Chapter 8
Protocol Layer

This chapter presents a bottom-up view of the USB protocol starting with field and packet definitions. This is followed by a description of packet transaction formats for different transaction types. Link layer flow control and transaction level fault recovery are then covered. The chapter finishes with a discussion of retry synchronization, babble, and loss of bus activity recovery.

8.1 Bit Ordering

Bits are sent out onto the bus LSB first, followed by next LSB, through to MSB last. In the following diagrams, packets are displayed such that both individual bits and fields are represented (in a left to right reading order) as they would move across the bus.

8.2 SYNC Field

All packets begin with a synchronization (SYNC) field, which is a coded sequence that generates a maximum edge transition density. The SYNC field appears on the bus as IDLE followed by the binary string “KJKJKJKK,” in its NRZI encoding. It is used by the input circuitry to align incoming data with the local clock and is defined to be eight bits in length. SYNC serves only as a synchronization mechanism and is not shown in the following packet diagrams (refer to Section 7.1.7). The last two bits in the SYNC field are a marker that is used to identify the first bit of the PID. All subsequent bits in the packet must be indexed from this point.

8.3 Packet Field Formats

Field formats for the token, data, and handshake packets are described in the following section. Packet bit definitions are displayed in unencoded data format. The effects of NRZI coding and bit stuffing have been removed for the sake of clarity. All packets have distinct start and end of packet delimiters. The start of packet (SOP) is part of the SYNC field, and the end of packet (EOP) delimiter is described in Chapter 7.

8.3.1 Packet Identifier Field

A packet identifier (PID) immediately follows the SYNC field of every USB packet. A PID consists of a four bit packet type field followed by a four-bit check field as shown in Figure 8-1. The PID indicates the type of packet and, by inference, the format of the packet and the type of error detection applied to the packet. The four-bit check field of the PID insures reliable decoding of the PID so that the remainder of the packet is interpreted correctly. The PID check field is generated by performing a ones complement of the packet type field.

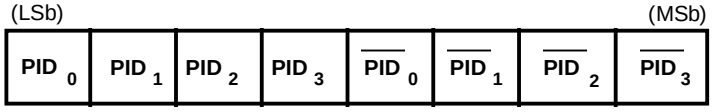

Figure 8-1. PID Format

The host and all functions must perform a complete decoding of all received PID fields. Any PID received with a failed check field or which decodes to a non-defined value is assumed to be corrupted and it, as well as the remainder of the packet, is ignored by the packet receiver. If a function receives an otherwise valid PID for a transaction type or direction that it does not support, the function must not respond. For example, an IN only endpoint must ignore an OUT token. PID types, codings, and descriptions are listed in Table 8-1.

Table 8-1. PID Types

PID Type	PID Name	PID[3:0]	Description
Token	OUT	b0001	Address + endpoint number in host -> function transaction
	IN	b1001	Address + endpoint number in function -> host transaction
	SOF	b0101	Start of frame marker and frame number
	SETUP	b1101	Address + endpoint number in host -> function transaction for setup to a control endpoint
Data	DATA0	b0011	Data packet PID even
	DATA1	b1011	Data packet PID odd
Handshake	ACK	b0010	Receiver accepts error free data packet
	NAK	b1010	Rx device cannot accept data or Tx device cannot send data
	STALL	b1110	Endpoint is stalled
Special	PRE	b1100	Host-issued preamble. Enables downstream bus traffic to low speed devices.

PIDs are divided into four coding groups: token, data, handshake, and special, with the first two transmitted PID bits (PID<1:0>) indicating which group. This accounts for the distribution of PID codes.

8.3.2 Address Fields

Function endpoints are addressed using two fields: the function address field and the endpoint field. A function needs to fully decode both address and endpoint fields. Address or endpoint aliasing is not permitted, and a mismatch on either field must cause the token to be ignored. Accesses to non-initialized endpoints will also cause the token to be ignored.

8.3.2.1 Address Field

The function address (ADDR) field specifies the function, via its address, that is either the source or destination of a data packet, depending on the value of the token PID. As shown in Figure 8-2, a total of 128 addresses are specified as ADDR<6:0>. The ADDR field is specified for IN, SETUP, and OUT tokens. By definition, each ADDR value defines a single function. Upon reset and power-up, a function's address defaults to a value of 0 and must be programmed by the host during the enumeration process. The 0 default address is reserved for default and cannot be assigned for normal operation.

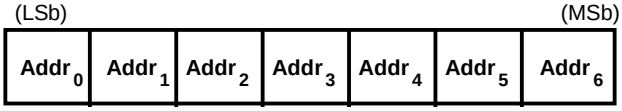

Figure 8-2. ADDR Field

8.3.2.2 Endpoint Field

An additional four-bit endpoint (ENDP) field, shown in Figure 8-3, permits more flexible addressing of functions in which more than one sub-channel is required. Endpoint numbers are function specific. The endpoint field is defined for IN, SETUP, and OUT token PIDs only. All functions must support one control endpoint at 0. Low speed devices support a maximum of two endpoint addresses per function: 0 plus one additional endpoint. Full speed functions may support up to the maximum of 16 endpoints.

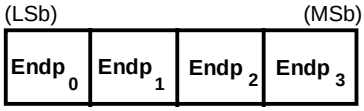

Figure 8-3. Endpoint Field

8.3.3 Frame Number Field

The frame number field is an 11-bit field that is incremented by the host on a per frame basis. The frame number field rolls over upon reaching its maximum value of x7FF, and is sent only for SOF tokens at the start of each frame.

8.3.4 Data Field

The data field may range from 0 to 1023 bytes and must be an integral numbers of bytes. Figure 8-4 shows the format for multiple bytes. Data bits within each byte are shifted out LSB first.

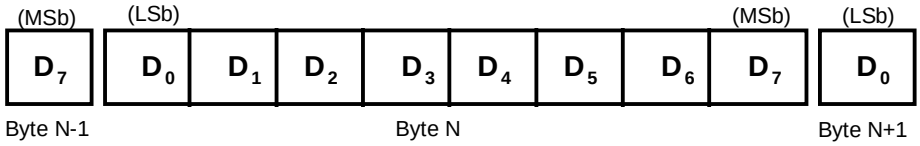

Figure 8-4. Data Field Format

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 the 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 0. If the result of that XOR is 1, 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.

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 generator polynomial is:

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

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

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 1000000000000101. If all data and CRC bits are received without error, the 16-bit residual will be 1000000000001101.

8.4 Packet Formats

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

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. 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. Only the host can issue token packets. IN PIDs define a data transaction from a function to the host. OUT and SETUP PIDs define data transactions from the host to a function.

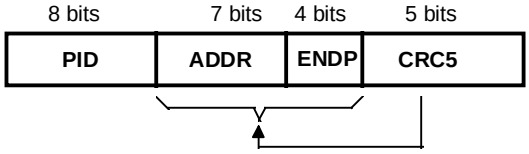

Figure 8-5. Token Format

Token packets have a five-bit CRC which 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 Start of Frame Packets

Start of Frame (SOF) packets are issued by the host at a nominal rate of once every 1.00 ms ±0.05. SOF packets consist of a PID indicating packet type followed by an 11-bit frame number field as illustrated in Figure 8-6.

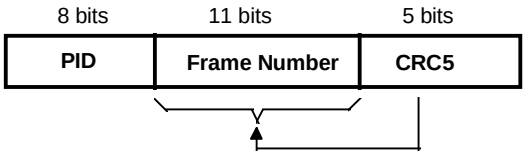

Figure 8-6. SOF Packet

The SOF token comprises the token-only transaction that distributes a start of frame marker and accompanying frame number at precisely timed intervals corresponding to the start of each frame. All full speed functions, including hubs, must receive and decode 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 a start of frame has occurred when it detects the SOF PID. Frame timing sensitive functions, which do not need to keep track of frame number, 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.

