2

Inside USB Transfers

This and the next three chapters are a tutorial on USB transfers. This chapter has essentials that apply to all transfers. The following chapters cover the four transfer types supported by USB, the enumeration process, and the standard requests used in control transfers.

You don't need to know every bit of this information to get a project up and running, but understanding something about how transfers work can help in deciding which transfer types to use, writing device firmware, and debugging.

The information in these chapters is dense. If you don't have a background in USB, you won't absorb it all in one reading. You should, however, get a feel for how USB works, and will know where to look later when you need to check the details.

The ultimate authority on the USB interface is the specification document, *Universal Serial Bus Specification,* available on the USB-IF's Web site. By design, the specification omits information and tips that are unique to any operating system or controller chip. This type of information is essential

when you're designing a product for the real world, so I include this information where relevant.

Transfer Basics

You can divide USB communications into two categories: communications used in enumerating the device and communications used by the applications that carry out the device's purpose. During enumeration, the host learns about the device and prepares it for exchanging data. Application communications occur when the host exchanges data that performs the functions the device is designed for. For example, for a keyboard, the application communications are the sending of keypress data to the host to tell an application to display a character or perform another action.

Enumeration Communications

During enumeration, the device's firmware responds to a series of standard requests from the host. The device must identify each request, return requested information, and take other actions specified by the requests.

On PCs, Windows performs the enumeration, so there's no user programming involved. However, to complete the enumeration, on first attachment, Windows must locate an INF file that identifies the file name and location of the device's driver. If the required files are available and the firmware functions correctly, the enumeration process is generally invisible to users. Chapter 9 has more details about device drivers and INF files.

Application Communications

After the host has exchanged enumeration information with the device and a device driver has been assigned and loaded, the application communications can begin. At the host, applications can use standard Windows API functions or other software components to read and write to the device. At the device, transferring data typically requires either placing data to send in the USB controller's transmit buffer or retrieving received data from the receive buffer, and on completing a transfer, ensuring that the device is ready

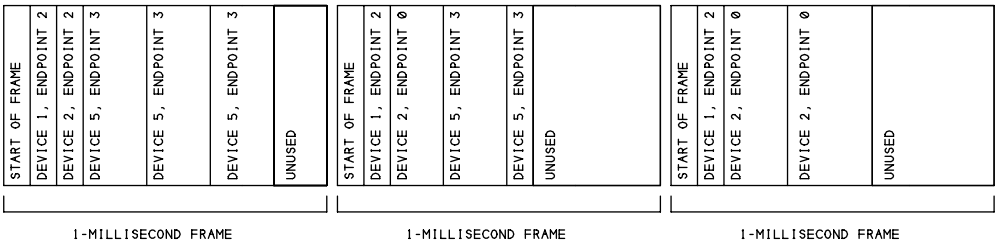

Figure 2-1: At low and full speeds, the host schedules transactions within 1-millisecond frames.The host may schedule transactions anywhere it wants within a frame. The process is similar at high speed, but using 125-microsecond microframes.

for the next transfer. Most devices also require additional firmware support for handling errors and other events. Each data transfer uses one of the four transfer types: control, interrupt, bulk, or isochronous. Each has a format and protocol to suit different needs.

Managing Data on the Bus

USB's two signal lines carry data to and from all of the devices on the bus. The wires form a single transmission path that all of the devices must share. (As explained later in this chapter, an exception is a cable segment between a 1.x device and a 2.0 hub on a high-speed bus, but even here, all data shares a path between the hub and host.) Unlike RS-232, which has a TX line to carry data in one direction and an RX line for the other direction, USB's pair of wires carries a single differential signal, with the directions taking turns.

The host is in charge of seeing that all transfers occur as quickly as possible. The host manages the traffic by dividing time into chunks called frames (at low and full speeds) or microframes (at high speed). The host allocates a portion of each frame or microframe to each transfer (Figure 2-1). A frame has a period of one millisecond. For high speed traffic, the host divides each frame into eight 125-microsecond microframes. Each frame or microframe begins with a Start-of-Frame timing reference.

Each transfer consists of one or more transactions. Control transfers always have multiple transactions because they have multiple stages, each consisting of one or more transactions. Other transfer types use multiple transactions when they have more data than will fit in a single transaction. Depending on how the host schedules the transactions and the speed of a device's response, a transfer's transactions may all be in a single frame or microframe, or they may be spread over multiple (micro)frames.

Because all of the traffic shares a data path, each transaction must include a device address that identifies the transaction's destination. Every device has a unique address assigned by the host, and all data travels to or from the host. Each transaction begins when the host sends a block of information that includes the address of the receiving device and a specific location, called an endpoint, within the device. Everything a device sends is in response to receiving a packet sent by the host.

Host Speed and Bus Speed

USB 2.0 hosts in general-purpose PCs support low, full, and high speeds. A 1.x host supports low and full speeds only. (Special-purpose hosts, typically found in small embedded systems, don't always support all speeds.)

A 1.x hub doesn't convert between speeds; it just passes received traffic up or down the bus, changing only the edge rate and signal polarity of traffic to and from attached low-speed devices. In contrast, a 2.0 hub acts as a remote processor with store-and-forward capabilities. The hub converts between high speed and low or full speed as needed and performs other functions that help make efficient use of the bus time. The added intelligence of 2.0 hubs is a major reason why the high-speed bus remains compatible with 1.x hardware.

