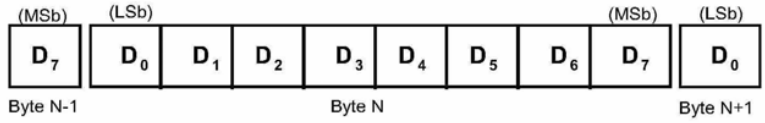

Figure 8-4. Data Field Format

Data packet size varies with the transfer type, as described in Chapter 5.

8.3.5 Cyclic Redundancy Checks

Cyclic redundancy checks (CRCs) are used to protect all non-PID fields in token and data packets. In this context, these fields are considered to be protected fields. The PID is not included in the CRC check of a packet containing a CRC. All CRCs are generated over their respective fields in the transmitter before bit stuffing is performed. Similarly, CRCs are decoded in the receiver after stuffed bits have been removed. Token and data packet CRCs provide 100% coverage for all single- and double-bit errors. A failed CRC is considered to indicate that one or more of the protected fields is corrupted and causes the receiver to ignore those fields and, in most cases, the entire packet.

For CRC generation and checking, the shift registers in the generator and checker are seeded with an all-ones pattern. For each data bit sent or received, the high order bit of the current remainder is XORed with the data bit and then the remainder is shifted left one bit and the low-order bit set to zero. If the result of that XOR is one, then the remainder is XORed with the generator polynomial.

When the last bit of the checked field is sent, the CRC in the generator is inverted and sent to the checker MSb first. When the last bit of the CRC is received by the checker and no errors have occurred, the remainder will be equal to the polynomial residual.

A CRC error exists if the computed checksum remainder at the end of a packet reception does not match the residual.

Bit stuffing requirements must be met for the CRC, and this includes the need to insert a zero at the end of a CRC if the preceding six bits were all ones.

8.3.5.1 Token CRCs

A five-bit CRC field is provided for tokens and covers the ADDR and ENDP fields of IN, SETUP, and OUT tokens or the time stamp field of an SOF token. The PING and SPLIT special tokens also include a five-bit CRC field. The generator polynomial is:

$$G(X) = X^5 + X^2 + 1$$

The binary bit pattern that represents this polynomial is 00101B. If all token bits are received without error, the five-bit residual at the receiver will be 01100B.

8.3.5.2 Data CRCs

The data CRC is a 16-bit polynomial applied over the data field of a data packet. The generating polynomial is:

$$G(X) = X^{16} + X^{15} + X^2 + 1$$

The binary bit pattern that represents this polynomial is 1000000000000101B. If all data and CRC bits are received without error, the 16-bit residual will be 1000000000001101B.

8.4 Packet Formats

This section shows packet formats for token, data, and handshake packets. Fields within a packet are displayed in these figures in the order in which bits are shifted out onto the bus.

8.4.1 Token Packets

Figure 8-5 shows the field formats for a token packet. A token consists of a PID, specifying either IN, OUT, or SETUP packet type and ADDR and ENDP fields. The PING special token packet also has the same fields as a token packet. For OUT and SETUP transactions, the address and endpoint fields uniquely identify the endpoint that will receive the subsequent Data packet. For IN transactions, these fields uniquely identify which endpoint should transmit a Data packet. For PING transactions, these fields uniquely identify which endpoint will respond with a handshake packet. Only the host can issue token packets. An IN PID defines a Data transaction from a function to the host. OUT and SETUP PIDs define Data transactions from the host to a function. A PING PID defines a handshake transaction from the function to the host.

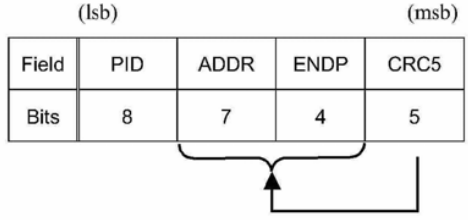

Figure 8-5. Token Format

Token packets have a five-bit CRC that covers the address and endpoint fields as shown above. The CRC does not cover the PID, which has its own check field. Token and SOF packets are delimited by an EOP after three bytes of packet field data. If a packet decodes as an otherwise valid token or SOF but does not terminate with an EOP after three bytes, it must be considered invalid and ignored by the receiver.

8.4.2 Split Transaction Special Token Packets

USB defines a special token for split transactions: SPLIT. This is a 4 byte token packet compared to other normal 3 byte token packets. The split transaction token packet provides additional transaction types with additional transaction specific information. The split transaction token is used to support split transactions between the host controller communicating with a hub operating at high speed with full-/low-speed devices to some of its downstream facing ports. There are two split transactions defined that use the SPLIT special token: a start-split transaction (SSPLIT) and a complete-split transaction (CSPLIT). A field in the SPLIT special token, described in the following sections, indicates the specific split transaction.

8.4.2.1 Split Transactions

A high-speed split transaction is used only between the host controller and a hub when the hub has full-/low-speed devices attached to it. This high-speed split transaction is used to initiate a full-/low-speed transaction via the hub and some full-/low-speed device endpoint. The high-speed split transaction also allows the completion status of the full-/low-speed transaction to be retrieved from the hub. This approach allows the host controller to start a full-/low-speed transaction via a high-speed transaction and then continue with other high-speed transactions without having to wait for the full-/low-speed transaction to proceed/complete at the slower speed. See Chapter 11 for more details about the state machines and transaction definitions of split transactions.

A high-speed split transaction has two parts: a start-split and a complete-split. Split transactions are only defined to be used between the host controller and a hub. No other high-speed or full-/low-speed devices ever use split transactions.

Figure 8-6 shows the packets composing a generic start-split transaction. There are two packets in the token phase: the SPLIT special token and a full-/low-speed token. Depending on the direction of data transfer and whether a handshake is defined for the transaction type, the token phase is optionally followed by a data packet and a handshake packet. Start split transactions can consist of 2, 3, or 4 packets as determined by the specific transfer type and data direction.

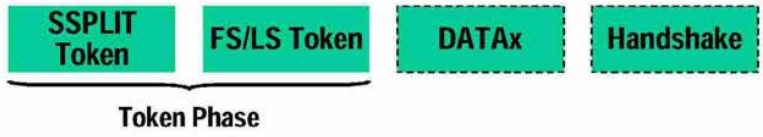

Figure 8-6. Packets in a Start-split Transaction

Figure 8-7 shows the packets composing a generic complete-split transaction. There are two packets in the token phase: the SPLIT special token and a full-/low-speed token. A data or handshake packet follows the token phase packets in the complete-split depending on the data transfer direction and specific transaction type. Complete split transactions can consist of 2 or 3 packets as determined by the specific transfer type and data direction.

Figure 8-7. Packets in a Complete-split Transaction

The results of a split transaction are returned by a complete-split transaction. Figure 8-8 shows this conceptual "conversion" for an example interrupt IN transfer type. The host issues a start-split (indicated with 1) to the hub and then can proceed with other high-speed transactions. The start-split causes the hub to issue a full-/low-speed IN token sometime later (indicated by 2). The device responds to the IN token (in this example) with a data packet and the hub responds with a handshake to the device. Finally, the host sometime later issues a complete-split (indicated by 3) to retrieve the data provided by the device. Note that in the example, the hub provided the full-/low-speed handshake (ACK in this example) to the device endpoint before the complete-split, and the complete-split did not provide a high-speed handshake to the hub.

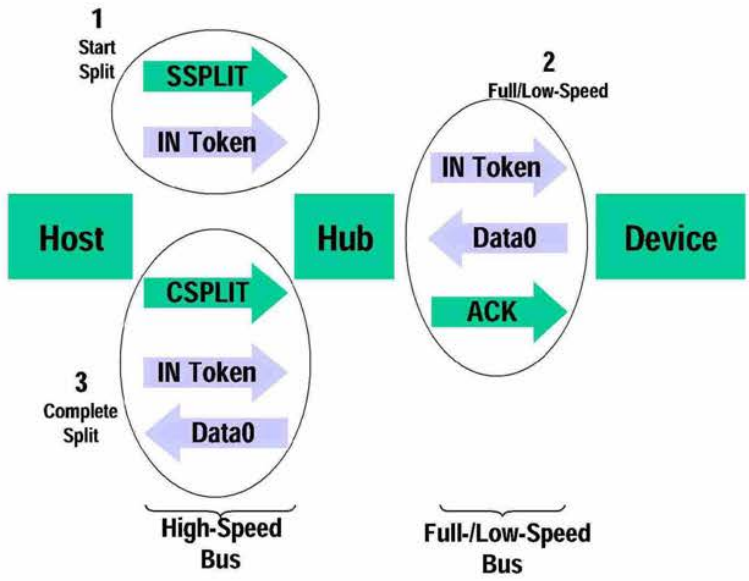

Figure 8-8. Relationship of Interrupt IN Transaction to High-speed Split Transaction

A normal full-/low-speed OUT transaction is similarly conceptually "converted" into start-split and complete-split transactions. Figure 8-9 shows this "conversion" for an example interrupt OUT transfer type. The host issues a start-split transaction consisting of a SSPLIT special token, an OUT token, and a DATA packet. The hub sometime later issues the OUT token and DATA packet on the full-/low-speed bus. The device responds with a handshake. Sometime later, the host issues the complete-split transaction and the hub responds with the results (either full-/low-speed data or handshake) provided by the device.

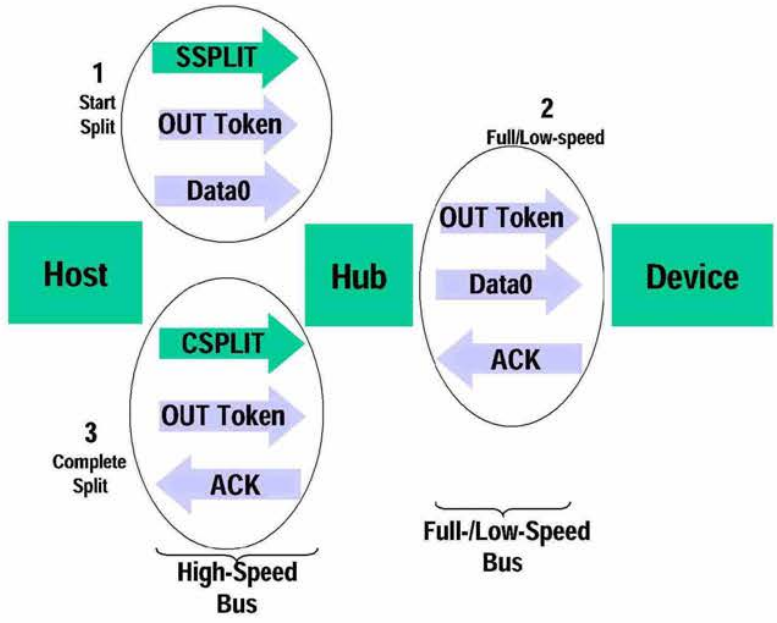

Figure 8-9. Relationship of Interrupt OUT Transaction to High-speed Split OUT Transaction

The next two sections describe the fields composing the detailed start- and complete-split token packets. Figure 8-10 and Figure 8-12 show the fields in the split-transaction token packet. The SPLIT special token follows the general token format and starts with a PID field (after a SYNC) and ends with a CRC5 field (and EOP). Start-split and complete-split token packets are both 4 bytes long. SPLIT transactions must only originate from the host. The start-split token is defined in Section 8.4.2.2 and the complete-split token is defined in Section 8.4.2.3.

8.4.2.2 Start-Split Transaction Token

	(lsb)							(msb)
Field	SPLIT PID	Hub Addr	SC	Port	S	E	ET	CRC5
Bits	8	7	1	7	1	1	2	5

Figure 8-10. Start-split (SSPLIT) Token

The Hub addr field contains the USB device address of the hub supporting the specified full-/low-speed device for this full-/low-speed transaction. This field has the same definition as the ADDR field definition in Section 8.3.2.1.

A SPLIT special token packet with the SC (Start/Complete) field set to zero indicates that this is a start-split transaction (SSPLIT).

The Port field contains the port number of the target hub for which this full-/low-speed transaction is destined. As shown in Figure 8-11, a total of 128 ports are specified as PORT<6:0>. The host must correctly set the port field for single and multiple TT hub implementations. A single TT hub implementation may ignore the port field.

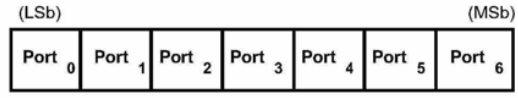

Figure 8-11. Port Field

The S (Speed) field specifies the speed for this interrupt or control transaction as follows:

- 0 – Full speed
- 1 – Low speed

For bulk IN/OUT and isochronous IN start-splits, the S field must be set to zero. For bulk/control IN/OUT, interrupt IN/OUT, and isochronous IN start-splits, the E field must be set to zero.

For full-speed isochronous OUT start-splits, the S[1] (Start) and E (End) fields specify how the high-speed data payload corresponds to data for a full-speed data packet as shown in Table 8-2.

Table 8-2. Isochronous OUT Payload Continuation Encoding

S	E	High-speed to Full-speed Data Relation
0	0	High-speed data is the middle of the full-speed data payload
0	1	High-speed data is the end of the full-speed data payload
1	0	High-speed data is the beginning of the full-speed data payload
1	1	High-speed data is all of the full-speed data payload.

Isochronous OUT start-split transactions use these encodings to allow the hub to detect various error cases due to lack of receiving start-split transactions for an endpoint with a data payload that requires multiple start-splits. For example, a large full-speed data payload may require three start-split transactions: a start-split/beginning, a start-split/middle and a start-split/end. If any of these transactions is not received by the hub, it will either ignore the full-speed transaction (if the start-split/beginning is not received), or it will force an error for the corresponding full-speed transaction (if one of the other two transactions are not received). Other error conditions can be detected by not receiving a start-split during a microframe.

The ET (Endpoint Type) field specifies the endpoint type of the full-/low-speed transaction as shown in Table 8-3.

[1] The S bit can be reused for these encodings since isochronous transactions must not be low speed.

Table 8-3. Endpoint Type Values in Split Special Token

ET value (msb:lsb)	Endpoint Type
00	Control
01	Isochronous
10	Bulk
11	Interrupt

This field tells the hub which split transaction state machine to use for this full-/low-speed transaction.

The full-/low-speed device address and endpoint number information is contained in the normal token packet that follows the SPLIT special token packet.

8.4.2.3 Complete-Split Transaction Token

(lsb) (msb)

Field	SPLIT PID	Hub Addr	SC	Port	S	U	ET	CRC5
Bits	8	7	1	7	1	1	2	5

Figure 8-12. Complete-split (CSPLIT) Transaction Token

A SPLIT special token packet with the SC field set to one indicates that this is a complete-split transaction (CSPLIT).

The U bit is reserved/unused and must be reset to zero(0B).

The other fields of the complete-split token packet have the same definitions as for the start-split token packet.

8.4.3 Start-of-Frame Packets

Start-of-Frame (SOF) packets are issued by the host at a nominal rate of once every 1.00 ms ±0.0005 ms for a full-speed bus and 125 µs ±0.0625 µs for a high-speed bus. SOF packets consist of a PID indicating packet type followed by an 11-bit frame number field as illustrated in Figure 8-13.

(lsb) (msb)

Field	PID	FrameNumber	CRC5
Bits	8	11	5

Figure 8-13. SOF Packet

The SOF token comprises the token-only transaction that distributes an SOF marker and accompanying frame number at precisely timed intervals corresponding to the start of each frame. All high-speed and full-speed functions, including hubs, receive the SOF packet. The SOF token does not cause any receiving function to generate a return packet; therefore, SOF delivery to any given function cannot be guaranteed.

The SOF packet delivers two pieces of timing information. A function is informed that an SOF has occurred when it detects the SOF PID. Frame timing sensitive functions, that do not need to keep track of frame number (e.g., a full-speed operating hub), need only decode the SOF PID; they can ignore the frame number and its CRC. If a function needs to track frame number, it must comprehend both the PID and the time stamp. Full-speed devices that have no particular need for bus timing information may ignore the SOF packet.

8.4.3.1 USB Frames and Microframes

USB defines a full-speed 1 ms frame time indicated by a Start Of Frame (SOF) packet each and every 1ms period with defined jitter tolerances. USB also defines a high-speed microframe with a 125 µs frame time with related jitter tolerances (See Chapter 7). SOF packets are generated (by the host controller or hub transaction translator) every 1ms for full-speed links. SOF packets are also generated after the next seven 125 µs periods for high-speed links.

Figure 8-14 shows the relationship between microframes and frames.

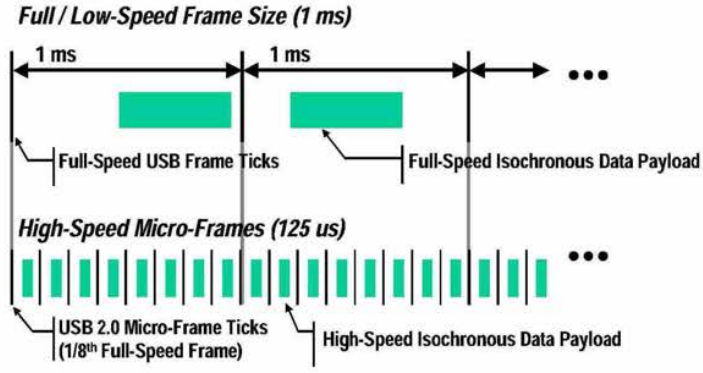

Figure 8-14. Relationship between Frames and Microframes

High-speed devices see an SOF packet with the same frame number eight times (every 125 µs) during each 1 ms period. If desired, a high-speed device can locally determine a particular microframe "number" by detecting the SOF that had a different frame number than the previous SOF and treating that as the zeroth microframe. The next seven SOFs with the same frame number can be treated as microframes 1 through 7.

8.4.4 Data Packets

A data packet consists of a PID, a data field containing zero or more bytes of data, and a CRC as shown in Figure 8-15. There are four types of data packets, identified by differing PIDs: DATA0, DATA1, DATA2 and MDATA. Two data packet PIDs (DATA0 and DATA1) are defined to support data toggle synchronization (refer to Section 8.6). All four data PIDs are used in data PID sequencing for high bandwidth high-speed isochronous endpoints (refer to Section 5.9). Three data PIDs (MDATA, DATA0, DATA1) are used in split transactions (refer to Sections 11.17-11.21).

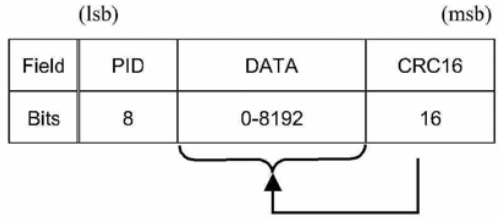

Figure 8-15. Data Packet Format

Data must always be sent in integral numbers of bytes. The data CRC is computed over only the data field in the packet and does not include the PID, which has its own check field.

The maximum data payload size allowed for low-speed devices is 8 bytes. The maximum data payload size for full-speed devices is 1023. The maximum data payload size for high-speed devices is 1024 bytes.

8.4.5 Handshake Packets

Handshake packets, as shown in Figure 8-16, consist of only a PID. Handshake packets are used to report the status of a data transaction and can return values indicating successful reception of data, command acceptance or rejection, flow control, and halt conditions. Only transaction types that support flow control can return handshakes. Handshakes are always returned in the handshake phase of a transaction and may be returned, instead of data, in the data phase. Handshake packets are delimited by an EOP after one byte of packet field. If a packet decodes as an otherwise valid handshake but does not terminate with an EOP after one byte, it must be considered invalid and ignored by the receiver.