8.4.3 Data Packets

A data packet consists of a PID, a data field, and a CRC as shown in Figure 8-7. There are two types of data packets, identified by differing PIDs: DATA0 and DATA1. Two data packet PIDs are defined to support data toggle synchronization (refer to Section 8.6).

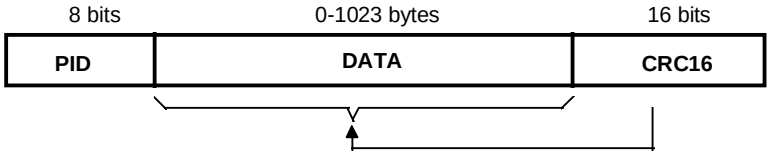

Figure 8-7. Data Packet Format

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.

8.4.4 Handshake Packets

Handshake packets, as shown in Figure 8-8, 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, flow control, and stall 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.

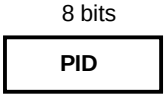

Figure 8-8. Handshake Packet

There are three types of handshake packets:

- **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 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 transactions. The host can never issue a 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. NAK is also used by interrupt endpoints to indicate that no interrupt is pending.
- **STALL** is returned by a function in response to an IN token or after the data phase of an OUT (see Figure 8-9 and Figure 8-13). STALL indicates that a function is unable to transmit or receive data, and that the condition requires host intervention to remove the stall. Once a function's endpoint is stalled, the function must continue returning STALL until the condition causing the stall has been cleared through host intervention. The host is not permitted to return a STALL under any condition.

8.4.5 Handshake Responses

Transmitting and receiving functions must return handshakes based upon an order of precedence detailed in Table 8-2 through Table 8-4. Not all handshakes are allowed, depending on the transaction type and whether the handshake is being issued by a function or the host.

8.4.5.1 Function Response to IN Transactions

Table 8-2 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 stall 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 8-2. Function Responses to IN Transactions

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

8.4.5.2 Host Response to IN Transactions

Table 8-3 shows the host response to an IN transaction. The host is able to return only one type of handshake, an 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 8-3. 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

8.4.5.3 Function Response to an OUT Transaction

Handshake responses for an OUT transaction are shown in Table 8-4. 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 stalled, the function returns a STALL handshake. 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 an ACK handshake. If the function cannot accept the data packet due to flow control reasons, it returns a NAK.

Table 8-4. Function Responses to OUT Transactions in Order of Precedence

Data Packet Corrupted	Receiver Stalled	Sequence Bits Mismatch	Function Can Accept Data	Handshake Returned by Function
Yes	N/A	N/A	N/A	None
No	Yes	N/A	N/A	STALL
No	No	Yes	N/A	ACK
No	No	No	Yes	ACK
No	No	No	No	NAK

8.4.5.4 Function Response to a SETUP Transaction

Setup defines a special type of host to function data transaction which permits the host to initialize an endpoint's synchronization bits to those of the host. Upon receiving a Setup transaction, a function must accept the data. Setup transactions cannot be stalled or NAKed and the receiving function must accept the Setup transfer's data. If a non-control endpoint receives a SETUP PID, it must ignore the transaction and return no response.

8.5 Transaction Formats

Packet transaction format varies depending on the endpoint type. There are four endpoint types: bulk, control, interrupt, and isochronous.

8.5.1 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 8-9. Under certain flow control and stall conditions, the data phase may be replaced with a handshake resulting in a two phase transaction in which no data is transmitted.

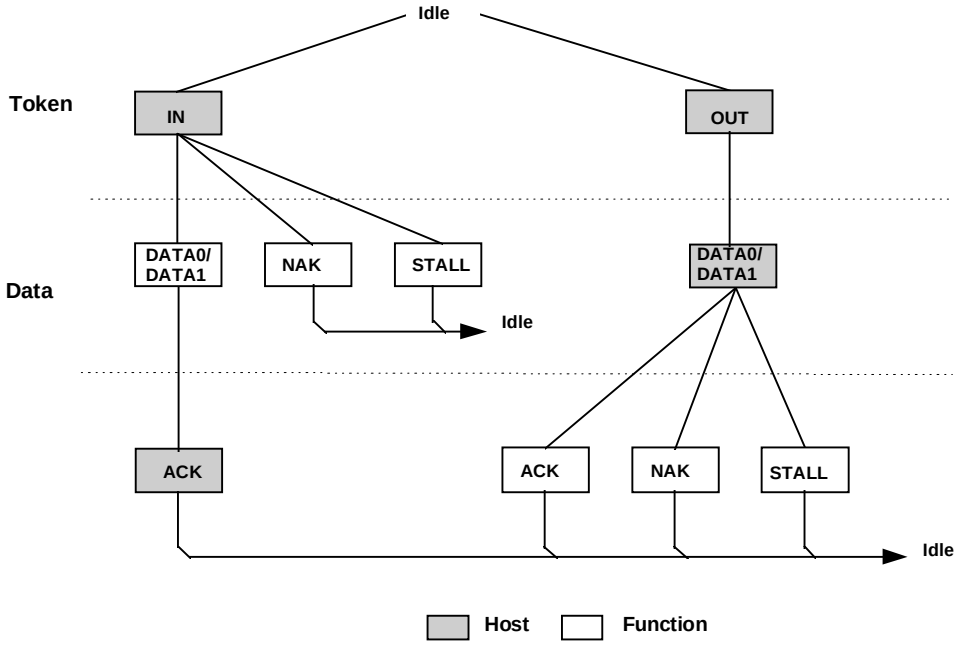

Figure 8-9. Bulk Transaction Format

When the host wishes 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. A NAK indicates that the function is temporarily unable to return data, while a STALL indicates that the endpoint is permanently stalled and requires host 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 wishes to transmit bulk data, it first issues an OUT token packet followed by a data packet. The function then returns one of three handshakes. ACK indicates that the data packet was received without errors and informs the host that 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 at this time (e.g., buffer full). If the endpoint was stalled, 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 8-10 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. Bulk endpoints must have their toggle sequence bits initialized via a separate control endpoint.

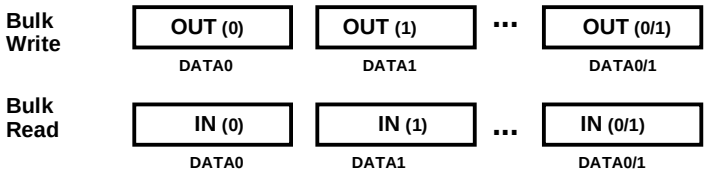

Figure 8-10. Bulk Reads and Writes

The host always initializes the first transaction of a bus transfer to the DATA0 PID. 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).

8.5.2 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 on OUT PID. Figure 8-11 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 an ACK handshake or, if the data is corrupted, discard the data and return no handshake.

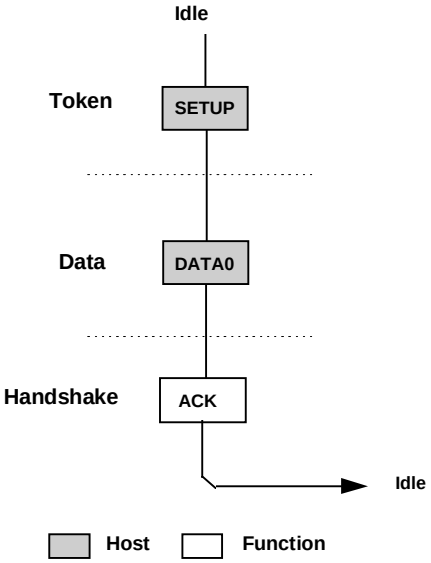

Figure 8-11. Control Setup Transaction

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 phase 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) which 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 operation in the sequence. 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 8-12 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.

