3

A Transfer Type for Every Purpose

This chapter takes a closer look at USB's four transfer types: control, bulk, interrupt, and isochronous. Each transfer type has abilities and limits that make the transfers suitable for different purposes. Table 3-1 compares the amount of data that each transfer type can move at each of the three speeds.

Control Transfers

Control transfers have two uses. Control transfers carry the requests that are defined by the USB specification and used by the host to learn about and configure devices. Control transfers can also carry requests defined by a class or vendor for any purpose.

Table 3-1: The maximum possible rate of data transfer varies greatly with the transfer type and bus speed.

Transfer Type	**Maximum data-transfer rate per endpoint (kilobytes/sec. with data payload/transfer = maximum packet size allowed for the speed)**		
	Low Speed	**Full Speed**	**High Speed**
Control	24	832	15,872
Interrupt	0.8	64	24,576
Bulk	not allowed	1216	53,248
Isochronous		1023	24,576

Availability

Every device must support control transfers over the default pipe at Endpoint 0. A device may also have additional pipes for control transfers, but in reality there's no need for more than one. Even if a device needs to send a lot of control requests, hosts allocate bandwidth for control transfers according to the number and size of requests, not by the number of control endpoints, so additional control endpoints offer no advantage.

Structure

Chapter 2 introduced control transfers and their stages: Setup, Data (optional), and Status. Each stage consists of one or more transactions.

Every control transfer must have a Setup stage and a Status stage. The Data stage is optional, though a particular request may require a Data stage. Because every control transfer requires transferring information in both directions, the control transfer's message pipe uses both the IN and OUT endpoint addresses.

In a control Write transfer, the data in the Data stage travels from the host to the device. Control transfers that have no Data stage are also considered to be control Write transfers. In a control Read transfer, the data in the Data stage travels from the device to the host. Figure 3-1 and Figure 3-2 show the stages of control Read and control Write transfers at low and full speeds on a low/full-speed bus. There are differences, described later in this chapter, for

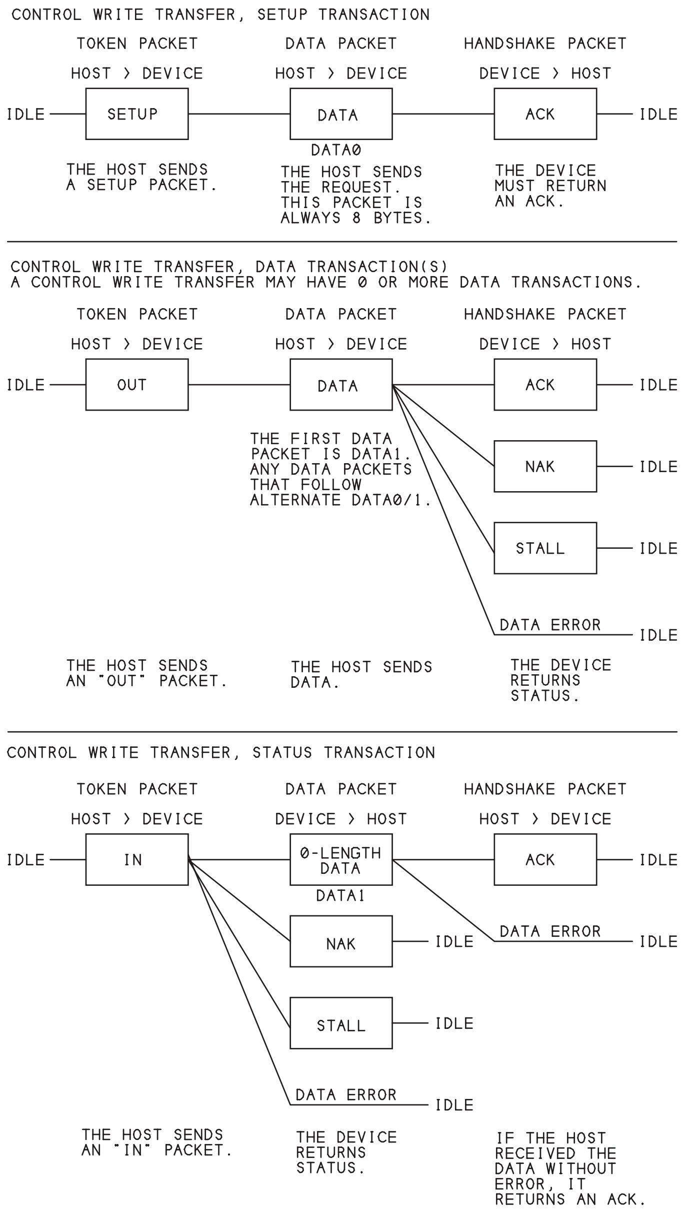
CONTROL WRITE TRANSFER, SETUP TRANSACTION
TOKEN PACKET
DATA PACKET
HANDSHAKE PACKET
HOST > DEVICE
HOST > DEVICE
DEVICE > HOST
IDLE
SETUP
DATA
ACK
IDLE
DATA0
THE HOST SENDS A SETUP PACKET.
THE HOST SENDS THE REQUEST. THIS PACKET IS ALWAYS 8 BYTES.
THE DEVICE MUST RETURN AN ACK.
CONTROL WRITE TRANSFER, DATA TRANSACTION(S)
A CONTROL WRITE TRANSFER MAY HAVE 0 OR MORE DATA TRANSACTIONS.
TOKEN PACKET
DATA PACKET
HANDSHAKE PACKET
HOST > DEVICE
HOST > DEVICE
DEVICE > HOST
IDLE
OUT
DATA
ACK
IDLE
NAK
IDLE
STALL
IDLE
DATA ERROR
IDLE
THE FIRST DATA PACKET IS DATA1. ANY DATA PACKETS THAT FOLLOW ALTERNATE DATA0/1.
THE HOST SENDS AN "OUT" PACKET.
THE HOST SENDS DATA.
THE DEVICE RETURNS STATUS.
CONTROL WRITE TRANSFER, STATUS TRANSACTION
TOKEN PACKET
DATA PACKET
HANDSHAKE PACKET
HOST > DEVICE
DEVICE > HOST
HOST > DEVICE
IDLE
IN
0-LENGTH DATA
ACK
IDLE
DATA1
NAK
IDLE
DATA ERROR
IDLE
STALL
IDLE
DATA ERROR
IDLE
THE HOST SENDS AN "IN" PACKET.
THE DEVICE RETURNS STATUS.
IF THE HOST RECEIVED THE DATA WITHOUT ERROR, IT RETURNS AN ACK.