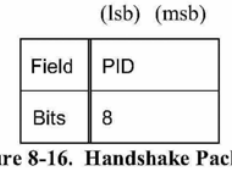

Figure 8-16. Handshake Packet

There are four types of handshake packets and one special handshake packet:

- **ACK** indicates that the data packet was received without bit stuff or CRC errors over the data field and that the data PID was received correctly. ACK may be issued either when sequence bits match and the receiver can accept data or when sequence bits mismatch and the sender and receiver must resynchronize to each other (refer to Section 8.6 for details). An ACK handshake is applicable only in transactions in which data has been transmitted and where a handshake is expected. ACK can be returned by the host for IN transactions and by a function for OUT, SETUP, or PING transactions.
- **NAK** indicates that a function was unable to accept data from the host (OUT) or that a function has no data to transmit to the host (IN). NAK can only be returned by functions in the data phase of IN transactions or the handshake phase of OUT or PING transactions. The host can never issue NAK.

NAK is used for flow control purposes to indicate that a function is temporarily unable to transmit or receive data, but will eventually be able to do so without need of host intervention.

- **STALL** is returned by a function in response to an IN token or after the data phase of an OUT or in response to a PING transaction (see Figure 8-30 and Figure 8-38). STALL indicates that a function is unable to transmit or receive data, or that a control pipe request is not supported. The state of a function after returning a STALL (for any endpoint except the default endpoint) is undefined. The host is not permitted to return a STALL under any condition.

  The STALL handshake is used by a device in one of two distinct occasions. The first case, known as "functional stall," is when the *Halt* feature associated with the endpoint is set. (The *Halt* feature is specified in Chapter 9 of this document.) A special case of the functional stall is the "commanded stall." Commanded stall occurs when the host explicitly sets the endpoint's *Halt* feature, as detailed in Chapter 9. Once a function's endpoint is halted, the function must continue returning STALL until the condition causing the halt has been cleared through host intervention.

  The second case, known as "protocol stall," is detailed in Section 8.5.3. Protocol stall is unique to control pipes. Protocol stall differs from functional stall in meaning and duration. A protocol STALL is returned during the Data or Status stage of a control transfer, and the STALL condition terminates at the beginning of the next control transfer (Setup). The remainder of this section refers to the general case of a functional stall.

- **NYET** is a high-speed only handshake that is returned in two circumstances. It is returned by a high-speed endpoint as part of the PING protocol described later in this chapter. NYET may also be returned by a hub in response to a split-transaction when the full-/low-speed transaction has not yet been completed or the hub is otherwise not able to handle the split-transaction. See Chapter 11 for more details.

- **ERR** is a high-speed only handshake that is returned to allow a high-speed hub to report an error on a full-/low-speed bus. It is only returned by a high-speed hub as part of the split transaction protocol. See Chapter 11 for more details.

8.4.6 Handshake Responses

Transmitting and receiving functions must return handshakes based upon an order of precedence detailed in Table 8-4 through Table 8-6. Not all handshakes are allowed, depending on the transaction type and whether the handshake is being issued by a function or the host. Note that if an error occurs during the transmission of the token to the function, the function will not respond with any packets until the next token is received and successfully decoded.

8.4.6.1 Function Response to IN Transactions

Table 8-4 shows the possible responses a function may make in response to an IN token. If the function is unable to send data, due to a halt or a flow control condition, it issues a STALL or NAK handshake, respectively. If the function is able to issue data, it does so. If the received token is corrupted, the function returns no response.

Table 8-4. Function Responses to IN Transactions

Token Received Corrupted	Function Tx Endpoint Halt Feature	Function Can Transmit Data	Action Taken
Yes	Don't care	Don't care	Return no response
No	Set	Don't care	Issue STALL handshake
No	Not set	No	Issue NAK handshake
No	Not set	Yes	Issue data packet

8.4.6.2 Host Response to IN Transactions

Table 8-5 shows the host response to an IN transaction. The host is able to return only one type of handshake: ACK. If the host receives a corrupted data packet, it discards the data and issues no response. If the host cannot accept data from a function, (due to problems such as internal buffer overrun) this condition is considered to be an error and the host returns no response. If the host is able to accept data and the data packet is received error-free, the host accepts the data and issues an ACK handshake.

Table 8-5. Host Responses to IN Transactions

Data Packet Corrupted	Host Can Accept Data	Handshake Returned by Host
Yes	N/A	Discard data, return no response
No	No	Discard data, return no response
No	Yes	Accept data, issue ACK

8.4.6.3 Function Response to an OUT Transaction

Handshake responses for an OUT transaction are shown in Table 8-6. Assuming successful token decode, a function, upon receiving a data packet, may return any one of the three handshake types. If the data packet was corrupted, the function returns no handshake. If the data packet was received error-free and the function's receiving endpoint is halted, the function returns STALL. If the transaction is maintaining sequence bit synchronization and a mismatch is detected (refer to Section 8.6 for details), then the function returns ACK and discards the data. If the function can accept the data and has received the data error-free, it returns ACK. If the function cannot accept the data packet due to flow control reasons, it returns NAK.

Table 8-6. Function Responses to OUT Transactions in Order of Precedence

Data Packet Corrupted	Receiver Halt Feature	Sequence Bits Match	Function Can Accept Data	Handshake Returned by Function
Yes	N/A	N/A	N/A	None
No	Set	N/A	N/A	STALL
No	Not set	No	N/A	ACK
No	Not set	Yes	Yes	ACK
No	Not set	Yes	No	NAK

8.4.6.4 Function Response to a SETUP Transaction

SETUP defines a special type of host-to-function data transaction that permits the host to initialize an endpoint's synchronization bits to those of the host. Upon receiving a SETUP token, a function must accept the data. A function may not respond to a SETUP token with either STALL or NAK, and the receiving function must accept the data packet that follows the SETUP token. If a non-control endpoint receives a SETUP token, it must ignore the transaction and return no response.

8.5 Transaction Packet Sequences

The packets that comprise a transaction varies depending on the endpoint type. There are four endpoint types: bulk, control, interrupt, and isochronous.

A host controller and device each require different state machines to correctly sequence each type of transaction. Figures in the following sections show state machines that define the correct sequencing of packets within a transaction of each type. The diagrams should not be taken as a required implementation, but to specify the required behavior.

Figure 8-17 shows the legend for the state machine diagrams. A circle with a three-line border indicates a reference to another (hierarchical) state machine. A circle with a two-line border indicates an initial state. A circle with a single-line border represents a simple state.

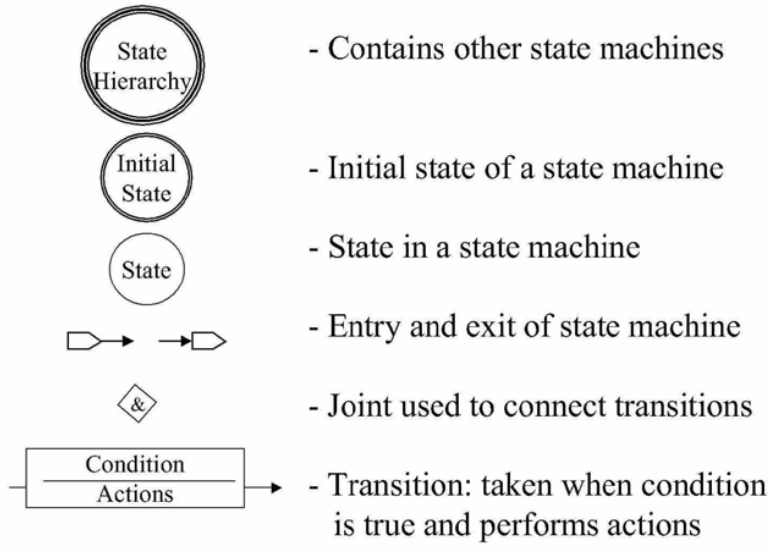

Figure 8-17. Legend for State Machines

The "tab" shapes with arrows are the entry or exit (respectively in the legend) to/from the state machine. The entry/exit relates to another state in a state machine at a higher level in the state machine hierarchy.

A diamond (joint) is used to join several transitions to a common point. A joint allows a single input transition with multiple output transitions or multiple input transitions and a single output transition. All conditions on the transitions of a path involving a joint must be true for the path to be taken. A path is simply a sequence of transitions involving one or more joints.

A transition is labeled with a block with a line in the middle separating the (upper) condition and the (lower) actions. The condition is required to be true to take the transition. The syntax for actions and conditions is VHDL. The actions are performed if the transition is taken. A circle includes a name in bold and optionally one or more actions that are performed upon entry to the state.

The host controller and device state machines are in a context as shown in Figure 8-18. The host controller determines the next transaction to run for an endpoint and issues a command (HC_cmd) to the host controller state machines. This causes the host controller state machines to issue one or more packets to move over the downstream bus (HSD1).

The device receives these packets from the bus (HSD2), reacts to the received packet, and interacts with its function(s) via the state of the corresponding endpoint (in the EP_array). Then the device may respond with a packet on the upstream bus (HSU1). The host controller state machines can receive a packet from the bus (HSU2) and provide a result of the transaction back to the host controller (HC_resp). The details of what packets are sent on the bus is determined by the transfer type for the endpoint and what bus activity the state machines observe.

The state machines are presented in a hierarchical form. Figure 8-19 shows the top level state machines for the host controller. The non-split transactions are presented in the remainder of this chapter. The split transaction state machines (HC_Do_start and HC_Do_complete) are described and shown in Chapter 11.

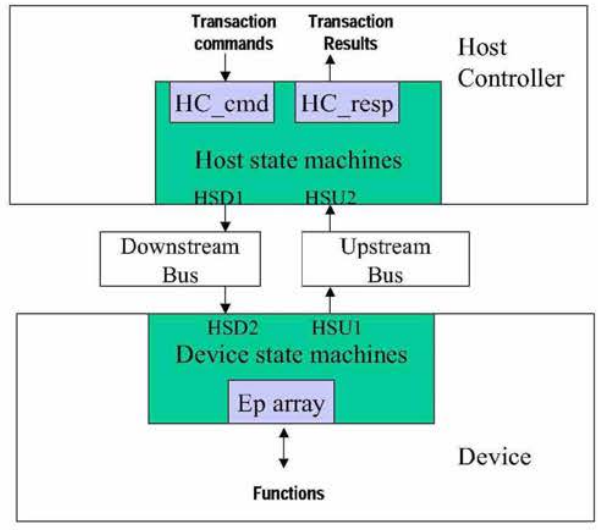

Figure 8-18. State Machine Context Overview

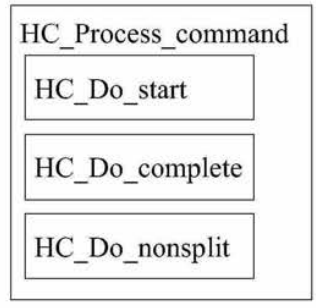

Figure 8-19. Host Controller Top Level Transaction State Machine Hierarchy Overview

The host controller state machines are located in the host controller. The host controller causes packets to be issued downstream (labeled as HSD1) and it receives upstream packets (labeled as HSU2).

The device state machines are located in the device. The device causes packets to be issued upstream (labeled as HSU1) and it receives downstream packets (labeled as HSD2).

The host controller has commands that tell it what transaction to issue next for an endpoint. The host controller tracks transactions for several endpoints. The host controller state machines sequence to determine what the host controller needs to do next for the current endpoint. The device has a state for each of its endpoints. The device state machines sequence to determine what reaction the device has to a transaction.

The appendix includes some declarations that were used in constructing the state machines and may be useful in understanding additional details of the state machines. There are several pseudo-code procedures and functions for conditions and actions. Simple descriptions of them are also included in the appendix.

Figure 8-20 shows an overview of the overall state machine hierarchy for the host controller for the non-split transaction types. Figure 8-21 shows the hierarchy of the device state machines. The state machines

common to endpoint types are presented first. The lowest level endpoint type specific state machines are presented in each following endpoint type section.

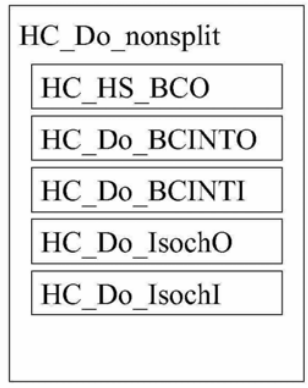

Figure 8-20. Host Controller Non-split Transaction State Machine Hierarchy Overview