The traffic on a bus segment is high speed only if the device is high speed and the host controller and all hubs between the host and device are 2.0-compliant. Figure 2-2 illustrates. A high-speed bus may also have 1.x hubs, and if so, any bus segments downstream from this hub (away from the host) are low or full speed. Traffic to and from low- and full-speed devices travels at high speed between the host and any 2.0 hubs that connect to the

host with no 1.x hubs between. Traffic between a 2.0 hub and a 1.x hub or another low- or full-speed device travels at low or full speed. A bus with only a 1.x host controller supports only low and full speeds, even if the bus has 2.0 hubs and high-speed-capable devices.

Elements of a Transfer

Every USB transfer consists of one or more transactions, and each transaction in turn contains packets that contain information. To understand transactions, packets, and their contents, you also need to know about endpoints and pipes. So that's where we'll begin.

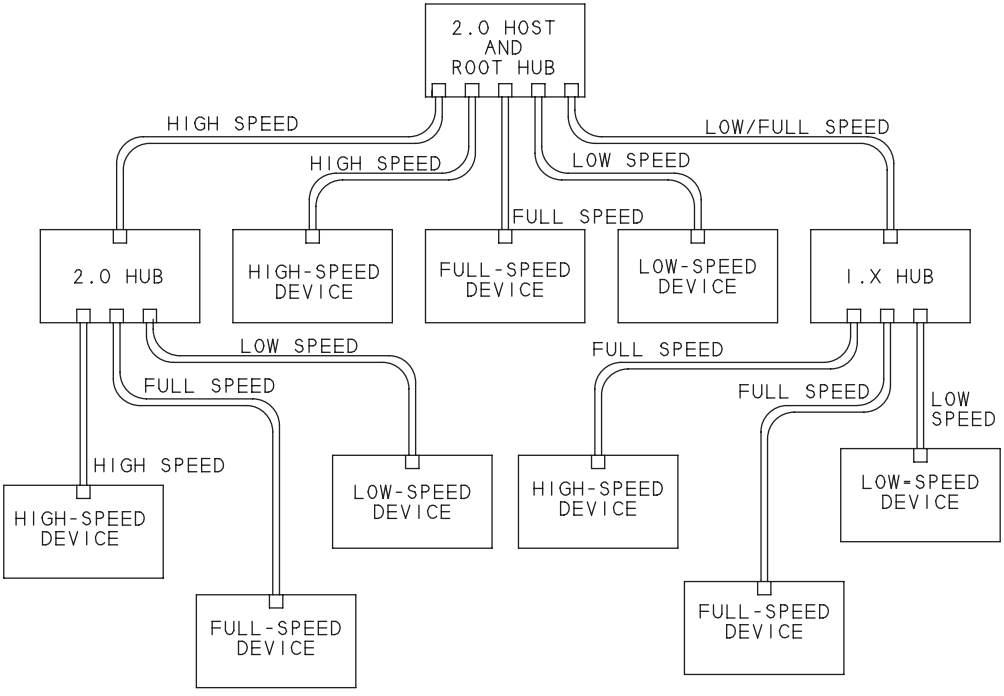

Figure 2-2: A USB 2.0 hub uses high speed whenever possible, switching to low and full speeds when necessary.

Device Endpoints: the Source and Sink of Data

All bus traffic travels to or from a device endpoint. The endpoint is a buffer that stores multiple bytes. Typically the endpoint is a block of data memory or a register in the controller chip. The data stored at an endpoint may be received data or data waiting to transmit. The host also has buffers that hold received data and data waiting to transmit, but the host doesn't have endpoints. Instead, the host serves as the start and finish for communications with device endpoints.

The USB specification defines a device endpoint as "a uniquely addressable portion of a USB device that is the source or sink of information in a communication flow between the host and device." This definition suggests that an endpoint carries data in one direction only. However, as I'll explain, a control endpoint is a special case that is bidirectional.

An endpoint's address consists of an endpoint number and direction. The number is a value from 0 to 15. The direction is defined from the host's perspective: an IN endpoint providess data to send to the host and an OUT endpoint stores data received from the host. An endpoint configured for control transfers must transfer data in both directions, so a control endpoint actually consists of a pair of IN and OUT endpoint addresses that share an endpoint number.

Every device must have Endpoint 0 configured as a control endpoint. There's rarely a need for additional control endpoints. Some controller chips support them, however.

Other types of transfers send data in one direction only, though status and control information may flow in the opposite direction. A single endpoint number can support both IN and OUT endpoint addresses. For example, a device might have an Endpoint 1 IN endpoint address for sending data to the host and an Endpoint 1 OUT endpoint address for receiving data from the host.

In addition to Endpoint 0, a full- or high-speed device can have up to 30 additional endpoint addresses (1 through 15, with each supporting both IN and OUT transfers). A low-speed device is limited to two additional end-

point addresses in any combination of directions (for example, Endpoint 1 IN and Endpoint 1 OUT or Endpoint 1 IN and Endpoint 2 IN).

Every transaction on the bus begins with a packet that contains an endpoint number and a code that indicates the direction of data flow and whether or not the transaction is initiating a control transfer. The codes are IN, OUT, and Setup:

Transaction Type	Source of Data	Types of Transfers that Use this Transaction Type	Contents
IN	device	all	data or status information
OUT	host	all	data or status information
Setup	host	control	a request

As with the endpoint directions, the naming convention for IN and OUT transactions is from the perspective of the host. In an IN transaction, data travels from the device to the host. In an OUT transaction, data travels from the host to the device.