Figure 3-1: A control Write transfer contains a Setup transaction, zero or more Data transactions, and a Status transaction. Not shown are the PING protocol used in some high-speed transfers with multiple data packets and the split transactions used with low- and full-speed devices on a high-speed bus.

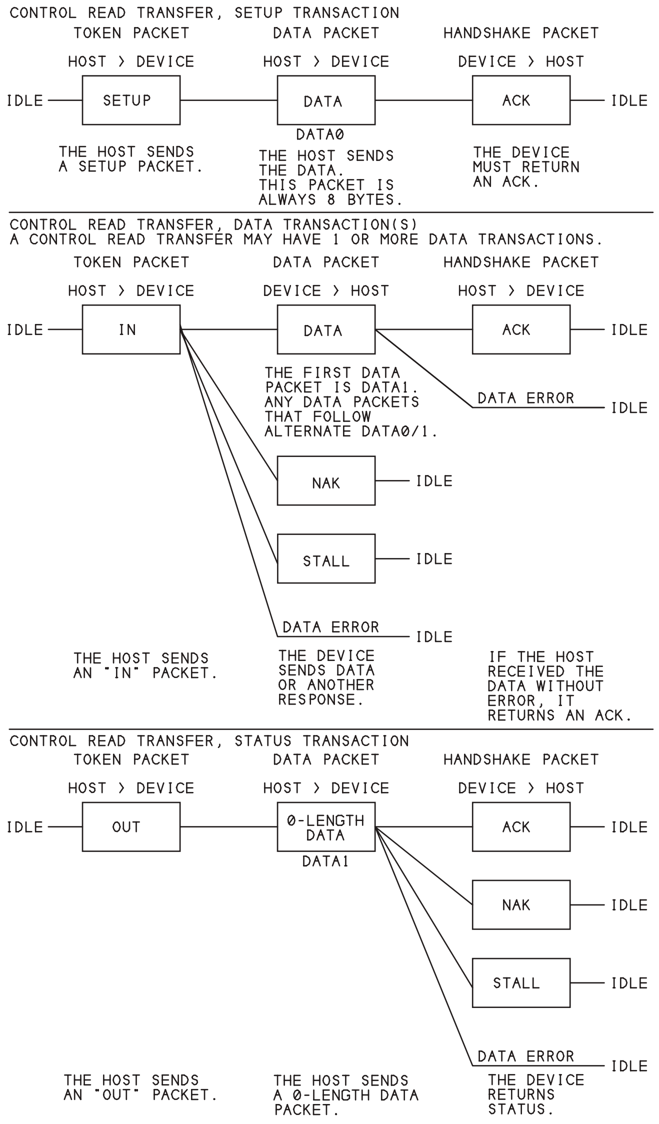
CONTROL READ TRANSFER, SETUP TRANSACTION
TOKEN PACKET
DATA PACKET
HANDSHAKE PACKET
HOST > DEVICE
HOST > DEVICE
DEVICE > HOST
IDLE
SETUP
DATA
ACK
IDLE
DATA0
THE HOST SENDS A SETUP PACKET.
THE HOST SENDS THE DATA. THIS PACKET IS ALWAYS 8 BYTES.
THE DEVICE MUST RETURN AN ACK.
CONTROL READ TRANSFER, DATA TRANSACTION(S)
A CONTROL READ TRANSFER MAY HAVE 1 OR MORE DATA TRANSACTIONS.
TOKEN PACKET
DATA PACKET
HANDSHAKE PACKET
HOST > DEVICE
DEVICE > HOST
HOST > DEVICE
IDLE
IN
DATA
ACK
IDLE
THE FIRST DATA PACKET IS DATA1. ANY DATA PACKETS THAT FOLLOW ALTERNATE DATA0/1.
DATA ERROR
IDLE
NAK
IDLE
STALL
IDLE
DATA ERROR
IDLE
THE HOST SENDS AN "IN" PACKET.
THE DEVICE SENDS DATA OR ANOTHER RESPONSE.
IF THE HOST RECEIVED THE DATA WITHOUT ERROR, IT RETURNS AN ACK.
CONTROL READ TRANSFER, STATUS TRANSACTION
TOKEN PACKET
DATA PACKET
HANDSHAKE PACKET
HOST > DEVICE
HOST > DEVICE
DEVICE > HOST
IDLE
OUT
0-LENGTH DATA
ACK
IDLE
DATA1
NAK
IDLE
STALL
IDLE
DATA ERROR
IDLE
THE HOST SENDS AN "OUT" PACKET.
THE HOST SENDS A 0-LENGTH DATA PACKET.
THE DEVICE RETURNS STATUS.

Figure 3-2: A control Read transfer contains a Setup transaction, one or more data transactions, and a status transaction. Not shown are the split transactions used with low- and full-speed devices on a high-speed bus.

some high-speed transfers and for low- and full-speed transfers with 2.0 hubs on high-speed buses.

