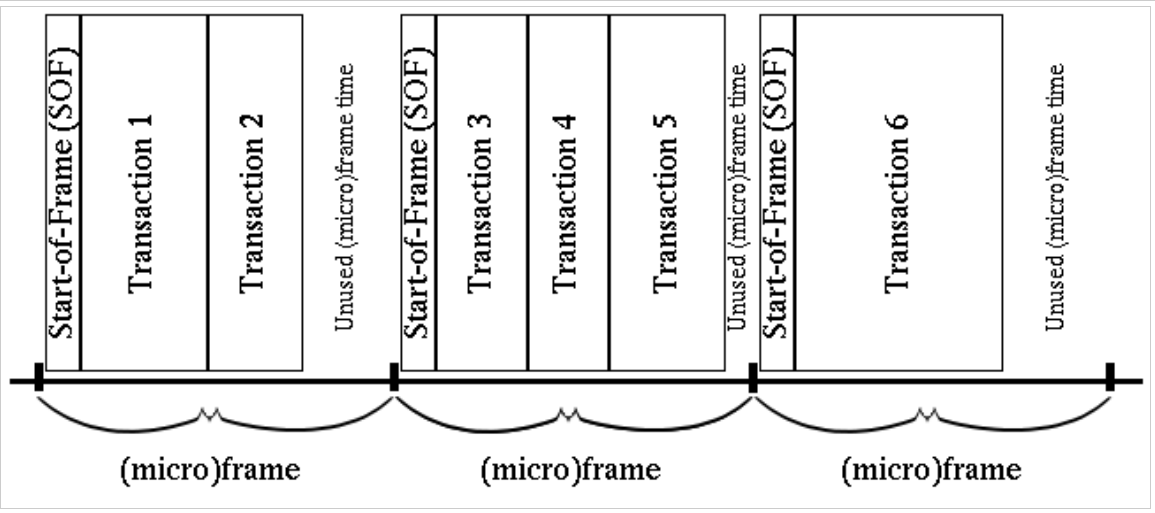


Figure 10: Illustration of USB (micro)frames.

It is important to realize that the host controller may rearrange transactions to make better use of the available bandwidth. Of course, two transactions through the same pipe must occur in the correct order, but the transactions of two separate transfers may be reordered at the host controller's discretion. Consider a pending bulk transfer and two pending control transfers. The host could potentially reorder the transfers on the bus as in Figure 11.

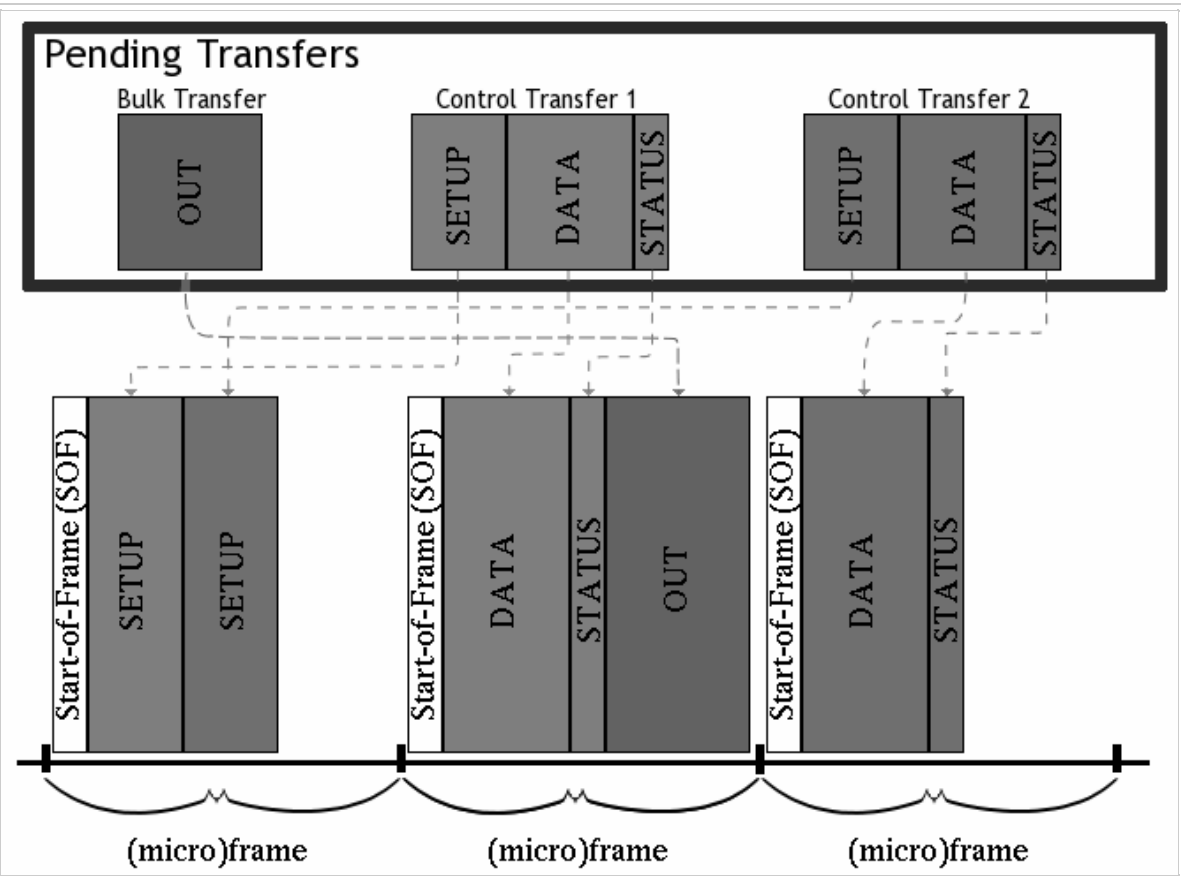


Figure 11: Illustrates how a host controller may potentially reorder a bulk transfer and two control transfers on the USB.

Bus Time Rationing

There are separate rules for the allocation of frames on a full-/low-speed bus, and for the allocation of microframes on a high-speed bus.

For full- or low- speed buses:

- If a control transfer requires less than 10% of a frame, the remaining bus time can be used to support bulk transfers
- If there are more control transfers than reserved time, yet additional frame time that is unused by interrupt or isochronous transfers, the host controller may move additional control transfers onto the bus.

- No more than 90% of a frame may be allocated for periodic (isochronous and interrupt) transfers.
- The host must not issue more than 1 transaction in a single frame for a specific isochronous endpoint.

For a high-speed bus:

- If a control transfer requires less than 20% of a microframe, the remaining bus time can be used to support bulk transfers.
- If there are more control transfers than reserved time, yet additional microframe time that is unused by interrupt or isochronous transfers, the host controller may move additional control transfers onto the bus.
- No more than 80% of a frame may be allocated for periodic (isochronous and interrupt) transfers.
- The host must not issue more than 1 transaction in a single microframe for a specific isochronous endpoint unless it is a high-speed, high-bandwidth endpoint.
- Split transaction bus access time is allocated from the 80% of the microframe guaranteed to periodic transfers.

High-Speed, High-Bandwidth Endpoints

High-speed interrupt or isochronous endpoints that require high bandwidth may specify that they support up to three transaction in a single (micro)frame. In this case, all but the last transaction in a particular (micro)frame must have a data payload of the maximum data payload size for that endpoint.

The host controller never retries a transaction with an isochronous endpoint. If a transaction with a high-speed, high-bandwidth interrupt endpoint fails, the host controller may retry the transaction during the same (micro)frame if the maximum number of transactions per (micro)frame has not been reached. Otherwise, the transaction is retried at the next period.

Supporting Isochronous Transfers

Recall that isochronous transfers occur over stream pipes, which provide one-way data transfer. On one of the pipe, called the **source**, data is produced, and on the other end, called the **sink**, data is delivered.

Devices that implement isochronous endpoints require that data be transmitted from source to sink at a certain rate, sometimes in large payloads (e.g, streaming audio or video). This section discusses how the USB accomplishes these requirements.

Synchronization

Due to application-specific sampling rates, different hardware clock designs, scheduling policies in the operating system, or even physical anomalies, the host and isochronous device could fall out of synchronization. Therefore, special consideration is required to maintain synchronization. Isochronous endpoints specify one of three synchronization types.

Asynchronous Endpoints

Asynchronous endpoints are incapable of synchronizing to SOF packet frequency (1ms periods for full-speed endpoints, 125 microsecond periods for high-speed endpoints). These endpoints have either:a set of one or more fixed data sampling rates, or a continuously programmable data rate. The device must report the programmability of an asynchronous endpoint in some manner (defined by the class of the device rather than by the USB specifications); if the data rate is programmable, then it must be set by the host during initialization of the isochronous endpoint.

Asynchronous source endpoints imply their data rate by the number of samples produced per (micro)frame. Asynchronous sink endpoints must provide explicit feedback to the source endpoint. When the source endpoint is the host, it is the responsibility of the device driver to process the explicit feedback properly. This feedback allows the host and device to make slight adjustments to the data rate in order to compensate for any clock drift.

Synchronous Endpoints

Synchronous endpoints must synchronize their data transmissions to the SOF packet frequency (1ms periods for full-speed endpoints, 125 microsecond periods for high-speed endpoints). These endpoints have either a set of one or more fixed data sampling rates, or a continuously programmable data rate. The device must report the programmability of a synchronous endpoint in some manner (defined by the class of the device rather than by the USB specifications); if the data rate is programmable, then it must be set by the host during initialization of the isochronous endpoint.

Adaptive Endpoints

Adaptive endpoints can source or sink data at any rate within their specified operating range. These endpoints may have an operating range that centers around a specific data rate, it may have a finite set of data rate ranges, or it may select between several programmable or auto-detecting data rates. The device must report the programmability of an adaptive endpoint in some manner (defined by the class of the device rather than by the USB specifications); unlike the previous synchronization types, adaptive endpoints may adjust it's instantaneous data rate during operation.

Adaptive sink endpoints provide explicit feedback to the source like asynchronous endpoints.