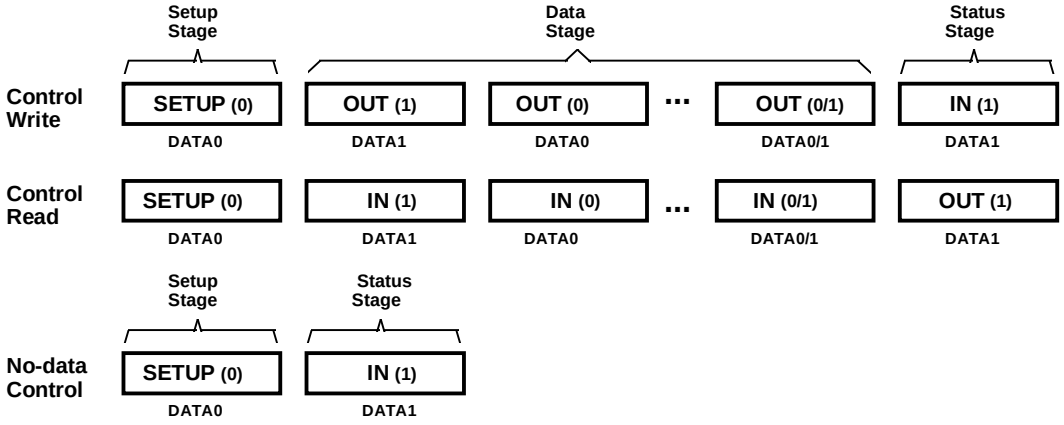

Figure 8-12. Control Read and Write Sequences

8.5.2.1 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 command.

Status reporting is always in the function to host direction. The following table summarizes the type of responses required for each. Control write transfers return status information on the data phase of the transfer. Control read transfers return status information on the handshake phase after the host has issued a zero length data packet during the previous data phase.

Table 8-5. Status Phase Responses

Status Response	Control Write Transfer (sent during data phase)	Control Read Transfer (send during handshake phase)
Function completes	0 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 sends a zero length data packet to the control endpoint. The endpoint's handshake response indicates the completion status. NAK indicates that the function is still processing the command and that the host should continue the status phase. ACK indicates that the function has completed the command and is ready to accept a new command and STALL indicates that the function has an error that prevents it from completing the command.

For control writes, the function responds with either a handshake or a zero length data packet to indicate its status. A NAK indicates that the function is still processing the command and that the host should continue the status phase, return of a zero length packet indicates normal completion of the command, and STALL indicates that the function has an error that prevents it from completing the command. Control write transfers which return a zero length data packet during the data phase always cause the host to return an ACK handshake to the function.

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

8.5.2.2 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 control transfer, 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, issues an OUT setup transfer, and the function, upon seeing that the token direction has toggled, interprets this action as proof that the host successfully received the data. In other words, the function interprets the toggling of the token direction as implicit proof of the host's successful receipt of the last ACK handshake. Therefore, when the function sees the OUT setup transaction, it advances to the status phase.

Control writes do not have this ambiguity. The host, by virtue of receiving the handshake, knows for sure if the last transaction was successful. If an ACK handshake on an OUT gets corrupted, the host does not advance to the status phase and retries the last data instead. A detailed analysis of retry policy is presented in Section 8.6.4.

8.5.3 Interrupt Transactions

Interrupt transactions consist solely of IN as shown in Figure 8-13. 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. A stalled interrupt endpoint causes the function to return a STALL handshake if it is permanently stalled and requires software intervention by the host. 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.

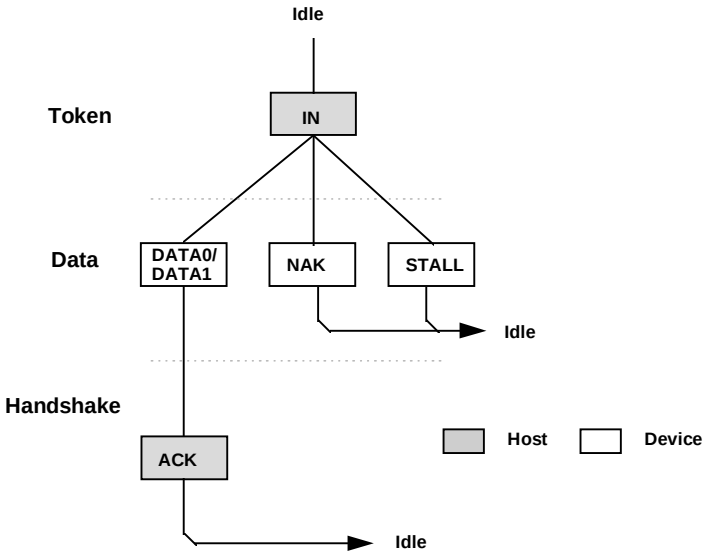

Figure 8-13. Interrupt Transaction Format

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 host software clears the interrupt condition. When used in the toggle mode, an interrupt endpoint is initialized to the DATA0 PID and behaves the same as the bulk IN transaction shown in Figure 8-10.

An interrupt endpoint may also be used to communicate rate feedback information for certain types of isochronous functions. When used in this mode, the data toggle bits should be changed after each data packet is sent to the host without regard to the presence or type of handshake packet.

8.5.4 Isochronous Transactions

ISO transactions have a token and data phase, but no handshake phase, as shown in Figure 8-14. 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. ISO transactions do not support a handshake phase or retry capability.

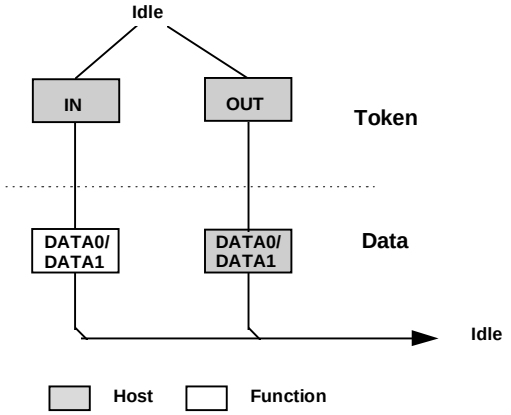

Figure 8-14. Isochronous Transaction Format

ISO transactions do not support toggle sequencing, and the data PID is always DATA0. The packet receiver does not examine the data PID.

8.6 Data Toggle Synchronization and Retry

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 ISO transfers.

8.6.1 Initialization via SETUP Token

Control transfers use the SETUP token for initializing host and function sequence bits. Figure 8-15 shows the host issuing a SETUP packet to a function followed by an OUT. The numbers in the circles represent the transmitter and receiver sequence bits. The function must accept the data and ACK the transaction. When the function accepts the transaction, it must reset its sequence bit so that both the host's and function's sequence bits are equal to 1 at the end of the SETUP transaction.

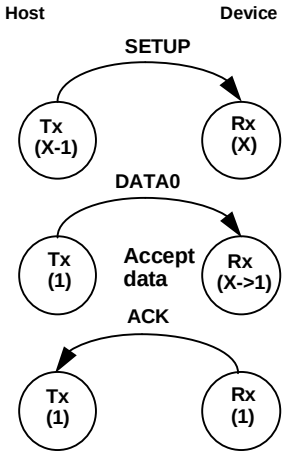

Figure 8-15. SETUP Initialization

8.6.2 Successful Data Transactions

Figure 8-16 shows the case where two successful transactions have occurred. For the data transmitter, this means that it toggles its sequence bit upon receipt of an ACK. The receiver toggles its sequence bit only if it receives a valid data packet and the packet's data PID matches the receiver's sequence bit.