In the Setup stage, the host begins a Setup transaction by sending information about the request. The token packet contains a PID that identifies the transfer as a control transfer. The data packet contains information about the request, including the request number, whether or not the transfer has a Data stage, and if so, in which direction the data will travel.

The USB 2.0 specification defines 11 standard requests. Successful enumeration requires specific responses to some requests, such as the request that sets a device's address. For other requests, a device can return a code that indicates that the request isn't supported. A specific class may require a device to support class-specific requests, and any device may support vendor-specific requests defined by a vendor-specific driver.

When present, the Data stage consists of one or more Data transactions, which may be IN or OUT transactions. Depending on the request, the host or peripheral may be the source of these transactions, but all data packets in this stage are in the same direction.

As described in Chapter 2, if a high-speed control Write transfer has more than one data packet in the Data stage, and if the device returns NYET after receiving a data packet, the host may use the PING protocol before sending the next data packet.

The Status stage consists of one IN or OUT transaction, also called the status transaction. In the Status stage, the device reports the success or failure of the previous stages. The source of the Status stage's data packet is the receiver of the data in the Data stage. When there is no Data stage, the device sends the Status stage's data packet. The data or handshake packet sent by the device in the Status stage contains a code that indicates the success or failure of the request.

If a host is performing a control transfer with a low- or full-speed device on a high-speed bus, the host uses the split transactions introduced in Chapter 2 for all of the transfer's transactions. To the device, the transaction is no different than a transaction with a 1.x host. The device's hub carries out the transaction with the device and reports back to the host when requested.

Data Size

The maximum size of the data packet in the Data stage varies with the device's speed. For low-speed devices, the maximum is 8 bytes. For full speed, the maximum may be 8, 16, 32, or 64 bytes. For high speed, the maximum must be 64 bytes. These bytes include only the information transferred in the data packet, excluding the PID and CRC bits.

In the Data stage, all data packets except the last must be the maximum packet size for the endpoint. The maximum packet size for the Default Control Pipe is in the device descriptor that the host retrieves during enumeration. If there are other control endpoints (this is rare), the size is in the endpoint descriptor. If a transfer has more data than will fit in one data transaction, the host sends or receives the data in multiple transactions.

In some control Read transfers, the amount of data returned by the device can vary. If the amount is less than the requested number of bytes and is an even multiple of the endpoint's maximum packet size, the device should indicate when it has no more data to send by returning a zero-length data packet in response to the next IN token packet that arrives after all of the data has been sent.

Speed

The host must make its best effort to ensure that all control transfers get through as quickly as possible. The host controller reserves a portion of the bus bandwidth for control transfers: 10 percent for low- and full-speed buses and 20 percent for high-speed buses. If the control transfers don't need this much time, bulk transfers may use what remains. If the bus has other unused bandwidth, control transfers may use more than the reserved amount.

The host attempts to parcel out the available time as fairly as possible to all devices. For each transfer, a single frame or microframe may contain multiple transactions, or the transactions may be in different (micro)frames.

There are two opinions on whether control transfers are appropriate for transferring data other than enumeration and configuration data. Some say that control transfers should be reserved as much as possible for servicing the standard USB requests and other performing other infrequent configuration tasks. This approach helps to ensure that the transfers complete quickly by keeping the bandwidth reserved for them as open as possible. But the USB specification doesn't forbid other uses for control transfers, and some believe that devices should be free to use control transfers for any purpose. Low-speed devices have no other choice except periodic interrupt transfers, which can waste bandwidth if data transfers are infrequent.

Control transfers aren't the most efficient way to transfer data. In addition to the data being transferred, each transfer with one data packet has an overhead of 63 bytes (low speed), 45 bytes (full speed), or 173 bytes (high speed). Each Data stage requires token and handshake packets, so stages with larger data packets are more efficient.

A single low-speed control transfer with 8 data bytes uses 29% of a frame's bandwidth, though the transfer's individual transactions may be spread among multiple frames. In a control transfer with multiple data packets in the Data stage, the data may travel in the same or different (micro)frames.

If the bus is very busy, all control transfers may have to share the reserved portion of the bandwidth. At low speed, one 8-byte transfer fits in the reserved portion of three frames. At full speed, one 64-byte transfer fits in the reserved portion of one frame (though again, any single transfer may be spread over multiple frames). At high speed, 512 transfers fit in the reserved portion of one frame.

Devices don't have to respond immediately to control-transfer requests. The USB specification includes timing limits that apply to most requests. A device class may require faster response to standard and class-specific requests. Where stricter timing isn't specified, in a transfer where the host requests data from the device, a device may delay as long as 500 milliseconds before making the data available to the host. To find out if data is available, the host sends a token packet requesting the data. If the data is ready, the device sends it immediately in that transaction's data packet. If not, the device returns a NAK to advise the host to retry later. The host keeps trying at intervals for up to 500 milliseconds. In a transfer where the host sends data to the device, the device can delay as long as 5 seconds before accepting all of the data and completing the Status stage (though the Status stage must complete within 50 milliseconds). The 5 seconds don't include any delays the host adds between packets. In a transfer with no Data stage, the device must complete the request and the Status stage within 50 milliseconds. The host and its drivers aren't required to enforce these limits.

Detecting and Handling Errors

If a device doesn't return an expected handshake packet during a control transfer, the host tries twice more. On receiving no response after a total of three tries, the host notifies the software that requested the transfer and stops communicating with the endpoint until the problem is corrected. The two retries include only those sent in response to no handshake at all. A NAK isn't an error.