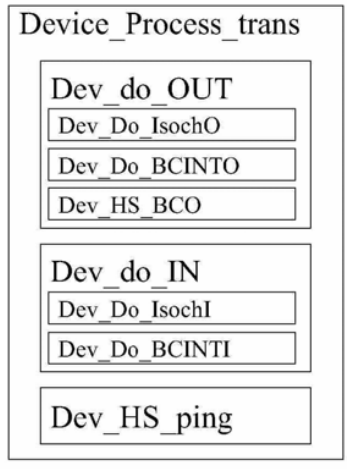

Figure 8-21. Device Transaction State Machine Hierarchy Overview

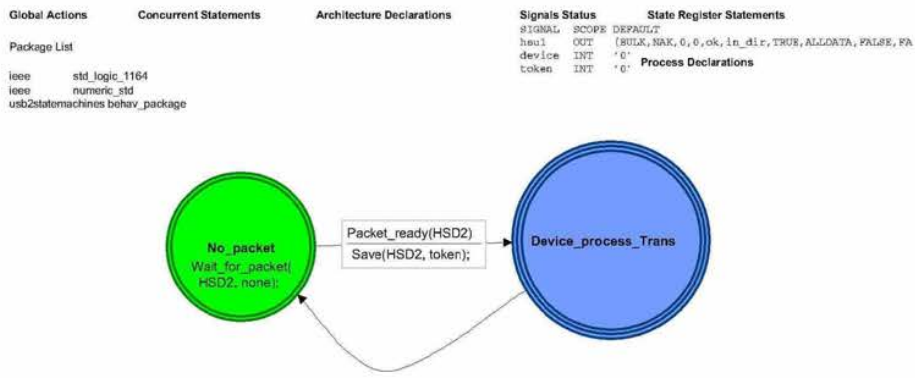

Figure 8-22. Device Top Level State Machine

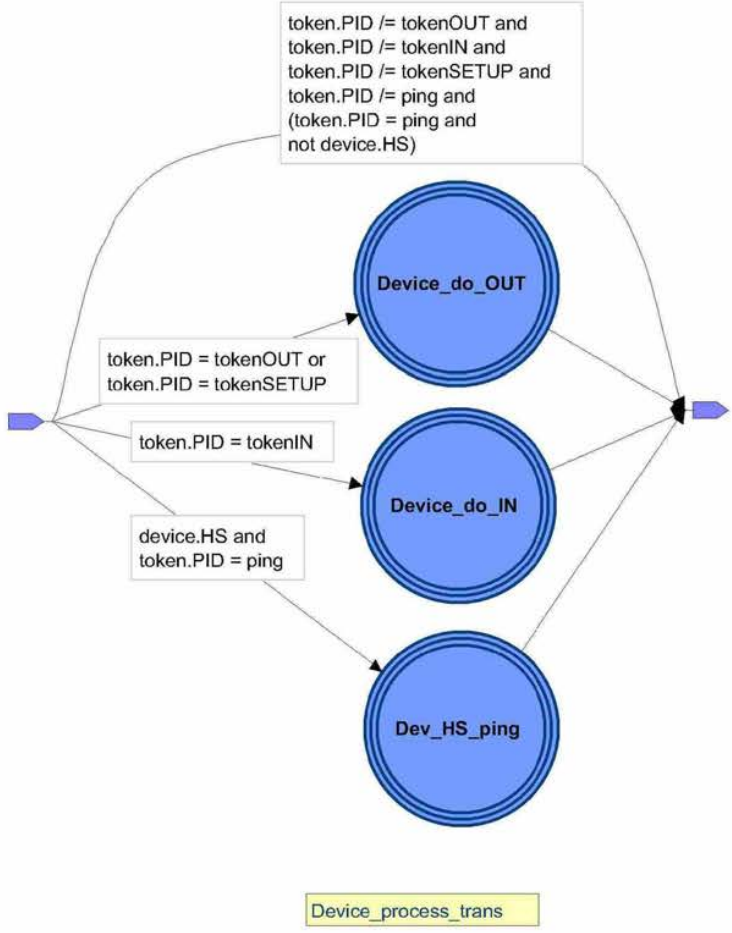

Figure 8-23. Device_process_Trans State Machine

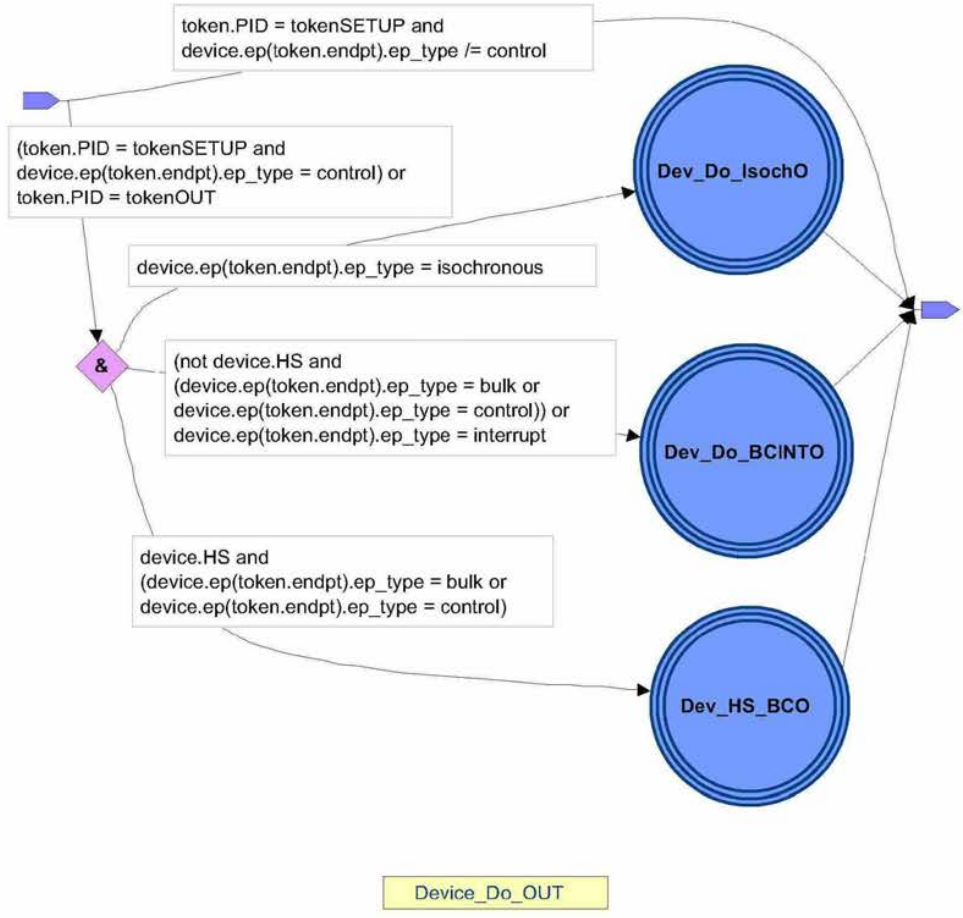

Figure 8-24. Dev_do_OUT State Machine

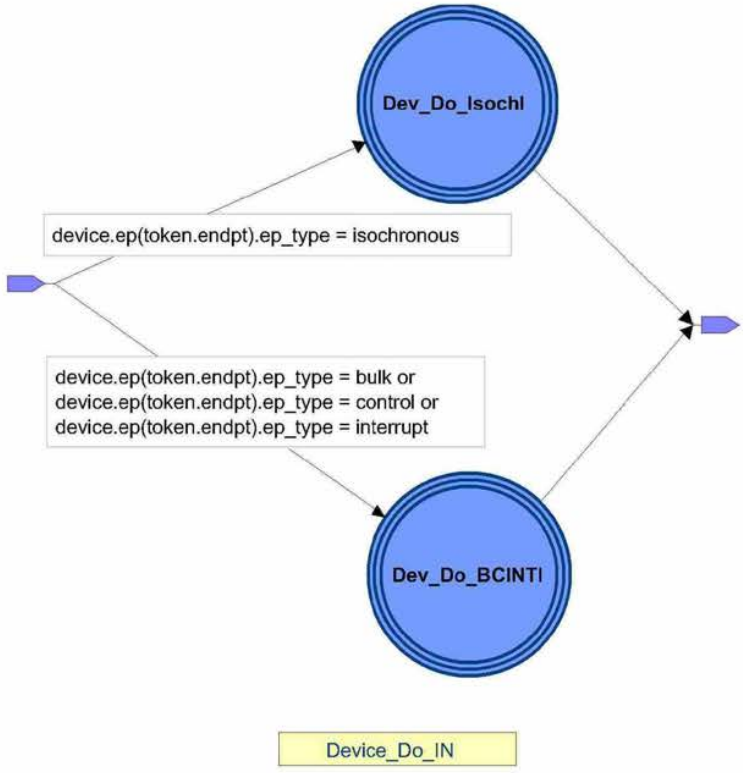

Figure 8-25. Dev_do_IN State Machine

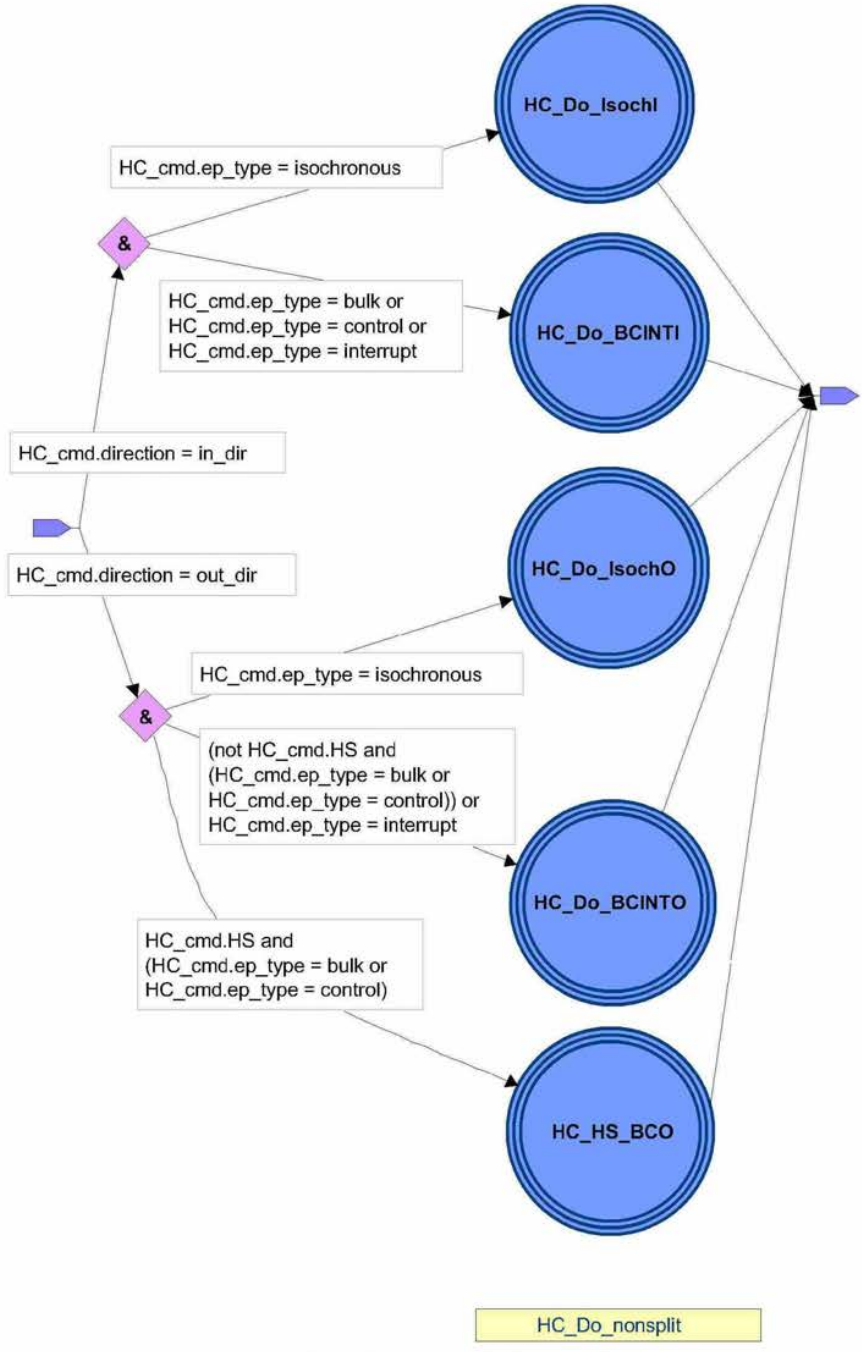

Figure 8-26. HC_Do_nonsplit State Machine

8.5.1 NAK Limiting via Ping Flow Control

Full-/low-speed devices can have bulk/control endpoints that take time to process their data and, therefore, respond to OUT transactions with a NAK handshake. This handshake response indicates that the endpoint did not accept the data because it did not have space for the data. The host controller is expected to retry the transaction at some future time when the endpoint has space available. Unfortunately, by the time the endpoint NAKs, most of the full-/low-speed bus time for the transaction had been used. This means that the full-/low-speed bus has poor utilization when there is a high frequency of NAK'd OUT transactions.

High-speed devices must support an improved NAK mechanism for Bulk OUT and Control endpoints and transactions. Control endpoints must support this protocol for an OUT transaction in the data and status stages. The control Setup stage must not support the PING protocol.

This mechanism allows the device to tell the host controller whether it has sufficient endpoint space for the next OUT transaction. If the device endpoint does not have space, the host controller can choose to delay a transaction attempt for this endpoint and instead try some other transaction. This can lead to improved bus utilization. The mechanism avoids using bus time to send data until the host controller knows that the endpoint has space for the data.

The host controller queries the high-speed device endpoint with a PING special token. The PING special token packet is a normal token packet as shown in Figure 8-5. The endpoint either responds to the PING with a NAK or an ACK handshake.

A NAK handshake indicates that the endpoint does not have space for a *wMaxPacketSize* data payload. The host controller will retry the PING at some future time to query the endpoint again. A device can respond to a PING with a NAK for long periods of time. A NAK response is not a reason for the host controller to retire a transfer request. If a device responds with a NAK in a (micro)frame, the host controller may choose to issue the next transaction in the next *bInterval* specified for the endpoint. However, the device must be prepared to receive PINGs as sequential transactions, e.g., one immediately after the other.

An ACK handshake indicates the endpoint has space for a *wMaxPacketSize* data payload. The host controller must generate an OUT transaction with a DATA phase as the next transaction to the endpoint. The host controller may generate other transactions to other devices or endpoints before the OUT/DATA transaction for this endpoint.

If the endpoint responds to the OUT/DATA transaction with an ACK handshake, this means the endpoint accepted the data successfully and has room for another *wMaxPacketSize* data payload. The host controller continues with OUT/DATA transactions (which are not required to be the next transactions on the bus) as long as it has transactions to generate.

If the endpoint instead responds to the OUT/DATA transaction with a NYET handshake, this means that the endpoint accepted the data but does not have room for another *wMaxPacketSize* data payload. The host controller must return to using a PING token until the endpoint indicates it has space.

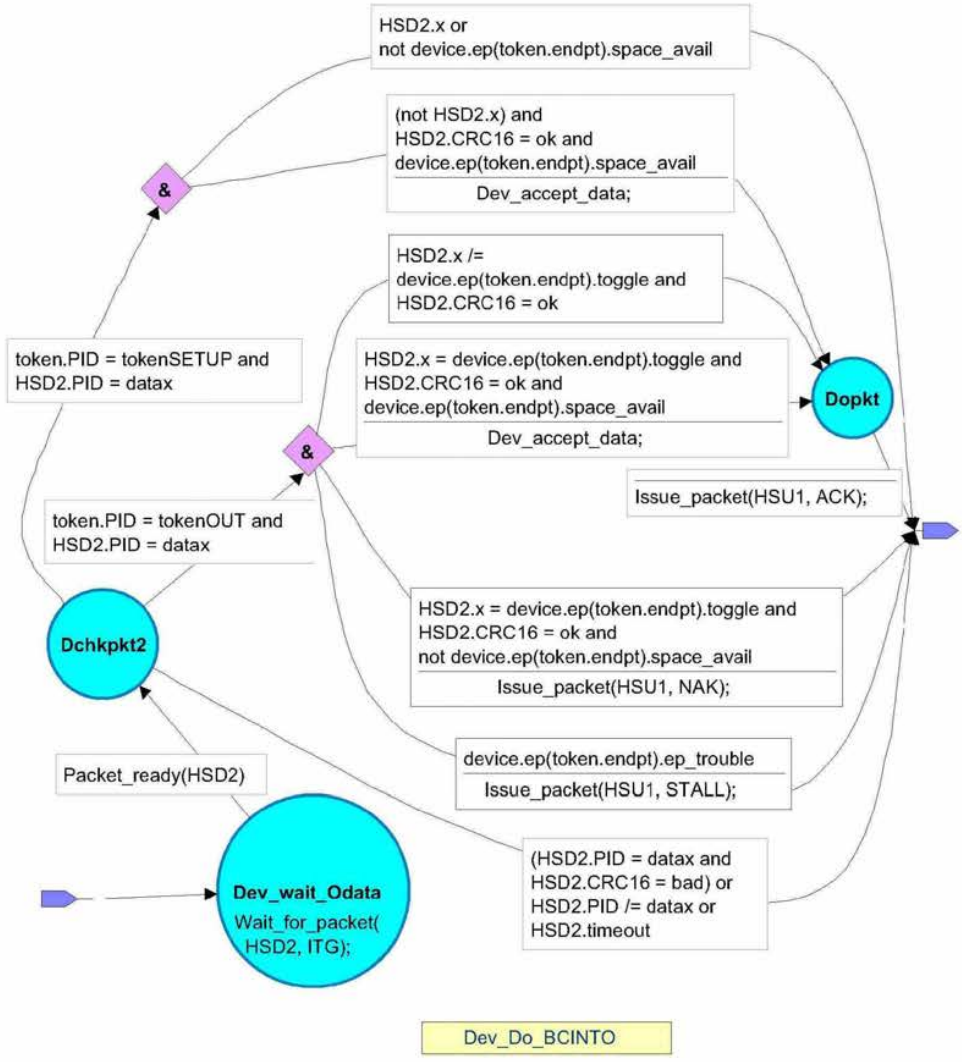

Figure 8-27. Host High-speed Bulk OUT/Control Ping State Machine

8.5.1.1 NAK Responses to OUT/DATA During PING Protocol

The endpoint may also respond to the OUT/DATA transaction with a NAK handshake. This means that the endpoint did not accept the data and does not have space for a *wMaxPacketSize* data payload at this time. The host controller must return to using a PING token until the endpoint indicates it has space.

A NAK response is expected to be an unusual occurrence. A high-speed bulk/control endpoint must specify its maximum NAK rate in its endpoint descriptor. The endpoint is allowed to NAK at most one time each *bInterval* period. A NAK suggests that the endpoint responded to a previous OUT or PING with an inappropriate handshake, or that the endpoint transitioned into a state where it (temporarily) could not

accept data. An endpoint can use a *bInterval* of zero to indicate that it never NAKs. An endpoint must always be able to accept a PING from the host, even if it never NAKs.

If a timeout occurs after the data phase, the host must return to using a PING token. Note that a transition back to the PING state does not affect the data toggle state of the transaction data phase.

Figure 8-27 shows the host controller state machine for the interactions and transitions between PING and OUT/DATA tokens and the allowed ACK, NAK, and NYET handshakes for the PING mechanism.

Figure 8-29 shows the device endpoint state machine for PING based on the buffer space the endpoint has available.

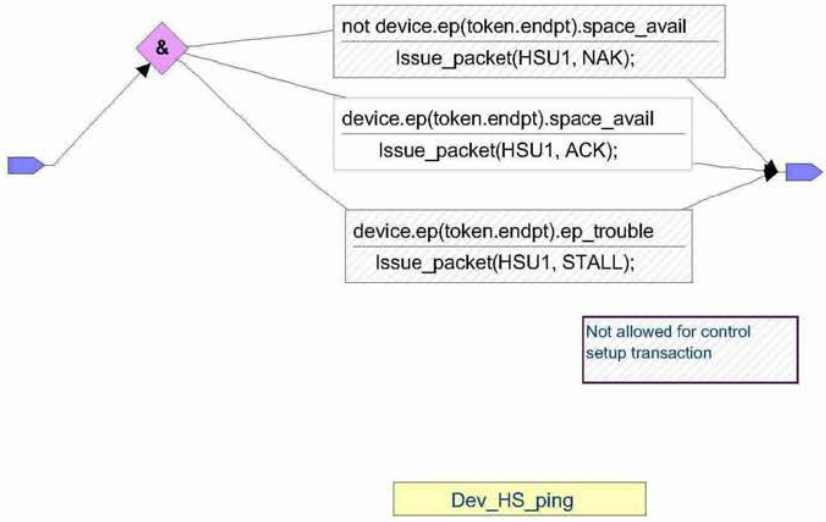

Figure 8-28. Dev_HS_ping State Machine

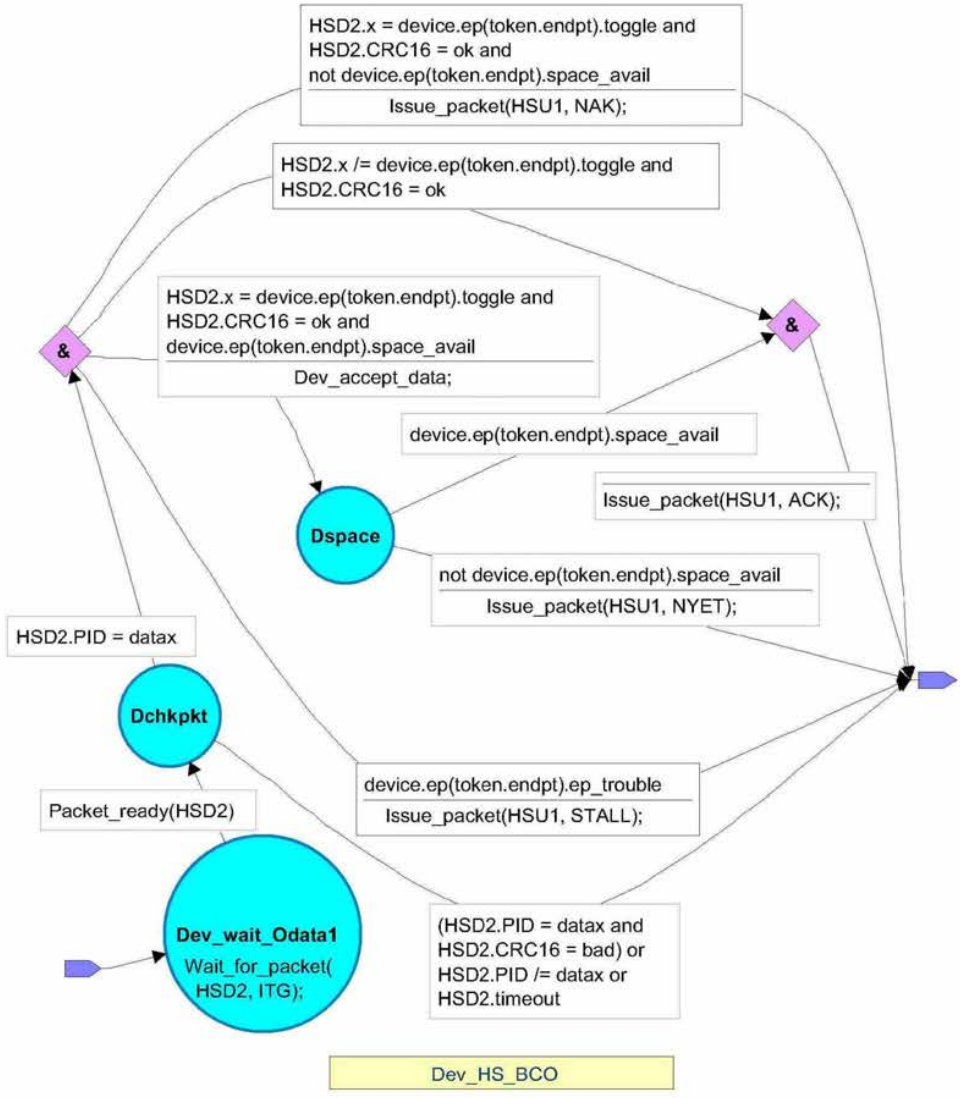

Figure 8-29. Device High-speed Bulk OUT /Control State Machine

Full-/low-speed devices/endpoints must not support the PING protocol. Host controllers must not support the PING protocol for full-/low-speed devices.

Note: The PING protocol is also not included as part of the split-transaction protocol definition. Some split-transactions have equivalent flow control without using PING. Other split-transactions will not benefit from PING as defined. In any case, split-transactions that can return a NAK handshake have small data payloads which should have minor high-speed bus impact. Hubs must support PING on their control endpoint, but PING is not defined for the split-transactions that are used to communicate with full-/low-speed devices supported by a hub.

8.5.2 Bulk Transactions

Bulk transaction types are characterized by the ability to guarantee error-free delivery of data between the host and a function by means of error detection and retry. Bulk transactions use a three-phase transaction consisting of token, data, and handshake packets as shown in Figure 8-30. Under certain flow control and halt conditions, the data phase may be replaced with a handshake resulting in a two-phase transaction in which no data is transmitted. The PING and NYET packets must only be used with devices operating at high-speed.

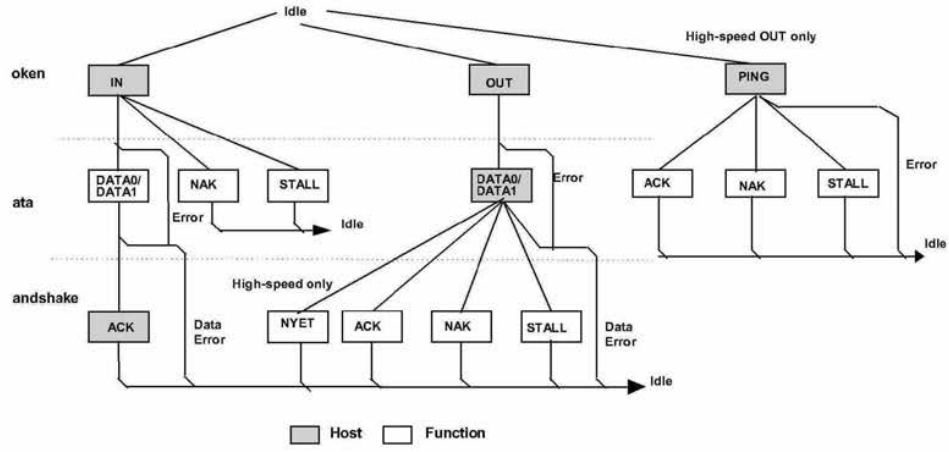

Figure 8-30. Bulk Transaction Format

When the host is ready to receive bulk data, it issues an IN token. The function endpoint responds by returning either a data packet or, should it be unable to return data, a NAK or STALL handshake. NAK indicates that the function is temporarily unable to return data, while STALL indicates that the endpoint is permanently halted and requires USB System Software intervention. If the host receives a valid data packet, it responds with an ACK handshake. If the host detects an error while receiving data, it returns no handshake packet to the function.

When the host is ready to transmit bulk data, it first issues an OUT token packet followed by a data packet (or PING special token packet, see Section 8.5.1). If the data is received without error by the function, it will return one of three (or four including NYET, for a device operating at high-speed) handshakes:

- ACK indicates that the data packet was received without errors and informs the host that it may send the next packet in the sequence.
- NAK indicates that the data was received without error but that the host should resend the data because the function was in a temporary condition preventing it from accepting the data (e.g., buffer full).
- If the endpoint was halted, STALL is returned to indicate that the host should not retry the transmission because there is an error condition on the function.