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 a NAK. If data can be accepted and the receiver's sequence bit matches the PID sequence bit, then data is accepted. Sequence bits may only change if a data packet is transmitted. Two-phase transactions in which there is no data packet leave the transmitter and receiver sequence bits unchanged.

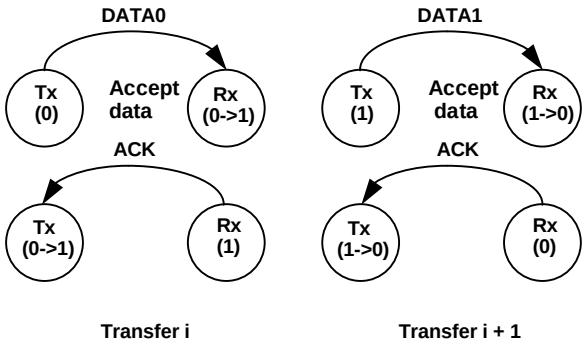

Figure 8-16. Consecutive Transactions

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 time out, depending on the circumstances, and the receiver will not toggle its sequence bit. Figure 8-17 shows the case where a transaction is NAKed and then retried. Any non-ACK handshake or time out 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.

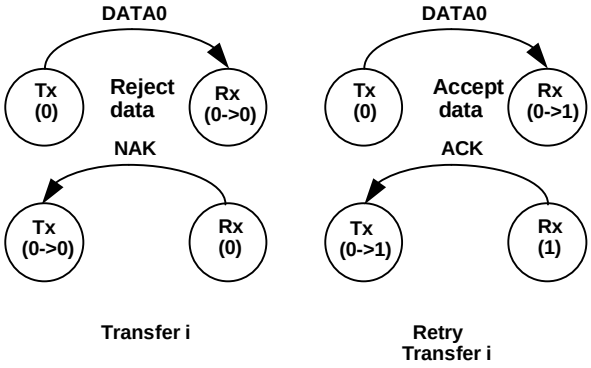

Figure 8-17. NAKed Transaction with Retry

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 8-18. Here the transmitter issues a valid data packet, which is successfully acquired by the receiver; however, the ACK handshake is corrupted.

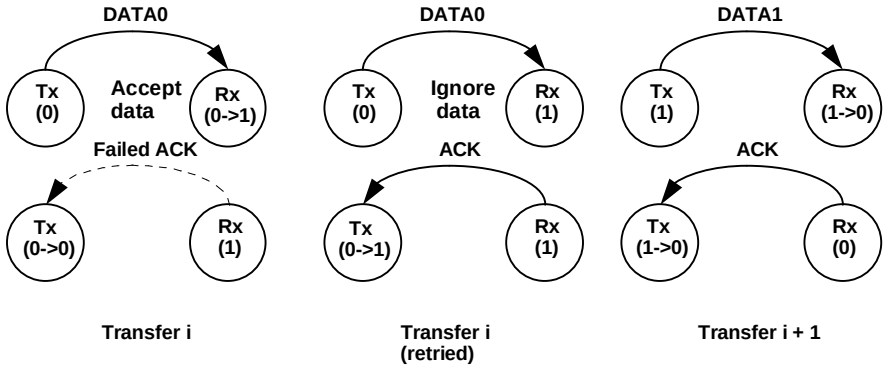

Figure 8-18. Corrupted ACK Handshake with Retry

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 an 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 be the same length 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. 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 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

USB supports signaling at two speeds: full speed signaling at 12.0 Mbs and low speed signaling at 1.5 Mbs. 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 an low speed device might misinterpret downstream a full speed packet as being addressed to it. Figure 8-19 shows an IN low speed transaction in which the host issues a token and handshake and receives a data packet.

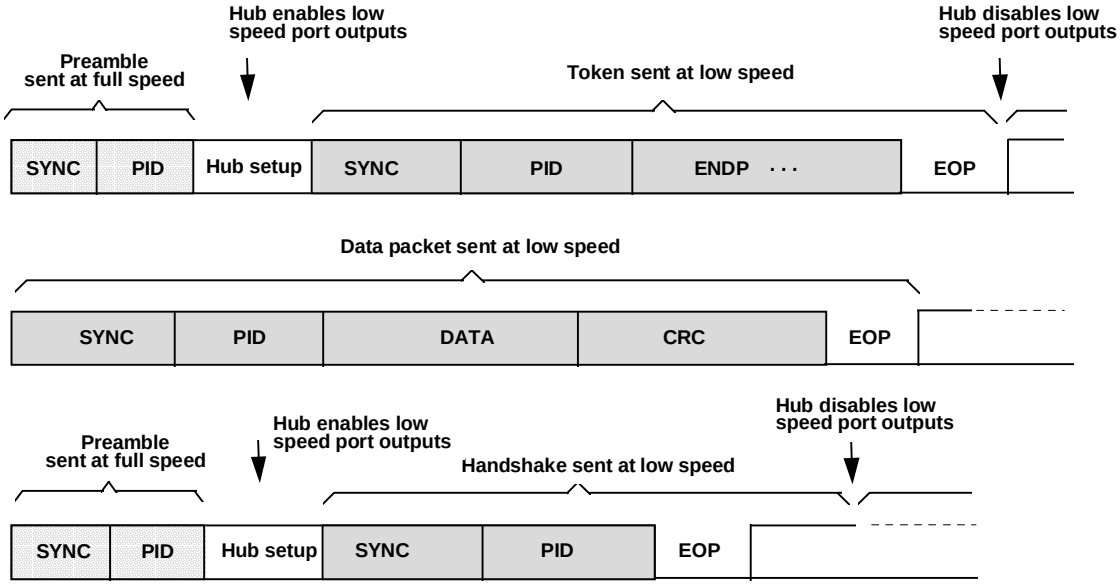

Figure 8-19. Low Speed Transaction

All downstream packets transmitted to low speed devices require a preamble. The preamble consists of a SYNC followed by a PID, both sent at full speed. Hubs must comprehend the PRE PID; all other USB devices must ignore it and treat it as undefined. After the end of the preamble PID, the host must wait at least four full speed bit times during which hubs must complete the process of configuring their repeater sections to accept low speed signaling. 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 accept low speed signaling from the host 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 issuing SYNC at low speed, followed by the remainder of the packet. The end of packet is identified by 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 limited to eight bytes, maximum.
- Low speed only supports interrupt and control types of transfers.
- The SOF packet is not received by low speed devices.

8.7 Error Detection and Recovery

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. ISO 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

USB employs three error detection mechanisms: bit stuff violations, PID check bits, and CRCs. A bit stuff violation exists if a packet receiver detects seven or more consecutive bit times without a differential (J -> K or K -> J) transition, as detected on the physical D+ and D- lines, between the start and end of a packet. A PID error exists if the four PID check bits are not complements of their respective packet identifier bits. A CRC error exists if the computed checksum remainder at the end of a packet reception is not zero.

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 8-6 lists error detection mechanisms, the types of packets to which they apply, and the appropriate packet receiver response.

Table 8-6. 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 Turnaround Timing

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 packet back. This time is referred to as the bus turnaround time and is tracked by the packet transmitter's bus turnaround timer. The timer starts counting on the SE0 to IDLE transition of the EOP strobe and stops counting when the IDLE to K SOP transition is detected. Both devices and the host require turnaround timers. The device bus turnaround time is defined by the worst case round trip delay plus the maximum device response delay (refer to Section 7.1.14). USB devices cannot time out earlier than 16 bit times after the end of the previous EOP and they must time out by 18 bit times. If the host wishes to indicate an error condition via a timeout, it must wait at least 18 bit times before issuing the next token to insure that all downstream devices have timed out.