Control transfers use data toggles to ensure that no data is lost. In the Data stage of a Control Read transfer, on receiving the data from the device, the host normally returns an ACK, then sends an OUT token packet to begin the Status stage. If the device for any reason doesn't see the ACK returned after the transfer's final data packet, the device must interpret a received OUT token packet as evidence that the handshake was returned and the Status stage can begin.

Devices must accept all Setup packets. If a new Setup packet arrives before a previous transfer completes, the device must abandon the previous transfer and start the new one.

Bulk Transfers

Bulk transfers are useful for transferring data when time isn't critical. A bulk transfer can send large amounts of data without clogging the bus because the transfers defer to the other transfer types and wait until time is available. Uses for bulk transfers include sending data from the host to a printer, sending data from a scanner to the host, and reading and writing to a disk. On an otherwise idle bus, bulk transfers are the fastest transfer type.

Availability

Only full- and high-speed devices can do bulk transfers. Devices aren't required to support bulk transfers, though a specific device class may require it. For example, a device in the mass-storage class must have a bulk endpoint in each direction.

Structure

A bulk transfer consists of one or more IN or OUT transactions (Figure 3-3). A bulk transfer is one-way: the transactions must all be IN transactions or all OUT transactions. Transferring data in both directions requires a separate pipe and transfer for each direction.

A bulk transfer ends in one of two ways: when the expected amount of data has transferred or when a transaction contains either zero data bytes or another number of bytes that is less than the endpoint's maximum packet size. The USB specification doesn't define a protocol for specifying the amount of data in a bulk transfer. When needed, the device and host can use a class-specific or vendor-specific protocol to pass this information. For example, a transfer can begin with a header that specifies the number of bytes to be transferred, or the device or host can use a class-specific or vendor-specific protocol to request a quantity of data.

To conserve bus time, the host may use the PING protocol in some high-speed bulk transfers. If a high-speed bulk OUT transfer has more than one data packet and the device returns NYET after receiving a packet, the host may use PING to find out when it's OK to begin the next data transaction. In a bulk transfer on a high-speed bus with a low- or full-speed device, the host uses split transactions for all of the transfer's transactions.

Data Size

A full-speed bulk transfer can have a maximum packet size of 8, 16, 32, or 64 bytes. For high speed, the maximum packet size must be 512 bytes. During enumeration, the host reads the maximum packet size for each bulk endpoint from the device's descriptors. The amount of data in a transfer may be less than, equal to, or greater than the maximum packet size. If the amount

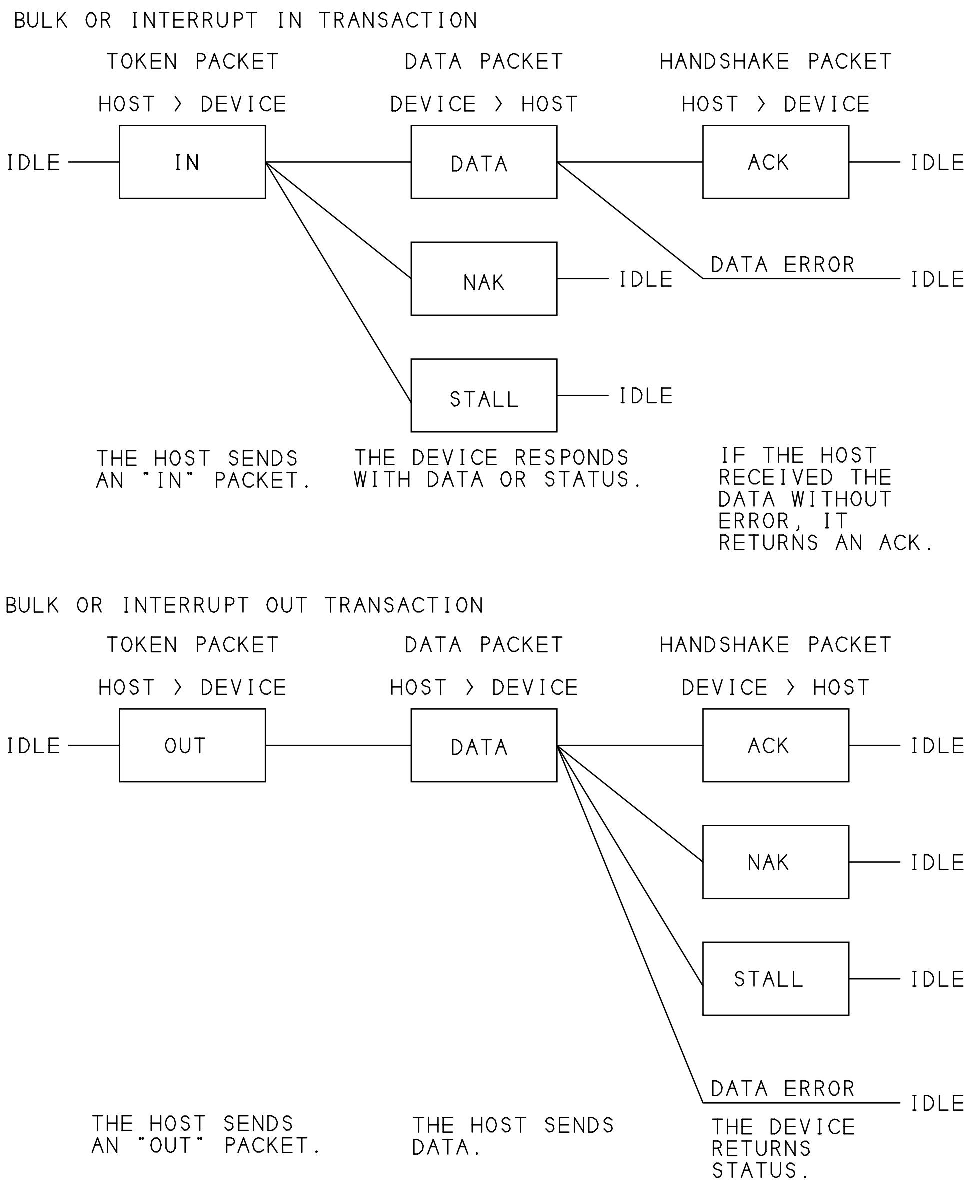

Figure 3-3: Bulk and interrupt transfers use IN and OUT transactions. Their structure is identical, but the host schedules them differently. Not shown are the PING protocol used in some high-speed bulk OUT transfers with multiple data packets or the split transactions used with low- and full-speed devices on a high-speed bus.