Handling Errors

Handshakes are not performed for isochronous transactions, therewith eliminating the bandwidth overhead of acknowledgment packets. Unlike other transfer types, the applications of isochronous endpoints are responsible for any error detection and handling. Although it may be more important to continue delivering streaming data rather than retransmit a missed data packet, applications of isochronous endpoints often still need to know that an error did occur in the stream.

The USB protocol highlights the following possible method for the host or a device to detect an error in an isochronous stream:

- High-speed, high-bandwidth isochronous transactions use data PID sequencing (data bit toggling), an isochronous sink can determine that a data packet was missed when it receives an invalid data PID sequence.
- The host controller and device can both see SOF packets on the bus. If the SOF packet is issued for a (micro)frame that is expected to carry the periodic data of an isochronous endpoint, but the data is not transmitted, then the hardware can determine that a packet was missed.
- The protocol provides CRC protection to ensure that the data has not been corrupted.
- If an endpoint sees the token packet but does not see the associated data packet within a bus transaction timeout period, then the data packet failed to transfer.

Once an application is aware that there is an error in the stream, it is up to the application to determine the next course of action.

USB Protocol

Packets

The atomic unit of data transfer is a packet. A packet is a bundle of organized data which typically contains three elements:

- Control information (e.g. source, destination, length of data)
- User/Application-specific data
- Error detection and correction bits

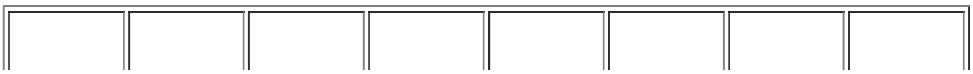
SYNC Field

The SYNC field is omitted from packet diagrams in the USB specifications, and usually in other material on USB meant for programmers. Here I will briefly describe the semantics of the SYNC field simply because USB sources often reference the SYNC field which may confuse the reader. However, for clarity, the system programmer (and probably most USB device firmware developers as well) does not need to know about the SYNC field.

All USB packets start with a SYNC field which serves, unsurprisingly, as a synchronization mechanism between the receiver and the sender. The SYNC field consists of 6 or 30 alternating bits for low- and full-speed or high-speed buses, respectively. The last two bits of the SYNC field are equal (and low). High-speed hubs may drop up to 4 bits of the SYNC field, so a receiving device may not see the entire field, but the final two bits are all the device needs to identify exactly where the SYNC field ends, and useful data begins.

Packet Identifier Field

The **Packet Identifier** (**PID**) immediately follow the SYNC field. There are a total of 17 defined PIDs (included the PID of 0000b, which is reserved), therefore a PID requires 4 bits to encode. If errors on the bus alter the PID field (changing an OUT PID to an IN PID, for example), the result could be anything from unexpected behavior to massive data loss. Due to the importance of the PID integrity, the PID field is 8 bits wide. The last 4 bits simply compliment the first four bits, this provides a means to determine if an error on the bus has altered the PID field. The PID Field is illustrated below.



PID_0	PID_1	PID_2	PID_3	$\overline{PID_0}$	$\overline{PID_1}$	$\overline{PID_2}$	$\overline{PID_3}$

Packet Identifier Field Format

PID codes are categorized into 4 groups which share the same two least-significant bits. USB 2.0 defines the PIDs in the following table.

PID Type	PID Name	PID [3:0]	Description
Token	OUT	0001b	The packet describes a host-to-function transaction.
	IN	1001b	The packet describes a function-to-host transaction.
	SOF	0101b	The packet marks the start of frame and specifies the frame number.
	SETUP	1101b	Packet describes a SETUP transaction from the host to the function via a control pipe.
Data	DATA0	0011b	This packet is an even data packet.
	DATA1	1011b	This packet is an odd data packet
	DATA2	0111b	This packet is only used in high-speed, high-bandwidth isochronous transfers.
	MDATA	1111b	This packet is only used in split transactions, or high-speed, high-bandwidth isochronous transfers.
Handshake	ACK	0010b	This packet acknowledges the successful receipt of a data packet.
	NAK	1010b	This packet indicates that data is not ready to be transmitted yet.
	STALL	1110b	This packet indicates that the endpoint has halted, or a control pipe does not support a certain request.
	NYET	0110b	The receiver has not yet responded, or the host should begin sending PING packets.
Special	PRE	1100b	This packet is a host-issued preamble for a split-transaction.
	ERR	1100b	This packet is a handshake response that a split transaction error occurred. Note that this PID is identical to the PID for a PRE packet.
	SPLIT	1000b	This packet supports split transactions between the host and a high-speed hub.
	PING	0100b	This packet is used for flow-control in high-speed control and bulk transfers.
	Reserved	0000b	This is a reserved PID and must not be used.

PID Types

Address Fields

Address fields select a specific endpoint on a specific function. Naturally, two such fields are defined: an address field and an endpoint field. All devices must fully decode these fields; a mismatch of either field (including an endpoint field which specifies an endpoint that have not been initialized) must be ignored.

Address Field

The address field is specified for the following PIDs:

- IN
- SETUP
- OUT
- PING
- SPLIT

The address field is 7 bits wide and illustrated below. Each possible value may only indicate a single function. Address zero is reserved as the **default address** and cannot be assigned to any function. All functions must respond to the default address upon reset and power-up until the host assigns the function a specific address. Therefore, one host controller can support up to

127 devices at one time.

$Addr_0$	$Addr_1$	$Addr_2$	$Addr_3$	$Addr_4$	$Addr_5$	$Addr_6$

Address Field

Endpoint Field

The endpoint field is specified for the following PIDs:

- IN
- SETUP
- OUT
- PING

The endpoint field is 4 bits wide and illustrated below. All functions must support at least endpoint number 0 (the default control pipe). Low-speed functions may only implement 2 additional pipes, while full- and high-speed devices are only limited by the width of the endpoint field. In other words, the width of the endpoint field is the reason that full- and high-speed devices are limited to implementing up to 15 additional IN endpoints, and 15 additional OUT endpoints, as noted above under Device Endpoints and Endpoint Numbers.

$Endp_0$	$Endp_1$	$Endp_2$	$Endp_3$

Endpoint Field

Data Field

The data field may range from zero to 1,024 bytes and must be an integral number of bytes. Data bytes are sent least-significant-bit first.

Cyclic Redundancy Checks

Cyclic Redundancy Checks (CRC) protect all non-PID fields and provide 100% coverage for all single- and double-bit errors. CRCs are provided for each token field as well as the data field. This provides a mechanism for the host or device to recognize and either correct or ignore corrupted fields or, in most cases, an entire corrupted packet.

Handshakes

Transaction types which support flow control return handshakes to indicate:

- Successful reception of data
- Command acceptance or rejection
- Flow control
- Halt conditions

Handshakes are always returned in the handshake phase of a transaction, but may also be returned in the data phase (in place of an expected data packet). To best understand a certain handshake response, it is useful to understand what each handshake packet type means, as well as the conditions under which each handshake response may be issued. This section is divided thusly.

Handshake Packets

All of the handshake packet types were listed previously and briefly in Packet Identifier Field. This section discusses those packet types in greater detail.

ACK

An ACK handshake is issued to communicate that a data packet was successfully received without any bit stuffing or CRC errors over the data field, and the PID field was not corrupted.

ACK Handshake Packet

May be issued by...	For these transactions
Host	IN

Function	OUT
	SETUP
	PING

ACK packets may be issued when the receiver's sequence bit matches the sequence bit of the received data packet (and the data can be accepted), but the an ACK packet may also be issued when the receiver's sequence bit does not match the sequence bit of the received data packet (and the data cannot be accepted). This may seem counter intuitive, but the reasoning will become clear in the sections discussing data toggling.

NAK

NAK Handshake Packet

May be issued by...	For these transactions
Function	IN
	OUT
	PING

The NAK handshake packet is generally used for flow control to indicate that a function is temporarily unable to transmit or receive data. The host never issues a NAK handshake packet to a device.

A function returns a NAK handshake packet to the host after an OUT transaction when the function is unable to receive data (usually because the function's internal buffer is currently full). This response is not an error, but instead indicates that the host should retry transmission later, allowing the function time to process the data currently in its buffer.

A function returns a NAK handshake packet to the host during the data phase of an IN transaction to indicate that the function does not have any data to transfer.

STALL

STALL Handshake Packet

May be issued by...	For these transactions
Function	IN
	OUT
	PING

A function uses the STALL handshake packet to indicate that it is unable to transmit or receive data. Besides the default control pipe, all of a function's endpoints are in an undefined state after the device issues a STALL handshake packet. The host must never issue a STALL handshake packet.

Typically, the STALL handshake indicates a functional stall. A **functional stall** occurs when the *halt feature* (which will be covered under "USB Framework") of an endpoint is set. In this circumstance, host intervention is required via the default control pipe to clear the *halt feature* of the halted endpoint.

Less often, the function returns a STALL handshake during a SETUP or DATA stage of a control transfer. This is called a **protocol stall** and is resolved when the host issues the next SETUP transaction.

NYET

NYET Handshake Packet

May be issued by...	For these transactions
Hub	SPLIT
Function	OUT

The NYET packet may be issued by a function as part of the PING protocol.

Hubs may issue a NYET handshake packet in response to a split transaction that has not yet completed on the low-/full-speed bus.

ERR

ERR Handshake Packet

May be issued by...	For these transactions
Hub	SPLIT

Hubs may issue the special ERR handshake packet to report an error on a low-/full-speed bus as part of the split transaction protocol.

Function/Host Response Circumstances

This section describes the functional circumstances that cause the host or a function to issue an expected response, no response, or certain handshake packet responses. The tables in this section are taken and slightly modified for clarity from the USB 2.0 specifications, section 8.4.6. Dashes denote a "don't care."