A Setup transaction is like an OUT transaction because data travels from the host to the device, but a Setup transaction is a special case because it initiates a control transfer. Devices need to identify Setup transactions because these are the only type of transactions that devices must always accept and because the device needs to identify and respond to the request contained in the received data. Any transfer type may use IN or OUT transactions.

Each transaction contains a device address and an endpoint address. When a device receives an OUT or Setup packet containing the device's address, the endpoint stores the data that follows the OUT or Setup packet and the hardware typically triggers an interrupt. An interrupt-service routine in the device can then process the received data and take any other required action. When a device receives an IN packet containing its device address, if the device has data ready to send to the host, the hardware sends the data from the specified endpoint onto the bus and typically triggers an interrupt. An

interrupt-service routine in the device can then do whatever is needed to get ready for the next IN transaction.

Pipes: Connecting Endpoints to the Host

Before a transfer can occur, the host and device must establish a pipe. A USB pipe is an association between a device's endpoint and the host controller's software.

The host establishes pipes during enumeration. If the device is removed from the bus, the host removes the no-longer-needed pipes. The host may also request new pipes or remove unneeded pipes at other times by requesting an alternate configuration or interface for a device. Every device has a Default Control Pipe that uses Endpoint 0.

The configuration information received by the host includes an endpoint descriptor for each endpoint that the device wants to use. Each endpoint descriptor is a block of information that tells the host what it needs to know about the endpoint in order to communicate with it. The information includes the endpoint address, the type of transfer to use, the maximum size of data packets, and, when appropriate, the desired interval for transfers.

Types of Transfers

USB is designed to handle many types of peripherals with varying requirements for transfer rate, response time, and error correcting. The four types of data transfers each handle different needs, and a device can support the transfer types that are best suited for its purpose. Table 2-1 summarizes the features and uses of each transfer type.

Control transfers are the only type that have functions defined by the USB specification. Control transfers enable the host to read information about a device, set a device's address, and select configurations and other settings. Control transfers may also send vendor-specific requests that send and receive data for any purpose. All USB devices must support control transfers.

Table 2-1: Each of the USB's four transfer types is suited for different uses.

Transfer Type	Control	Bulk	Interrupt	Isochronous
Typical Use	Identification and configuration	Printer, scanner, drive	Mouse, keyboard	Streaming audio, video
Required?	yes	no	no	no
Low speed allowed?	yes	no	yes	no
Data bytes/millisecond per transfer, maximum possible per pipe (high speed).*	15,872 (thirty-one 64-byte transactions/ microframe)	53,248 (thirteen 512-byte transactions/ microframe)	24,576 (three 1024-byte transactions/ microframe)	24,576 (three 1024-byte transactions/ microframe)
Data bytes/millisecond per transfer, maximum possible per pipe (full speed).*	832 (thirteen 64-byte transactions/ frame)	1216 (nineteen 64-byte transactions/ frame)	64 (one 64-byte transaction/ frame)	1023 (one 1023-byte transaction/ frame)
Data bytes/millisecond per transfer, maximum possible per pipe (low speed).*	24 (three 8-byte transactions)	not allowed	0.8 (8 bytes per 10 milliseconds)	not allowed
Direction of data flow	IN and OUT	IN or OUT	IN or OUT (USB 1.0 supports IN only)	IN or OUT
Reserved bandwidth for all transfers of the type (percent)	10 at low/full speed, 20 at high speed (minimum)	none	90 at low/full speed, 80 at high speed (isochronous & interrupt combined, maximum)	
Error correction?	yes	yes	yes	no
Message or Stream data?	message	stream	stream	stream
Guaranteed delivery rate?	no	no	no	yes
Guaranteed latency (maximum time between transfers)?	no	no	yes	yes
*Assumes transfers use maximum packet size.				

Bulk transfers are intended for situations where the rate of transfer isn't critical, such as sending a file to a printer, receiving data from a scanner, or accessing files on a drive. For these applications, quick transfers are nice but the data can wait if necessary. If the bus is very busy, bulk transfers are delayed, but if the bus is otherwise idle, bulk transfers are very fast. Only

full- and high-speed devices can do bulk transfers. Devices aren't required to support bulk transfers, but a specific device class might require them.

Interrupt transfers are for devices that must receive the host's or device's attention periodically. Other than control transfers, interrupt transfers are the only way that low-speed devices can transfer data. Keyboards and mice use interrupt transfers to send keypress and mouse-movement data. Interrupt transfers can use any speed. Devices aren't required to support interrupt transfers, but a specific device class might require them.

Isochronous transfers have guaranteed delivery time but no error correcting. Data that might use isochronous transfers incudes audio or video to be played in real time. Isochronous is the only transfer type that doesn't support automatic re-transmitting of data received with errors, so occasional errors must be acceptable. Only full- and high-speed devices can do isochronous transfers. Devices aren't required to support isochronous transfers, but a specific device class might require them.

Stream and Message Pipes

In addition to classifying a pipe by the type of transfer it carries, the USB specification defines pipes as either stream or message, according to whether or not information travels in one or both directions. Control transfers use bidirectional message pipes; all other transfer types use unidirectional stream pipes.