of data won't fit in a single packet, the host completes the transfer using multiple transactions.

Speed

The host controller guarantees that bulk transfers will complete eventually but doesn't reserve any bandwidth for the transfers. Control transfers are guaranteed to have 10 percent of the bandwidth at low and full speeds, and 20 percent at high speed. Interrupt and isochronous transfers may use the rest. So if a bus is very busy, a bulk transfer may take very long.

However, when the bus is otherwise idle, bulk transfers can use the most bandwidth of any type, and they have a low overhead, so they're the fastest of all. When an endpoint's maximum packet size is less than the maximum size allowed for the speed, some host controllers schedule no more than one packet per frame, even if more bandwidth is available. So it's best to specify the maximum allowed packet size for bulk endpoints if possible.

At full speed on an otherwise idle bus, up to nineteen 64-byte bulk transfers can transfer up to 1216 data bytes per frame, for a data rate of 1.216 Megabytes/sec. This leaves 18% of the bus bandwidth free for other uses. The protocol overhead for a bulk transfer with one data packet is 13 bytes at full speed and 55 bytes at high speed.

At high speed on an otherwise idle bus, up to thirteen 512-byte bulk transfers can transfer up to 6656 data bytes per microframe, for a data rate of 53.248 Megabytes/sec., using all but 2% of the bus bandwidth. The protocol overhead for a bulk transfer with one data packet is 55 bytes. Real-world performance varies with the host-controller hardware and driver and the host architecture, including latencies when accessing system memory. At this writing, some high-speed hosts can perform a single transfer at up to around 35 Megabytes/sec.

Detecting and Handling Errors

Bulk transfers use error detecting. If a device doesn't return an expected handshake packet, the host tries up to twice more. The host also retries on receiving NAK handshakes. The host's driver determines whether the host eventually gives up on receiving multiple NAKs. Bulk transfers use data toggles to ensure that no data is lost.

Interrupt Transfers

Interrupt transfers are useful when data has to transfer within a specific amount of time. Typical applications include keyboards, pointing devices, game controllers, and hub status reports. Users don't want a noticeable delay between pressing a key or moving a mouse and seeing the result on screen. A hub needs to report the attachment or removal of devices promptly. Low-speed devices, which support only control and interrupt transfers, are likely to use interrupt transfers for generic data.

At low and full speeds, the bandwidth available for an interrupt endpoint is limited, but high speed loosens the limits and enables an interrupt endpoint to transfer almost 400 times as much data as full speed per unit of time.

The name *interrupt transfer* suggests that a device might spontaneously send data that triggers a hardware interrupt on the host. But interrupt transfers, like all other USB transfers, occur only when the host polls a device. The transfers are interrupt-like, however, because they guarantee that the host requests or sends data with minimal delay.

Availability

All three speeds allow interrupt transfers. Devices aren't required to support interrupt transfers, but a device class may require it. For example, a HID-class device must support interrupt IN transfers for sending data to the host.

Structure

An interrupt transfer consists of one or more IN transactions or one or more OUT transactions. On the bus, interrupt transactions are identical to bulk transactions (Figure 3-3). The only difference is the scheduling. An interrupt transfer is one-way; the transactions must be all IN transactions, or all

OUT transactions. Transferring data in both directions requires a separate transfer and pipe for each direction.

An interrupt transfer ends in one of two ways: when the expected amount of data has transferred, or when a transaction contains either zero data bytes or another number of bytes that is less than the endpoint's maximum packet size. The USB specification doesn't define a protocol for specifying the amount of data in an interrupt transfer. When needed, the device and host can use a class-specific or vendor-specific protocol to pass this information. For example, a transfer can begin with a header that specifies the number of bytes to be transferred, or the device or host can use a class-specific or vendor-specific protocol to request a quantity of data.

In an interrupt transfer on a high-speed bus with a low- or full-speed device, the host uses the split transactions introduced in Chapter 2 for all of the transfer's transactions. Unlike high-speed bulk OUT transfers, high-speed interrupt OUT transfers can't use the PING protocol when a transfer has multiple transactions.

Data Size

For low-speed devices, the maximum packet size can be any value from 1 to 8 bytes. For full speed, the maximum packet size can range from 1 to 64 bytes. For high speed, the allowed range is 1 to 1024 bytes. In a device's default interface, interrupt endpoints must have a maximum packet size of 64 bytes or less. If the amount of data in a transfer won't fit in a single transaction, the host uses multiple transactions to complete the transfer.

Speed

An interrupt transfer guarantees a maximum latency, or time between transaction attempts. In other words, there is no guaranteed transfer rate, just the guarantee that there will be no more than the requested maximum latency period between transaction attempts.