Function Response to IN Transactions

Token received corrupted	Function Tx endpoint halt feature	Function can transmit data	Action taken by function
Yes	-	-	Return no response
No	Set	-	Issue STALL handshake
No	Not Set	No	Issue NAK handshake
No	Not Set	Yes	Issue data packet

Host Response to IN Transactions

Data packet corrupted	Host can accept data	Action taken by host	Handshake returned by host
Yes	-	Discard Data	Return no response
No	No	Discard Data	Return no response
No	Yes	Accept Data	Issue ACK handshake

Function Response to OUT Transactions

Data packet corrupted	Receiver halt feature	Sequence bits match	Function can accept data	Action taken by function
Yes	-	-	-	Return no response
No	Set	-	-	Issue STALL handshake
No	Not Set	No	-	Issue ACK handshake
No	Not Set	Yes	Yes	Issue ACK handshake
No	Not Set	Yes	No	Issue NAK handshake

Function Response to SETUP Transactions

A function must always accept data in a SETUP transaction, and must never issue a STALL or NAK handshake in response. All non-control endpoints must simply ignore any SETUP transaction addressed to that endpoint. This allows SETUP transactions to function as a (re)synchronization mechanism between the host and a function's control endpoint.

PING Transaction Protocol

Consider a USB mass storage device. During a transfer from the host to the function, the function's buffer fills up with data that is pending being committed to the physical media. When the function's buffer is full, the function cannot accept new data until some of the buffer is committed, so if the host continues sending OUT transactions, the function must NAK them.

The problem with this OUT/NAK model is that a function must wait for the handshake stage of the OUT transaction before responding with a NAK. Since the handshake stage occurs after the data stage, this can waste a significant amount of bandwidth. Low- and full-speed buses suffer from this problem, but the USB 2.0 specification introduced the PING transaction protocol for high-speed buses.

The PING transaction protocol is very straightforward. Rather than an OUT transaction, the host issues a PING transaction to the function when the host wishes to send data. The function responds with either NAK to indicate that it is not ready to receive data (specifically, the function's buffer cannot accommodate the endpoint's maximum data payload amount of data), or ACK to indicate that the host may start sending data.

The USB 2.0 framework allows endpoints to specify an interval, in terms of microframes, which is the amount of microframes that the host should wait before issuing another PING packet to the endpoint. However, the host is not required to wait this interval before issuing the next PING packet.

During a high-speed control or bulk transfer from the host to function, when an OUT transactions causes a function's free buffer space to drop below the endpoint's maximum data payload, the function responds with a NYET handshake packet. This indicates that the host should start issuing PING packets rather than additional OUT transactions.

Data Toggle Synchronization

During a transfer, the host and function must remain synchronized. The ability to maintain synchronization means that the host or function can detect when synchronization has been lost and, in most cases, resynchronize.

Every endpoint maintains, internally (in the function's hardware), a data toggle bit, also called a data sequence bit. The host also maintains a data toggle bit for every endpoint with which it communicates. The state of the data toggle bit on the sender is indicated by which DATA PID the sender uses.

The receiver toggles its data sequence bit when it is able to accept data and it receives an error-free data packet with the expected DATA PID. The sender toggles its data sequence bit only upon receiving a valid ACK handshake. This data toggling scheme requires that the sender and receiver synchronize their data toggle bits at the start of a transaction.

Data toggle synchronization works differently depending on the type of transfer used:

- Control transfers initialize the endpoint's data toggle bits to 0 with a SETUP packet.
- Interrupt and Bulk endpoints initialize their data toggle bits to 0 upon any configuration event.
- Isochronous transfers do not perform a handshake and thus do not support data toggle synchronization.
- High-speed, high-bandwidth isochronous transfers do support data sequencing within a microframe.

The remainder of this section illustrates how the sending and receiving devices each manage their data toggle bits during different transmission scenarios. Black arrows signify the intended data transmission on the USB. Gray arrows signify that the intended data transmission completed without error. Red, discontinuous arrows signify that the intended data was corrupted during transmission or entirely failed to transmit.

Successful transmissions

Figure 12 illustrates a successful data transfer. Both devices have data toggle bits set to 0 at the beginning of transfer i. Accordingly, the sending device issues a DATA0 PID followed by the data packet. The receiving device successful reads the DATA0 PID as well as the data packet. Since the receiver's data toggle bit matches the DATA0 PID and there were no errors in transmitting the remaining data, the receiver toggles its data toggle bit to 1 and issues an ACK handshake response. The sender receives the ACK handshake without error, and thus toggles its data toggle bit to 1.

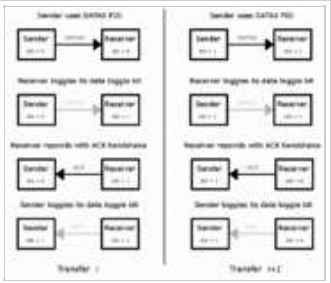


Figure 12: Illustration of how the sender and receiver manage their data toggle bits during a successful data transfer

Supposing that the next transfer occurs without error as well, the only difference is that the DATA1 PID is used rather than DATA0, and the sending and receiving devices toggle their data toggle bits from 1 to 0 in the same stages that the same bit toggled to a 1 in the previous transfer.

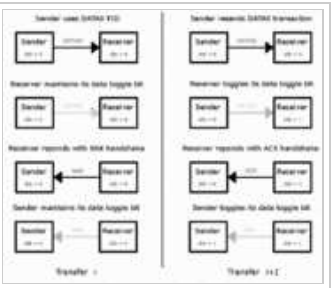


Figure 13: Illustration of how the sender and receiver manage their data toggle bits during a failed or corrupt data transfer

Failed or corrupted data transmissions

Figure 13 illustrates a failed or corrupted data transmission. Both devices have data toggle bits set to 0 at the beginning of transfer i. Accordingly, the sending device issues a DATA0 PID followed by the data packet. The receiving device either does not see the data packet, or reads a corrupted data packet. The receiver maintains its data toggle bit and issues a NAK handshake. The sender successfully sees the NAK handshake and thus does not toggle its data toggle bit.

At the beginning of the next transfer, both the sending and receiving device have data toggle bits still set to 0. Supposing this transfer completes successfully, it is carried as as described above, under successful transmissions.

Failed or corrupted ACK handshake

Figure 14 illustrates a failed or corrupted ACK handshake. Both devices have data toggle bits set to 0 at the beginning of transfer i. Accordingly, the sending device issues a DATA0 PID followed by the data packet. The receiving device successfully reads the DATA0 PID as well as the data packet. Since the receiver's data toggle bit matches the DATA0 PID and there were no errors in transmitting the remaining data, the receiver toggles its data toggle bit to 1 and issues an ACK handshake response. The sender does not receive, or receives a corrupted ACK response, and thus discards the packet without modifying it's data toggle bit.

At this point, the sending device's data toggle bit is still 0, and the receiving device's data toggle bit has been set to 1. The sender, having not seen a valid ACK response for transfer i, reattempts transfer i. With a data toggle bit of 0, the sender issues a DATA0 PID followed by the data packet. The receiving device successfully reads the DATA0 PID as well as the data

packet. Since the receiver's data toggle bit does not match the DATA0 PID, the receiver maintains it's data toggle bit value of 1 and issues an ACK handshake response. The sender receives the ACK response without error, and thus toggles its data toggle bit to 1.

Supposing that the next transfer occurs without error, it begins with both device's data toggle bits set to 1 and ends with them toggling to 0 at the appropriate stage of the transfer.

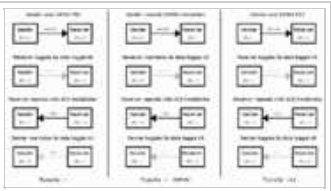


Figure 14: Illustration of how the sender and receiver manage their data toggle bits during a failed or corrupt ACK response

USB Transfers Revisited

A lot of information has been introduced since Basics of USB Transfers, and it is very easy to get lost in the details. With even a decent understanding of the four types of USB transfers, it is often difficult to extrapolate from the intricacies of the USB protocol to an understanding of just how everything fits together. For these reasons, this section intends to clarify some potentially confusing concepts both explicitly and implicitly by revisiting the four transfer types in context of all the information covered since first discussing them.

An apprehensive reader may have noticed that some terms like SETUP and DATA are used both in referring to packet identifiers, and in referring to types of transactions. This wiki entry may very well be the first and only source of USB information that takes a moment to specifically differentiate between the two.

Under Basics of USB Transfers, USB transactions were mentioned only briefly as has been reproduced below:

```
Most USB transactions consist of three packets:
* A token packet indicates the type and direction of the transaction, the device address, and an
  endpoint number.
* Depending on the direction of the transaction, either the host or the function sends a data packet
  (which may simply indicate that there is no data to send).
* The receiving device responds with a handshake packet to indicate if the transfer was successful.
```

Then, under Packets, a packet was described as "the atomic unit of data transfer."

If a packet is an atom, then a transaction would be a molecule. That is, a transaction is made up of several packets in a specific order, and the packets which make up a transaction cannot be reordered or separated and still yield the same transaction. Transactions are normally named after their token packet (or their "special" packet, in the case of PING or SPLIT because these special packets play the same role as token packets), with the exception that IN or OUT transactions are often referred to, collectively, as DATA transactions. In examples, transactions that contain a data stage often indicate the type of DATA PID used by either appending 0, 1, 2, or M to the name, or adding it in parenthesis (e.g, SETUP(0) or SETUP0, OUT1 or OUT(1)).

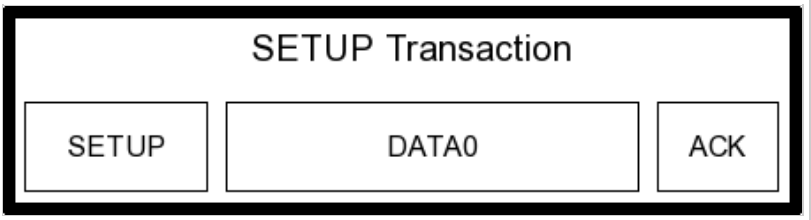


Figure 15: Illustrates a SETUP transaction, which consists of a SETUP packet, a DATA0 packet, and a handshake packet (in this case, an ACK handshake).

An example of a single SETUP transaction is depicted in figure 15. This transaction contains the typical three packets. The token packet has a SETUP PID, the data packet has a DATA0 PID (recall that a SETUP packet initializes both the function's and the host's data toggle bits to 0), and the handshake response has an ACK PID.

Transfers are made up of transactions. Transactions may not be reordered within a transfer but, as discussed in Frames and Microframes, the transactions of a particular transfer may or may not be sent over the bus in a continuous fashion. The rest of this section looks at the transactions involved in the four transfer types.