As shown in Figure 8-20, the device uses its bus turnaround 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 must wait before sending out the next token. This wait interval guarantees that the host does not attempt to issue a token immediately after a false EOP.

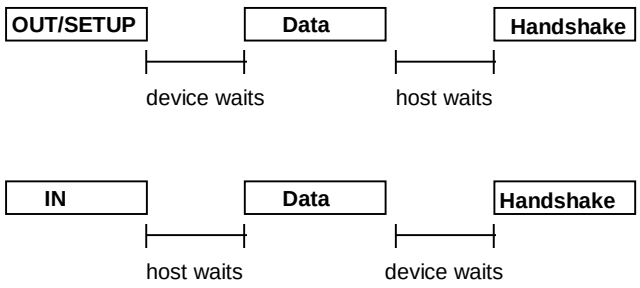

Figure 8-20. Bus Turnaround Timer Usage

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 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 sees 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 time out.

If the host receives a corrupted 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 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 time-out interval (with no bus transitions) greater than the worst case 6-bit time bit stuff interval. The second condition is that the transmitting device's bus turnaround timer must be guaranteed to expire. Note that the time-out interval is transaction speed sensitive. For full speed transactions, the host must wait 16 full speed bit times; for low speed transactions, it must wait 16 low speed bit times.

If the host receives a data packet with a valid CRC, it assumes that the packet is complete and need not delay in issuing the next token.

8.7.4 Babble and Loss of Activity Recovery

USB must be able to detect and recover from conditions which leave it waiting indefinitely for an end of packet or which leave the bus in something other than the idle state at the end of a frame. Loss of activity (LOA) is characterized by SOP followed by lack of bus activity (bus remains in J or K state) and no EOP at the end of a frame. Babble is characterized by an SOP followed by the presence of bus activity past the end of a frame. LOA and babble have the potential to either deadlock the bus or force out the beginning of the next 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 frame are prevented from transmitting past a 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 11.4.3.

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 and 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). Note: USB devices perform a reset operation in response to a Reset request to the upstream port from the host. When reset signaling has completed, the USB device is reset.

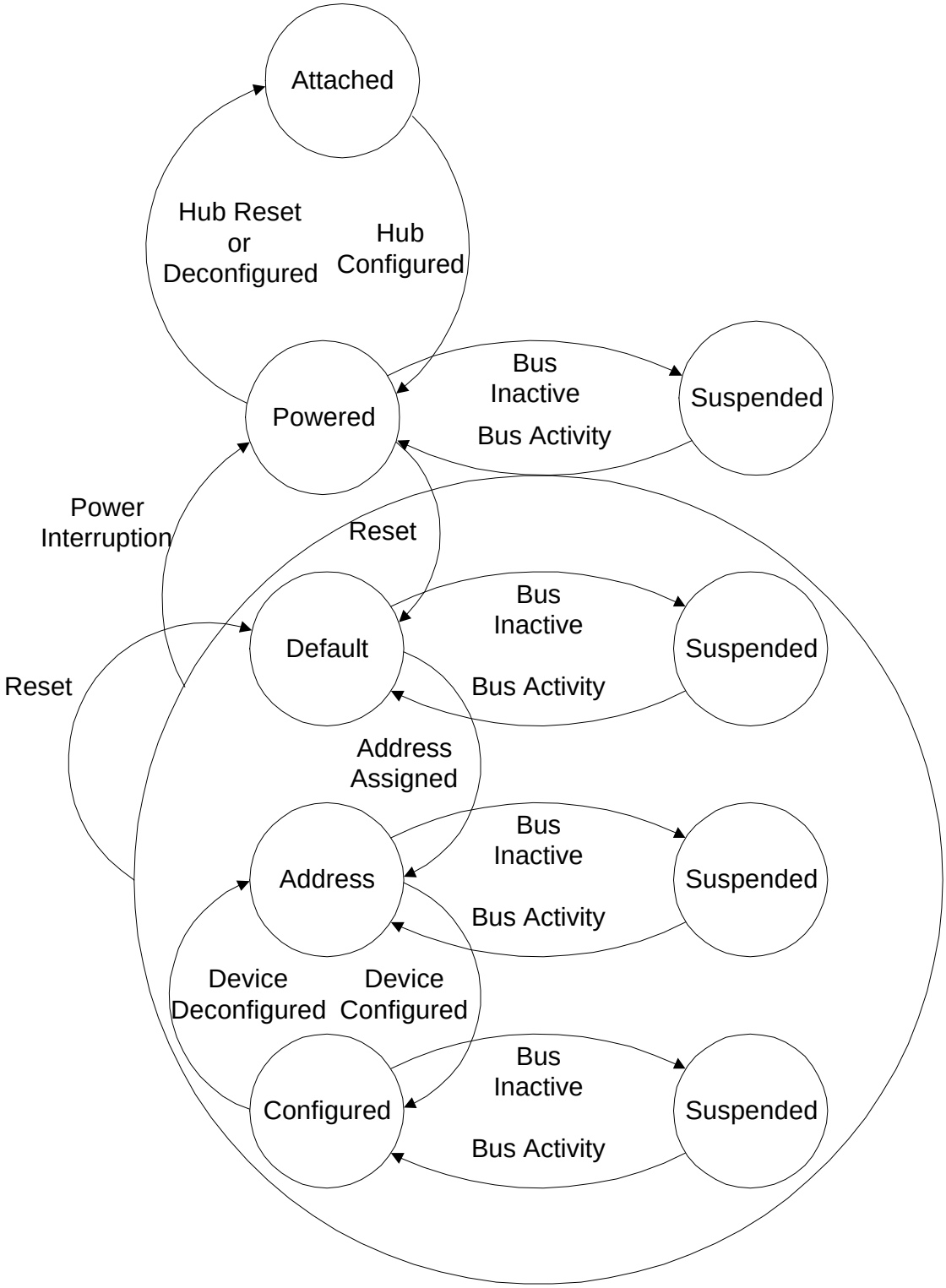

Figure 9-1. Device State Diagram

Table 9-1. Visible Device States

Attached	Powered	Default	Address	Configured	Suspended	State
No	--	--	--	--	--	Device is not attached to USB. Other attributes are not significant.
Yes	No	--	--	--	--	Device is attached to USB, but is not powered. Other attributes are not significant.
Yes	Yes	No	--	--	--	Device is attached to USB and powered, but has not been reset.
Yes	Yes	Yes	No	--	--	Device is attached to 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 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 USB, powered, has been reset, has unique address, is configured, and is not suspended. Host may now use the function provided by the device.
Yes	Yes	Yes	Yes	Yes	Yes	Device is, at minimum, attached to USB, has been reset, and is powered at the minimum suspend level. It may also have a unique address and be configured for use. However, since 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 USB through the hub to which they are attached. Externally powered USB devices are termed self-powered. These devices may already be powered before they are attached to the USB. A device may support both self-powered and bus-powered configurations. Some device configurations support either power source. Other device configurations may only be available if the device is externally 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 USB-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 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 loose its current configuration and address and return to the attached state.

A hub port must be powered in order to detect port status changes, including attach and detach. 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 shall 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.

9.1.1.4 Address Assigned

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 the USB device's function may be used, the device must be configured. From the device's perspective, configuration involves writing a non-zero value to the device configuration register. 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 exit suspend mode or a 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.

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, the following actions are undertaken:

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 Chapter 11 for more information). At this point, the USB device is in the attached 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 issues a port enable and reset command to that port.
4. The hub maintains the reset signal to that port for 10 ms. When the reset signal is released, the port has been enabled and the hub provides 100 mA of bus power to the USB device. The USB device is now in the powered state. All of its registers and state have been reset and it answers to the default address.
5. Before the USB device receives a unique address, it's default 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.
6. The host assigns a unique address to the USB device, moving the device to the addressed state.
7. The host reads the configuration information from the device by reading each configuration zero to n. This process may take several frames 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 configuration descriptor. 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 unconfigured
- Is not initially suspended

When a device is removed from a hub port, the host is notified of the removal. The host responds by disabling the hub port where the device was attached.