High-speed interrupt transfers can be very fast. A high-speed endpoint can request up to three 1024-byte packets in each 125-microsecond microframe, which works out to 24.576 Megabytes/sec. An endpoint that requests more than 1024 bytes per microframe is called a high-bandwidth endpoint. Windows XP/Windows Server and earlier don't support high-bandwidth interrupt endpoints, however, so the achievable maximum for these operating systems is 8.192 Megabytes/sec. If the host's driver doesn't support alternate interfaces, the maximum is 64 kilobytes/sec. A full-speed endpoint can request up to 64 bytes in each 1-millisecond frame, or 64 kilobytes/sec. A low-speed endpoint can request up to 8 bytes every 10 milliseconds, or 800 bytes/sec.

The endpoint descriptor stored in the device specifies the maximum latency period. For low-speed devices, the maximum latency can be any value from 10 to 255 milliseconds. For full speed, the value can range from 1 to 255 milliseconds. For high speed, the range is 125 microseconds to 4 seconds, in increments of 125 microseconds. In addition, a high-speed interrupt endpoint with a maximum latency of 125 microseconds can request 1, 2, or 3 transactions per interval. The host controller ensures that transaction attempts occur within the specified period.

The host may begin each transaction at any time up to the specified maximum latency since the previous transaction began. So, for example, with a 10-millisecond maximum at full speed, five transfers could take as long as 50 milliseconds or as little as 5 milliseconds. OHCI host controllers use values that correspond to powers of 2, with a maximum of 32 milliseconds. So for a full-speed device that requests a maximum anywhere from 8 to 15 milliseconds, an OHCI host will begin a transaction every 8 milliseconds, and a maximum latency anywhere from 32 to 255 will cause a transaction attempt every 32 milliseconds. However, devices shouldn't rely on behavior that is specific to a type of host controller and should assume only that the host complies with the specification. (Chapter 8 has more about host-controller types.)

Because the host is free to transfer data more quickly than the requested rate, interrupt transfers don't guarantee a precise rate of delivery. The only exceptions are when the maximum latency equals the fastest possible rate. For example, with a 1.x host, a full-speed interrupt pipe configured for 1 transaction per millisecond will have bandwidth reserved for one transaction in each frame.

An otherwise idle bus can carry up to six low-speed, 8-byte transactions per frame. Note, however, that the maximum bandwidth that a single low-speed interrupt endpoint can request is 8 bytes every 10 milliseconds, and a low-speed device can have no more than two interrupt endpoints. Devices that need to transfer more than 800 bytes/sec. in each direction should be full or high speed. The reason for the limitation on low-speed endpoints is that low-speed traffic uses much more bandwidth compared to sending the same amount of data at full or high speed. Limiting the amount of bus time available to low-speed endpoints helps keep the bus available for other devices.

At full speed, nineteen 64-byte transactions can fit in a frame. Since the minimum time between transfers is one millisecond or more, each transaction in the frame would have to be for a different endpoint address. In reality, a host may not be able to schedule nineteen full-speed interrupt transactions in a single frame, so the practical maximum number of interrupt transactions per frame is likely to be less.

At high speed, the limit is two transfers per microframe, with each transfer consisting of three 1024-byte transactions.

The protocol overhead per transfer with one data packet is 19 bytes at low speed, 13 bytes at full speed, and 55 bytes at high speed. High-speed interrupt and isochronous transfers combined can use no more than 80 percent of a microframe. Full-speed isochronous transfers and low- and full-speed interrupt transfers combined can use no more than 90 percent of a frame. The section *More about Time-critical Transfers* later in this chapter has more about the capabilities and limits of interrupt transfers.

Detecting and Handling Errors

If a device doesn't return an expected handshake packet, host controllers in PCs will retry up to twice more. The host typically retries without limit on receiving NAKs. For example, a keyboard might sit idle for days before someone presses a key.

Interrupt transfers can use data toggles to ensure that all data is received without errors. A receiver that cares only about the most recent data can ignore the data toggle.

Isochronous Transfers

Isochronous transfers are streaming, real-time transfers that are useful when data must arrive at a constant rate, or by a specific time, and where occasional errors can be tolerated. At full speed, isochronous transfers can transfer more data per frame than interrupt transfers, but there is no provision for retransmitting data received with errors.

Examples of uses for isochronous transfers include encoded voice and music to be played in real time. But data that will eventually be consumed at a constant rate doesn't always require an isochronous transfer. For example, a host can use a bulk transfer to send a music file to a device. After receiving the file, the device can play the music at the appropriate rate.

Nor does the data in an isochronous transfer have to be consumed at a constant rate. An isochronous transfer is a way to ensure that a large block of data gets through quickly on a busy bus, even if the data doesn't need to transfer in real time. Unlike with bulk transfers, once an isochronous transfer begins, the host guarantees that the time will be available to send the data at a constant rate, so the completion time is predictable.

Availability

Only full- and high-speed devices can do isochronous transfers. Devices aren't required to support isochronous transfers but a device class may require it. For example, many audio- and video-class devices use isochronous endpoints.

Structure

Isochronous means that the data has a fixed transfer rate, with a defined number of bytes transferring in every frame or microframe. None of the other transfer types guarantee bandwidth for a specific number of bytes in each frame (except interrupt transfers with the shortest maximum latency).

A full-speed isochronous transfer consists of one IN or OUT transaction per frame in one or more frames at equal intervals. High-speed isochronous transfers are more flexible. They can request as many as three transactions per microframe or as little as one transaction every 32,768 microframes. Figure 3-4 shows the packets in full-speed isochronous IN and OUT transactions. An isochronous transfer is one-way. The transactions in a transfer must all be IN transactions or all OUT transactions. Transferring data in both directions requires a separate pipe and transfer for each direction.