Control Transfers

Control transfers are the only transfers that use the SETUP transaction. Control transfers take place in up to three stages:

- The SETUP stage consists simply of a SETUP transaction
- The DATA stage is optional. If used, it may contain either one or more IN transactions, or one or more OUT transactions. The first of these IN or OUT transactions uses the DATA1 PID. The second, if present, uses the DATA0

PID, the third DATA1, and so on.
- The STATUS stage consists of a single IN or a single OUT transaction, which must use the DATA1 PID. If the DATA stage is present, then the STATUS stage uses the opposite type of transaction as the DATA stage (i.e, if the DATA stage consists of one or more OUT transactions, the STATUS stage consists of a single IN transaction, and vice versa). When the DATA stage is omitted, the STATUS stage uses a single IN transaction.

Figure 16 is taken from Figure 8-37 of the USB 2.0 specification and illustrates the transaction order, data sequence bit value, and DATA PID type for control read and write sequences.

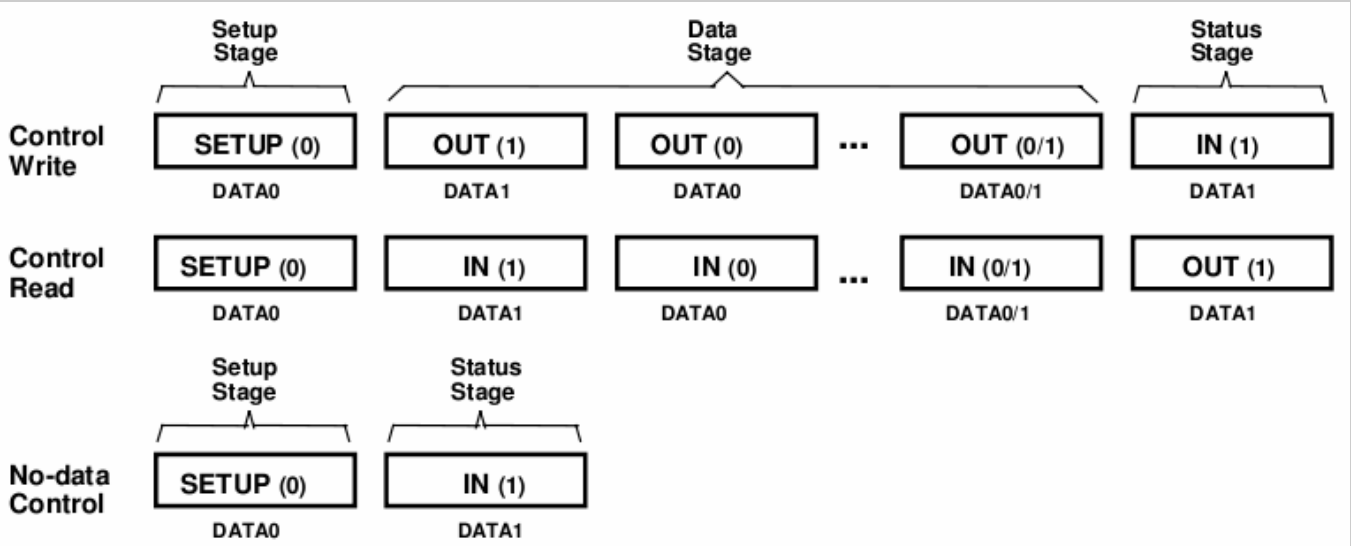


Figure 16: Control read and write sequences

Bulk and Interrupt Transfers

In the context of the USB protocol, the only difference between bulk and interrupt transfers is that bulk transfers, when operating at high-speed, support the PING Transaction Protocol. Note that in a general context, these two transfer types are also different in that they are scheduled differently by the host (refer to Bus Time Rationing).

All bulk and interrupt endpoints transfer in one direction. The data toggle bits for these endpoints are initialized to zero after any configuration event. Figure 17 is taken from Figure 8-35 of the USB 2.0 specification and illustrates the bulk and interrupt transactions for both IN and OUT endpoints. Note that, even though the figure only mentions bulk reads and bulk writes, the USB 2.0 specification references the same figure from section 8.5.4, on Interrupt Transactions.

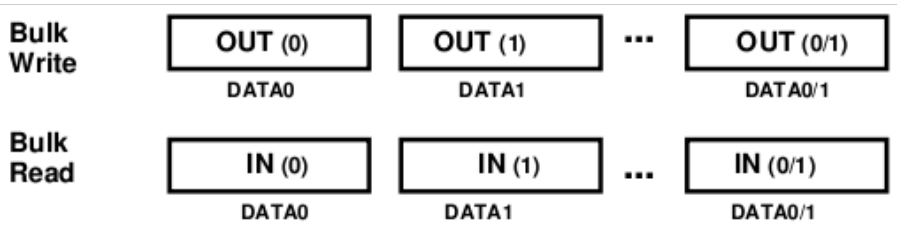


Figure 17: Bulk and interrupt transactions

Isochronous Transfers

Isochronous transfers are the only type of transfers whose transactions do not have a handshake phase. Isochronous transfers should only use DATA0 PIDs, however the host controller must support DATA1 PIDs as well, even though isochronous transfers do not use a data synchronization bit mechanism.

High-Speed, High-Bandwidth Isochronous Transfers

High-speed, high-bandwidth isochronous transfers are a special case of isochronous transfers, where up to 3 transactions may occur in one microframe. As a specific type of isochronous transfer, high-speed, high-bandwidth isochronous transfers omit the handshake phase of their transactions. Since up to 3 transactions may occur in one microframe, high-speed, high-bandwidth isochronous transfers, it is necessary to use a data sequencing mechanism like the other transfer types.

USB 2.0 does implement a data sequencing mechanism for high-speed, high-bandwidth isochronous transfers, but it works a little differently than as in other transfer types. In fact, data sequencing works differently depending on whether an endpoint is an IN, or an OUT high-speed, high-bandwidth isochronous endpoint.

For IN high-speed, high-bandwidth isochronous endpoints, the data sequencing is depicted in figure 18, which has been taken from figure 5-11 of the USB 2.0 specifications. The last transaction in a microframe always uses the DATA0 PID. The second-to-last transaction in a microframe uses the DATA1 PID, and the third-to-last transaction in a microframe always uses the DATA2 PID.

Figure 18: Data Phase Sequence for Isochronous IN High Bandwidth Endpoints

For OUT high-speed, high-bandwidth isochronous endpoints, the data sequencing is depicted in figure 19, which has been taken from figure 5-12 of the USB 2.0 specifications. All transactions but the last transaction use the MDATA PID. The last transaction uses either the DATA0, DATA1, or DATA2 PID, depending on how many transactions were intended to take place during the microframe. If one transaction was meant to take place, it is also the last transaction and uses a DATA0 PID. If two transactions were meant to take place, the last transaction uses a DATA1 PID. If three transactions were meant to take place, the last transaction uses a DATA2 PID.

Figure 19: Data Phase Sequence for Isochronous OUT High Bandwidth Endpoints

USB Device Framework

The USB device framework is the thing that makes USB support so appealing. The transfer types and USB protocol are well-designed, of course, but the USB device framework defines standard device states that all devices must support, as well as standard requests and responses that allow the host to retrieve more than enough information about a device to determine the correct device driver and report information about the device even if the correct device driver isn't available (e.g, the manufacturer's name, the product's name, etc).

Functions, Configurations, Interfaces, and Endpoints

All USB devices, or functions, have at least one configuration, and every configuration has at least one interface. An interface may define zero or more endpoints. This relationship is illustrated in figure 20.

Although configurations descriptors are addressed sequentially starting with configuration descriptor zero, each configuration specifies a unique (within the scope of the function), none-zero configuration value. The **configuration value** is what the host needs to know in order to apply a certain configuration to a device. When asking for the current configuration of a device, a returned value of zero indicates that the device is not configured and is thus in the address state.

An **interface** defines the functional use of a set of endpoints and may imply that certain class-specific requests can be executed via the default control pipe. Thus, an interface need not necessarily define any additional endpoints. No interface may define the functional use of endpoint zero.

Each interface describes a unique set of endpoints within the scope of the configuration. However, an interface may provide one or more **alternate settings**, which may have different definitions for the same set of endpoints. When the host selects an alternate setting for an interface, the alternate setting's definitions are used instead of the default settings of the same interface.

USB Device States

A USB device may define states that are internal to the device, however the USB device framework defines a set of states that are visible to both the host and the device. Those visible states are the following:

- **Attached -** Immediately after the USB device is attached to the USB system, it is in this state. The USB specifications do not define the state of a USB device that is detached from a USB system.
- **Powered -** A device is in this state after it has both been attached to the bus, and the V_{BUS} line is applied to the device (the host controller drives the V_{BUS} at +5V, however this is only particularly important for hardware developers). In this state, the device must not respond to any bus transactions. The USB specification recognizes three potential scenarios with respect to how a device draws power:
  - *Self-Powered Devices* draw power from an external power source (e.g, a USB printer plugs into the wall as well as a USB port). Although the device may be considered technically "powered" even before attachment to the USB, it is still only considered powered after the V_{BUS} line is applied to the device.
  - *Bus-Powered Devices* draw power solely from the USB up to 100mA.
  - *Self- or Bus-Powered Devices* may draw power from either the bus or an external power source, depending on the configuration. These devices may change power source at any time. If a device is currently self-powered and requires more than 100mA of power, but switches to being bus-powered, then the device must return to the Address state.
- **Default -** A device in the powered state enters the default state after receiving a bus reset. In this state, the device is addressable at the default, reserved address of 0. At this point, the device is operating at the correct speed. The host is expected to allow 10 milliseconds before expecting the device to respond to data transfers after reset.