Control Transfers Use Message Pipes

In a message pipe, each transfer begins with a Setup transaction containing a request. To complete the transfer, the host and device may exchange data and status information, or the device may just send status information. Each control transfer has at least one transaction that sends information in each direction.

If a device supports a received request, the device takes the requested action. If a device doesn't support the request, the device responds with a code to indicate that the request isn't supported.

All Other Transfers Use Stream Pipes

In a stream pipe, the data has no structure defined by the USB specification. The receiving host or device just accepts whatever arrives. The device firmware or host software can then process the data in whatever way is appropriate for the application.

Of course, even with stream data, the sending and receiving devices will need to agree on a format of some type. For example, a host application may define a code that requests a device to send a series of bytes indicating a temperature reading and the time of the reading. Although the host could use control transfers with a vendor-defined Get_Temperature request, interrupt transfers may be preferable because of their guaranteed bandwidth.

Initiating a Transfer

When a device driver in the host wants to communicate with a device, the driver initiates a transfer. The USB specification defines a transfer as the process of making and carrying out a communication request. A transfer may be very short, sending as little as a byte of application data, or very long, sending the contents of a large file.

A Windows application can open communications with a device using a handle retrieved using standard API functions. To begin a transfer, an application may use the handle in calling an API function to request the transfer from the device's driver. An application might request to "send the contents of the file *data.txt* to the device" or "get the contents of Input Report 1 from the device." When an application requests a transfer, the operating system passes the request to the appropriate device driver, which in turn passes the request to other system-level drivers and on to the host controller. The host controller then initiates the transfer on the bus.

For devices in standard classes, a programming language can provide alternate ways to access a device. For example, the .NET Framework includes Directory and File classes for accessing files on drives. A vendor-supplied driver can also define its own API functions. For example, devices that use controllers from FTDI Chip can use FTDI's D2XX driver, which exposes a

series of functions for setting communications parameters and exchanging data.

In some cases, a driver is configured to request periodic transfers, and applications can read the retrieved data or provide data to send in these transfers. During enumeration, the operating system initiates transfers. Other transfers require an application to request to send or receive data.

Transactions: the Building Blocks of a Transfer

Figure 2-3 shows the elements of a typical transfer. A lot of the terminology here begins to sound the same. There are transfers and transactions, stages and phases, data transactions and data packets, Status stages and handshake

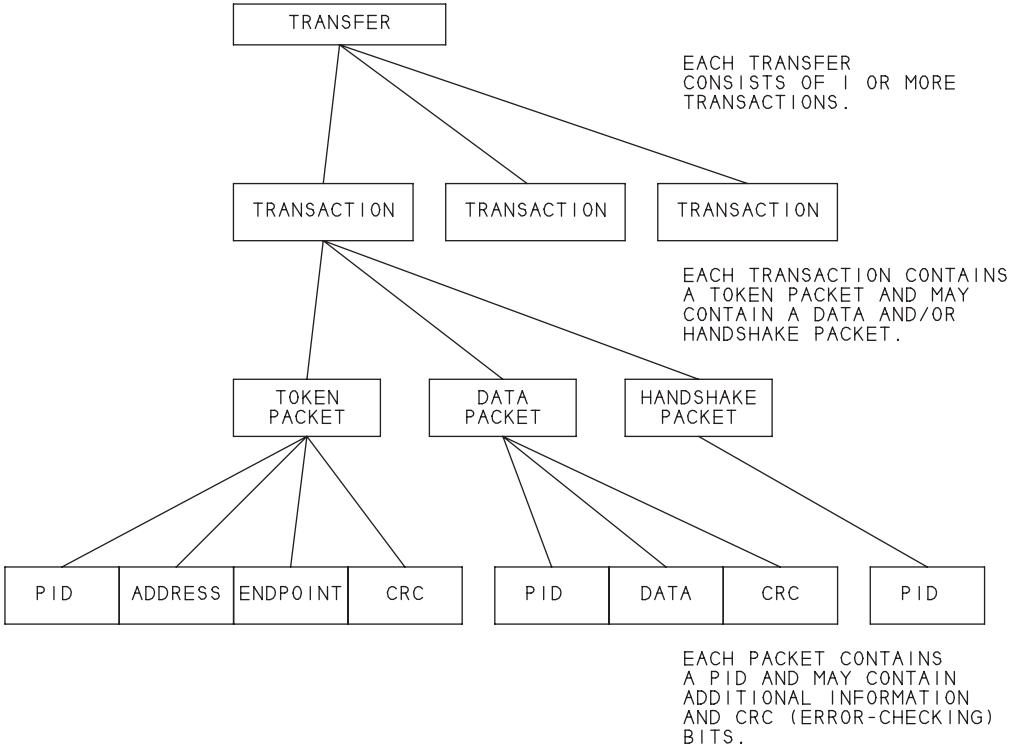

Figure 2-3: A USB transfer consists of transactions. The transactions in turn contain packets, and the packets contain a packet identifier (PID), PID-check bits, and sometimes additional information.

phases. Data stages have handshake packets and Status stages have data packets. It can take a while to absorb it all. Table 2-2 lists the elements that make up each of the four transfer types and may help in keeping the terms straight.

Each transfer consists of one or more transactions, and each transaction in turn consists of one, two, or three packets.

The three transaction types are defined by their purpose and direction of data flow. Setup transactions send control-transfer requests to a device. OUT transactions send other data or status information to the device. IN transactions send data or status information to the host.