The USB specification doesn't define a protocol for specifying the amount of data in an isochronous transfer. When needed, the device and host can use a class-specific or vendor-specific protocol to pass this information. For

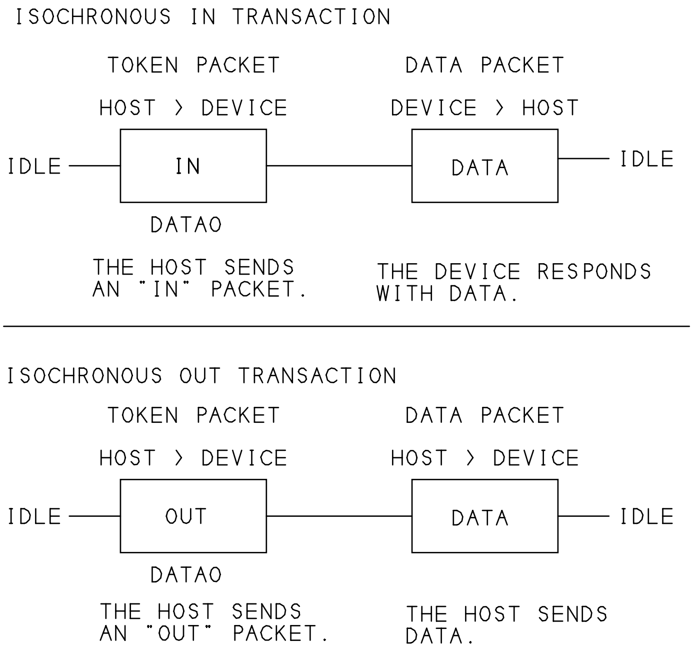

Figure 3-4: Isochronous transfers don't have handshake packets, so occasional errors must be acceptable. Not shown are the split transactions used with full-speed devices on a high-speed bus or the data PID sequencing in high-speed transfers with multiple data packets per microframe.

example, a transfer can begin with a header that specifies the number of bytes to be transferred, or the device or host can use a class-specific or vendor-specific protocol to request a quantity of data.

Before selecting a device configuration that consumes isochronous bandwidth, the host controller determines whether the requested bandwidth is available by comparing the available unreserved bus bandwidth with the maximum packet size and transfer rate of the configuration's isochronous endpoint(s). A full-speed transfer with the maximum 1023 bytes per frame uses 69 percent of the bus's bandwidth. If two full-speed devices want to transfer 1023 bytes per frame, a 1.x host will refuse to configure the second device because the data won't fit in the remaining bandwidth.

Every USB 2.0 device with isochronous endpoints must have an interface that requests no isochronous bandwidth so the host can configure the device even if there is no reservable bandwidth available. In addition to this interface and an interface that requests the optimum bandwidth for a device, a device can have alternate interfaces that have smaller isochronous data packets or use fewer isochronous packets per microframe. The device driver can then request to use an interface that transfers data at a lower rate when needed. Or the driver can try again later in the hope that the bandwidth will be available. After the host configures the device, the transfers are guaranteed to have the time they need.

Although isochronous transfers may send a fixed number of bytes per frame, the data doesn't transfer at a constant number of bits per second. Each transaction has overhead and must share the bus with other devices. So the data is actually a burst at 12 Megabits/sec. or 480 Megabits/sec. and may occur any time within the frame or microframe. To use the data at a constant rate, such as sending the data to a speaker, the receiver must convert the received bits to signals that span the interval.

Isochronous transfers may also synchronize to another data source or recipient, or to the bus's Start-of-Frame signals. For example, a microphone's input may synchronize to the output of speakers. The USB specification describes several methods of synchronizing to internal and external clocks. The descriptor for a USB 2.0 isochronous endpoint can specify a synchronization type and a usage value that indicates whether the endpoint is contains data or feedback information used to maintain synchronization.

If a host is performing an isochronous transfer on a high-speed bus with a full-speed device, the host uses the split transactions introduced in Chapter 2 for all of the transfer's transactions. Isochronous OUT transactions use start-split transactions, but not complete-splits because there is no status information to report back to the host. Isochronous transfers don't use the PING protocol.

Data Size

For full-speed endpoints, the maximum packet size can range from 0 to 1023 data bytes. High-speed endpoints can have a maximum packet size up to 1024 bytes. If the amount of data won't fit in a single packet, the host completes the transfer in multiple transactions.

Within a transfer, the amount of data in each frame doesn't have to be the same. For example, data at 44,100 samples per second could use a sequence of 9 frames containing 44 samples each, followed by 1 frame containing 45 samples.

Speed

A full-speed isochronous transaction can transfer up to 1023 bytes per frame, or up to 1.023 Megabytes/sec. This leaves 31% of the bus bandwidth free for other uses. The protocol overhead is 9 bytes per transfer for a transfer with one data packet, or less than 1% for a single 1023-byte transaction. The minimum requested bandwidth for a full-speed transfer is one byte per frame, which is 1 kilobyte per second.

A high-speed isochronous transaction can transfer up to 1024 bytes. An isochronous endpoint that requires more than 1024 bytes per microframe can request 2 or 3 transactions per microframe, for a maximum rate of 24.576 Megabytes/sec. An endpoint that requires multiple transactions per microframe is called a high-bandwidth endpoint. The protocol overhead is 38 bytes per transfer for a transfer with one data packet.

Because high-speed isochronous transfers don't have to do a transaction in every frame or microframe, they can request less bandwidth than full-speed transfers. The minimum requested bandwidth is one byte every 32,678 microframes, which works out to one byte every 4.096 seconds. However, any endpoint can transfer less data than the maximum reserved bandwidth by skipping available transactions or by transferring less than the maximum data per transfer.