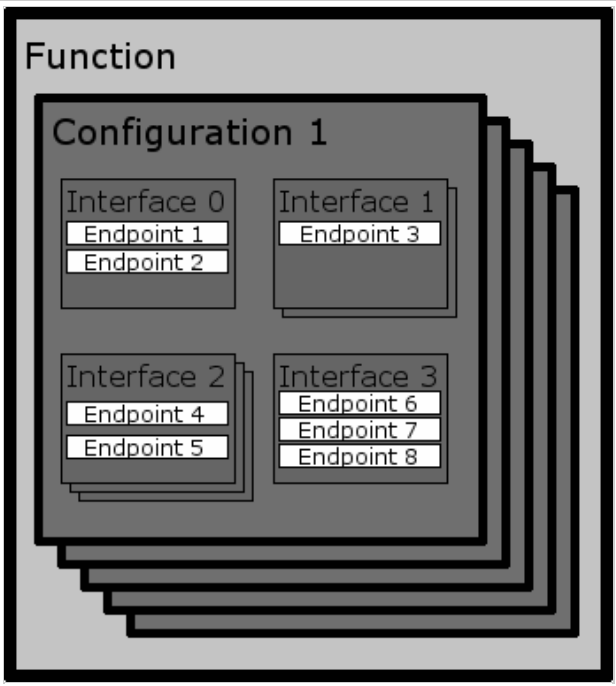


Figure 20: Illustration of the relationship between functions, configurations, interfaces, and endpoints.

- **Address -** A device enters this state after the host assigns it an address via the default control pipe, which is always accessible whether the device's address has been set or not.
- **Configured -** A device is in this state after the host examines its possible configurations and selects one. All endpoint's data toggle bits are initialized to zero when a device enters this state.
- **Suspended -** When no traffic is observed on the bus for a period of 1 millisecond, a USB device enters this state, characterized by its low power consumption. The device's address and configuration settings are maintained while suspended. A device exits the suspended state as soon as it begins seeing bus activity again. The host is expected to allow 10 milliseconds before expecting the device to respond to data transfers after resume.

Remote Wakeup Capability

One of the reasons a USB device may stop seeing USB traffic and thus enter a suspended state is because the host may have entered a suspended state as well. Some devices, typically keyboards and mice, support the ability to issue a remote wakeup signal to the host. In case the host software does not support remote wakeup, this capability must be disabled when a USB device is reset. If the host does support remote wakeup, then it may selectively enable the remote wakeup capability for specific devices (typically as chosen by the user). Then, these devices may issue a remote wakeup signal while in a suspended state to request that the host exits its own suspended state.

USB Device Enumeration

The following describes the process of bus enumeration, which occurs after a device is connected to a powered port:

1. The hub to which the device has been attached notifies the host via its status change pipe. The newly attached device is in the powered state at this point, and the port to which it has been attached is disabled.
2. The host queries more information from the hub to determine that a device has been attached, and to which port.

3. The host must wait at least 100 millisecond to allow a device to complete its insertion process, and for power to stabilize at the device. After the delay, the host enables the port and issues a reset signal to the device for at least 50 milliseconds.
4. The hub performs any required reset processing. After the reset signal has been released, the port is enabled and the device enters the default state.
5. The host assigns the device a unique address, thereby transitioning the device into the address state.
6. The host requests the device descriptor from the device via the default control pipe in order to determine the actual maximum data payload size of the default control pipe for the device. This step may occur before or after the host assigns the device an address.
7. The host reads all the possible device configuration information.
8. The host selects a certain configuration from the list of configurations supported by the device and sets the device to use that configuration. Optionally, the host may also select alternate interface settings within a configuration. All endpoints are initialized as described by the selected configuration, and the device is ready to use.

USB Device Requests

Standard, class-specific, and vendor-specific requests are made to the USB device over the default control pipe. The SETUP transaction always has a data payload size of 8 bytes, as noted in the Maximum Data Payload Size section of Control Transfers. The format of the setup data is as follows:

<table>
<tr><th>Offset</th><th>Field</th><th>Size</th><th>Type</th><th>Description</th></tr>
<tr><td>0</td><td>bmRequestType</td><td>1</td><td>Bitmap</td><td>
<table>
<tr><td>D7</td><td>D6</td><td>D5</td><td>D4</td><td>D3</td><td>D2</td><td>D1</td><td>D0</td></tr>
</table>
D7 Data transfer direction * The value of this bit is ignored when wLength is zero
0b = Host-to-device
1b = Device-to-host
D6...5 Type of request
00b = Standard
01b = Class
10b = Vendor
11b = Reserved
D4...0 Recipient
00000b = Device
00001b = Interface
00010b = Endpoint
00011b = Other
00100b to 11111b = Reserved
</td></tr>
<tr><td>1</td><td>bRequest</td><td>1</td><td>Value</td><td>Specific request</td></tr>
<tr><td>2</td><td>wValue</td><td>2</td><td>Value</td><td>Word-sized field the may (or may not) serve as a parameter to the request, depending on the specific request.</td></tr>
<tr><td>4</td><td>wIndex</td><td>2</td><td>Index or offset</td><td>Word-sized field that may (or may not) serve as a parameter to the request, depending on the specific request. Typically this field holds an index or an offset value.

When bmRequestType specifies an endpoint as the recipient, the format of this field is as follows:
<table>
<tr><td>D7</td><td>D6</td><td>D5</td><td>D4</td><td>D3</td><td>D2</td><td>D1</td><td>D0</td></tr>
<tr><td>Direction</td><td colspan="3">Reserved (reset to zero)</td><td colspan="4">Endpoint Number</td></tr>
<tr><td>D15</td><td>D14</td><td>D13</td><td>D12</td><td>D11</td><td>D10</td><td>D9</td><td>D8</td></tr>
</table>
</td></tr>
</table>

				Reserved (reset to zero) The direction bit (bit **D7**) is set to zero to indicate the OUT endpoint with the specified endpoint number, or it is set to one to indicate the IN endpoint with the specified endpoint number. The host should always set the direction bit to zero (but the device should accept either value) when the endpoint is part of a control pipe. When bmRequestType specifies an interface as the recipient, the format of this field is as follows: **D7** **D6** **D5** **D4** **D3** **D2** **D1** **D0** Interface Number **D15** **D14** **D13** **D12** **D11** **D10** **D9** **D8** Reserved (reset to zero)
6	wLength	2	Count	Number of bytes to transfer if there is a DATA stage. ▪ If this field is non-zero, and bmRequestType indicates a transfer from device-to-host, then the device must never return more than *wLength* bytes of data. However, a device may return less. ▪ If this field is non-zero, and the bmRequestType indicates a transfer from host-to-device, then the host must send exactly *wLength* bytes of data. If the host sends more than *wLength* bytes, the behavior of the device is undefined.

When a device receives a request that is undefined, is inappropriate given the current setting or state of the device, or uses values that are inappropriate for the particular request, then a **Request Error** exists. The device handles a Request Error by returning a STALL PID to the next DATA or STATUS stage, preferably at the next DATA stage transaction.

Standard Requests

The standard requests are defined for all USB devices, and all USB devices must respond to these standard requests even if the device hasn't been assigned an address, or the device hasn't been configured. To issue a certain request, the software creates the SETUP stage's DATA packet using the appropriate **request code**, a valid bmRequestType, the appropriate parameter values (or zero, if not applicable) for wValue and wIndex, and the amount of data bytes to be transfered for wLength. To the right are the standard USB device request codes, and the remainder of this section discusses each request.

bRequest	Value
GET_STATUS	0
CLEAR_FEATURE	1
Reserved	2
SET_FEATURE	3
Reserved	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNC_FRAME	12

Standard USB Request Codes

SET_ADDRESS

The SET_ADDRESS request has the following SETUP DATA packet format:

bmRequestType	bRequest	wValue	wIndex	wLength
00000000b	SET_ADDRESS 5	Device Address	Zero	Zero

SET_ADDRESS SETUP DATA Packet Format

This request does not have a DATA stage, only a SETUP and STATUS stage.

wValue specifies the address to be assigned to the device. The behavior of a device is undefined when *wValue* specifies an address greater than 127.

The exact behavior of the device after the SET_ADDRESS requests depends on the current state of the device:

- When a device is in the default state, a non-zero *wValue* causes the device to transition into the address state. When a device is in the default state, a *wValue* of zero has no effect.
- When a device is in the address state, a non-zero *wValue* keeps a device in the Address state, but the device responds to the newly set address. When a device is in the address state, a *wValue* of zero transitions the device into the default state.
- When a device is in the configurd state, device behavior is not defined for the SET_ADDRESS request.

This is the only request that is complete after the STATUS stage completes successfully. After the reset/resume recovery interval (10 milliseconds), a device is expected to be able to complete the STATUS stage of this request within 50 milliseconds. After the STATUS stage is complete, the device is allowed a 2 millisecond recovery interval before it must be able to accept farther SETUP packets for additional requests.

GET_DESCRIPTOR

The GET_DESCRIPTOR request has the following SETUP DATA packet format:

Descriptor Type	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER	8

Standard USB Descriptor Types

bmRequestType	bRequest	wValue		wIndex	wLength
10000000b	GET_DESCRIPTOR 6	Descriptor Type	Descriptor Index	Zero or Language ID	Descriptor Length

GET_DESCRIPTOR SETUP DATA Packet Format

The high-order byte of *wValue* specifies the descriptor type (see table of usb standard descriptor types, to the right). The low-order byte of *wValue* is only used for selecting a specific STRING or CONFIGURATION descriptor, and should be reset to zero otherwise.

The *wIndex* field is only used for STRING descriptors to specify the desired language and should be reset to zero for other descriptor types.

Different descriptor types have different lengths which will be discussed soon. If *wLength* is less than the size of the descriptor being returned, then the device only returns the first *wLength* bytes of the descriptor data. If *wLength* is larger than the size of the descriptor being returned, than the full descriptor is returned, followed by a short packet (a packet shorter than the maximum data payload size, including a length of 0 bytes).

All USB devices must support requests for DEVICE, CONFIGURATION, and STRING descriptors. All high-speed devices must support basic operations at full-speed; such devices also support DEVICE_QUALIFIER and OTHER_SPEED_CONFIGURATION descriptors which return the same data that the device would return for DEVICE and CONFIGURATION descriptor requests, respectively, if the device were operating at the speed at which it is not currently operating.

A request for a CONFIGURATION descriptor also returns all the INTERFACE descriptors for the specified configuration descriptor index (i.e, the low-order byte of *wValue*), as well as all of the ENDPOINT descriptors associated with all of the returned INTERFACE descriptors, all in a single request.

GET_DESCRIPTOR is a valid request for a device in the default, address, or configured state.