9.2.2 Address Assignment

When a USB device is attached, the host is responsible for assigning a unique address to the device 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 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, sets the maximum packet size for endpoints that require such limitation.

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 class 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 shall support the Get Interface and Set Interface requests to report or select a specific alternative setting for a specific 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 related class of devices that has been defined as a part of the USB specification. A class of devices is further subdivided into subclasses and within a class or subclass a protocol code defines how host software communicates with the device.

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. Some endpoints may be capable of different types of data transfers. However, once configured, a USB device endpoint uses only one data transfer method.

9.2.5 Power Management

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

9.2.5.1 Power Budgeting

For bus-powered devices, power is a limited resource. When a host detects the attachment of a bus-powered USB device, the host needs to evaluate the power requirements of the device. If USB device power requirements exceed available power, the device is not configured.

No USB device may require more than 100 mA when first attached. A configured bus-powered USB device attached to a self-powered hub may use up to 500 mA; however, some ports may not be able supply this much power and thus the device will not be usable.

All USB devices must support a suspended mode that requires less than 500 µA. A USB device automatically suspends when the bus is inactive, as previously described.

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 elsewhere in this document.

9.3 USB Device Requests

All USB devices respond to requests from the host on the device's default 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 following fields. Every setup packet has eight bytes, used as follows:

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bit-map	Characteristics of request D7 Data xfer 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 9-2)
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 phase

9.3.1 bmRequestType

This bit-mapped 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 phase.

The USB Specification defines a series of Standard requests that all devices must support. 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 only defines values for the *bRequest* field when the bits are reset to zero indicating a standard request (refer to Table 9-2).

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.

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 *bRequestType* field. If this field is zero, there is no data transfer phase.

9.4 Standard Device Requests

This section describes the standard device requests defined for all USB devices (refer to Table 9-2).

USB devices must respond to standard device requests whether the device has been assigned a non-default address or the device is currently configured.

Table 9-2. 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 9-3. 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 9-4. Descriptor Types

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5

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 below.

Feature Selector	Recipient	Value
DEVICE_REMOTE_WAKEUP	Device	1
ENDPOINT_STALL	Endpoint	0

If an unsupported or invalid request is made to a USB device, the device responds by indicating a stall condition on the pipe used for the request. Control pipes, including the default pipe, must accept a setup transaction even if they are stalled. The ClearStall request is used to clear a stalled pipe. After the stall condition is cleared by the host, system software continues normal accesses to the control pipe. If for any reason, the device becomes unable to communicate via its default pipe due to an error condition, the device must be reset to clear the condition and restart the default 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 Section 9.4 for a definition of which feature selector values are defined for which recipients.

A ClearFeature request that references a feature that cannot be cleared or that does not exist will cause a stall.

9.4.2 Get Configuration

This request returns the current device configuration.

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.

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 9.6.5)	Descriptor Length	Descriptor

The *wValue* field specifies the descriptor type in the high byte and the descriptor index in the low byte (refer to Table 9-4). 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 NULL data packet (refer to Chapter 5).

The standard request to a device supports three types of descriptors: DEVICE, CONFIGURATION, and STRING. 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 immediately 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.

All devices must provide a device descriptor and at least one configuration descriptor. If a device does not support a requested descriptor, it responds by stalling the pipe used for the request. A non-zero value as the first byte of a descriptor indicates the buffer contains a valid descriptor.

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 alternative setting.

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 *bRequestType* field specify the desired recipient. The data returned is the current status of the specified recipient.

A GetStatus request to a device returns the following information in little-endian order:

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)							

The *Self Powered* field indicates whether the device is currently bus-powered or 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 following information in little-endian order:

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

If the request is made to an endpoint, then the endpoint must be specified in the *wIndex* field. The upper byte of *xIndex* is reset to zero and the lower byte contains the endpoint number as follows:

D7	D6	D5	D4	D3	D2	D1	D0
Direction	Reserved (Reset to zero)			Endpoint Number			

For IN endpoints, D7 is set to one. For OUT endpoints, D7 is reset to zero.

A GetStatus request to an endpoint returns the following information:

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

If the endpoint is currently stalled, the *Stall* field is set to one. Otherwise the *Stall* field is reset to zero. The *Stall* field may be changed with the SetFeature and ClearFeature requests with the ENDPOINT_STALL feature selector. 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 stall has been removed, clearing the stall field results in the endpoint no longer returning a stall status. For this endpoints using a data toggle, clearing a stalled endpoint results in the data toggle being reinitialized to DATA0. This field is reset to zero after either the SetConfiguration or SetInterface request.

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.

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 *wValue* field specifies the desired configuration. This value must be zero or match a configuration value from a configuration descriptor. If the value is zero, the device is placed in its unconfigured state.

9.4.8 Set Descriptor

This request is optional. If a device supports this request, existing descriptors may be updated 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 9.6.5)	Descriptor Length	Descriptor

The *wValue* field specifies the descriptor type in the high byte and the descriptor index in the low byte (refer to Table 9-4). 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.

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	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 Section 9.4 for a definition of which feature selector values are defined for which recipients. A SetFeature request that references a feature that cannot be set or that does not exist causes a stall.

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	Alternative 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.

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
00000010B	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. This request causes the endpoint to begin monitoring the SOF frame number to track the position of a given frame in its pattern. The number of the frame in which the pattern began is returned to the host. This frame number is the one conveyed to the endpoint by the last SOF prior to the first frame of the pattern.

This value is only used for isochronous data transfers using implicit pattern synchronization. If the endpoint is not isochronous or is not using this method, this request is not supported by the endpoint and returns a stall.

The starting frame is reset to zero by a device reset or by configuring the endpoint either via a SetConfiguration or a SetInterface request.

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.

Class and vendor specific descriptors may be returned in one of two ways. Class and vendor specific descriptors that are related to standard descriptors are returned in the same data buffer as the standard descriptor immediately following the related standard descriptor.

If, for example, a class or vendor specific descriptor is related to an interface descriptor, the related class or vendor specific descriptor is placed between the interface descriptor and the interface's endpoint descriptors in the buffer returned in response to a GET_CONFIGURATION_DESCRIPTOR request. The length of a standard descriptor is not increased to accommodate device class or vendor specific extensions. Class or vendor specific descriptors follow the same format as standard descriptors with the length and type fields as the first two bytes of the specific descriptor.

Class or vendor specific descriptors that are not related to a standard descriptor are returned using class or vendor specific requests.

9.6 Standard USB Descriptor Definitions

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.

All USB devices have an endpoint zero used by the default pipe. The maximum packet size of a device's endpoint zero 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 endpoint zero. Other than the maximum packet size, the characteristics of endpoint zero are defined by this specification and are the same for all USB devices.

The *bNumConfigurations* field identifies the number of configurations the device supports.

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 0x210). This field identifies the release of the USB Specification that the device and its descriptors are compliant with.
4	*bDeviceClass*	1	Class	Class code (assigned by USB). If this field is reset to 0, 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 0xFE, 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. (For example, a CD-ROM device with audio and digital data interfaces that require transport control to eject CDs or start them spinning.) If this field is set to 0xFF, the device class is vendor specific.