The USB specification defines a transaction as the delivery of service to an endpoint. *Service* in this case can mean either the host's sending information to the device, or the host's requesting and receiving information from the device.

Each transaction includes identifying, error-checking, status, and control information as well as any data to be exchanged. A complete transfer may take place over multiple frames or microframes, but a transaction must complete uninterrupted. No other communication on the bus can break into the middle of a transaction. Devices thus must be able to respond quickly with requested data or status information in a transaction. Device firmware typically configures, or arms, an endpoint to respond to received packets, and the hardware responds to the packets when they arrive.

A transfer with a small amount of data may require just one transaction. Other transfers require multiple transactions with a portion of the data in each.

Transaction Phases

Each transaction has up to three phases, or parts that occur in sequence: token, data, and handshake. Each phase consists of one or two transmitted packets. Each packet is a block of information with a defined format. All packets begin with a Packet ID (PID) that contains identifying information (shown in Table 2-3). Depending on the transaction, the PID may be fol-

Table 2-2: Each of the four transfer types consists of one or more transactions, with each transaction containing two or three phases. (This table doesn't show the additional transactions required for the split transactions and PING protocol used in some transfers.)

Transfer Type		**Transactions**	**Phases (packets). Each downstream, low-speed packet is also preceded by a PRE packet.**
Control	Setup Stage	One transaction	Token
			Data
			Handshake
	Data Stage	Zero or more transactions (IN or OUT)	Token
			Data
			Handshake
	Status Stage	One transaction (opposite direction of transaction(s) in the Data stage or IN if there is no Data stage)	Token
			Data
			Handshake
Bulk		One or more transactions (IN or OUT)	Token
			Data
			Handshake
Interrupt		One or more transactions (IN or OUT)	Token
			Data
			Handshake
Isochronous		One or more transactions (IN or OUT)	Token
			Data

lowed by an endpoint address, data, status information, or a frame number, along with error-checking bits.

In the token phase of a transaction, the host initiates a communication by sending a token packet. The PID indicates the transaction type, such as Setup, IN, OUT, or Start-of-Frame.

In the data phase, the host or device may transfer any kind of information in a data packet. The PID includes a data-toggle or data-sequencing value used to guard against lost or duplicated data when a transfer has multiple data packets.

Table 2-3: The PID (packet identifier) provides information about a transaction. (Sheet 1 of 2)

Packet Type	PID Name	Value	Transfer types used in	Source	Bus Speed	Description
Token (identifies transaction type)	OUT	0001	all	host	all	Device and endpoint address for OUT (host-to-device) transaction.
	IN	1001	all	host	all	Device and endpoint address for IN (device-to-host) transaction.
	SOF	0101	Start-of-Frame	host	all	Start-of-Frame marker and frame number.
	SETUP	1101	control	host	all	Device and endpoint address for Setup transaction.
Data (carries data or status code)	DATA0	0011	all	host, device	all	Data toggle, data PID sequencing
	DATA1	1011	all	host, device	all	Data toggle, data PID sequencing
	DATA2	0111	isoch.	host, device	high	Data PID sequencing
	MDATA	1111	isoch., split transac-tions	host, device	high	Data PID sequencing
Handshake (carries status code)	ACK	0010	all	host, device	all	Receiver accepts error-free data packet.
	NAK	1010	control, bulk, interrupt	device	all	Receiver can't accept data or sender can't send data or has no data to transmit.
	STALL	1110	control, bulk, interrupt	device	all	A control request isn't sup-ported or the endpoint is halted.
	NYET	0110	control Write, bulk OUT, split transac-tions	device	high	Device accepts error-free data packet but isn't yet ready for another or a hub doesn't yet have complete-split data.

Table 2-3: The PID (packet identifier) provides information about a transaction. (Sheet 2 of 2)

Packet Type	PID Name	Value	Transfer types used in	Source	Bus Speed	Description
Special	PRE	1100	control, interrupt	host	full	Preamble issued by host to indicate that the next packet is low speed.
	ERR	1100	all	hub	high	Returned by a hub to report a low- or full-speed error in a split transaction.
	SPLIT	1000	all	host	high	Precedes a token packet to indicate a split transaction.
	PING	0100	control Write, bulk OUT	host	high	Busy check for bulk OUT and control Write data transactions after NYET.
	reserved	0000	–	–	–	For future use.

In the handshake phase, the host or device sends status information in a handshake packet. The PID contains a status code (ACK, NAK, STALL, or NYET). The USB specification sometimes uses the terms *status phase* and *status packet* to refer to the handshake phase and packet.

The token phase has one additional use. A token packet may carry a Start-of-Frame (SOF) marker, which is a timing reference that the host sends at 1-millisecond intervals at full speed and at 125-microsecond intervals at high speed. This packet also contains a frame number that increments and rolls over on reaching the maximum. The number indicates the frame count, so the eight microframes within a frame all have the same number. An endpoint may synchronize to the Start-of-Frame packet or use the frame count as a timing reference. The Start-of-Frame marker also keeps devices from entering the low-power Suspend state when there is no other USB traffic.

Low-speed devices don't see the SOF packet. Instead, the hub that the device attaches to uses a simpler End-of-Packet (EOP) signal called the low-speed keep-alive signal, sent once per frame. As the SOF does for

full-speed devices, the low-speed keep-alive keeps low-speed devices from entering the Suspend state.