If the data packet was received with a CRC or bit stuff error, no handshake is returned.

Figure 8-31 and Figure 8-32 show the host and device state machines respectively for bulk, control, and interrupt OUT full/low-speed transactions. Figure 8-27, Figure 8-28, and Figure 8-29 show the state machines for high-speed transactions. Figure 8-33 and Figure 8-34 show the host and device state machines respectively for bulk, control, and interrupt IN transactions.

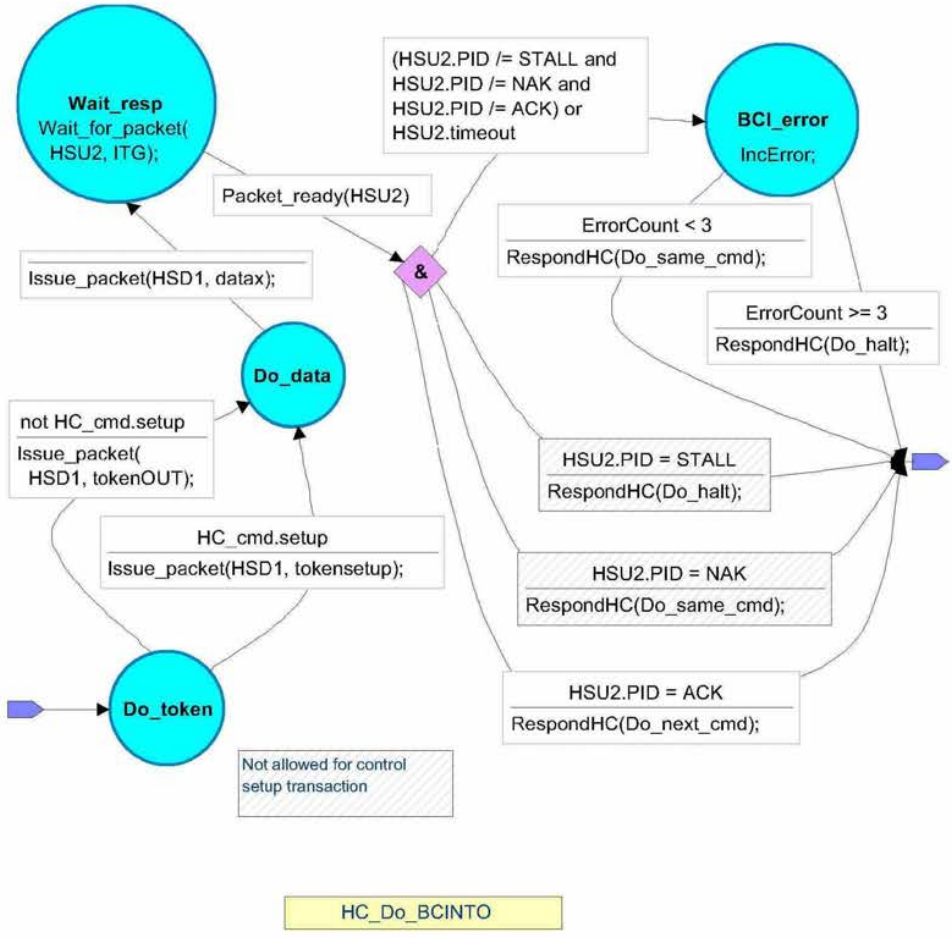

Figure 8-31. Bulk/Control/Interrupt OUT Transaction Host State Machine

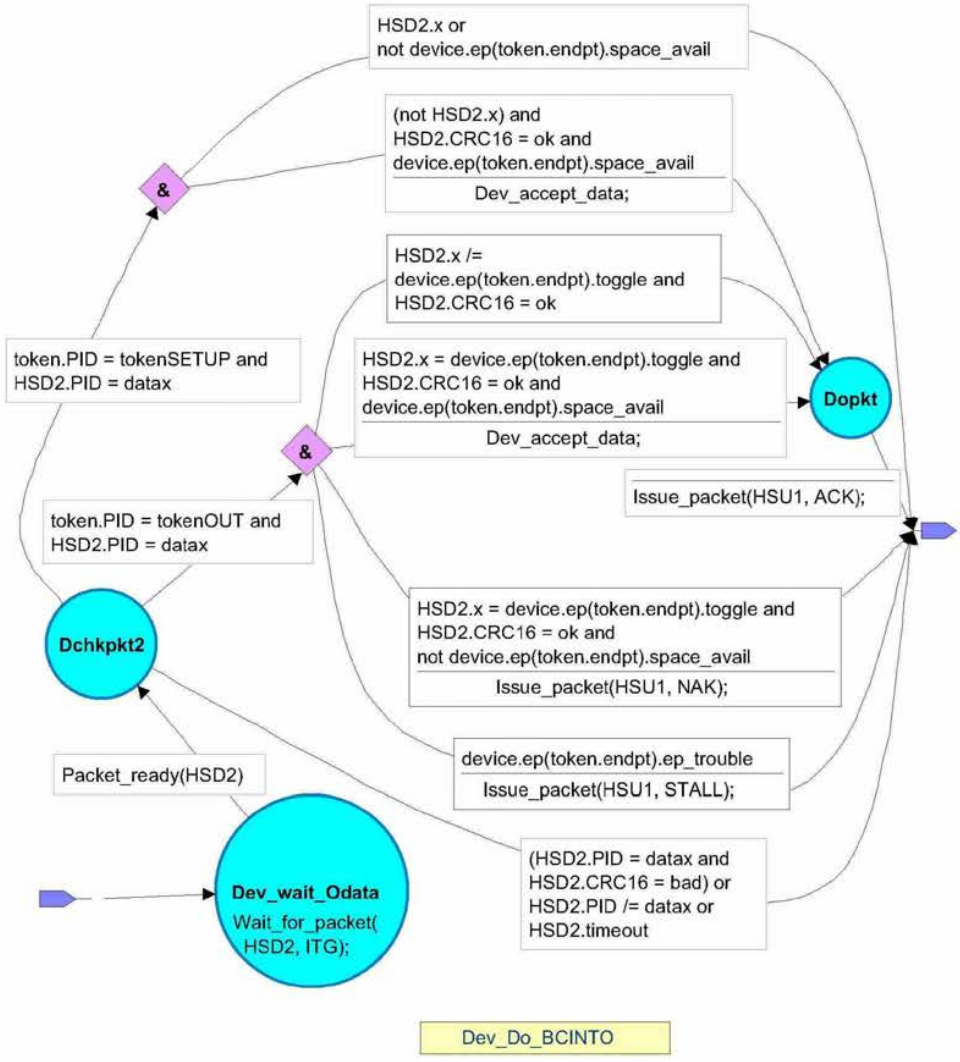

Figure 8-32. Bulk/Control/Interrupt OUT Transaction Device State Machine

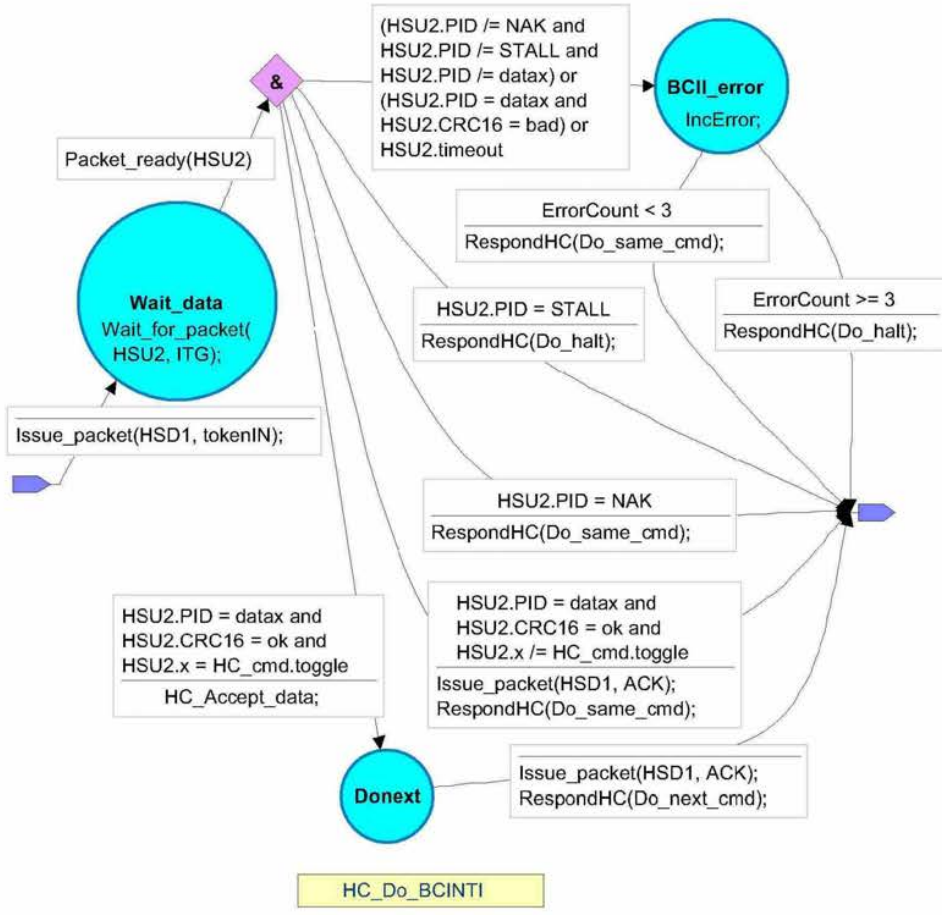

Figure 8-33. Bulk/Control/Interrupt IN Transaction Host State Machine

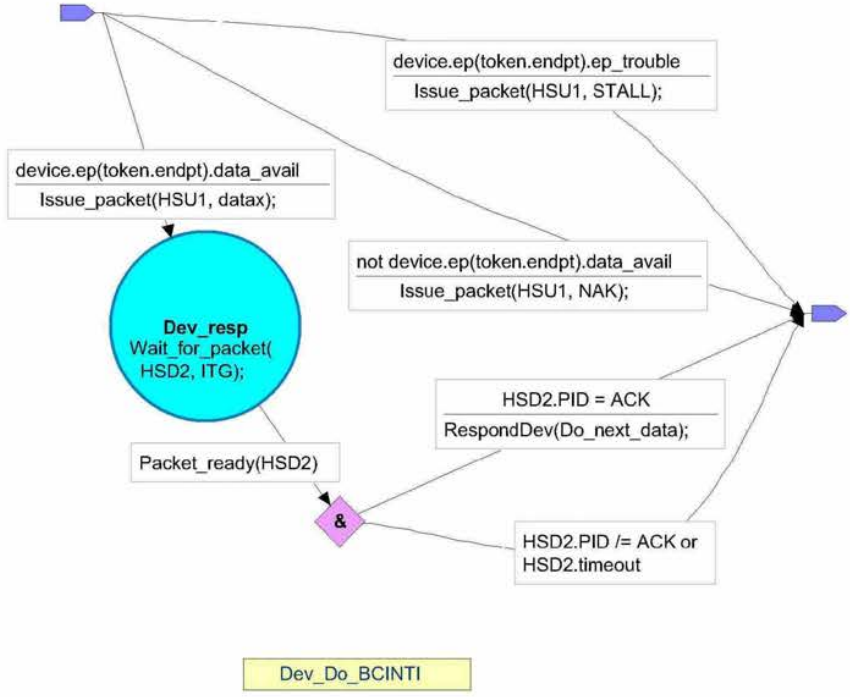

Figure 8-34. Bulk/Control/Interrupt IN Transaction Device State Machine

Figure 8-35 shows the sequence bit and data PID usage for bulk reads and writes. Data packet synchronization is achieved via use of the data sequence toggle bits and the DATA0/DATA1 PIDs. A bulk endpoint's toggle sequence is initialized to DATA0 when the endpoint experiences any configuration event (configuration events are explained in Sections 9.1.1.5 and 9.4.5). Data toggle on an endpoint is NOT initialized as the direct result of a short packet transfer or the retirement of an IRP.

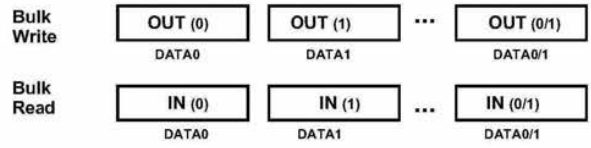

Figure 8-35. Bulk Reads and Writes

The host always initializes the first transaction of a bus transfer to the DATA0 PID with a configuration event. The second transaction uses a DATA1 PID, and successive data transfers alternate for the remainder of the bulk transfer. The data packet transmitter toggles upon receipt of ACK, and the receiver toggles upon receipt and acceptance of a valid data packet (refer to Section 8.6).

8.5.3 Control Transfers

Control transfers minimally have two transaction stages: Setup and Status. A control transfer may optionally contain a Data stage between the Setup and Status stages. During the Setup stage, a SETUP transaction is used to transmit information to the control endpoint of a function. SETUP transactions are similar in format to an OUT but use a SETUP rather than an OUT PID. Figure 8-36 shows the SETUP transaction format. A SETUP always uses a DATA0 PID for the data field of the SETUP transaction. The

function receiving a SETUP must accept the SETUP data and respond with ACK; if the data is corrupted, discard the data and return no handshake.

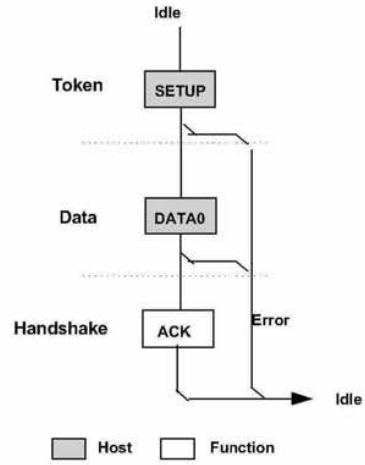

Figure 8-36. Control SETUP Transaction

The Data stage, if present, of a control transfer consists of one or more IN or OUT transactions and follows the same protocol rules as bulk transfers. All the transactions in the Data stage must be in the same direction (i.e., all INs or all OUTs). The amount of data to be sent during the data stage and its direction are specified during the Setup stage. If the amount of data exceeds the prenegotiated data packet size, the data is sent in multiple transactions (INs or OUTs) that carry the maximum packet size. Any remaining data is sent as a residual in the last transaction.

The Status stage of a control transfer is the last transaction in the sequence. The status stage transactions follow the same protocol sequence as bulk transactions. Status stage for devices operating at high-speed also includes the PING protocol. A Status stage is delineated by a change in direction of data flow from the previous stage and always uses a DATA1 PID. If, for example, the Data stage consists of OUTs, the status is a single IN transaction. If the control sequence has no Data stage, then it consists of a Setup stage followed by a Status stage consisting of an IN transaction.

Figure 8-37 shows the transaction order, the data sequence bit value, and the data PID types for control read and write sequences. The sequence bits are displayed in parentheses.

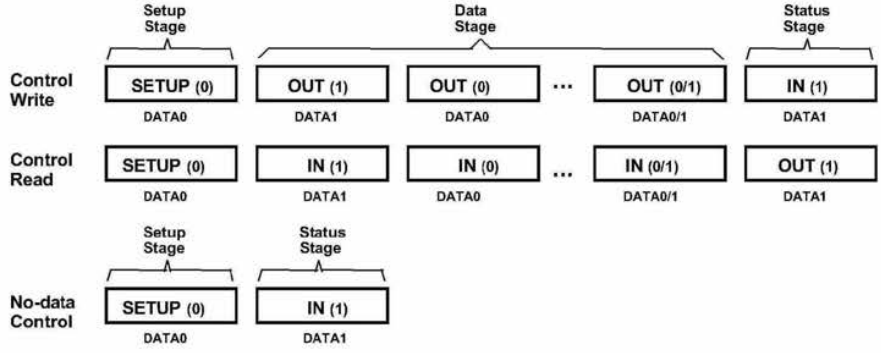

Figure 8-37. Control Read and Write Sequences

When a STALL handshake is sent by a control endpoint in either the Data or Status stages of a control transfer, a STALL handshake must be returned on all succeeding accesses to that endpoint until a SETUP PID is received. The endpoint is not required to return a STALL handshake after it receives a subsequent SETUP PID. For the default endpoint, if an ACK handshake is returned for the SETUP transaction, the host expects that the endpoint has automatically recovered from the condition that caused the STALL and the endpoint must operate normally.

8.5.3.1 Reporting Status Results

The Status stage reports to the host the outcome of the previous Setup and Data stages of the transfer. Three possible results may be returned:

- The command sequence completed successfully.
- The command sequence failed to complete.
- The function is still busy completing the command.

Status reporting is always in the function-to-host direction. Table 8-7 summarizes the type of responses required for each. Control write transfers return status information in the data phase of the Status stage transaction. Control read transfers return status information in the handshake phase of a Status stage transaction, after the host has issued a zero-length data packet during the previous data phase.

Table 8-7. Status Stage Responses

Status Response	Control Write Transfer (sent during data phase)	Control Read Transfer (sent during handshake phase)
Function completes	Zero-length data packet	ACK handshake
Function has an error	STALL handshake	STALL handshake
Function is busy	NAK handshake	NAK handshake

For control reads, the host must send either an OUT token or PING special token (for a device operating at high-speed) to the control pipe to initiate the Status stage. The host may only send a zero-length data packet in this phase but the function may accept any length packet as a valid status inquiry. The pipe's handshake response to this data packet indicates the current status. NAK indicates that the function is still processing the command and that the host should continue the Status stage. ACK indicates that the function has completed the command and is ready to accept a new command. STALL indicates that the function has an error that prevents it from completing the command.

For control writes, the host sends an IN token to the control pipe to initiate the Status stage. The function responds with either a handshake or a zero-length data packet to indicate its current status. NAK indicates that the function is still processing the command and that the host should continue the Status stage; return of a zero-length packet indicates normal completion of the command; and STALL indicates that the function cannot complete the command. The function expects the host to respond to the data packet in the Status stage with ACK. If the function does not receive ACK, it remains in the Status stage of the command and will continue to return the zero-length data packet for as long as the host continues to send IN tokens.

If during a Data stage a command pipe is sent more data or is requested to return more data than was indicated in the Setup stage (see Section 8.5.3.2), it should return STALL. If a control pipe returns STALL during the Data stage, there will be no Status stage for that control transfer.

8.5.3.2 Variable-length Data Stage

A control pipe may have a variable-length data phase in which the host requests more data than is contained in the specified data structure. When all of the data structure is returned to the host, the function should indicate that the Data stage is ended by returning a packet that is shorter than the *MaxPacketSize* for the pipe. If the data structure is an exact multiple of *wMaxPacketSize* for the pipe, the function will return a zero-length packet to indicate the end of the Data stage.

8.5.3.3 Error Handling on the Last Data Transaction

If the ACK handshake on an IN transaction is corrupted, the function and the host will temporarily disagree on whether the transaction was successful. If the transaction is followed by another IN, the toggle retry mechanism will detect the mismatch and recover from the error. If the ACK was on the last IN of a Data stage, the toggle retry mechanism cannot be used and an alternative scheme must be used.

The host that successfully received the data of the last IN will send ACK. Later, the host will issue an OUT token to start the Status stage of the transfer. If the function did not receive the ACK that ended the Data stage, the function will interpret the start of the Status stage as verification that the host successfully received the data. Control writes do not have this ambiguity. If an ACK handshake on an OUT gets corrupted, the host does not advance to the Status stage and retries the last data instead. A detailed analysis of retry policy is presented in Section 8.6.4.