Offset	Field	Size	Value	Description
5	*bDeviceSubClass*	1	SubClass	Subclass code (assigned by USB). These codes are qualified by the value of the *bDeviceClass* field. If the *bDeviceClass* field is reset to 0, this field must also be reset to 0. If the *bDeviceClass* field is not set to 0xFF, all values are reserved for assignment by USB.
6	*bDeviceProtocol*	1	Protocol	Protocol code (assigned by USB). 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 0, 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 0xFF, 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 USB)
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 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 Set Configuration 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 kBs 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 kBs bi-directional channel.

When the host requests the configuration descriptor, all related interface and endpoint descriptors are returned (refer to Section 9.4.2).

A USB device has one or more configuration descriptors. Each configuration has one or more interfaces and each interface has one 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. Within an interface, an isochronous endpoint's maximum packet size may also be adjusted.

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	CONFIGURATION
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 Set Configuration to select this configuration
6	*iConfiguration*	1	Index	Index of string descriptor describing this configuration
7	*bmAttributes*	1	Bitmap	Configuration characteristics D7 Bus Powered D6 Self Powered D5 Remote Wakeup D4..0 Reserved (reset to 0) A device configuration that uses power from the bus and a local source sets both D7 and D6. The actual power source at runtime may be determined using the Get Status device request. If a device configuration supports remote wakeup, D5 is set to 1.

Offset	Field	Size	Value	Description
8	*MaxPower*	1	mA	Maximum power consumption of 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. Host software may determine the cause of the failure by checking the status and noting the loss of the device's power source.

9.6.3 Interface

This descriptor describes a specific interface provided by the associated configuration. A configuration provides one or more interfaces, each with its own endpoint descriptors describing a unique set of endpoints within the configuration. When a configuration supports more than one interface, the endpoints for a particular interface immediately follow the interface descriptor in the data returned by the Get Configuration request. An interface descriptor is always returned as part of a configuration descriptor. It cannot be directly accessed with a Get or Set Descriptor 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 Set Interface request is used to select an alternate setting or to return to the default setting. The Get Interface 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 only uses endpoint zero, no endpoint descriptors follow the interface descriptor and the interface identifies a request interface that uses the default pipe attached to endpoint zero. In this case, the *bNumEndpoints* field shall be set to zero.

An interface descriptor never includes endpoint zero in the number of endpoints.

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	INTERFACE Descriptor Type
2	*bInterfaceNumber*	1	Number	Number of interface. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration.
3	*bAlternateSetting*	1	Number	Value used to select alternate setting for the interface identified in the prior field
4	*bNumEndpoints*	1	Number	Number of endpoints used by this interface (excluding endpoint zero). If this value is 0, this interface only uses endpoint zero.
5	*bInterfaceClass*	1	Class	Class code (assigned by USB) If this field is reset to 0, the interface does not belong to any USB specified device class. If this field is set to 0xFF, the interface class is vendor specific. All other values are reserved for assignment by USB.
6	*bInterfaceSubClass*	1	SubClass	Subclass code (assigned by USB). These codes are qualified by the value of the *bInterfaceClass* field. If the *bInterfaceClass* field is reset to 0, this field must also be reset to 0. If the *bInterfaceClass* field is not set to 0xFF, all values are reserved for assignment by USB.

Offset	Field	Size	Value	Description
7	*bInterfaceProtocol*	1	Protocol	Protocol code (assigned by USB). These codes are qualified by the value of the *bInterfaceClass* and the *bInterfaceSubClass* fields. If an interface supports class-specific requests, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to 0, the device does not use a class specific protocol on this interface. If this field is set to 0xFF, the device uses a vendor specific protocol for this interface.
8	*iInterface*	1	Index	Index of string descriptor describing this interface

9.6.4 Endpoint

Each endpoint used for an interface has its own descriptor. This descriptor contains the information required by the host to determine the bandwidth requirements of each endpoint. An endpoint descriptor is always returned as part of a configuration descriptor. It cannot be directly accessed with a Get or Set Descriptor request. There is never an endpoint descriptor for endpoint zero.

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	ENDPOINT Descriptor Type
2	*bEndpointAddress*	1	Endpoint	The address of the endpoint on the USB device described by this descriptor. The address is encoded as follows: Bit 0..3: The endpoint number Bit 4..6: Reserved, reset to 0 Bit 7: Direction, ignored for control endpoints 0 OUT endpoint 1 IN endpoint

Offset	Field	Size	Value	Description
3	*bmAttributes*	1	Bit Map	This field describes the endpoint's attributes when it is configured using the *bConfigurationValue*. Bit 0 .. 1: Transfer Type 00 Control 01 Isochronous 10 Bulk 11 Interrupt All other bits are reserved
4	*wMaxPacketSize*	2	Number	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. For isochronous endpoints, this value is used to reserve the bus time in the schedule, required for the per frame data payloads. The pipe may, on an ongoing basis, actually use less bandwidth than that reserved. The device reports, if necessary, the actual bandwidth used via its normal, non-USB defined mechanisms. For interrupt, bulk, and control endpoints smaller data payloads may be sent, but will terminate the transfer and may or may not require intervention to restart. Refer to Chapter 5 for more information.
6	*bInterval*	1	Number	Interval for polling endpoint for data transfers. Expressed in milliseconds. This field is ignored for bulk and control endpoints. For isochronous endpoints this field must be set to 1. For interrupt endpoints, this field may range from 1 to 255.

9.6.5 String

String descriptors are optional. As noted previously, if a device does not support string descriptors, all references to string descriptors within device, configuration, and interface descriptors must be reset to zero.

String descriptors use UNICODE encodings as defined by *The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volumes 1 and 2*, The Unicode Consortium, Addison-Wesley Publishing Company, Reading, Massachusetts. The strings in a USB device may support multiple languages. When requesting a string descriptor, the requester specifies the desired language using a sixteen-bit language ID (LANGID) defined by Microsoft for Windows as described in *Developing International Software for Windows 95 and Windows NT*, Nadine Kano, Microsoft Press, Redmond, Washington. String index 0 for

all languages returns an array of two-byte LANGID codes supported by the device. A USB device may omit all string descriptors.

The UNICODE string descriptor is not NULL terminated. The string length is computed by subtracting two from the value of the first byte of the descriptor.

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

9.7 Device Class Definitions

All devices must support the above registers and descriptor definitions. Most devices provide additional registers and possibly, descriptors for device-specific extensions. In addition, devices may provide extended services which are common to a group of devices. In order to define a class of devices, the following information must be provided to completely define the appearance and behavior of the device class.

9.7.1 Descriptors

If the class requires any specific definition of the standard descriptors, the class definition must include those requirements as part of the class definition. In addition, if the class defines a standard extended set of descriptors, they must also be fully defined in the class definition. Any extended descriptor definitions should follow the approach used for standard descriptors; for example, all descriptors should begin with a length field.

9.7.2 Interface(s) and Endpoint Usage

When a class of devices is standardized, the interfaces used by the devices including how endpoints are used must be included in the device class definition. Devices may further extend a class definition with proprietary features as long as they meet the base definition of the class.

9.7.3 Requests

All of the requests specific to the class must be defined.

9.8 Device Communications

The USB communications model characterizes data and control traffic between the host and a given device across the USB interconnect. The host and the device are divided into the distinct layers shown in Figure 9-2.