Of the four special PIDs, one is used only with low-speed devices, one is used only with high-speed devices, and two are used when a low- or full-speed device's 2.0 hub communicates at high speed with the host.

The special low-speed PID is PRE, which contains a preamble code that tells hubs that the next packet is low speed. On receiving a PRE PID, the hub enables communications with any attached low-speed devices. On a low- and full-speed bus, the PRE PID precedes all token, data, and handshake packets directed to low-speed devices. High-speed buses encode the PRE in the SPLIT packet, rather than sending the PRE separately. Low-speed packets sent by a device don't require a PRE PID.

The PID used only with high-speed devices is PING. In a bulk or control transfer with multiple data packets, before sending the second and any subsequent data packets, the host may send a PING to find out if the endpoint is ready to receive more data. The device responds with a status code.

The SPLIT PID identifies a token packet as part of a split transaction, as explained later in this chapter. The ERR PID is used only in split transactions. A 2.0 hub uses this PID to report an error to the host in a low- or full-speed transaction. The ERR and PRE PIDs have the same value but won't be confused because a hub never sends a PRE to the host or an ERR to a device. Also, ERR is used only on high-speed segments and PRE is never used on high-speed segments.

Packet Sequences

Every transaction has a token packet. The host is always the source of this packet, which sets up the transaction by identifying the packet type, the receiving device and endpoint, and the direction of any data the transaction will transfer. For low-speed transactions on a full-speed bus, a PRE packet precedes the token packet. For split transactions, a SPLIT packet precedes the token packet.

Depending on the transfer type and whether the host or device has information to send, a data packet may follow the token packet. The direction spec-

ified in the token packet determines whether the host or device sends the data packet.

In all transfer types except isochronous, the receiver of the data packet (or the device if there is no data packet) returns a handshake packet containing a code that indicates the success or failure of the transaction. The absence of an expected handshake packet indicates a more serious error.

Timing Constraints and Guarantees

The allowed delays between the token, data, and handshake packets of a transaction are very short, intended to allow only for cable delays and switching times plus a brief time to allow the hardware to prepare a response, such as a status code, in response to a received packet.

A common mistake in writing firmware is to assume that the firmware should wait for an interrupt before providing data to send to the host. Instead, before the host requests the data, the firmware must copy the data to send into the endpoint's buffer and configure the endpoint to send the data on receiving an IN token packet. The interrupt occurs after the transaction completes, to tell the firmware that the endpoint's buffer can store data for the next transaction. If the firmware waits for an interrupt before providing the initial data, the interrupt never happens and no data is transferred.

A single transaction can carry an amount of data up to the maximum packet size specified for the endpoint. A data packet that is less than the maximum packet size is a *short packet*. A transfer with multiple transactions may take place over multiple frames or microframes, which don't have to be contiguous. For example, in a full-speed bulk transfer of 512 bytes, the maximum number of bytes in a single transaction is 64, so transferring all of the data requires at least 8 transactions, which may occur in one or more (micro)frames.

Split Transactions

A 2.0 hub communicates with a 2.0 host at high speed unless a 1.x hub lies between them. When a low- or full-speed device is attached to a 2.0 hub, the hub converts between speeds as needed. But speed conversion isn't the

only thing a hub does to manage multiple speeds. High speed is 40 times faster than full speed and 320 times faster than low speed. It doesn't make sense for the entire bus to wait while a hub exchanges low- or full-speed data with a device.

The solution is split transactions. A 2.0 host uses split transactions when communicating with a low- or full-speed device on a high-speed bus. What would be a single transaction at low or full speed usually requires two types of split transactions: one or more start-split transactions to send information to the device and one or more complete-split transactions to receive information from the device. The exception is isochronous OUT transactions, which don't use complete-split transactions because the device has nothing to send.

Even though they require more transactions, split transactions make better use of bus time because they minimize the amount of time spent waiting for a low- or full-speed device to transfer data. The USB 2.0 host controller and the closest 2.0 hub upstream from the low- or full-speed device are entirely responsible for performing split transactions. The device and its firmware don't have to know or care whether the host is using split transactions. The transactions at the device are identical whether the host is using split transactions or not. At the host, device drivers and application software don't have to know or care whether the host is using split transactions because the protocol is handled at a lower level. Chapter 15 has more about how split transactions work.

Ensuring that Transfers Are Successful

USB transfers use handshaking and error-checking to help ensure that data gets to its destination as quickly as possible and without errors.

Handshaking

Like other interfaces, USB uses status and control, or handshaking, information to help to manage the flow of data. In hardware handshaking, dedicated lines carry the handshaking information. An example is the RTS and

CTS lines in the RS-232 interface. In software handshaking, the same lines that carry the data also carry handshaking codes. An example is the XON and XOFF codes transmitted on the data lines in RS-232 links.

USB uses software handshaking. A code indicates the success or failure of all transactions except in isochronous transfers. In addition, in control transfers, the Status stage enables a device to report the success or failure of an entire transfer.

Handshaking signals transmit in the handshake or data packet. The defined status codes are ACK, NAK, STALL, NYET, and ERR. The absence of an expected handshake code indicates a more serious error. In all cases, the expected receiver of the handshake uses the information to help decide what to do next. Table 2-4 shows the status indicators and where they transmit in each transaction type.