8.5.3.4 STALL Handshakes Returned by Control Pipes

Control pipes have the unique ability to return a STALL handshake due to function problems in control transfers. If the device is unable to complete a command, it returns a STALL in the Data and/or Status stages of the control transfer. Unlike the case of a functional stall, protocol stall does not indicate an error with the device. The protocol STALL condition lasts until the receipt of the next SETUP transaction, and the function will return STALL in response to any IN or OUT transaction on the pipe until the SETUP transaction is received. In general, protocol stall indicates that the request or its parameters are not understood by the device and thus provides a mechanism for extending USB requests.

A control pipe may also support functional stall as well, but this is not recommended. This is a degenerative case, because a functional stall on a control pipe indicates that it has lost the ability to communicate with the host. If the control pipe does support functional stall, then it must possess a *Halt* feature, which can be set or cleared by the host. Chapter 9 details how to treat the special case of a *Halt* feature on a control pipe. A well-designed device will associate all of its functions and *Halt* features with non-control endpoints. The control pipes should be reserved for servicing USB requests.

8.5.4 Interrupt Transactions

Interrupt transactions may consist of IN or OUT transfers. Upon receipt of an IN token, a function may return data, NAK, or STALL. If the endpoint has no new interrupt information to return (i.e., no interrupt is pending), the function returns a NAK handshake during the data phase. If the *Halt* feature is set for the interrupt endpoint, the function will return a STALL handshake. If an interrupt is pending, the function returns the interrupt information as a data packet. The host, in response to receipt of the data packet, issues either an ACK handshake if data was received error-free or returns no handshake if the data packet was received corrupted. Figure 8-38 shows the interrupt transaction format.

Section 5.9.1 contains additional information about high-speed, high-bandwidth interrupt endpoints. Such endpoints use multiple transactions in a microframe as defined in that section. Each transaction for a high-bandwidth endpoint follows the transaction format shown in Figure 8-38.

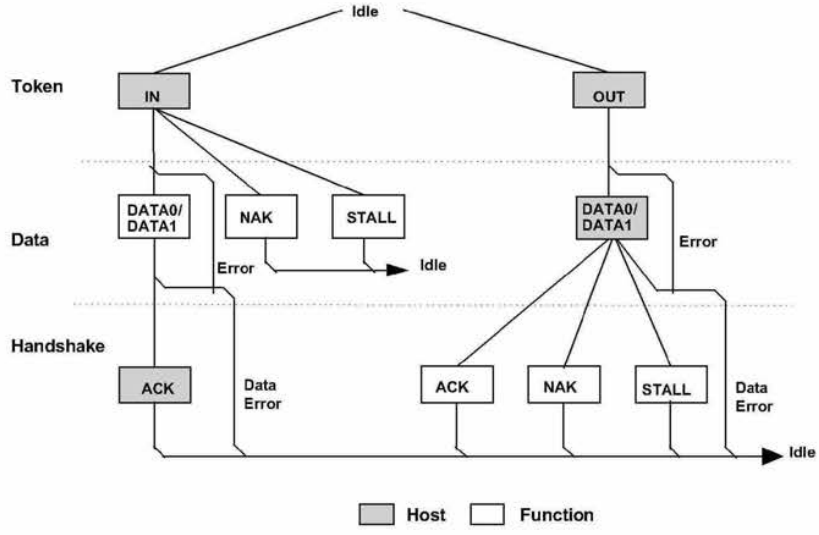

Figure 8-38. Interrupt Transaction Format

When an endpoint is using the interrupt transfer mechanism for actual interrupt data, the data toggle protocol must be followed. This allows the function to know that the data has been received by the host and the event condition may be cleared. This "guaranteed" delivery of events allows the function to only send the interrupt information until it has been received by the host rather than having to send the interrupt data every time the function is polled and until the USB System Software clears the interrupt condition. When used in the toggle mode, an interrupt endpoint is initialized to the DATA0 PID by any configuration event on the endpoint and behaves the same as the bulk transactions shown in Figure 8-35.

8.5.5 Isochronous Transactions

Isochronous transactions have a token and data phase, but no handshake phase, as shown in Figure 8-39. The host issues either an IN or an OUT token followed by the data phase in which the endpoint (for INs) or the host (for OUTs) transmits data. Isochronous transactions do not support a handshake phase or retry capability.

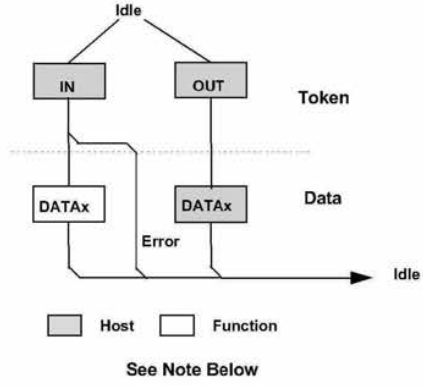

Figure 8-39. Isochronous Transaction Format

Note: A full-speed device or Host Controller should be able to accept either DATA0 or DATA1 PIDs in data packets. A full-speed device or Host Controller should only send DATA0 PIDs in data packets. A high-speed Host Controller must be able to accept and send DATA0, DATA1, DATA2, or MDATA PIDs in data packets. A high-speed device with at most 1 transaction per microframe must only send DATA0 PIDs in data packets. A high-speed device with high-bandwith endpoints (e.g., one that has more than 1 transaction per microframe) must be able to accept and/or send DATA0, DATA1, DATA2, or MDATA PIDs in data packets.

Full-speed isochronous transactions do not support toggle sequencing. High-speed isochronous transactions with a single transaction per microframe do not support toggle sequencing. High bandwidth, high-speed isochronous transactions support data PID sequencing (see Section 5.9.1 for more details).

Figure 8-40 and Figure 8-41 show the host and device state machines respectively for isochronous OUT transactions. Figure 8-42 and Figure 8-43 show the host and device state machines respectively for isochronous IN transactions.

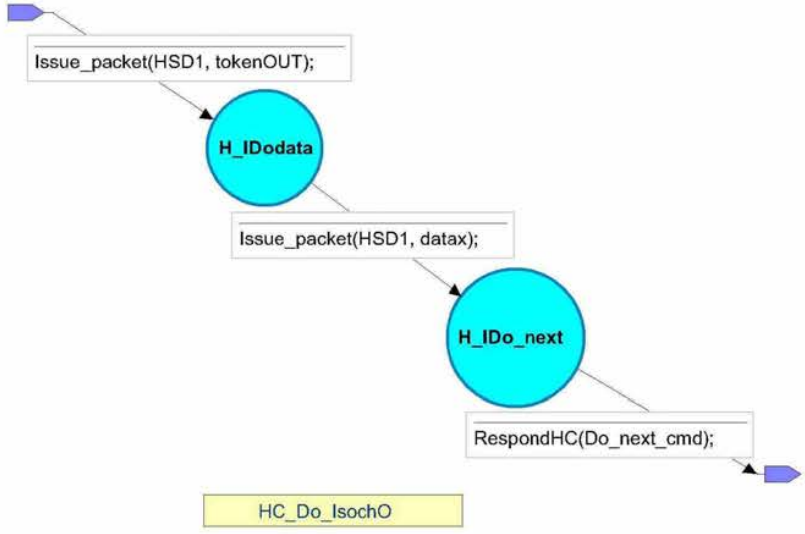

Figure 8-40. Isochronous OUT Transaction Host State Machine

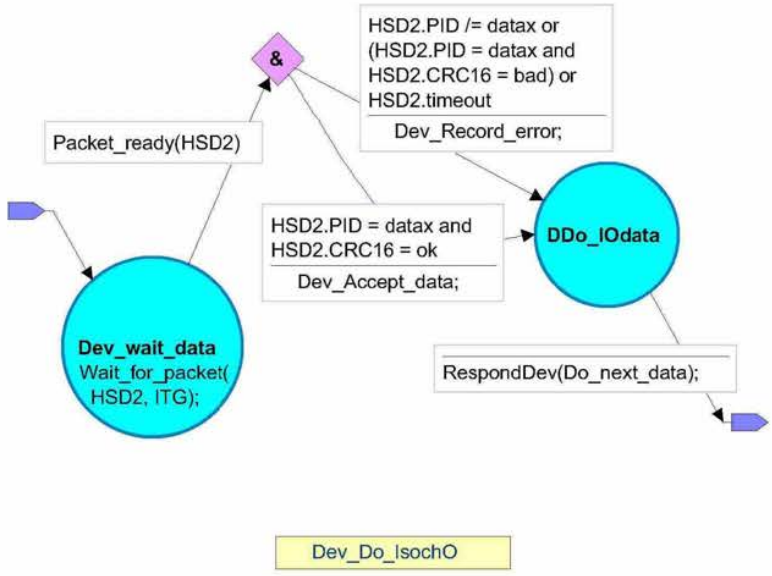

Figure 8-41. Isochronous OUT Transaction Device State Machine

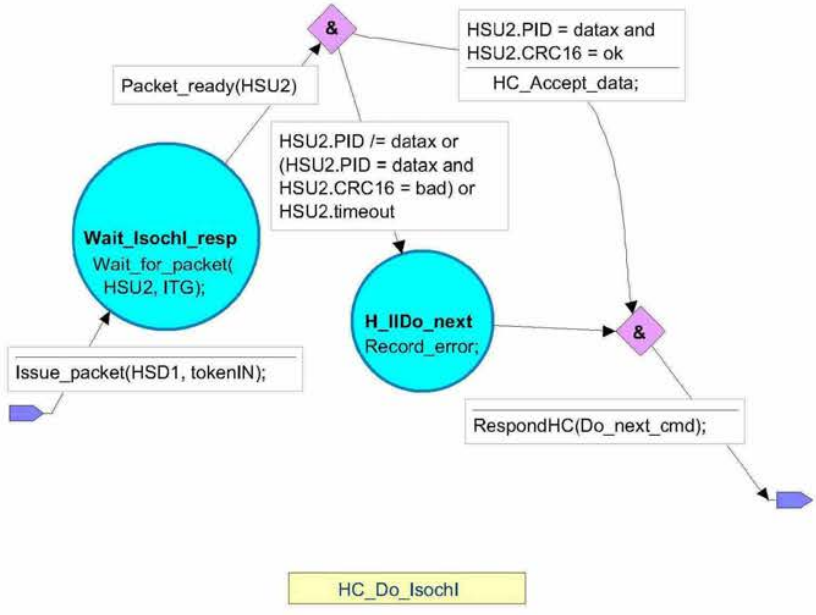

Figure 8-42. Isochronous IN Transaction Host State Machine

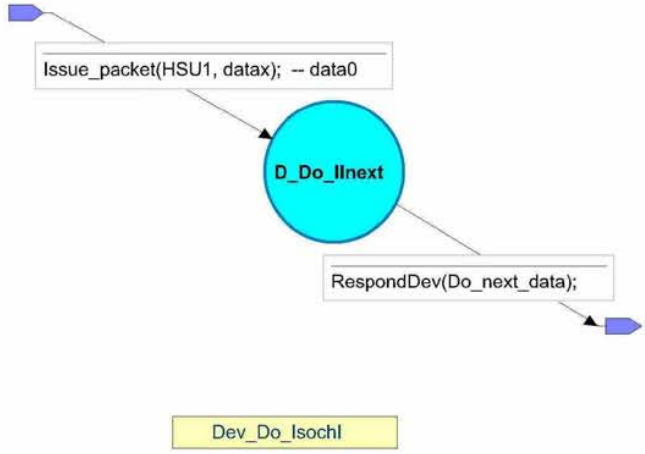

Figure 8-43. Isochronous IN Transaction Device State Machine

8.6 Data Toggle Synchronization and Retry

The USB provides a mechanism to guarantee data sequence synchronization between data transmitter and receiver across multiple transactions. This mechanism provides a means of guaranteeing that the handshake phase of a transaction was interpreted correctly by both the transmitter and receiver. Synchronization is achieved via use of the DATA0 and DATA1 PIDs and separate data toggle sequence bits for the data transmitter and receiver. Receiver sequence bits toggle only when the receiver is able to accept data and receives an error-free data packet with the correct data PID. Transmitter sequence bits toggle only when the data transmitter receives a valid ACK handshake. The data transmitter and receiver must have their sequence bits synchronized at the start of a transaction. The synchronization mechanism used varies with the transaction type. Data toggle synchronization is not supported for isochronous transfers.

The state machines contained in this chapter and in Chapter 11 describe data toggle synchronization in a more compact form. Instead of explicitly identifying DATA0 and DATA1, it uses a value "DATAx" to represent either/both DATA0/DATA1 PIDs. In some cases where the specific data PID is important, another variable labeled "x" is used that has the value 0 for DATA0 and 1 for DATA1.

High-speed, high-bandwidth isochronous and interrupt endpoints support a similar but different data synchronization technique called data PID sequencing. That technique is used instead of data toggle synchronization. Section 5.9.1 defines data PID sequencing.

8.6.1 Initialization via SETUP Token

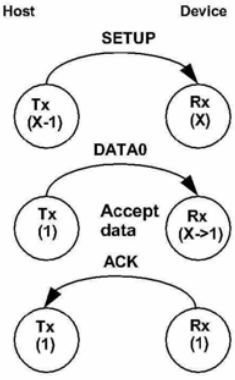

Figure 8-44. SETUP Initialization

Control transfers use the SETUP token for initializing host and function sequence bits. Figure 8-44 shows the host issuing a SETUP packet to a function followed by an OUT transaction. The numbers in the circles represent the transmitter and receiver sequence bits. The function must accept the data and return ACK. When the function accepts the transaction, it must set its sequence bit so that both the host's and function's sequence bits are equal to one at the end of the SETUP transaction.

8.6.2 Successful Data Transactions

Figure 8-45 shows the case where two successful transactions have occurred. For the data transmitter, this means that it toggles its sequence bit upon receipt of ACK. The receiver toggles its sequence bit only if it receives a valid data packet and the packet's data PID matches the current value of its sequence bit. The transmitter only toggles its sequence bit after it receives an ACK to a data packet.

During each transaction, the receiver compares the transmitter sequence bit (encoded in the data packet PID as either DATA0 or DATA1) with its receiver sequence bit. If data cannot be accepted, the receiver must issue NAK and the sequence bits of both the transmitter and receiver remain unchanged. If data can be accepted and the receiver's sequence bit matches the PID sequence bit, then data is accepted and the sequence bit is toggled. Two-phase transactions in which there is no data packet leave the transmitter and receiver sequence bits unchanged.

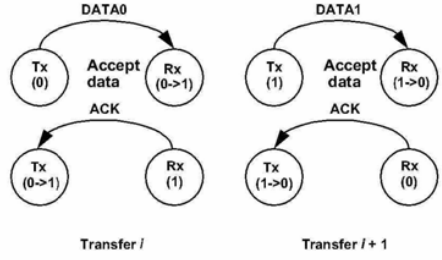

Figure 8-45. Consecutive Transactions

8.6.3 Data Corrupted or Not Accepted

If data cannot be accepted or the received data packet is corrupted, the receiver will issue a NAK or STALL handshake, or timeout, depending on the circumstances, and the receiver will not toggle its sequence bit.

Figure 8-46 shows the case where a transaction is NAKed and then retried. Any non-ACK handshake or timeout will generate similar retry behavior. The transmitter, having not received an ACK handshake, will not toggle its sequence bit. As a result, a failed data packet transaction leaves the transmitter's and receiver's sequence bits synchronized and untoggled. The transaction will then be retried and, if successful, will cause both transmitter and receiver sequence bits to toggle.

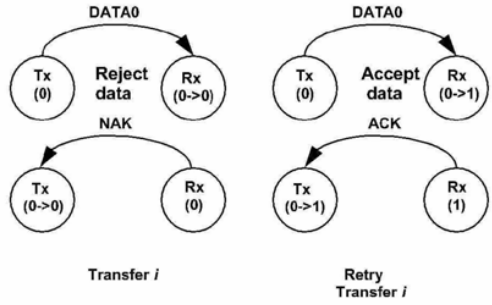

Figure 8-46. NAKed Transaction with Retry

8.6.4 Corrupted ACK Handshake

The transmitter is the last and only agent to know for sure whether a transaction has been successful, due to its receiving an ACK handshake. A lost or corrupted ACK handshake can lead to a temporary loss of synchronization between transmitter and receiver as shown in Figure 8-47. Here the transmitter issues a valid data packet, which is successfully acquired by the receiver; however, the ACK handshake is corrupted.

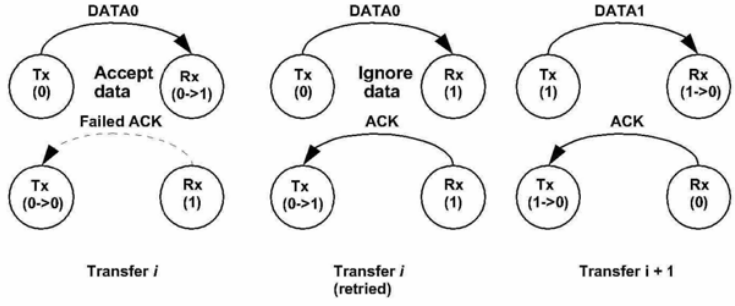

Figure 8-47. Corrupted ACK Handshake with Retry

At the end of transaction *i*, there is a temporary loss of coherency between transmitter and receiver, as evidenced by the mismatch between their respective sequence bits. The receiver has received good data, but the transmitter does not know whether it has successfully sent data. On the next transaction, the transmitter will resend the previous data using the previous DATA0 PID. The receiver's sequence bit and the data PID will not match, so the receiver knows that it has previously accepted this data. Consequently, it discards the incoming data packet and does not toggle its sequence bit. The receiver then issues ACK, which causes the transmitter to regard the retried transaction as successful. Receipt of ACK causes the transmitter to toggle its sequence bit. At the beginning of transaction *i*+1, the sequence bits have toggled and are again synchronized.

The data transmitter must guarantee that any retried data packet is identical (same length and content) as that sent in the original transaction. If the data transmitter is unable, because of problems such as a buffer underrun condition, to transmit the identical amount of data as was in the original data packet, it must abort

the transaction by generating a bit stuffing violation for full-/low-speed. An error for high-speed must be forced by taking the currently calculated CRC and complementing it before transmitting it. This causes a detectable error at the receiver and guarantees that a partial packet will not be interpreted as a good packet. The transmitter should not try to force an error at the receiver by sending a constant known bad CRC. A combination of a bad packet with a "bad" CRC may be interpreted by the receiver as a good packet.

8.6.5 Low-speed Transactions

The USB supports signaling at three speeds: high-speed signaling at 480 Mb/s, full-speed signaling at 12.0 Mb/s, and low-speed signaling at 1.5 Mb/s. Hubs isolate high-speed signaling from full-/low-speed signaling environments.

Within a full-/low-speed signaling environment, hubs disable downstream bus traffic to all ports to which low-speed devices are attached during full-speed downstream signaling. This is required both for EMI reasons and to prevent any possibility that a low-speed device might misinterpret downstream a full-speed packet as being addressed to it.

Figure 8-48 shows an IN low-speed transaction in which the host (or TT) issues a token and handshake and receives a data packet.