On a high-speed bus, interrupt and isochronous transfers can use no more than 80 percent of a microframe. On a full-speed bus, isochronous transfers and low- and full-speed interrupt transfers combined can use no more than 90 percent of a frame. An otherwise idle high-speed bus can carry two isochronous transfers at the maximum rate.

The section *More about Time-critical Transfers* later in this chapter has more about the capabilities of isochronous transfers.

Detecting and Handling Errors

The price to pay for guaranteed on-time delivery of large blocks of data is no error correcting. Isochronous transfers are intended for uses where occasional, small errors are acceptable. For example, listeners may tolerate or not even notice a short dropout in voice or music. And in reality, under normal circumstances, a USB transfer should experience no more than a very occasional error due to line noise. Because isochronous transfers must keep to a schedule, the receiver can't request the sender to retransmit if the receiver is busy or detects an error. A receiver that suspects errors could ask the sender to resend the entire transfer, but this approach isn't very efficient.

More about Time-critical Transfers

Just because an endpoint is capable of a rate of data transfer doesn't mean that a particular device and host will be able to achieve the rate. Several things can limit an application's ability to send or receive data at the rate that a device requests. The limiting factors include bus bandwidth, the device's capabilities, the capabilities of the device driver and application software, and latencies in the host's hardware and software.

Bus Bandwidth

When a device requests more interrupt or isochronous bandwidth than is available, the host refuses to configure the device. Low- and full-speed interrupt transfers use little bandwidth, so the host isn't likely to deny a configuration due to their requirements. High-speed interrupt transfers are a different story. A high-speed endpoint can request up to three 1024-byte data packets in each microframe, using as much as 40 percent of the bus bandwidth. To help ensure that devices can enumerate without problems, the interrupt endpoints in a device's default interface must specify a maximum packet size no larger than 64 bytes. The device driver is then free to try to increase the endpoint's reserved bandwidth by requesting alternate interface settings or configurations.

Isochronous endpoints can also cause bandwidth problems. A frequent problem with isochronous endpoints on 1.x devices is that devices request more bandwidth than is available. The host properly refuses to configure the device and the user is left with a device that doesn't work without knowing why.

To help ensure that devices will enumerate without problems, the default interface setting of a 2.0-compliant device must request no isochronous bandwidth. In other words, the default interface can transfer no isochronous data at all. An obvious way to comply is to include no isochronous endpoints in the default interface. After enumeration, the device driver is free to attempt to request isochronous bandwidth by requesting an alternate interface or configuration with an isochronous endpoint. Note that even full-speed endpoints must meet this requirement to comply with USB 2.0.

Device Capabilities

If the host has promised that the requested USB bandwidth will be available, there's still no guarantee that the device will be ready to send or receive data when needed.

To use interrupt and isochronous transfers effectively, both the sender and receiver have to be capable of sending and receiving at the desired rate. A device that is sending data must write the data to send into the endpoint's transmit buffer in time to enable the controller to place the data on the bus on receiving an IN token packet. A device that is receiving data must read the previous data from the endpoint's buffer before the new data arrives. Otherwise either the old data will be overwritten or the device will NAK or drop the new data.

One way to help ensure that the device is always ready for a transfer is to use double (or quadruple) buffering, as described in Chapter 6. Multiple buffers give the firmware extra time to load the next data to transfer or to retrieve just-received data.

Host Capabilities

The capabilities of the device driver and application software on the host can also can affect whether all available transfers take place.

A device driver requests a transfer by submitting an I/O request packet (IRP) to a lower-level driver. For interrupt and isochronous transfers, if there is no outstanding IRP for an endpoint when its scheduled time comes up, the host controller skips the transaction attempt. To ensure that no transfer opportunities are missed, drivers typically submit a new IRP immediately on completing the previous one.

The application software that uses the data also has to be able to keep up with the transfers. For example, the driver for HID-class devices places report data received in interrupt transfers in a buffer, and applications use ReadFile to retrieve reports from the buffer. If the buffer is full when a new report arrives, the driver discards the oldest report and replaces it with the newest one. If the application can't keep up, some reports are lost. A solution is to increase the size of the buffer the driver uses to store received data or to read multiple reports at once.

One way to help ensure that an application sends or receives data with minimal delays is to place the code that communicates with the device driver in its own program thread. The thread should have few responsibilities other than managing these communications.

Doing fewer, larger transfers rather than multiple, small transfers can also help. An application can typically send or request a few large chunks of data more quickly than it can send or request many smaller chunks. When there are multiple transactions per transfer, the lower-level drivers take care of the scheduling.

Host Latencies

Another factor in the performance of time-critical USB transfers is the latencies due to how Windows handles multi-tasking. Windows was never designed as a real-time operating system that could guarantee a rate of data transfer with a peripheral.

Multi-tasking means that multiple program threads run at the same time. The operating system grants a portion of the available time to each thread. Different threads can have different priorities, but under Windows, no thread can be guaranteed CPU time at a defined, precise rate, such as once per millisecond.

Latencies under Windows are often well under 1 millisecond, but in some cases a thread can keep other code from executing for over 100 milliseconds. Newer Windows editions tend to have improved performance over older editions.

A USB device and its software have no control over what other tasks the host CPU is performing and how fast the CPU can perform them, so dealing with these latencies can be a challenge when timing is critical.

In general, it's best to let the device handle any required real-time processing and make the timing of the host communications as non-critical as possible. For example, imagine a full-speed device that reads a sensor once per millisecond. The device could attempt to send each reading to the host in a separate interrupt transfer, but if a transfer is skipped for any reason, the transfers will never catch up. If the device instead collects a series of readings and transfers them using less frequent, but larger transfers, the timing of the bus transfers is less critical. Data compression can also help by reducing the amount of data that must transfer.