ACK

ACK (acknowledge) indicates that a host or device has received data without error. Devices must return ACK in the handshake packets of Setup transactions when the token and data packets were received without error. Devices may also return ACK in the handshake packets of OUT transactions. The host returns ACK in the handshake packets of IN transactions.

NAK

NAK (negative acknowledge) means the device is busy or has no data to return. If the host sends data at a time when the device is too busy to accept the data, the device returns a NAK in the handshake packet. If the host requests data from the device when the device has nothing to send, the device returns a NAK in the data packet. In either case, NAK indicates a temporary condition, and the host retries later.

Hosts never send NAK. Isochronous transactions don't use NAK because they have no handshake packet for returning a NAK. If a device or the host doesn't receive transmitted isochronous data, it's gone.

Table 2-4: The location, source, and contents of the handshake signal depend on the type of transaction.

Transaction type or PING query	Data packet source	Data packet contents	Handshake packet source	Handshake packet contents
Setup	host	data	device	ACK
OUT	host	data	device	ACK, NAK, STALL, NYET (high speed only), ERR (from hub in complete split)
IN	device	data, NAK, STALL, ERR (from hub in complete split)	host	ACK
PING (high speed only)	none	none	device	ACK, NAK, STALL

STALL

The STALL handshake can have any of three meanings: unsupported control request, control request failed, or endpoint failed.

When a device receives a control-transfer request that the device doesn't support, the device returns a STALL to the host. The device also returns a STALL if the device supports the request but for some reason can't take the requested action. For example, if the host sends a Set_Configuration request that requests the device to set its configuration to 2, and the device supports only configuration 1, the device returns a STALL. To clear this type of stall, the host just needs to send another Setup packet to begin a new control transfer. The USB specification calls this type of stall a protocol stall.

Another use of STALL is to respond when the endpoint's Halt feature is set, which means that the endpoint is unable to send or receive data at all. The USB specification calls this type of stall a functional stall.

Bulk and interrupt endpoints must support the functional stall. Although control endpoints may also support this use of STALL, it's not recommended. A control endpoint in a functional stall must continue to respond normally to other requests related to controlling and monitoring the stall condition. And an endpoint that is capable of responding to these requests is clearly capable of communicating and shouldn't be stalled. Isochronous transactions don't use STALL because they have no handshake packet for returning the STALL.

On receiving a functional STALL, the host drops all pending requests to the device and doesn't resume communications until the host has sent a successful request to clear the Halt feature on the device. Hosts never send STALL.

NYET

Only high-speed devices use NYET, which stands for *not yet*. High-speed bulk and control transfers have a protocol that enables the host to find out before sending data if a device is ready to receive the data. At full and low speeds, when the host wants to send data in a control, bulk, or interrupt transfer, the host sends the token and data packets and receives a reply from the device in the handshake packet of the transaction. If the device isn't ready for the data, the device returns a NAK and the host tries again later. This can waste a lot of bus time if the data packets are large and the device is often not ready.

High-speed bulk and control transactions with multiple data packets have a better way. After receiving a data packet, a device endpoint can return a NYET handshake, which says that the endpoint accepted the data but is not yet ready to receive another data packet. When the host thinks the device might be ready, the host can send a PING token packet, and the endpoint returns either an ACK to indicate the device is ready for the next data packet or NAK or STALL if the device isn't ready. Sending a PING is more efficient than sending the entire data packet only to find out the device wasn't ready and having to resend later. Even after responding to a PING or OUT with ACK, an endpoint is allowed to return NAK on receiving the data packet that follows but should do so rarely. The host then tries again with another PING. The use of PING by the host is optional.

A 2.0 hub may also use NYET in complete-split transactions. Hosts and low- and full-speed devices never send NYET.

ERR

The ERR handshake is used only by high-speed hubs in complete-split transactions. ERR indicates the device didn't return an expected handshake in the transaction the hub is completing with the host.

No Response

Another type of status indication occurs when the host or a device expects to receive a handshake but receives nothing. This lack of response can occur if the receiver's error-checking calculation detected an error. On receiving no response, the sender knows that it should try again. If if multiple tries fail, the sender can take other action. (If the receiver ACKs the data but doesn't use it, the problem is probably in the data-toggle value.)

Reporting the Status of Control Transfers

In addition to reporting the status of transactions, the same ACK, NAK, and STALL codes report the success or failure of complete control transfers. An additional status code is a zero-length data packet (ZLP), which reports successful completion of a control transfer. A transaction with a zero-length data packet is a transaction whose Data phase consists of a Data PID and error-checking bits but no data. Table 2-5 shows the status indicators for control transfers.

For control Write transfers, where the device receives data in the Data stage, the device returns the transfer's status in the data packet of the Status stage. A zero-length data packet means the transfer was successful, a STALL indicates that the device can't complete the transfer, and a NAK indicates that the device isn't ready to complete the transfer. The host returns an ACK in the handshake packet of the Status stage to indicate that the host received the response.

For control Read transfers, where the host receives data in the Data stage, the device returns the status of the transfer in the handshake packet of the

Table 2-5: Depending on the direction of the Data stage, the status information for a control transfer may be in the data or handshake packet of the Status stage.