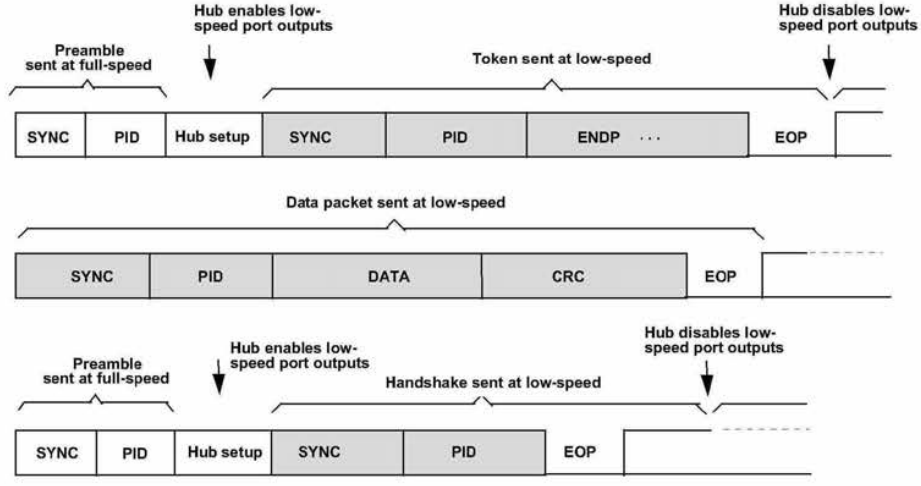

Figure 8-48. Low-speed Transaction

All downstream packets transmitted to low-speed devices within a full-/low-speed signaling environment require a preamble. Preambles are never used in a high-speed signaling environment. The preamble consists of a SYNC followed by a PRE PID, both sent at full-speed. Hubs must comprehend the PRE PID; all other USB devices may ignore it and treat it as undefined. At the end of the preamble PID, the host (or TT) drives the bus to the Idle state for at least one full-speed bit time. This Idle period on the bus is termed the hub setup interval and lasts for at least four full-speed bit times. During this hub setup interval, hubs must drive their full-speed and low-speed ports to their respective Idle states. Hubs must be ready to repeat low-speed signaling on low-speed ports before the end of the hub setup interval. Low-speed connectivity rules are summarized below:

1. Low-speed devices are identified during the connection process, and the hub ports to which they are connected are identified as low-speed.
2. All downstream low-speed packets must be prefaced with a preamble (sent at full-speed), which turns on the output buffers on low-speed hub ports.

3. Low-speed hub port output buffers are turned off upon receipt of EOP and are not turned on again until a preamble PID is detected.
4. Upstream connectivity is not affected by whether a hub port is full- or low-speed.

Low-speed signaling begins with the host (or TT) issuing SYNC at low-speed, followed by the remainder of the packet. The end of the packet is identified by an End-of-Packet (EOP), at which time all hubs tear down connectivity and disable any ports to which low-speed devices are connected. Hubs do not switch ports for upstream signaling; low-speed ports remain enabled in the upstream direction for both low-speed and full-speed signaling.

Low-speed and full-speed transactions maintain a high degree of protocol commonality. However, low-speed signaling does have certain limitations which include:

- Data payload is limited to eight bytes, maximum.
- Only interrupt and control types of transfers are supported.
- The SOF packet is not received by low-speed devices.

8.7 Error Detection and Recovery

The USB permits reliable end-to-end communication in the presence of errors on the physical signaling layer. This includes the ability to reliably detect the vast majority of possible errors and to recover from errors on a transaction-type basis. Control transactions, for example, require a high degree of data reliability; they support end-to-end data integrity using error detection and retry. Isochronous transactions, by virtue of their bandwidth and latency requirements, do not permit retries and must tolerate a higher incidence of uncorrected errors.

8.7.1 Packet Error Categories

The USB employs three error detection mechanisms: bit stuff violations, PID check bits, and CRCs. Bit stuff violations are defined in Section 7.1.9. PID errors are defined in Section 8.3.1. CRC errors are defined in Section 8.3.5.

With the exception of the SOF token, any packet that is received corrupted causes the receiver to ignore it and discard any data or other field information that came with the packet. Table 8-8 lists error detection mechanisms, the types of packets to which they apply, and the appropriate packet receiver response.

Table 8-8. Packet Error Types

Field	Error	Action
PID	PID Check, Bit Stuff	Ignore packet
Address	Bit Stuff, Address CRC	Ignore token
Frame Number	Bit Stuff, Frame Number CRC	Ignore Frame Number field
Data	Bit Stuff, Data CRC	Discard data

8.7.2 Bus Turn-around Timing

Neither the device nor the host will send an indication that a received packet had an error. This absence of positive acknowledgement is considered to be the indication that there was an error. As a consequence of this method of error reporting, the host and USB function need to keep track of how much time has elapsed from when the transmitter completes sending a packet until it begins to receive a response packet. This time is referred to as the bus turn-around time. Devices and hosts require turn-around timers to measure this time.

For full-/low-speed transactions, the timer starts counting on the SE0-to-'J' transition of the EOP strobe and stops counting when the Idle-to-'K' SOP transition is detected. For high-speed transactions, the timer starts counting when the data lines return to the squelch level and stops counting when the data lines leave the squelch level.

The device bus turn-around time is defined by the worst case round trip delay plus the maximum device response delay (refer to Sections 7.1.18 and 7.1.19 for specific bus turn-around times). If a response is not received within this worst case timeout, then the transmitter considers that the packet transmission has failed.

Timeout is used and interpreted as a transaction error condition for many transfer types. If the host wishes to indicate an error condition for a transaction via a timeout, it must wait the full bus turn-around time before issuing the next token to ensure that all downstream devices have timed out.

As shown in Figure 8-49, the device uses its bus turn-around timer between token and data or data and handshake phases. The host uses its timer between data and handshake or token and data phases.

If the host receives a corrupted data packet, it may require additional wait time before sending out the next token. This additional wait interval guarantees that the host properly handles false EOPs.

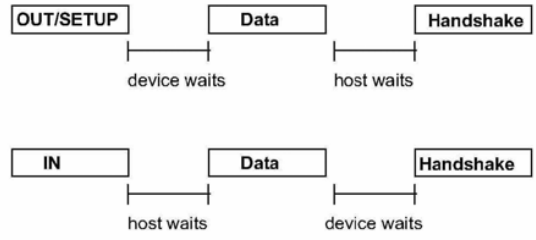

Figure 8-49. Bus Turn-around Timer Usage

8.7.3 False EOPs

False EOPs must be handled in a manner which guarantees that the packet currently in progress completes before the host or any other device attempts to transmit a new packet. If such an event were to occur, it would constitute a bus collision and have the ability to corrupt up to two consecutive transactions. Detection of false EOP relies upon the fact that a packet into which a false EOP has been inserted will appear as a truncated packet with a CRC failure. (The last 16 bits of the data packet will have a very low probability of appearing to be a correct CRC.)

The host and devices handle false EOP situations differently. When a device receives a corrupted data packet, it issues no response and waits for the host to send the next token. This scheme guarantees that the device will not attempt to return a handshake while the host may still be transmitting a data packet. If a false EOP has occurred, the host data packet will eventually end, and the device will be able to detect the next token. If a device issues a data packet that gets corrupted with a false EOP, the host will ignore the

packet and not issue the handshake. The device, expecting to see a handshake from the host, will timeout the transaction.

If the host receives a corrupted full-/low-speed data packet, it assumes that a false EOP may have occurred and waits for 16 bit times to see if there is any subsequent upstream traffic. If no bus transitions are detected within the 16 bit interval and the bus remains in the Idle state, the host may issue the next token.

Otherwise, the host waits for the device to finish sending the remainder of its full-/low-speed packet. Waiting 16 bit times guarantees two conditions:

- The first condition is to make sure that the device has finished sending its packet. This is guaranteed by a timeout interval (with no bus transitions) greater than the worst case six-bit time bit stuff interval.
- The second condition is that the transmitting device's bus turn-around timer must be guaranteed to expire.

Note that the timeout interval is transaction speed sensitive. For full-speed transactions, the host must wait full-speed bit times; for low-speed transactions, it must wait low-speed bit times.

If the host receives a corrupted high-speed data packet, it ignores any data until the data lines return to the squelch level before issuing the next token. For high-speed transactions, the host does not need to wait additional time (beyond the normal inter-transaction gap time) after the data lines return to the squelch level.

If the host receives a data packet with a valid CRC, it assumes that the packet is complete and requires no additional delay (beyond normal inter-transaction gap time) in issuing the next token.

8.7.4 Babble and Loss of Activity Recovery

The USB must be able to detect and recover from conditions which leave it waiting indefinitely for a full-/low-speed EOP or which leave the bus in something other than the Idle state at the end of a (micro)frame.

- Full-/low-speed loss of activity (LOA) is characterized by an SOP followed by lack of bus activity (bus remains driven to a 'J' or 'K') and no EOP at the end of a frame.
- Full-/low-speed babble is characterized by an SOP followed by the presence of bus activity past the end of a frame.
- High-speed babble/LOA is characterized by the data lines being at an unsquelched level at the end of a microframe.

LOA and babble have the potential to either deadlock the bus or delay the beginning of the next (micro)frame. Neither condition is acceptable, and both must be prevented from occurring. As the USB component responsible for controlling connectivity, hubs are responsible for babble/LOA detection and recovery. All USB devices that fail to complete their transmission at the end of a (micro)frame are prevented from transmitting past a (micro)frame's end by having the nearest hub disable the port to which the offending device is attached. Details of the hub babble/LOA recovery mechanism appear in Section 11.2.5.

Chapter 9
USB Device Framework

A USB device may be divided into three layers:

- The bottom layer is a bus interface that transmits and receives packets.
- The middle layer handles routing data between the bus interface and various endpoints on the device. An endpoint is the ultimate consumer or provider of data. It may be thought of as a source or sink for data.
- The top layer is the functionality provided by the serial bus device, for instance, a mouse or ISDN interface.

This chapter describes the common attributes and operations of the middle layer of a USB device. These attributes and operations are used by the function-specific portions of the device to communicate through the bus interface and ultimately with the host.

9.1 USB Device States

A USB device has several possible states. Some of these states are visible to the USB and the host, while others are internal to the USB device. This section describes those states.

9.1.1 Visible Device States

This section describes USB device states that are externally visible (see Figure 9-1). Table 9-1 summarizes the visible device states.

Note: USB devices perform a reset operation in response to reset signaling on the upstream facing port. When reset signaling has completed, the USB device is reset.

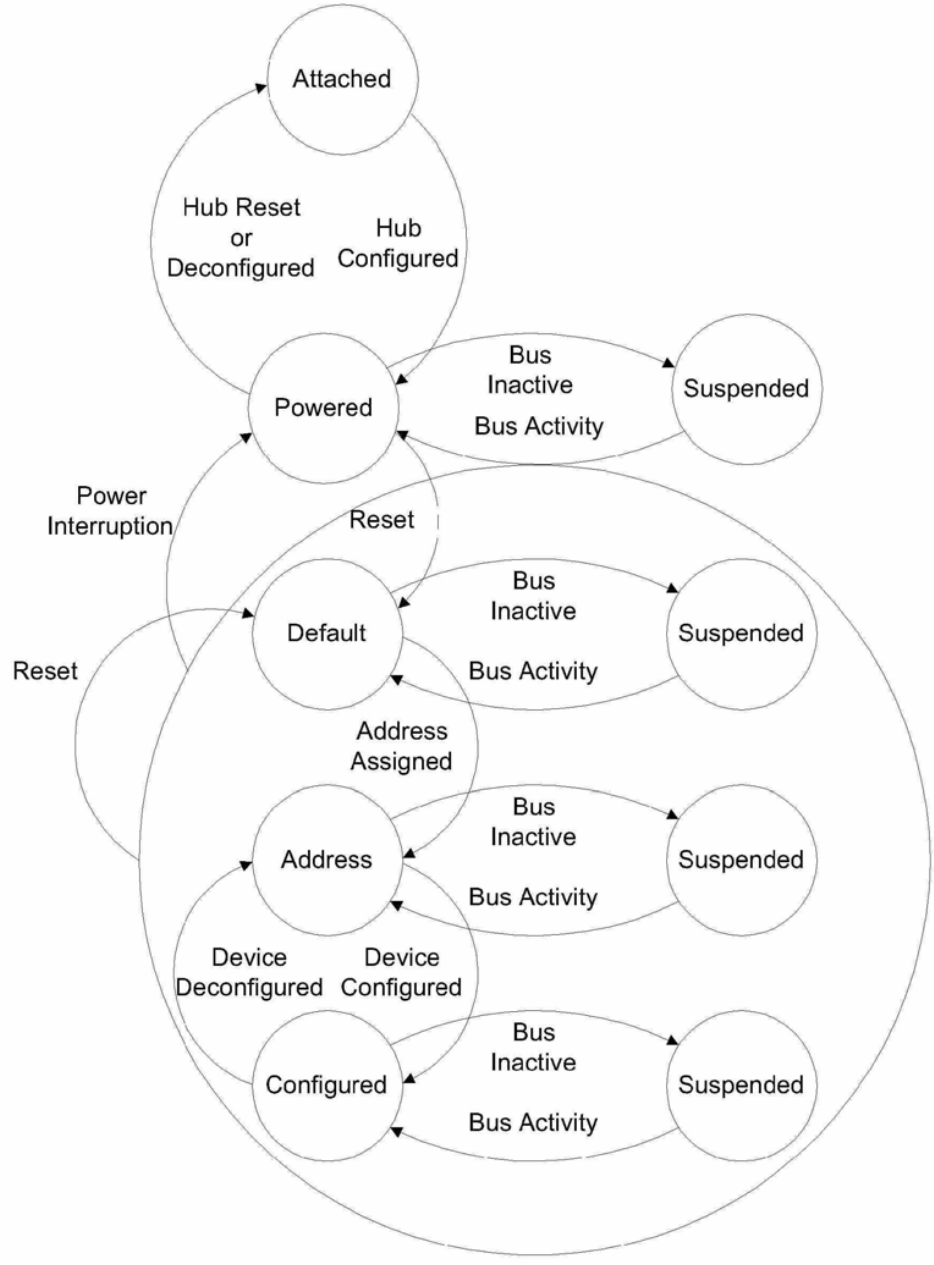

Figure 9-1. Device State Diagram

Table 9-1. Visible Device States

Attached	Powered	Default	Address	Configured	Suspended	State
No	--	--	--	--	--	Device is not attached to the USB. Other attributes are not significant.
Yes	No	--	--	--	--	Device is attached to the USB, but is not powered. Other attributes are not significant.
Yes	Yes	No	--	--	--	Device is attached to the USB and powered, but has not been reset.
Yes	Yes	Yes	No	--	--	Device is attached to the USB and powered and has been reset, but has not been assigned a unique address. Device responds at the default address.
Yes	Yes	Yes	Yes	No	--	Device is attached to the USB, powered, has been reset, and a unique device address has been assigned. Device is not configured.
Yes	Yes	Yes	Yes	Yes	No	Device is attached to the USB, powered, has been reset, has a unique address, is configured, and is not suspended. The host may now use the function provided by the device.
Yes	Yes	--	--	--	Yes	Device is, at minimum, attached to the USB and is powered and has not seen bus activity for 3 ms. It may also have a unique address and be configured for use. However, because the device is suspended, the host may not use the device's function.

9.1.1.1 Attached

A USB device may be attached or detached from the USB. The state of a USB device when it is detached from the USB is not defined by this specification. This specification only addresses required operations and attributes once the device is attached.

9.1.1.2 Powered

USB devices may obtain power from an external source and/or from the USB through the hub to which they are attached. Externally powered USB devices are termed self-powered. Although self-powered devices may already be powered before they are attached to the USB, they are not considered to be in the Powered state until they are attached to the USB and VBUS is applied to the device.

A device may support both self-powered and bus-powered configurations. Some device configurations support either power source. Other device configurations may be available only if the device is self-powered. Devices report their power source capability through the configuration descriptor. The current power source is reported as part of a device's status. Devices may change their power source at any time, e.g., from self- to bus-powered. If a configuration is capable of supporting both power modes, the power maximum reported for that configuration is the maximum the device will draw from VBUS in either mode. The device must observe this maximum, regardless of its mode. If a configuration supports only one power mode and the power source of the device changes, the device will lose its current configuration and address and return to the Powered state. If a device is self-powered and its current configuration requires more than 100 mA, then if the device switches to being bus-powered, it must return to the Address state. Self-powered hubs that use VBUS to power the Hub Controller are allowed to remain in the Configured state if local power is lost. Refer to Section 11.13 for details.

A hub port must be powered in order to detect port status changes, including attach and detach. Bus-powered hubs do not provide any downstream power until they are configured, at which point they will provide power as allowed by their configuration and power source. A USB device must be able to be addressed within a specified time period from when power is initially applied (refer to Chapter 7). After an attachment to a port has been detected, the host may enable the port, which will also reset the device attached to the port.

9.1.1.3 Default

After the device has been powered, it must not respond to any bus transactions until it has received a reset from the bus. After receiving a reset, the device is then addressable at the default address.

When the reset process is complete, the USB device is operating at the correct speed (i.e., low-/full-/high-speed). The speed selection for low- and full-speed is determined by the device termination resistors. A device that is capable of high-speed operation determines whether it will operate at high-speed as a part of the reset process (see Chapter 7 for more details).

A device capable of high-speed operation must reset successfully at full-speed when in an electrical environment that is operating at full-speed. After the device is successfully reset, the device must also respond successfully to device and configuration descriptor requests and return appropriate information. The device may or may not be able to support its intended functionality when operating at full-speed.

9.1.1.4 Address

All USB devices use the default address when initially powered or after the device has been reset. Each USB device is assigned a unique address by the host after attachment or after reset. A USB device maintains its assigned address while suspended.

A USB device responds to requests on its default pipe whether the device is currently assigned a unique address or is using the default address.

9.1.1.5 Configured

Before a USB device's function may be used, the device must be configured. From the device's perspective, configuration involves correctly processing a SetConfiguration() request with a non-zero configuration value. Configuring a device or changing an alternate setting causes all of the status and configuration values associated with endpoints in the affected interfaces to be set to their default values. This includes setting the data toggle of any endpoint using data toggles to the value DATA0.

9.1.1.6 Suspended

In order to conserve power, USB devices automatically enter the Suspended state when the device has observed no bus traffic for a specified period (refer to Chapter 7). When suspended, the USB device maintains any internal status, including its address and configuration.

All devices must suspend if bus activity has not been observed for the length of time specified in Chapter 7. Attached devices must be prepared to suspend at any time they are powered, whether they have been assigned a non-default address or are configured. Bus activity may cease due to the host entering a suspend mode of its own. In addition, a USB device shall also enter the Suspended state when the hub port it is attached to is disabled. This is referred to as selective suspend.

A USB device exits suspend mode when there is bus activity. A USB device may also request the host to exit suspend mode or selective suspend by using electrical signaling to indicate remote wakeup. The ability of a device to signal remote wakeup is optional. If a USB device is capable of remote wakeup signaling, the device must support the ability of the host to enable and disable this capability. When the device is reset, remote wakeup signaling must be disabled.

9.1.2 Bus Enumeration

When a USB device is attached to or removed from the USB, the host uses a process known as bus enumeration to identify and manage the device state changes necessary. When a USB device is attached to a powered port, the following actions are taken:

1. The hub to which the USB device is now attached informs the host of the event via a reply on its status change pipe (refer to Section 11.12.3 for more information). At this point, the USB device is in the Powered state and the port to which it is attached is disabled.
2. The host determines the exact nature of the change by querying the hub.
3. Now that the host knows the port to which the new device has been attached, the host then waits for at least 100 ms to allow completion of an insertion process and for power at the device to become stable. The host then issues a port enable and reset command to that port. Refer to Section 7.1.7.5 for sequence of events and timings of connection through device reset.
4. The hub performs the required reset processing for that port (see Section 11.5.1.5). When the reset signal is released, the port has been enabled. The USB device is now in the Default state and can draw no more than 100 mA from VBUS. All of its registers and state have been reset and it answers to the default address.
5. The host assigns a unique address to the USB device, moving the device to the Address state.
6. Before the USB device receives a unique address, its Default Control Pipe is still accessible via the default address. The host reads the device descriptor to determine what actual maximum data payload size this USB device's default pipe can use.
7. The host reads the configuration information from the device by reading each configuration zero to *n*-1, where *n* is the number of configurations. This process may take several milliseconds to complete.

8. Based on the configuration information and how the USB device will be used, the host assigns a configuration value to the device. The device is now in the Configured state and all of the endpoints in this configuration have taken on their described characteristics. The USB device may now draw the amount of VBUS power described in its descriptor for the selected configuration. From the device's point of view, it is now ready for use.