SET_DESCRIPTOR

The SET_DESCRIPTOR request is optional; when it is supported, it may be used to update descriptors or add new ones.

The SET_DESCRIPTOR request has the following SETUP DATA packet format:

bmRequestType	bRequest	wValue		wIndex	wLength
00000000b	SET_DESCRIPTOR 7	Descriptor Type	Descriptor Index	Zero or Language ID	Descriptor Length

SET_DESCRIPTOR SETUP DATA Packet Format

The high-order byte of *wValue* specifies the descriptor type. The low-order byte of *wValue* is only used for selecting a specific STRING or CONFIGURATION descriptor, and should be reset to zero otherwise.

The *wIndex* field is only used for STRING descriptors to specify the desired language and should be reset to zero for other descriptor types.

The *wLength* field specifies how many bytes will be transfered from the host to the device.

This request only supports DEVICE, CONFIGURATIOn, and STRING descriptor types.

If this request is not supported, the device responds with a Reuqest Error.

If this request is supported, it is only valid when the device is in the address or configured state; the behavior of the device is undefined if this request is made while the device is in the default state.

GET_CONFIGURATION

The GET_CONFIGURATION request has the following SETUP DATA packet format:

bmRequestType	bRequest	wValue	wIndex	wLength
10000000b	GET_CONFIGURATION 8	Zero	Zero	One

GET_CONFIGURATION SETUP DATA Packet Format

The device sends a one-byte DATA packet during the DATA phase of the control transfer. This byte is the value of the current configuration of the device. A value of zero indicates that the device has not yet been configured (it is in the address state). Behavior of a device is undefined if this request is issued while the device is in the default state.

SET_CONFIGURATION

The SET_CONFIGURATION request has the following SETUP DATA packet format:

bmRequestType	bRequest	wValue		wIndex	wLength
00000000b	SET_CONFIGURATION 9	*Reserved*	Configuration Value	Zero	Zero

SET_CONFIGURATION SETUP DATA Packet Format

The low-order byte of *wValue* specifies the desired configuration value. The low-order byte of *wValue* must either be zero, or it must match the configuration value field of a configuration descriptor returned by the device. Specifying a configuration value of zero sets the device into the address state.

If the device is in the default state, or if the high-order byte of *wValue* is not zero, *wIndex* is not zero, or *wLength* is not zero, then the behavior after issuing this request is undefined.

If the specified configuration value is neither zero nor a valid configuration value specified by a configuration descriptor of the device, the device responds with a Request Error.

GET_INTERFACE

The GET_INTERFACE request has the following SETUP DATA packet format:

bmRequestType	bRequest	wValue	wIndex	wLength

10000001b	GET_INTERFACE 10	Zero	Interface	One

GET_INTERFACE SETUP DATA Packet Format

The host uses this request to determine which alternate setting (as described in Functions, Configurations, Interfaces, and Endpoints) is used for a particular interface of the current configuration. The device responds with a one-byte long DATA packet during the data phase, the transfered byte being the alternate setting value for the interface specified in this request.

If *wValue* is not zero, *wLength* is not one, *wIndex* specifies an invalid interface, or the device is in the address state, then the device responds with a Request Error.

The behavior of a device in the default state after receiving this request is undefined.

This request is valid for a device in the configured state.

SET_INTERFACE

The SET_INTERFACE request has the following SETUP DATA packet format:

bmRequestType	bRequest	wValue	wIndex	wLength
00000001b	SET_INTERFACE 11	Alternate Setting	Interface	Zero

SET_INTERFACE SETUP DATA Packet Format

The host uses this request to select an alternate setting (as described in Functions, Configurations, Interfaces, and Endpoints) to be used for a particular interface of the current configuration. If the interface specified only supports a default setting, then the device may return a STALL handshake during the STATUS stage of the request.

If the interface or alternate setting do not exist, or if the device is in the address state, then the device responds with a Request Error. The behavior of the device is undefined if *wLength* is not zero, or the device is in the default state.

This is a valid request when the device is in the configured state.

CLEAR_FEATURE

The CLEAR_FEATURE request has the following SETUP DATA packet format:

Feature Selector	Recipient	Value
DEVICE_REMOTE_WAKEUP	Device	1
ENDPOINT_HALT	Endpoint	0
TEST_MODE	Device	2

Standard USB Feature Selectors

bmRequestType	bRequest	wValue	wIndex	wLength
00000000b 00000001b 00000010b	CLEAR_FEATURE 1	Feature Selector	Zero or Interface or Endpoint	Zero

CLEAR_FEATURE SETUP DATA Packet Format

The host uses this request to clear or disable a specific feature.

wValue must contain a feature selector (see table of Standard USB Feature Selectors, to the right) which corresponds with the recipient as specified in the *bmRequestType* value.

Issuing this request while referencing a feature that cannot be cleared or does not exist, or referencing an interface or endpoint that does not exist will cause the device to respond with a Request Error.

If the device is in the default state, or *wLength* is non-zero, then the behavior of the device is undefined.

This request is valid when the device is in the configured state. When the device is in the address state, this request is only valid when referencing endpoint zero, otherwise the device responds with a Request Error.

The TEST_MODE feature cannot be cleared by this request.

SET_FEATURE

The SET_FEATURE request has the following SETUP DATA packet format:

Value	Description
00h	*Reserved*
01h	Test_J
02h	Test_K
03h	Test_SE0_NAK
04h	Test_Packet
05h	Test_Force_Enable
06h-3Fh	*Reserved for standard test selectors*
3Fh-BFh	*Reserved*
C0h-FFh	*Reserved for vendor-specific test modes*

Standard USB Test Selectors

bmRequestType	bRequest	wValue	wIndex		wLength
00000000b 00000001b 00000010b	SET_FEATURE 3	Feature Selector	Test selector	Zero or Interface or Endpoint	Zero

SET_FEATURE SETUP DATA Packet Format

The host uses this request to set or enable a specific feature.

wValue must contain a feature selector which corresponds with the recipient as specified in the *bmRequestType* value.

When *wValue* selects the TEST_MODE feature, *bmRequestType* and the low-order byte of *wIndex* must both be reset to zero. The high-order byte of *wIndex* must be a valid test selector (see table of Standard USB Test Selectors, to the right), or the device respond with Request Error. The device must set its upstream-facing port into test mode no longer than 3milliseconds after completing the STATUS stage of this request. In order to exit test mode, the power to the device must be cycled. A device must support the TEST_MODE feature in the default, address, and configured high-speed device states.

If this request references a feature that does not exist or cannot be set, then the devices responds with a STALL handshake during the STATUS stage.

If an endpoint or interface is specified that does not exist, or if the device is in the address state and an endpoint other than endpoint zero is specified, the device responds with a Request Error.

Besides requests which select the TEST_MODE feature, issuing this request to a device in the default state results in undefined behavior. A non-zero value for *wLength* also results in undefined behavior.

This request is valid when a device is in the configured state, or when the device is in the address state and only endpoint zero is referenced.

GET_STATUS

The GET_STATUS request has the following SETUP DATA packet format:

bmRequestType	bRequest	wValue	wIndex	wLength
10000000b 10000001b 10000010b	GET_STATUS 0	Zero	Zero or Interface or Endpoint	Two

GET_STATUS SETUP DATA Packet Format

The host uses this request to learn the status of the recipient, as specified by the *bmRequestType* field and, in the case of an interface or endpoint recipient, the *wIndex* field.

If *wValue* is not zero, *wLength* is not two, or *wIndex* is non-zero when *bmRequestType* specifies a device recipient, or the device is in the default state, then the behavior of the device is undefined.

If this request references an endpoint or interface that does not exist (including any endpoint other than endpoint zero when the device is in the address state), then the device responds with a Request Error.

In response to this request, the device issues a 2 byte long data transfer during the DATA stage to the host. These two bytes represent the requested status and the meaning depends on the recipient type.

Device Recipient

When the recipient was a device, the two bytes describe the status as follows:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Reserved (reset to zero)						Remote Wakeup	Self Powered
D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
Reserved (reset to zero)							

The *Self Powered* field is set to 1 to indicate that the device is currently powered by an external power source, or 0 to indicate that the device is currently running on power supplied by the bus.

The *Remote Wakeup* field is set to 0 when the device is reset and indicates whether or not the device is currently enabled to perform remote wakeup signaling (see Remote Wakeup Capability). The host may modify the value of the *Remote Wakeup* field by issuing either a CLEAR_FEATURE or SET_FEATURE request using the DEVICE_REMOTE_WAKEUP feature selector.

Interface Recipient

When the recipient was an interface, the two bytes describe the status as follows:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Reserved (reset to zero)							
D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
Reserved (reset to zero)							

Endpoint Recipient

When the recipient was an endpoint, the two bytes describe the status as follows:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Reserved (reset to zero)							Halt
D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
Reserved (reset to zero)							

All interrupt and bulk endpoint types must implement the halt feature, otherwise it is optional. The *Halt* field reflects the status of the halt feature of the endpoint. A value of 0 in the *Halt* field indicates that the endpoint is not halted, and a value of 1 in the *Halt* field indicates that the endpoint is halted.

The host may set the halt feature of an endpoint with the SET_FEATURE request using the ENDPOINT_HALT feature selector, or the host may clear the halt feature of an endpoint with the CLEAR_FEATURE request using the ENDPOINT_HALT feature selector. When the CLEAR_FEATURE request is used in this manner, and the endpoint uses a data toggle bit, the data toggle bit is reset to zero.

The default control pipe is not required nor recommended to implement the halt feature, but some devices may choose to use the halt feature on the default control pipe to reflect a functional error condition. When the halt feature is set on the default control pipe, the device responds with a STALL handshake during the DATA or STATUS stage of all transfers with the exception of the GET_STATUS, CLEAR_FEATURE, and SET_FEATURE standard requests. Additionally, the device is not required to stall vendor- or class-specific requests when the halt feature is set.