Transfer Type and Direction	Status Stage Direction	Status stage's data packet	Status stage's handshake packet
Control Write (Host sends data to device or no Data stage)	IN	Device sends status: zero-length data packet (success), NAK (busy), or STALL (failed)	Host returns ACK
Control Read (Device sends data to host)	OUT	Host sends zero-length data packet	Device sends status: ACK (success), NAK (busy), or STALL (failed)

Status stage. The host normally waits to receive all of the packets in the Data stage, then returns a zero-length data packet in the Status stage. The device responds with ACK, NAK, or STALL. However, if the host begins the Status stage before all of the requested data packets have been sent, the device must abandon the Data stage and return a status code.

Error Checking

The USB specification spells out hardware requirements that ensure that errors due to line noise will be rare. Still, there is a chance that a noise glitch or unexpectedly disconnected cable could corrupt a transmission. For this reason, USB packets include error-checking bits that enable a receiver to identify just about any received data that doesn't match what was sent. In addition, for transfers that require multiple transactions, a data-toggle value keeps the transmitter and receiver synchronized to ensure that no transactions are missed entirely.

Error-checking Bits

All token, data, and Start-of-Frame packets include bits for use in error-checking. The bit values are calculated using a mathematical algorithm called the cyclic redundancy check (CRC). The USB specification explains

how the CRC is calculated. The hardware handles the calculations, which must be done quickly to enable the device to respond appropriately.

The CRC is applied to the data to be checked. The transmitting device performs the calculation and sends the result along with the data. The receiving device performs the identical calculation on the received data. If the results match, the data has arrived without error and the receiving device returns an ACK. If the results don't match, the receiving device sends no handshake. The absence of the expected handshake tells the sender to retry.

Typically, the host tries a total of three times, but the USB specification gives the host some flexibility in determining the number of retries. On giving up, the host informs the driver of the problem.

The PID field in token packets uses a simpler form of error checking. The lower four bits in the field are the PID, and the upper four bits are its complement. The receiver can check the integrity of the PID by complementing the upper four bits and ensuring that they match the PID. If not, the packet is corrupted and is ignored.

The Data Toggle

In transfers that require multiple transactions, the data-toggle value can ensure that no transactions are missed by keeping the transmitting and receiving devices synchronized. The data-toggle value is included in the PID field of the token packets for IN and OUT transactions. DATA0 is a code of 0011, and DATA1 is 1011. In controller chips, a register bit often indicates the data-toggle state, so the data-toggle value is often referred to as the data-toggle bit. Each endpoint maintains its own data toggle.

Both the sender and receiver keep track of the data toggle. A Windows host handles the data toggles without requiring any user programming. Some device controller chips also handle the data toggles completely automatically, while others require some firmware control. If you're debugging a device where it appears that the proper data is transmitting on the bus but the receiver is discarding the data, chances are good that the device isn't sending or expecting the correct data toggle.

When the host configures a device on power up or attachment, the host and device each set their data toggles to DATA0 for all except some high-speed isochronous endpoints. On detecting an incoming data packet, the host or device compares the state of its data toggle with the received data toggle. If the values match, the receiver toggles its value and returns an ACK handshake packet to the sender. The ACK causes the sender to toggle its value for the next transaction.

The next received packet in the transfer should contain a data toggle of DATA1, and again the receiver toggles its bit and returns an ACK. The data toggle continues to alternate until the transfer completes. (An exception is control transfers, where the Status stage always uses DATA1.)

If the receiver is busy and returns a NAK, or if the receiver detects corrupted data and returns no response, the sender doesn't toggle its bit and instead tries again with the same data and data toggle.

If a receiver returns an ACK but for some reason the sender doesn't see the ACK, the sender will think that the receiver didn't get the data and will try again using the same data and data-toggle bit. In this case, the receiver of the repeated data doesn't toggle the data toggle and ignores the data but returns an ACK. The ACK re-synchronizes the data toggles. The same thing happens if the sender mistakenly sends the same data toggle twice in a row.

Control transfers always use DATA0 in the Setup stage, use DATA1 in the first transaction of the Data stage, toggle the bit in any additional Data-stage transactions, and use DATA1 in the Status stage. Bulk endpoints toggle the bit in every transaction, resetting the data toggle only after completing a Set_Configuration, Set_Interface, or Clear_Feature(ENDPOINT HALT) request. Interrupt endpoints can behave the same as bulk endpoints. Or an interrupt IN endpoint can toggle its data toggle in each transaction without checking for the host's ACKs, at the risk of losing some data. Full-speed isochronous transfers always use DATA0. Isochronous transfers can't use the data toggle to correct errors because there is no handshake packet for returning an ACK or NAK and no time to resend missed data.

Some high-speed isochronous transfers use DATA0, DATA1, and additional PIDs of DATA2 and MDATA. High-speed isochronous IN transfers that

have two or three transactions per microframe use DATA0, DATA1, and DATA2 encoding to indicate a transaction's position in the microframe:

Number of IN Transactions in the Microframe	Data PID		
	First Transaction	Second Transaction	Third Transaction
1	DATA0	–	–
2	DATA1	DATA0	–
3	DATA2	DATA1	DATA0

High-speed isochronous OUT transfers that have two or three transactions per microframe use DATA0, DATA1, and MDATA encoding to indicate whether more data will follow in the microframe:

Number of OUT Transactions in the Microframe	Data PID:		
	First Transaction	Second Transaction	Third Transaction
1	DATA0	–	–
2	MDATA	DATA1	–
3	MDATA	MDATA	DATA2

This use of the data toggle and other PIDs is called data PID sequencing.