When the USB device is removed, the hub again sends a notification to the host. Detaching a device disables the port to which it had been attached. Upon receiving the detach notification, the host will update its local topological information.

9.2 Generic USB Device Operations

All USB devices support a common set of operations. This section describes those operations.

9.2.1 Dynamic Attachment and Removal

USB devices may be attached and removed at any time. The hub that provides the attachment point or port is responsible for reporting any change in the state of the port.

The host enables the hub port where the device is attached upon detection of an attachment, which also has the effect of resetting the device. A reset USB device has the following characteristics:

- Responds to the default USB address
- Is not configured
- Is not initially suspended

When a device is removed from a hub port, the hub disables the port where the device was attached and notifies the host of the removal.

9.2.2 Address Assignment

When a USB device is attached, the host is responsible for assigning a unique address to the device. This is done after the device has been reset by the host, and the hub port where the device is attached has been enabled.

9.2.3 Configuration

A USB device must be configured before its function(s) may be used. The host is responsible for configuring a USB device. The host typically requests configuration information from the USB device to determine the device's capabilities.

As part of the configuration process, the host sets the device configuration and, where necessary, selects the appropriate alternate settings for the interfaces.

Within a single configuration, a device may support multiple interfaces. An interface is a related set of endpoints that present a single feature or function of the device to the host. The protocol used to communicate with this related set of endpoints and the purpose of each endpoint within the interface may be specified as part of a device class or vendor-specific definition.

In addition, an interface within a configuration may have alternate settings that redefine the number or characteristics of the associated endpoints. If this is the case, the device must support the GetInterface() request to report the current alternate setting for the specified interface and SetInterface() request to select the alternate setting for the specified interface.

Within each configuration, each interface descriptor contains fields that identify the interface number and the alternate setting. Interfaces are numbered from zero to one less than the number of concurrent interfaces supported by the configuration. Alternate settings range from zero to one less than the number of alternate

settings for a specific interface. The default setting when a device is initially configured is alternate setting zero.

In support of adaptive device drivers that are capable of managing a related group of USB devices, the device and interface descriptors contain *Class*, *SubClass*, and *Protocol* fields. These fields are used to identify the function(s) provided by a USB device and the protocols used to communicate with the function(s) on the device. A class code is assigned to a group of related devices that has been characterized as a part of a USB Class Specification. A class of devices may be further subdivided into subclasses, and, within a class or subclass, a protocol code may define how the Host Software communicates with the device.

Note: The assignment of class, subclass, and protocol codes must be coordinated but is beyond the scope of this specification.

9.2.4 Data Transfer

Data may be transferred between a USB device endpoint and the host in one of four ways. Refer to Chapter 5 for the definition of the four types of transfers. An endpoint number may be used for different types of data transfers in different alternate settings. However, once an alternate setting is selected (including the default setting of an interface), a USB device endpoint uses only one data transfer method until a different alternate setting is selected.

9.2.5 Power Management

Power management on USB devices involves the issues described in the following sections.

9.2.5.1 Power Budgeting

USB bus power is a limited resource. During device enumeration, a host evaluates a device's power requirements. If the power requirements of a particular configuration exceed the power available to the device, Host Software shall not select that configuration.

USB devices shall limit the power they consume from VBUS to one unit load or less until configured. Suspended devices, whether configured or not, shall limit their bus power consumption as defined in Chapter 7. Depending on the power capabilities of the port to which the device is attached, a USB device may be able to draw up to five unit loads from VBUS after configuration.

9.2.5.2 Remote Wakeup

Remote wakeup allows a suspended USB device to signal a host that may also be suspended. This notifies the host that it should resume from its suspended mode, if necessary, and service the external event that triggered the suspended USB device to signal the host. A USB device reports its ability to support remote wakeup in a configuration descriptor. If a device supports remote wakeup, it must also allow the capability to be enabled and disabled using the standard USB requests.

Remote wakeup is accomplished using electrical signaling described in Section 7.1.7.7.

9.2.6 Request Processing

With the exception of SetAddress() requests (see Section 9.4.6), a device may begin processing of a request as soon as the device returns the ACK following the Setup. The device is expected to "complete" processing of the request before it allows the Status stage to complete successfully. Some requests initiate operations that take many milliseconds to complete. For requests such as this, the device class is required to define a method other than Status stage completion to indicate that the operation has completed. For example, a reset on a hub port takes at least 10 ms to complete. The SetPortFeature(PORT_RESET) (see Chapter 11) request "completes" when the reset on the port is initiated. Completion of the reset operation is

signaled when the port's status change is set to indicate that the port is now enabled. This technique prevents the host from having to constantly poll for a completion when it is known that the request will take a relatively long period of time.

9.2.6.1 Request Processing Timing

All devices are expected to handle requests in a timely manner. USB sets an upper limit of 5 seconds as the upper limit for any command to be processed. This limit is not applicable in all instances. The limitations are described in the following sections. It should be noted that the limitations given below are intended to encompass a wide range of implementations. If all devices in a USB system used the maximum allotted time for request processing, the user experience would suffer. For this reason, implementations should strive to complete requests in times that are as short as possible.

9.2.6.2 Reset/Resume Recovery Time

After a port is reset or resumed, the USB System Software is expected to provide a "recovery" interval of 10 ms before the device attached to the port is expected to respond to data transfers. The device may ignore any data transfers during the recovery interval.

After the end of the recovery interval (measured from the end of the reset or the end of the EOP at the end of the resume signaling), the device must accept data transfers at any time.

9.2.6.3 Set Address Processing

After the reset/resume recovery interval, if a device receives a SetAddress() request, the device must be able to complete processing of the request and be able to successfully complete the Status stage of the request within 50 ms. In the case of the SetAddress() request, the Status stage successfully completes when the device sends the zero-length Status packet or when the device sees the ACK in response to the Status stage data packet.

After successful completion of the Status stage, the device is allowed a SetAddress() recovery interval of 2 ms. At the end of this interval, the device must be able to accept Setup packets addressed to the new address. Also, at the end of the recovery interval, the device must not respond to tokens sent to the old address (unless, of course, the old and new address is the same).

9.2.6.4 Standard Device Requests

For standard device requests that require no Data stage, a device must be able to complete the request and be able to successfully complete the Status stage of the request within 50 ms of receipt of the request. This limitation applies to requests to the device, interface, or endpoint.

For standard device requests that require data stage transfer to the host, the device must be able to return the first data packet to the host within 500 ms of receipt of the request. For subsequent data packets, if any, the device must be able to return them within 500 ms of successful completion of the transmission of the previous packet. The device must then be able to successfully complete the status stage within 50 ms after returning the last data packet.

For standard device requests that require a data stage transfer to the device, the 5-second limit applies. This means that the device must be capable of accepting all data packets from the host and successfully completing the Status stage if the host provides the data at the maximum rate at which the device can accept it. Delays between packets introduced by the host add to the time allowed for the device to complete the request.

9.2.6.5 Class-specific Requests

Unless specifically exempted in the class document, all class-specific requests must meet the timing limitations for standard device requests. If a class document provides an exemption, the exemption may only be specified on a request-by-request basis.

A class document may require that a device respond more quickly than is specified in this section. Faster response may be required for standard and class-specific requests.

9.2.6.6 Speed Dependent Descriptors

A device capable of operation at high-speed can operate in either full- or high-speed. The device always knows its operational speed due to having to manage its transceivers correctly as part of reset processing (See Chapter 7 for more details on reset). A device also operates at a single speed after completing the reset sequence. In particular, there is no speed switch during normal operation. However, a high-speed capable device may have configurations that are speed dependent. That is, it may have some configurations that are only possible when operating at high-speed or some that are only possible when operating at full-speed. High-speed capable devices must support reporting their speed dependent configurations.

A high-speed capable device responds with descriptor information that is valid for the current operating speed. For example, when a device is asked for configuration descriptors, it only returns those for the current operating speed (e.g., full speed). However, there must be a way to determine the capabilities for both high- and full-speed operation.

Two descriptors allow a high-speed capable device to report configuration information about the other operating speed. The two descriptors are: the (other_speed) device_qualifier descriptor and the other_speed_configuration descriptor. These two descriptors are retrieved by the host by using the GetDescriptor request with the corresponding descriptor type values.

Note: These descriptors are not retrieved unless the host explicitly issues the corresponding GetDescriptor requests. If these two requests are not issued, the device would simply appear to be a single speed device.

Devices that are high-speed capable must set the version number in the *bcdUSB* field of their descriptors to 0200H. This indicates that such devices support the other_speed requests defined by USB 2.0. A device with descriptor version numbers less than 0200H should cause a Request Error response (see next section) if it receives these other_speed requests. A USB 1.x device (i.e., one with a device descriptor version less than 0200H) should not be issued the other_speed requests.

9.2.7 Request Error

When a request is received by a device that is not defined for the device, is inappropriate for the current setting of the device, or has values that are not compatible with the request, then a Request Error exists. The device deals with the Request Error by returning a STALL PID in response to the next Data stage transaction or in the Status stage of the message. It is preferred that the STALL PID be returned at the next Data stage transaction, as this avoids unnecessary bus activity.

9.3 USB Device Requests

All USB devices respond to requests from the host on the device's Default Control Pipe. These requests are made using control transfers. The request and the request's parameters are sent to the device in the Setup packet. The host is responsible for establishing the values passed in the fields listed in Table 9-2. Every Setup packet has eight bytes.

Table 9-2. Format of Setup Data

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

9.3.1 bmRequestType

This bitmapped field identifies the characteristics of the specific request. In particular, this field identifies the direction of data transfer in the second phase of the control transfer. The state of the *Direction* bit is ignored if the *wLength* field is zero, signifying there is no Data stage.

The USB Specification defines a series of standard requests that all devices must support. These are enumerated in Table 9-3. In addition, a device class may define additional requests. A device vendor may also define requests supported by the device.

Requests may be directed to the device, an interface on the device, or a specific endpoint on a device. This field also specifies the intended recipient of the request. When an interface or endpoint is specified, the *wIndex* field identifies the interface or endpoint.

9.3.2 bRequest

This field specifies the particular request. The *Type* bits in the *bmRequestType* field modify the meaning of this field. This specification defines values for the *bRequest* field only when the bits are reset to zero, indicating a standard request (refer to Table 9-3).

9.3.3 wValue

The contents of this field vary according to the request. It is used to pass a parameter to the device, specific to the request.

9.3.4 wIndex

The contents of this field vary according to the request. It is used to pass a parameter to the device, specific to the request.

The *wIndex* field is often used in requests to specify an endpoint or an interface. Figure 9-2 shows the format of *wIndex* when it is used to specify an endpoint.

D7	D6	D5	D4	D3	D2	D1	D0
Direction	Reserved (Reset to zero)			Endpoint Number			
D15	D14	D13	D12	D11	D10	D9	D8
Reserved (Reset to zero)							

Figure 9-2. wIndex Format when Specifying an Endpoint

The *Direction* bit is set to zero to indicate the OUT endpoint with the specified *Endpoint Number* and to one to indicate the IN endpoint. In the case of a control pipe, the request should have the *Direction* bit set to zero but the device may accept either value of the *Direction* bit.

Figure 9-3 shows the format of *wIndex* when it is used to specify an interface.

D7	D6	D5	D4	D3	D2	D1	D0
Interface Number							
D15	D14	D13	D12	D11	D10	D9	D8
Reserved (Reset to zero)							

Figure 9-3. wIndex Format when Specifying an Interface

9.3.5 wLength

This field specifies the length of the data transferred during the second phase of the control transfer. The direction of data transfer (host-to-device or device-to-host) is indicated by the *Direction* bit of the *bmRequestType* field. If this field is zero, there is no data transfer phase.

On an input request, a device must never return more data than is indicated by the *wLength* value; it may return less. On an output request, *wLength* will always indicate the exact amount of data to be sent by the host. Device behavior is undefined if the host should send more data than is specified in *wLength*.

9.4 Standard Device Requests

This section describes the standard device requests defined for all USB devices. Table 9-3 outlines the standard device requests, while Table 9-4 and Table 9-5 give the standard request codes and descriptor types, respectively.

USB devices must respond to standard device requests, even if the device has not yet been assigned an address or has not been configured.

Table 9-3. Standard Device Requests

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

Table 9-4. Standard Request Codes

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

Table 9-5. Descriptor Types

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

[1] The INTERFACE_POWER descriptor is defined in the current revision of the *USB Interface Power Management Specification*.

Feature selectors are used when enabling or setting features, such as remote wakeup, specific to a device, interface, or endpoint. The values for the feature selectors are given in Table 9-6.

Table 9-6. Standard Feature Selectors

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

If an unsupported or invalid request is made to a USB device, the device responds by returning STALL in the Data or Status stage of the request. If the device detects the error in the Setup stage, it is preferred that the device returns STALL at the earlier of the Data or Status stage. Receipt of an unsupported or invalid request does NOT cause the optional *Halt* feature on the control pipe to be set. If for any reason, the device becomes unable to communicate via its Default Control Pipe due to an error condition, the device must be reset to clear the condition and restart the Default Control Pipe.

9.4.1 Clear Feature

This request is used to clear or disable a specific feature.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None

Feature selector values in *wValue* must be appropriate to the recipient. Only device feature selector values may be used when the recipient is a device, only interface feature selector values may be used when the recipient is an interface, and only endpoint feature selector values may be used when the recipient is an endpoint.

Refer to Table 9-6 for a definition of which feature selector values are defined for which recipients.

A ClearFeature() request that references a feature that cannot be cleared, that does not exist, or that references an interface or endpoint that does not exist, will cause the device to respond with a Request Error.

If *wLength* is non-zero, then the device behavior is not specified.

Default state: Device behavior when this request is received while the device is in the Default state is not specified.

Address state: This request is valid when the device is in the Address state; references to interfaces or to endpoints other than endpoint zero shall cause the device to respond with a Request Error.

Configured state: This request is valid when the device is in the Configured state.

Note: The Test_Mode feature cannot be cleared by the ClearFeature() request.

9.4.2 Get Configuration

This request returns the current device configuration value.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value

If the returned value is zero, the device is not configured.

If *wValue*, *wIndex*, or *wLength* are not as specified above, then the device behavior is not specified.

Default state: Device behavior when this request is received while the device is in the Default state is not specified.

Address state: The value zero must be returned.

Configured state: The non-zero *bConfigurationValue* of the current configuration must be returned.

9.4.3 Get Descriptor

This request returns the specified descriptor if the descriptor exists.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID (refer to Section 9.6.7)	Descriptor Length	Descriptor

The *wValue* field specifies the descriptor type in the high byte (refer to Table 9-5) and the descriptor index in the low byte. The descriptor index is used to select a specific descriptor (only for configuration and string descriptors) when several descriptors of the same type are implemented in a device. For example, a device can implement several configuration descriptors. For other standard descriptors that can be retrieved via a GetDescriptor() request, a descriptor index of zero must be used. The range of values used for a descriptor index is from 0 to one less than the number of descriptors of that type implemented by the device.

The *wIndex* field specifies the Language ID for string descriptors or is reset to zero for other descriptors. The *wLength* field specifies the number of bytes to return. If the descriptor is longer than the *wLength* field, only the initial bytes of the descriptor are returned. If the descriptor is shorter than the *wLength* field, the device indicates the end of the control transfer by sending a short packet when further data is requested. A short packet is defined as a packet shorter than the maximum payload size or a zero length data packet (refer to Chapter 5).

The standard request to a device supports three types of descriptors: device (also device_qualifier), configuration (also other_speed_configuration), and string. A high-speed capable device supports the device_qualifier descriptor to return information about the device for the speed at which it is not operating (including *wMaxPacketSize* for the default endpoint and the number of configurations for the other speed). The other_speed_configuration returns information in the same structure as a configuration descriptor, but for a configuration if the device were operating at the other speed. A request for a configuration descriptor returns the configuration descriptor, all interface descriptors, and endpoint descriptors for all of the

interfaces in a single request. The first interface descriptor follows the configuration descriptor. The endpoint descriptors for the first interface follow the first interface descriptor. If there are additional interfaces, their interface descriptor and endpoint descriptors follow the first interface's endpoint descriptors. Class-specific and/or vendor-specific descriptors follow the standard descriptors they extend or modify.

All devices must provide a device descriptor and at least one configuration descriptor. If a device does not support a requested descriptor, it responds with a Request Error.

Default state: This is a valid request when the device is in the Default state.

Address state: This is a valid request when the device is in the Address state.

Configured state: This is a valid request when the device is in the Configured state.

9.4.4 Get Interface

This request returns the selected alternate setting for the specified interface.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Setting

Some USB devices have configurations with interfaces that have mutually exclusive settings. This request allows the host to determine the currently selected alternate setting.

If *wValue* or *wLength* are not as specified above, then the device behavior is not specified.

If the interface specified does not exist, then the device responds with a Request Error.

Default state: Device behavior when this request is received while the device is in the Default state is not specified.

Address state: A Request Error response is given by the device.

Configured state: This is a valid request when the device is in the Configured state.

9.4.5 Get Status

This request returns status for the specified recipient.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status

The *Recipient* bits of the *bmRequestType* field specify the desired recipient. The data returned is the current status of the specified recipient.

If *wValue* or *wLength* are not as specified above, or if *wIndex* is non-zero for a device status request, then the behavior of the device is not specified.

If an interface or an endpoint is specified that does not exist, then the device responds with a Request Error.

Default state: Device behavior when this request is received while the device is in the Default state is not specified.

Address state: If an interface or an endpoint other than endpoint zero is specified, then the device responds with a Request Error.

Configured state: If an interface or endpoint that does not exist is specified, then the device responds with a Request Error.

A GetStatus() request to a device returns the information shown in Figure 9-4.

D7	D6	D5	D4	D3	D2	D1	D0
Reserved (Reset to zero)						Remote Wakeup	Self Powered
D15	D14	D13	D12	D11	D10	D9	D8
Reserved (Reset to zero)							

Figure 9-4. Information Returned by a GetStatus() Request to a Device

The *Self Powered* field indicates whether the device is currently self-powered. If D0 is reset to zero, the device is bus-powered. If D0 is set to one, the device is self-powered. The *Self Powered* field may not be changed by the SetFeature() or ClearFeature() requests.

The *Remote Wakeup* field indicates whether the device is currently enabled to request remote wakeup. The default mode for devices that support remote wakeup is disabled. If D1 is reset to zero, the ability of the device to signal remote wakeup is disabled. If D1 is set to one, the ability of the device to signal remote wakeup is enabled. The *Remote Wakeup* field can be modified by the SetFeature() and ClearFeature() requests using the DEVICE_REMOTE_WAKEUP feature selector. This field is reset to zero when the device is reset.

A GetStatus() request to an interface returns the information shown in Figure 9-5.

D7	D6	D5	D4	D3	D2	D1	D0
Reserved (Reset to zero)							
D15	D14	D13	D12	D11	D10	D9	D8
Reserved (Reset to zero)							

Figure 9-5. Information Returned by a GetStatus() Request to an Interface

A GetStatus() request to an endpoint returns the information shown in Figure 9-6.

D7	D6	D5	D4	D3	D2	D1	D0
Reserved (Reset to zero)							Halt
D15	D14	D13	D12	D11	D10	D9	D8
Reserved (Reset to zero)							

Figure 9-6. Information Returned by a GetStatus() Request to an Endpoint

The *Halt* feature is required to be implemented for all interrupt and bulk endpoint types. If the endpoint is currently halted, then the *Halt* feature is set to one. Otherwise, the *Halt* feature is reset to zero. The *Halt* feature may optionally be set with the SetFeature(ENDPOINT_HALT) request. When set by the SetFeature() request, the endpoint exhibits the same stall behavior as if the field had been set by a hardware condition. If the condition causing a halt has been removed, clearing the *Halt* feature via a ClearFeature(ENDPOINT_HALT) request results in the endpoint no longer returning a STALL. For endpoints using data toggle, regardless of whether an endpoint has the *Halt* feature set, a ClearFeature(ENDPOINT_HALT) request always results in the data toggle being reinitialized to DATA0. The *Halt* feature is reset to zero after either a SetConfiguration() or SetInterface() request even if the requested configuration or interface is the same as the current configuration or interface.