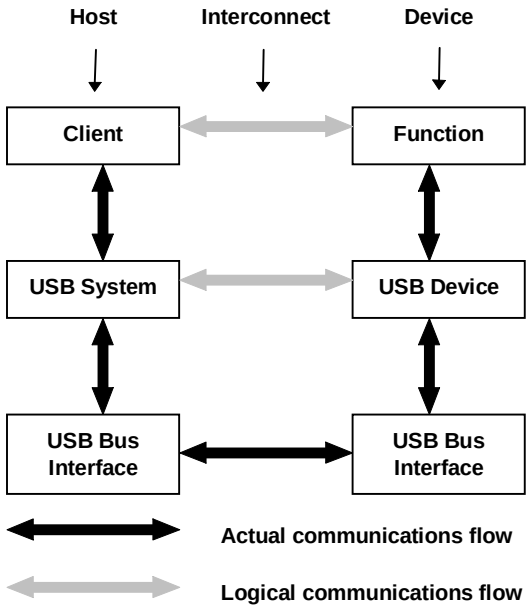

Figure 9-2. Interlayer Communications Model

The actual communication on the host, as indicated by vertical arrows, takes place via SPIs. The interlayer relationships on the device are implementation-specific. Between the host and device, all communications must ultimately occur on the physical USB wire. However, there are logical host-device interfaces between horizontal layers. Between client software, resident on the host, and the function provided by the device, communications are typified by a contract based on the needs of the application currently using the device and the capabilities provided by the device. This client-function interaction creates the requirements for all of the underlying layers and their interfaces.

This section describes the communications model from the point of view of the device and its layers. Figure 9-3 illustrates, based on the overall view introduced in Chapter 8, the device's view of its communication with the host.

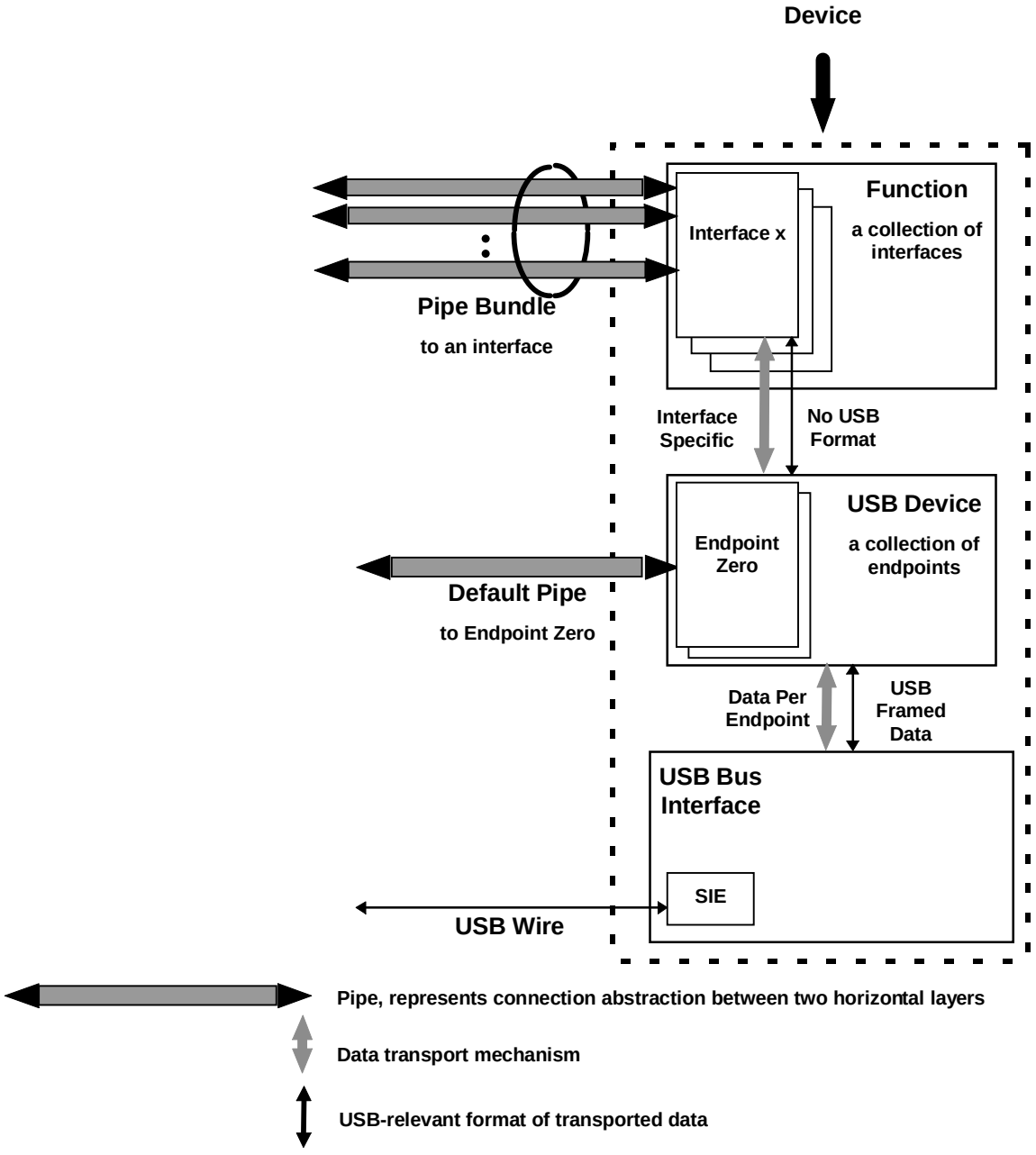

Figure 9-3. Device Communications

The USB bus interface handles interactions among the electrical and protocol layers (refer to Chapters 7 and 8). The USB device layer presents a uniform abstraction of the USB device to the host. It is this layer that is primarily described here. The function layer uses the capabilities provided by the USB device layer, combined into a given interface, to support the requirements of a host-based application.

A USB device acts as a collection of endpoints, each capable of supporting different types of pipes. Each pipe can support one of the following transfer types at a time:

- Control
- Isochronous
- Interrupt
- Bulk

These transfer types are described in more detail in Chapter 5. Each of the transfer types, however, requires the associated endpoint to behave in a certain fashion. A given endpoint may support a variety of transfer types. However, once a pipe is associated with an endpoint, the endpoint only uses a single transfer type. In this section, when discussing the behavior of an endpoint for a given transfer type, it is assumed that the endpoint has been associated with a pipe supporting that specific transfer type. The basic communication mechanisms used by endpoints are:

- Pipe mode
- Start of Frame (SOF) synchronization
- Handshakes
- Data toggles

The mode of a pipe indicates whether the data flow across the pipe is stream or message mode. Devices may use the SOF as generated by the host to synchronize their internal clocks. Devices may use handshakes and data toggles to implement error and flow control.

Traffic between a client and a function may require a certain transport rate. The client, USB and the function all will be using, at best, slightly different clock rates. To ensure that all of the required data can be delivered with minimum buffering required, the various clocks must be synchronized. Refer to Chapter 5 for a discussion of the synchronization options. Additionally, in order to support the just-in-time delivery capability implied by clock synchronization, the size of the data packets transmitted between the host and the device will be normalized such that variations in size over time are minimized. To support data flows in which the loss of data is acceptable as long as the loss can be accurately communicated, sample headers may be used by the host and the device to communicate the expected transmission volumes. Refer to Chapter 5 for the definition of sample headers.

These basic communication mechanisms are described, from the device's point of view, in greater detail below. Each of the different transfer types uses these basic communication mechanisms in different ways.

9.8.1 Basic Communication Mechanisms

This section describes in more detail the basic communication mechanisms as supported by the USB Device layer.

9.8.1.1 Pipe Mode

A pipe supports either stream or message mode transfers. In stream mode, the data flow is considered to be a unidirectional serial stream of samples. In message mode, data is delivered as a related set of bytes. Message mode pipes are always considered to be capable of being bi-directional.

A stream mode pipe is always unidirectional. When in the stream mode, the endpoint expects to receive a token either requesting the endpoint to send data or alerting it that data will be sent to it. The amount of data sent will always be equal to or less than the current MaxPacketSize for the endpoint.

Message transfers begin with a command from the host to the device. The device may respond to the command with data, the host may follow the command with data for the device, or the command may