SYNCH_FRAME

The SYNCH_FRAME request has the following SETUP DATA packet format:

bmRequestType	bRequest	wValue	wIndex	wLength
10000010b	SYNCH_FRAME 12	Zero	Endpoint	Two

SYNCH_FRAME SETUP DATA Packet Format

This request is only used for isochronous endpoints that use implicit pattern synchronization. That is, some isochronous endpoints require per-frame transfers to vary in size according to a specific pattern (in order to attain an application-specific bit rate, for example). This call causes the device to send the host a 2 byte value that is the number of the frame where the pattern began.

High-speed isochronous endpoints that support this request must synchronize to the zeroth microframe, as well as have a time notion of classic frames (1 millisecond as opposed to 125 microsecond intervals).

If *wValue* is not zero, *wLength* is not two, or the device is in the default state, then the behavior of the device is undefined.

If the specified endpoint does not support this request, or the device is in the address state, then the device responds with a Request Error.

This request is valid when the device is in the configured state.

Standard USB Descriptors

A **descriptor** is a data structure with a defined format. All standard descriptors begin with two bytes. The first byte specifies the total length of the descriptor in bytes, including the first two mandatory bytes. The second byte identifies the type of the descriptor.

Some descriptors contain fields which specify an index of a STRING descriptor, but it is optional for a device to support STRING descriptors. If a device does not support STRING descriptors, then all fields which reference an index of a STRING descriptor should be reset to zero. Thus, a value of zero in any field that is meant to supply an index of a STRING descriptor indicates that no such STRING descriptor is available.

If the second byte of a descriptor identifies that descriptor as one of the standard USB descriptors, but the first byte of that descriptor specifies a length less than the lengths defined in the USB 2.0 specifications (and, transitively, here), then the descriptor should be rejected by the host. If the length field reports that the descriptor is longer than expected, then the extra data should be ignored, but still considered part of the descriptor (this is important when the device is returning multiple descriptors, as is the case when the host requests a CONFIGURATION descriptor).

If class- or vendor-specific descriptors use the same format as standard descriptors (i.e, the two mandatory bytes at the beginning of the descriptor), then the class- or vendor-specific descriptors are interleaved within the results when the host requests a CONFIGURATION descriptor. Otherwise, the class- or vendor-specific descriptors are accessed by passing a class- or vendor-specific descriptor type in a GET_DESCRIPTOR request.

The remainder of this section serves to catalog the standard USB device descriptors and very closely mirrors section 9.6 of the USB 2.0 specifications. These descriptor definitions supplement the GET_DESCRIPTOR request.

DEVICE

Every USB device has exactly one DEVICE descriptor. This descriptor provides general information about the device, as well as information that applies globally to the device and all of its configurations.

Offset	Field	Size	Type	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	DEVICE Descriptor Type
2	bcdUSB	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e, 2.10 is expressed as 210h). Identifies the release of the USB Specification with with the device and its descriptors are compliant.
4	bDeviceClass	1	Class	Class code (assigned by the USB-IF) ▪ This field is reset to zero if each interface within a configuration specifies its own class information and the various interfaces operate independently. ▪ A value of FFh in this field indicates the device class is vendor-specific.
5	bDeviceSubClass	1	SubClass	Subclass Code (assigned by the USB-IF) ▪ The subclass code of a device is qualified by the class code of that device. ▪ If *bDeviceClass* is reset to zero, then this field must also be reset to zero. ▪ When *bDeviceClass* is not set to FFh, then all values for this field are reserved for assignment by the USB-IF.
6	bDeviceProtocol	1	Protocol	Protocol code (assigned by the USB-IF) ▪ The protocol code of a device is qualified by both the class and subclass codes of that device. ▪ A value of 00h in this field means that the device may specify class-specific protocols on an interface basis, though this is not a requirement. ▪ If this field is set to FFh, then the device uses a vendor-specific protocol.
7	bMaxPacketSize0	1	Number	Maximum packet size for endpoint zero (8, 16, 32, or 64 are the only valid options)
8	idVendor	2	ID	Vendor ID (assigned by the USB-IF)
10	idProduct	2	ID	Product ID (assigned by the USB-IF)
12	bcdDevice	2	BCD	Device release number in binary-coded decimal
14	iManufacturer	1	Index	Index of STRING descriptor describing manufacturer
15	iProduct	1	Index	Index of STRING descriptor describing product
16	iSerialNumber	1	Index	Index of STRING descriptor describing the device's serial number
17	bNumConfigurations	1	Number	Number of possible configurations

DEVICE_QUALIFIER

A high-speed capable device that has different device information depending on the speed in which the device operating, then that device must also have a DEVICE_QUALIFIER descriptor. This descriptor provides information about the device that would change if the device were operating at the alternate speed (i.e, when the device is operating at high-speed, this descriptor provides the differences if the device were operating a full-speed, and vice versa). This descriptor leaves out fields from the DEVICE descriptor that would not reasonably depend on the speed of the device (e.g, index of the STRING descriptor describing the product).

If a full-speed only device with a *bcdUSB* field of at least 0200h in its DEVICE descriptor receives a request for a DEVICE_QUALIFIER descriptor, it must respond with a Request Error.

Offset	Field	Size	Type	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	DEVICE_QUALIFIER Descriptor Type
2	bcdUSB	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e, 2.00 is expressed as 200h). Identifies the release of the USB Specification with with

				the device and its descriptors are compliant. This field must be at least 0200h.
4	bDeviceClass	1	Class	Class code (assigned by the USB-IF)
5	bDeviceSubClass	1	SubClass	Subclass Code (assigned by the USB-IF)
6	bDeviceProtocol	1	Protocol	Protocol code (assigned by the USB-IF)
7	bMaxPacketSize0	1	Number	Maximum packet size for endpoint zero (8, 16, 32, or 64 are the only valid options)
8	bNumConfigurations	1	Number	Number of possible configurations
9	bReserved	1	-	Reserved for future uses, must be zero.

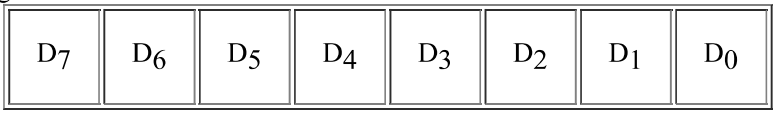
CONFIGURATION

All USB devices have at least one CONFIGURATION descriptor. The host may request a specific CONFIGURATION descriptor by its descriptor indexx, which is zero based and has *bNumConfigurations* (as returned in the DEVICE descriptor) used indices. That is, the valid values to be used as a descriptor index when requesting a CONFIGURATION descriptor are any integer in the range of 0 to *bNumConfigurations*-1, inclusive.

Each CONFIGURATION descriptor has at least one INTERFACE descriptor, and each INTERFACE descriptor may have up to 15 ENDPOINT descriptors. When the host requests a certain CONFIGURATION descriptor, the device returns the CONFIGURATION descriptor followed immediately by the first INTERFACE descriptor, followed immediately by all of the ENDPOINT descriptors for endpoints that the interface defines (which may be none). This is followed immediately by the next INTERFACE descriptor if one exists, and then by its ENDPOINT descriptors if applicable. This pattern continues until all the information within the scope of the specific configuration is transfered.

When a device has vendor- or class-specific descriptors that conform to the standard USB descriptor format (that is, the first byte of the descriptor determines the length of the descriptor, and the second byte identifies the type of descriptor), those descriptors are also returned interleaved among the CONFIGURATION, INTERFACE, and ENDPOINT descriptors when the host requests a specific CONFIGURATION descriptor. Therefore, the system software cannot assume continuous standard descriptors as implied by the previous paragraph; instead, the system software should check the descriptor type and skip that descriptor if it is not a standard descriptor. The software should also check that standard descriptors report at least the expected length.

Note that the CONFIGURATION descriptor index is not the same as the value *bConfigurationValue* in the CONFIGURATION descriptor. *bConfigurationValue* is the value that the host passes as a parameter with the SET_CONFIGURATION request in order to select a particular configuration, whereas this cannot be done using the CONFIGURATION descriptor index.

Offset	Field	Size	Type	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	CONFIGURATION Descriptor Type
2	wTotalLength	2	Number	The total combined length in bytes of all the descriptors returned with the request for this CONFIGURATION descriptor (including CONFIGURATION, INTERFACE, ENDPOINT, class- and vendor-specific descriptors).
4	bNumInterfaces	1	Number	Number of interfaces supported by this configuration
5	bConfigurationValue	1	Number	Value which when used as an argument in the SET_CONFIGURATION request, causes the device to assume the configuration described by this descriptor.
6	iConfiguration	1	Index	Index of STRING descriptor describing this configuration.
7	bmAttributes	1	Bitmap	Configuration Characteristics  **D7** *Reserved, must be set to one for historical reasons* **D6** **Self-Powered** ▪ 0 = Device runs on power supplied by the bus

				▪ 1 = Device provides a local power source, if *bMaxPower* is non-zero, the device also may use bus power. **D_5 Remote Wakeup** ▪ 0 = Remote Wakeup not supported ▪ 1 = Remote Wakeup supported **$D_{4...0}$** *Reserved, reset to zero*
8	bMaxPower	1	mA	Maximum power consumption of this device from the bus when fully operational and using this configuration. ▪ Expressed in units of 2mA (i.e., a value of 50 in this field indicates 100mA). ▪ A device reports with the *bmAttributes* field whether the configuration is bus- or self-powered, but the device status (retrieved with a GET_STATUS request) reports whether the device is currently self-powered. ▪ If a device is disconnected from an external power source, it may not draw more power from the bus than specified in this field. ▪ Some devices may be able to operate solely on bus power. A device that cannot and has lost its external power source will fail the operations it can no longer support. It is up to the software on the host to determine when this is the case, which may be accomplished with a GET_STATUS request.

OTHER_SPEED_CONFIGURATION

This descriptor describes the configuration of a high-speed device if it were operating at it's alternative speed. The host should not request this descriptor unless it already successfully received a DEVICE_QUALIFIER descriptor from the device. The structure of the OTHER_SPEED_CONFIGURATION is identical to that of the CONFIGURATION descriptor shown above. The only difference is that the *bDescriptorType* field reflects that the descriptor is an OTHER_SPEED_CONFIGURATION descriptor rather than a CONFIGURATION descriptor.