It is neither required nor recommended that the *Halt* feature be implemented for the Default Control Pipe. However, devices may set the *Halt* feature of the Default Control Pipe in order to reflect a functional error condition. If the feature is set to one, the device will return STALL in the Data and Status stages of each standard request to the pipe except GetStatus(), SetFeature(), and ClearFeature() requests. The device need not return STALL for class-specific and vendor-specific requests.

9.4.6 Set Address

This request sets the device address for all future device accesses.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None

The *wValue* field specifies the device address to use for all subsequent accesses.

As noted elsewhere, requests actually may result in up to three stages. In the first stage, the Setup packet is sent to the device. In the optional second stage, data is transferred between the host and the device. In the final stage, status is transferred between the host and the device. The direction of data and status transfer depends on whether the host is sending data to the device or the device is sending data to the host. The Status stage transfer is always in the opposite direction of the Data stage. If there is no Data stage, the Status stage is from the device to the host.

Stages after the initial Setup packet assume the same device address as the Setup packet. The USB device does not change its device address until after the Status stage of this request is completed successfully. Note that this is a difference between this request and all other requests. For all other requests, the operation indicated must be completed before the Status stage.

If the specified device address is greater than 127, or if *wIndex* or *wLength* are non-zero, then the behavior of the device is not specified.

Device response to SetAddress() with a value of 0 is undefined.

Default state: If the address specified is non-zero, then the device shall enter the Address state; otherwise, the device remains in the Default state (this is not an error condition).

Address state: If the address specified is zero, then the device shall enter the Default state; otherwise, the device remains in the Address state but uses the newly-specified address.

Configured state: Device behavior when this request is received while the device is in the Configured state is not specified.

9.4.7 Set Configuration

This request sets the device configuration.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None

The lower byte of the *wValue* field specifies the desired configuration. This configuration value must be zero or match a configuration value from a configuration descriptor. If the configuration value is zero, the device is placed in its Address state. The upper byte of the *wValue* field is reserved.

If *wIndex*, *wLength*, or the upper byte of *wValue* is non-zero, then the behavior of this request is not specified.

Default state: Device behavior when this request is received while the device is in the Default state is not specified.

Address state: If the specified configuration value is zero, then the device remains in the Address state. If the specified configuration value matches the configuration value from a configuration descriptor, then that configuration is selected and the device enters the Configured state. Otherwise, the device responds with a Request Error.

Configured state: If the specified configuration value is zero, then the device enters the Address state. If the specified configuration value matches the configuration value from a configuration descriptor, then that configuration is selected and the device remains in the Configured state. Otherwise, the device responds with a Request Error.

9.4.8 Set Descriptor

This request is optional and may be used to update existing descriptors or new descriptors may be added.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Language ID (refer to Section 9.6.7) or zero	Descriptor Length	Descriptor

The *wValue* field specifies the descriptor type in the high byte (refer to Table 9-5) and the descriptor index in the low byte. The descriptor index is used to select a specific descriptor (only for configuration and string descriptors) when several descriptors of the same type are implemented in a device. For example, a device can implement several configuration descriptors. For other standard descriptors that can be set via a SetDescriptor() request, a descriptor index of zero must be used. The range of values used for a descriptor index is from 0 to one less than the number of descriptors of that type implemented by the device.

The *wIndex* field specifies the Language ID for string descriptors or is reset to zero for other descriptors. The *wLength* field specifies the number of bytes to transfer from the host to the device.

The only allowed values for descriptor type are device, configuration, and string descriptor types.

If this request is not supported, the device will respond with a Request Error.

Default state: Device behavior when this request is received while the device is in the Default state is not specified.

Address state: If supported, this is a valid request when the device is in the Address state.

Configured state: If supported, this is a valid request when the device is in the Configured state.

9.4.9 Set Feature

This request is used to set or enable a specific feature.

bmRequestType	bRequest	wValue	wIndex		wLength	Data
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Test Selector	Zero Interface Endpoint	Zero	None

Feature selector values in *wValue* must be appropriate to the recipient. Only device feature selector values may be used when the recipient is a device; only interface feature selector values may be used when the recipient is an interface, and only endpoint feature selector values may be used when the recipient is an endpoint.

Refer to Table 9-6 for a definition of which feature selector values are defined for which recipients.

The TEST_MODE feature is only defined for a device recipient (i.e., bmRequestType = 0) and the lower byte of wIndex must be zero. Setting the TEST_MODE feature puts the device upstream facing port into test mode. The device will respond with a request error if the request contains an invalid test selector. The transition to test mode must be complete no later than 3 ms after the completion of the status stage of the request. The transition to test mode of an upstream facing port must not happen until after the status stage of the request. The power to the device must be cycled to exit test mode of an upstream facing port of a device. See Section 7.1.20 for definitions of each test mode. A device must support the TEST_MODE feature when in the Default, Address or Configured high-speed device states.

A SetFeature() request that references a feature that cannot be set or that does not exist causes a STALL to be returned in the Status stage of the request.

Table 9-7. Test Mode Selectors

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.

If the feature selector is *TEST_MODE*, then the most significant byte of *wIndex* is used to specify the specific test mode. The recipient of a SetFeature(TEST_MODE…) must be the device; i.e., the lower byte of *wIndex* must be zero and the *bmRequestType* must be set to zero. The device must have its power cycled to exit test mode. The valid test mode selectors are listed in Table 9-7. See Section 7.1.20 for more information about the specific test modes.

If *wLength* is non-zero, then the behavior of the device is not specified.

If an endpoint or interface is specified that does not exist, then the device responds with a Request Error.

Default state: A device must be able to accept a SetFeature(TEST_MODE, TEST_SELECTOR) request when in the Default State. Device behavior for other SetFeature requests while the device is in the Default state is not specified.

Address state: If an interface or an endpoint other than endpoint zero is specified, then the device responds with a Request Error.

Configured state: This is a valid request when the device is in the Configured state.

9.4.10 Set Interface

This request allows the host to select an alternate setting for the specified interface.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None

Some USB devices have configurations with interfaces that have mutually exclusive settings. This request allows the host to select the desired alternate setting. If a device only supports a default setting for the specified interface, then a STALL may be returned in the Status stage of the request. This request cannot be used to change the set of configured interfaces (the SetConfiguration() request must be used instead).

If the interface or the alternate setting does not exist, then the device responds with a Request Error. If *wLength* is non-zero, then the behavior of the device is not specified.

Default state: Device behavior when this request is received while the device is in the Default state is not specified.

Address state: The device must respond with a Request Error.

Configured state: This is a valid request when the device is in the Configured state.

9.4.11 Synch Frame

This request is used to set and then report an endpoint's synchronization frame.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

When an endpoint supports isochronous transfers, the endpoint may also require per-frame transfers to vary in size according to a specific pattern. The host and the endpoint must agree on which frame the repeating pattern begins. The number of the frame in which the pattern began is returned to the host.

If a high-speed device supports the Synch Frame request, it must internally synchronize itself to the zeroth microframe and have a time notion of classic frame. Only the frame number is used to synchronize and reported by the device endpoint (i.e., no microframe number). The endpoint must synchronize to the zeroth microframe.

This value is only used for isochronous data transfers using implicit pattern synchronization. If *wValue* is non-zero or *wLength* is not two, then the behavior of the device is not specified.

If the specified endpoint does not support this request, then the device will respond with a Request Error.

Default state: Device behavior when this request is received while the device is in the Default state is not specified.

Address state: The device shall respond with a Request Error.

Configured state: This is a valid request when the device is in the Configured state.

9.5 Descriptors

USB devices report their attributes using descriptors. A descriptor is a data structure with a defined format. Each descriptor begins with a byte-wide field that contains the total number of bytes in the descriptor followed by a byte-wide field that identifies the descriptor type.

Using descriptors allows concise storage of the attributes of individual configurations because each configuration may reuse descriptors or portions of descriptors from other configurations that have the same characteristics. In this manner, the descriptors resemble individual data records in a relational database.

Where appropriate, descriptors contain references to string descriptors that provide displayable information describing a descriptor in human-readable form. The inclusion of string descriptors is optional. However, the reference fields within descriptors are mandatory. If a device does not support string descriptors, string reference fields must be reset to zero to indicate no string descriptor is available.

If a descriptor returns with a value in its length field that is less than defined by this specification, the descriptor is invalid and should be rejected by the host. If the descriptor returns with a value in its length

field that is greater than defined by this specification, the extra bytes are ignored by the host, but the next descriptor is located using the length returned rather than the length expected.

A device may return class- or vendor-specific descriptors in two ways:

1. If the class or vendor specific descriptors use the same format as standard descriptors (e.g., start with a length byte and followed by a type byte), they must be returned interleaved with standard descriptors in the configuration information returned by a GetDescriptor(Configuration) request. In this case, the class or vendor-specific descriptors must follow a related standard descriptor they modify or extend.
2. If the class or vendor specific descriptors are independent of configuration information or use a non-standard format, a GetDescriptor() request specifying the class or vendor specific descriptor type and index may be used to retrieve the descriptor from the device. A class or vendor specification will define the appropriate way to retrieve these descriptors.

9.6 Standard USB Descriptor Definitions

The standard descriptors defined in this specification may only be modified or extended by revision of the Universal Serial Bus Specification.

Note: An extension to the USB 1.0 standard endpoint descriptor has been published in Device Class Specification for Audio Devices Revision 1.0. This is the only extension defined outside USB Specification that is allowed. Future revisions of the USB Specification that extend the standard endpoint descriptor will do so as to not conflict with the extension defined in the Audio Device Class Specification Revision 1.0.

9.6.1 Device

A device descriptor describes general information about a USB device. It includes information that applies globally to the device and all of the device's configurations. A USB device has only one device descriptor.

A high-speed capable device that has different device information for full-speed and high-speed must also have a device_qualifier descriptor (see Section 9.6.2).

The DEVICE descriptor of a high-speed capable device has a version number of 2.0 (0200H). If the device is full-speed only or low-speed only, this version number indicates that it will respond correctly to a request for the device_qualifier desciptor (i.e., it will respond with a request error).

The *bcdUSB* field contains a BCD version number. The value of the *bcdUSB* field is 0xJJMN for version JJ.M.N (JJ – major version number, M – minor version number, N – sub-minor version number), e.g., version 2.1.3 is represented with value 0x0213 and version 2.0 is represented with a value of 0x0200.

The *bNumConfigurations* field indicates the number of configurations at the current operating speed. Configurations for the other operating speed are not included in the count. If there are specific configurations of the device for specific speeds, the *bNumConfigurations* field only reflects the number of configurations for a single speed, not the total number of configurations for both speeds.

If the device is operating at high-speed, the *bMaxPacketSize0* field must be 64 indicating a 64 byte maximum packet. High-speed operation does not allow other maximum packet sizes for the control endpoint (endpoint 0).

All USB devices have a Default Control Pipe. The maximum packet size of a device's Default Control Pipe is described in the device descriptor. Endpoints specific to a configuration and its interface(s) are described in the configuration descriptor. A configuration and its interface(s) do not include an endpoint descriptor for the Default Control Pipe. Other than the maximum packet size, the characteristics of the Default Control Pipe are defined by this specification and are the same for all USB devices.

The *bNumConfigurations* field identifies the number of configurations the device supports. Table 9-8 shows the standard device descriptor.

Table 9-8. Standard Device Descriptor

Offset	Field	Size	Value	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 210H). This field identifies the release of the USB Specification with which the device and its descriptors are compliant.
4	*bDeviceClass*	1	Class	Class code (assigned by the USB-IF). If this field is reset to zero, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and FEH, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to FFH, the device class is vendor-specific.
5	*bDeviceSubClass*	1	SubClass	Subclass code (assigned by the USB-IF). These codes are qualified by the value of the *bDeviceClass* field. If the *bDeviceClass* field is reset to zero, this field must also be reset to zero. If the *bDeviceClass* field is not set to FFH, all values are reserved for assignment by the USB-IF.

Table 9-8. Standard Device Descriptor (Continued)

Offset	Field	Size	Value	Description
6	*bDeviceProtocol*	1	Protocol	Protocol code (assigned by the USB-IF). These codes are qualified by the value of the *bDeviceClass* and the *bDeviceSubClass* fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use class-specific protocols on a device basis. However, it may use class-specific protocols on an interface basis. If this field is set to FFH, the device uses a vendor-specific protocol on a device basis.
7	*bMaxPacketSize0*	1	Number	Maximum packet size for endpoint zero (only 8, 16, 32, or 64 are valid)
8	*idVendor*	2	ID	Vendor ID (assigned by the USB-IF)
10	*idProduct*	2	ID	Product ID (assigned by the manufacturer)
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

9.6.2 Device_Qualifier

The device_qualifier descriptor describes information about a high-speed capable device that would change if the device were operating at the other speed. For example, if the device is currently operating at full-speed, the device_qualifier returns information about how it would operate at high-speed and vice-versa. Table 9-9 shows the fields of the device_qualifier descriptor.

Table 9-9. Device_Qualifier Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of descriptor
1	*bDescriptorType*	1	Constant	Device Qualifier Type
2	*bcdUSB*	2	BCD	USB specification version number (e.g., 0200H for V2.00)
4	*bDeviceClass*	1	Class	Class Code
5	*bDeviceSubClass*	1	SubClass	SubClass Code
6	*bDeviceProtocol*	1	Protocol	Protocol Code
7	*bMaxPacketSize0*	1	Number	Maximum packet size for other speed
8	*bNumConfigurations*	1	Number	Number of Other-speed Configurations
9	*bReserved*	1	Zero	Reserved for future use, must be zero

The vendor, product, device, manufacturer, product, and serialnumber fields of the standard device descriptor are not included in this descriptor since that information is constant for a device for all supported speeds. The version number for this descriptor must be at least 2.0 (0200H).

The host accesses this descriptor using the GetDescriptor() request. The descriptor type in the GetDescriptor() request is set to device_qualifier (see Table 9-5).

If a full-speed only device (with a device descriptor version number equal to 0200H) receives a GetDescriptor() request for a device_qualifier, it must respond with a request error. The host must not make a request for an other_speed_configuration descriptor unless it first successfully retrieves the device_qualifier descriptor.

9.6.3 Configuration

The configuration descriptor describes information about a specific device configuration. The descriptor contains a *bConfigurationValue* field with a value that, when used as a parameter to the SetConfiguration() request, causes the device to assume the described configuration.

The descriptor describes the number of interfaces provided by the configuration. Each interface may operate independently. For example, an ISDN device might be configured with two interfaces, each providing 64 Kb/s bi-directional channels that have separate data sources or sinks on the host. Another configuration might present the ISDN device as a single interface, bonding the two channels into one 128 Kb/s bi-directional channel.

When the host requests the configuration descriptor, all related interface and endpoint descriptors are returned (refer to Section 9.4.3).

A USB device has one or more configuration descriptors. Each configuration has one or more interfaces and each interface has zero or more endpoints. An endpoint is not shared among interfaces within a single configuration unless the endpoint is used by alternate settings of the same interface. Endpoints may be shared among interfaces that are part of different configurations without this restriction.

Once configured, devices may support limited adjustments to the configuration. If a particular interface has alternate settings, an alternate may be selected after configuration. Table 9-10 shows the standard configuration descriptor.

Table 9-10. Standard Configuration Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	CONFIGURATION Descriptor Type
2	*wTotalLength*	2	Number	Total length of data returned for this configuration. Includes the combined length of all descriptors (configuration, interface, endpoint, and class- or vendor-specific) returned for this configuration.
4	*bNumInterfaces*	1	Number	Number of interfaces supported by this configuration
5	*bConfigurationValue*	1	Number	Value to use as an argument to the SetConfiguration() request to select this configuration
6	*iConfiguration*	1	Index	Index of string descriptor describing this configuration

Table 9-10. Standard Configuration Descriptor (Continued)

Offset	Field	Size	Value	Description
7	*bmAttributes*	1	Bitmap	Configuration characteristics D7: Reserved (set to one) D6: Self-powered D5: Remote Wakeup D4...0: Reserved (reset to zero) D7 is reserved and must be set to one for historical reasons. A device configuration that uses power from the bus and a local source reports a non-zero value in *bMaxPower* to indicate the amount of bus power required and sets D6. The actual power source at runtime may be determined using the GetStatus(DEVICE) request (see Section 9.4.5). If a device configuration supports remote wakeup, D5 is set to one.
8	*bMaxPower*	1	mA	Maximum power consumption of the USB device from the bus in this specific configuration when the device is fully operational. Expressed in 2 mA units (i.e., 50 = 100 mA). Note: A device configuration reports whether the configuration is bus-powered or self-powered. Device status reports whether the device is currently self-powered. If a device is disconnected from its external power source, it updates device status to indicate that it is no longer self-powered. A device may not increase its power draw from the bus, when it loses its external power source, beyond the amount reported by its configuration. If a device can continue to operate when disconnected from its external power source, it continues to do so. If the device cannot continue to operate, it fails operations it can no longer support. The USB System Software may determine the cause of the failure by checking the status and noting the loss of the device's power source.

9.6.4 Other_Speed_Configuration

The other_speed_configuration descriptor shown in Table 9-11 describes a configuration of a high-speed capable device if it were operating at its other possible speed. The structure of the other_speed_configuration is identical to a configuration descriptor.

Table 9-11. Other_Speed_Configuration Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of descriptor
1	*bDescriptorType*	1	Constant	Other_speed_Configuration Type
2	*wTotalLength*	2	Number	Total length of data returned
4	*bNumInterfaces*	1	Number	Number of interfaces supported by this speed configuration
5	*bConfigurationValue*	1	Number	Value to use to select configuration
6	*iConfiguration*	1	Index	Index of string descriptor
7	*bmAttributes*	1	Bitmap	Same as Configuration descriptor
8	*bMaxPower*	1	mA	Same as Configuration descriptor

The host accesses this descriptor using the GetDescriptor() request. The descriptor type in the GetDescriptor() request is set to other_speed_configuration (see Table 9-5).

9.6.5 Interface

The interface descriptor describes a specific interface within a configuration. A configuration provides one or more interfaces, each with zero or more endpoint descriptors describing a unique set of endpoints within the configuration. When a configuration supports more than one interface, the endpoint descriptors for a particular interface follow the interface descriptor in the data returned by the GetConfiguration() request. An interface descriptor is always returned as part of a configuration descriptor. Interface descriptors cannot be directly accessed with a GetDescriptor() or SetDescriptor() request.

An interface may include alternate settings that allow the endpoints and/or their characteristics to be varied after the device has been configured. The default setting for an interface is always alternate setting zero. The SetInterface() request is used to select an alternate setting or to return to the default setting. The GetInterface() request returns the selected alternate setting.

Alternate settings allow a portion of the device configuration to be varied while other interfaces remain in operation. If a configuration has alternate settings for one or more of its interfaces, a separate interface descriptor and its associated endpoints are included for each setting.

If a device configuration supported a single interface with two alternate settings, the configuration descriptor would be followed by an interface descriptor with the *bInterfaceNumber* and *bAlternateSetting* fields set to zero and then the endpoint descriptors for that setting, followed by another interface descriptor and its associated endpoint descriptors. The second interface descriptor's *bInterfaceNumber* field would also be set to zero, but the *bAlternateSetting* field of the second interface descriptor would be set to one.

If an interface uses only endpoint zero, no endpoint descriptors follow the interface descriptor. In this case, the *bNumEndpoints* field must be set to zero.

An interface descriptor never includes endpoint zero in the number of endpoints. Table 9-12 shows the standard interface descriptor.