INTERFACE

INTERFACE descriptors are only returned following a CONFIGURATION descriptor when the host requests a specific CONFIGURATION descriptor; it is not possible to directly request a specific INTERFACE descriptor. An interface may provide alternate settings within a configuration that allow the endpoints and/or their characteristics to be varied. A default interface has the *bAlternateSetting* field in its INTERFACE descriptor reset to zero.

Offset	Field	Size	Type	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	INTERFACE Descriptor Type
2	bInterfaceNumber	1	Number	Number of this interface. Zero-based value which identifies the index of this interface in the array of interfaces supported within a configuration.
3	bAlternateSetting	1	Number	Value used to select the alternate settings described by this INTERFACE descriptor for the interface with the *bInterfaceNumber* in the previous field. This value is zero if this descriptor describes the default settings for a particular interface.
4	bNumEndpoints	1	Number	Number of endpoints used by this interface, not including endpoint zero.
5	bInterfaceClass	1	Class	Class code (assigned by the USB-IF) ▪ A value of zero here is reserved for future standardization. ▪ If this value is FFh, the interface class is vendor-specific. ▪ All other values are reserved for assignment by the USB-IF.
6	bInterfaceSubClass	1	SubClass	Subclass code (assigned by the USB-IF)

				▪ The subclass code in this field is qualified by the value of the *bInterfaceClass* field. ▪ If *bInterfaceClass* is reset to zero, then this field must also be reset to zero. ▪ If *bInterfaceClass* is not set to the value of FFh, then all values of this field are reserved for assignment by the USB-IF.
7	bInterfaceProtocol	1	Protocol	Protocol code (assigned by the USB-IF) ▪ The protocol code in this field is qualified by the values of the *bInterfaceClass* and *bInterfaceSubClass* fields. ▪ If an interface supports class-specific requests, then this field identifies the protocols that the device uses as defined by the specifications of the device class. ▪ If this field is reset to zero, then the device does not use a class-specific protocol on this interface. ▪ If this field is set to FFh, then the devices uses a vendor-specific protocol on this interface.
8	iInterface	1	Index	Index of STRING descriptor describing this interface

ENDPOINT

Each endpoint used for a particular interface has a descriptor which follows after that particular interface's descriptor when the host requests a specific CONFIGURATION descriptor; the host cannot request a specific ENDPOINT descriptor explicitly. And ENDPOINT descriptor never describes endpoint zero.

Offset	Field	Size	Type	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	ENDPOINT Descriptor Type
2	bEndpointAddress	1	Endpoint	The address of the endpoint on the USB device described by this descriptor. This field has the following format: D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0 **D_7** **Direction** (ignored for control endpoints) ▪ 0 = OUT endpoint ▪ 1 = IN endpoint **$D_{6...4}$** *Reserved, reset to zero* **$D_{3...0}$** **Endpoint Number**
3	bmAttributes	1	Bitmap	This field describes the endpoint's attributes as follows: D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0 **$D_{7...6}$** *Reserved, reset to zero* **$D_{5...4}$** **Usage Type** (Isochronous endpoints only; reserved and reset to zero for other endpoints) ▪ 00 = Data endpoint ▪ 01 = Feedback endpoint ▪ 10 = Implicit feedback data endpoint ▪ 11 = Reserved **$D_{3...2}$** **Synchronization Type** (Isochronous endpoints only; reserved and reset to zero for other endpoint types) ▪ 00 = No synchronization ▪ 01 = Asynchronous

				▪ 10 = Adaptive ▪ 11 = Synchronous **$D_{1...0}$ Transfer Type** ▪ 00 = Control ▪ 01 = Isochronous ▪ 10 = Bulk ▪ 11 = Interrupt
4	wMaxPacketSize	2	Number	Maximum packet size that this endpoint is capable of sending or receiving. ▪ For isochronous endpoints, this value is used to reserve bus time; the pipe, however, may not always use all of the reserved bus time. ▪ Bits 10...0 specify the maximum packet size in bytes. ▪ For high-speed isochronous and interrupt endpoints, bits 12...11 specify the number of additional transaction opportunities per microframe (see High-Speed, High-Bandwidth Endpoints). The format is as follows: ▪ 00 = None (1 transaction per microframe) ▪ 01 = 1 additional (2 transactions per microframe) ▪ 10 = 2 additional (3 transactions per microframe) ▪ 11 = Reserved ▪ Bits 15...13 are reserved and must be reset to zero.
6	bInterval	1	Number	Interval for polling a device during a data transfer, expressed in units of microframes for high-speed devices, and frames for low- and full-speed devices. The exact meaning of the value in this field depends on the endpoint type and the operating speed of the device: ▪ Full- and High-speed isochronous endpoints, and high-speed interrupt endpoints: ▪ This field must be in the range from 1 to 16. ▪ This field is used to calculate the period as $2^{bInterval - 1}$. That is, a value of 4 calculates to $2^{4 - 1} = 2^3 = 8$. ▪ Full- and Low-speed interrupt endpoints: ▪ This field must be in the range from 1 to 255. ▪ High-speed bulk and control OUT endpoints: ▪ This field must be in the range from 0 to 255. ▪ This field specifies the maximum NAK rate of the endpoint. ▪ A value of zero indicates that the endpoint never NAKs ▪ Other values indicate at most 1 NAK each *bInterval* number of microframes. ▪ See PING Transaction Protocol

STRING

Devices may optionally support STRING descriptors. If a device does not support STRING descriptors, any field which references the index of a STRING descriptor must be reset to zero. STRING descriptors use unicode encodings and may support multiple languages. The host requests a STRING descriptor with the GET_DECRIPTOR request and must pass the 16-bit LANGID (as defined by the USB-IF) of the desired language in the *wIndex* field. The list of currently accepted LANGIDs is located here (http://www.usb.org/developers/docs/USB_LANGIDs.pdf) .

String index 0 for all languages returns a STRING descriptor that contains an array of all the two-byte LANGID codes that the device supports.

Whether requesting a string or an array of LANGIDs, the data is not NULL-terminated. Instead, the host determines the length of the data by subtracting 2 from the *bLength* field of the descriptor.

When the host requests string index 0, the following descriptor is returned:

Offset	Field	Size	Type	Description
0	bLength	1	Number	Size of this descriptor in bytes

1	bDescriptorType	1	Constant	STRING Descriptor Type
2	wLangID[0]	2	Number	LANGID code zero
...	...	...	...	...
N	wLangID[x]	2	Number	LANGID code x

When the host requests a valid string index other than string index 0 for a supported LANGID, the following descriptor is returned:

Offset	Field	Size	Type	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	STRING Descriptor Type
2	bString	N	Number	Unicode string

Typical organization of system software

This section discusses how system software is typically, reasonably organized. This section also serves as an index to the wiki entries which provide, or will provide, farther information and perhaps programming examples.

USB Device Drivers

As with any device driver, a USB device driver abstracts away from the low-level details on just how a specific device is being accessed, and provides the rest of the system and applications with a common interface (e.g, a file manager shouldn't have to know whether it is dealing with an external versus internal hard drive).

USB device drivers typically implement a certain class of device as per the appropriate specifications. Such classes of USB devices include, but are not limited to:

- USB Mass Storage Class Devices
- USB Human Input Devices

USB Driver

Even a USB device driver need not be concerned with some of the lower-level details. For instance, it shouldn't matter to the device driver if a device is connected directly to the root hub, or if it lies behind 3 hubs. The device driver shouldn't worry about how much power the device needs from the bus. This is where the USB driver comes in.

The USB driver essentially provides the USB framework interface to device drivers. The USB driver also handles connect and disconnect events on the USB, as well as determining which device driver is needed (according to the Class, Subclass, and Protocol codes), and if that device driver even exists.

USB Hub Driver

Although the USB Driver knows some details about the USB topography, the responsibility of hub-specific communication (including split-transactions) is often separated from the USB Driver into another module called the USB Hub Driver.

Depending on the design of the system, the USB Driver might bypass the USB Hub Driver when communicating with devices on the root hub, or the system may use the reserved address of 0 to indicate the root hub to the USB Hub Driver (it appears that Linux does this).

Details on USB Hubs will eventually be discussed in the USB Hubs wiki entry.

Host Controller Driver

As a request for a data transfer moves from the device driver, through the USB Driver, and through the USB Hub Driver, the request gains all the information needed for the host controller to generate the appropriate transactions on the bus. However, depending on the host controller, this information needs to be formatted in a certain way and added for scheduling by the host controller.

This task if given to the host controller driver. Requests reach the host controller driver in a system-defined format, often called a USB Request Block (URB), or an I/O Request Packet (IRP).

Additionally, host controller drivers are loaded by the PCI subsystem when a corresponding host controller is discovered during PCI enumeration. The host controller driver is thus also responsible for initializing the host controller and perhaps loading the USB Hub Driver and the USB driver. Combined, the USB driver, USB hub driver, and the host controller driver make up a USB subsystem.

Links

- USB.org (http://www.usb.org/home)
- USB Universal Serial Bus Revision 2.0 Specification (http://www.usb.org/developers/docs/usb_20_110512.zip)
- Univseral Serial Bus Revision 3.0 Specification (http://www.usb.org/developers/docs/usb_30_spec_122012.zip)
- Wireless USB Specification Revision 1.1 (http://www.usb.org/developers/wusb/wusb1_1_20100910.zip)
- The Linux kernel (http://www.kernel.org/) (though things tends to be confusing there, and you have to be careful with educating yourself from Linux sources if your project isn't GPL'ed).
- USB in a Nutshell (http://www.beyondlogic.org/usbnutshell/usb1.htm) may also interest you. It looks like a really good tutorial giving all the required knowledge to understand any other USB documentation/source code in a couple of HTML pages ...
- Currently accepted LANGIDs (http://www.usb.org/developers/docs/USB_LANGIDs.pdf)

Forum Topics

- Implementing USB Support - Briefly covers the necessary steps in EHCI configuration to ensure that the UHCI or OHCI companion controllers can handle all USB devices (e.g, before EHCI support is implemented where some BIOS may configure a device via EHCI during boot). May also be helpful for those new to EHCI who aren't sure how to manipulate/use the EHCI's registers.
- USB driver - suggests a few startup links about USB
- Collecting links about USB - PypeClicker and Df's collection of links about USB

Categories: USB | Buses
