Note on USB 2.0 Bit Rate: This specification draft calls out a data rate of 480Mb/s. This is the target rate for which the Electrical Working Group is designing and prototyping; this rate needs to be confirmed with completed validation of prototype IC's operating on test boards.

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, and high-speed PING protocol.

8.1 Bit Ordering

Bits are sent out onto the bus least-significant bit (LSb) first, followed by the next LSb, through to the most-significant bit (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 7.1.10). The last two bits in the SYNC field are a marker that is used to identify the end of the SYNC field and, by inference, the start of the PID.

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) ~~delimeter~~delimiter 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 . 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 ensures reliable decoding of the PID so that the remainder of the packet is interpreted correctly. The PID check field is generated by performing a one's complement of the packet type field. A PID error exists if the four PID check bits are not complements of their respective packet identifier bits.

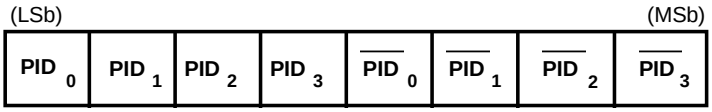

Figure 8-1. PID Format

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	0001B	Address + endpoint number in host-to-function transaction
	IN	1001B	Address + endpoint number in function-to-host transaction
	SOF	0101B	Start-of-Frame marker and frame number
	SETUP	1101B	Address + endpoint number in host-to-function transaction for SETUP to a control pipe
~~Data~~	~~DATA0~~	~~0011B~~	~~Data packet PID even~~
	~~DATA1~~	~~1011B~~	~~Data packet PID odd~~
Data	DATA0	0011B	Data packet PID even
	DATA1	1011B	Data packet PID odd
	DATA2	0111B	Data packet PID high-speed, high bandwidth isochronous transaction in a microframe (see section 5.9.2 for more information)
	MDATA	1111B	Data packet PID high-speed for split and high bandwidth isochronous transactions (see sections 5.9.2 and 11.19,11.20 for more information)

PID Type	PID Name	PID[3:0]*	Description
~~Handshake~~	~~ACK~~	~~0010B~~	~~Receiver accepts error-free data packet~~
	~~NAK~~	~~1010B~~	~~Rx device cannot accept data or Tx device cannot send data~~
	~~STALL~~	~~1110B~~	~~Endpoint is halted or a control pipe request is not supported.~~
Handshake	ACK	0010B	Receiver accepts error-free data packet
	NAK	1010B	Receiving device cannot accept data or transmitting device cannot send data
	STALL	1110B	Endpoint is halted or a control pipe request is not supported.
	NYET	0110B	No response yet from receiver (see sections 8.5.1 and 11.17-11.20)
~~Special~~	~~PRE~~	~~1100B~~	~~Host-issued preamble. Enables downstream bus traffic to low-speed devices.~~
Special	PRE	1100B	Host-issued preamble. Enables downstream bus traffic to low-speed devices.
	ERR	1100B	Split Transaction Error Handshake (reuses PRE value)
	XOUT	0000B	High-speed Extended OUT Token (see section 8.4.2)
	XIN	1000B	High-speed Extended IN Token (see section 8.4.2)
	PING	0100B	High-speed flow control probe for a bulk/control endpoint (see section 8.5.1)

*Note: PID bits are shown in MSb order. When sent on the USB, the rightmost bit (bit 0) will be sent first.

PIDs are divided into four coding groups: token, data, handshake, and special, with the first two transmitted PID bits (PID<0:1>) 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 zero and must be programmed by the host during the enumeration process. Function address zero is reserved as the default address and may not be assigned to any other use.

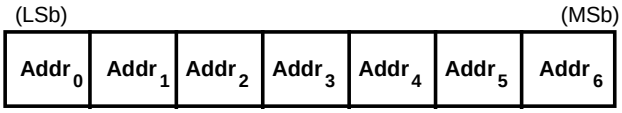

Figure 8-2. ADDR Field

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 endpoint is required. Except for endpoint address zero, endpoint numbers are function-specific. The endpoint field is defined for IN, SETUP, and OUT token PIDs only. All functions must support a control pipe at endpoint number zero (the Default Control Pipe). Low-speed devices support a maximum of three pipes per function: a control pipe at endpoint number zero plus two additional pipes (either two control pipes, a control pipe and a interrupt endpoint, or two interrupt endpoints). Full-speed functions may support up to the maximum of 16 endpoint numbers of any type.

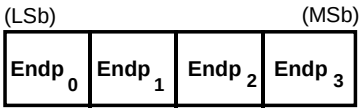

Figure 8-3. Endpoint Field

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 7FFH, and is sent only in SOF tokens at the start of each frame.

8.3.4 Data Field

The data field may range from zero to ~~1,023~~1,024 bytes and must be an integral number of bytes. Figure 8-4 shows the format for multiple bytes. Data bits within each byte are shifted out LSb first.

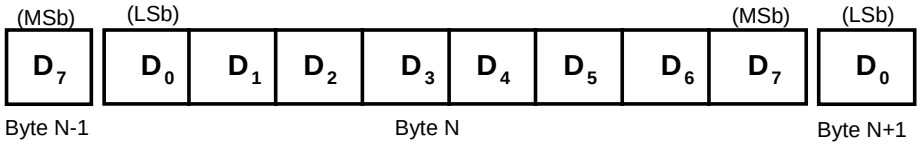

Figure 8-4. Data Field Format

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

For CRC generation and checking, the shift registers in the generator and checker are seeded with an all-ones pattern. For each data bit sent or received, the high order bit of the current remainder is XORed with

the data bit and then the remainder is shifted left one bit and the low-order bit set to zero. If the result of that XOR is one, then the remainder is XORed with the generator polynomial.

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

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

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

8.3.5.1 Token CRCs

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

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

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

8.3.5.2 Data CRCs

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

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

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

8.4 Packet Formats

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

8.4.1 Token Packets

Figure 8-5 shows the field formats for a token packet. A token consists of a PID, specifying either IN, OUT, or SETUP packet type; and ADDR and ENDP fields. 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.

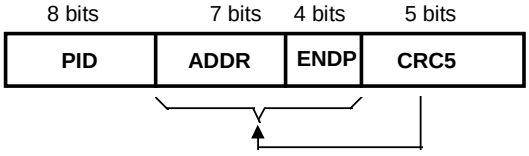

Figure 8-5. Token Format

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

8.4.2 Extended Token Packets

USB defines two extended token PIDs: XIN and XOUT. These are 4 byte token packets compared to other normal 3 byte token packets. The extended token packets provide additional transaction types with additional transaction specific information. The two extended tokens are used to support split transactions between the host controller communicating with a hub operating at high speed with full/low speed devices attached below. The two extended tokens for split transactions are: start-split (XOUT) and complete-split (XIN) transactions. Other possible extended token encodings are reserved.

8.4.2.1 Split Transactions

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

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

<<This is too confusing and needs to be rewritten.>>

A normal full-/low-speed IN transaction form is conceptually “converted” into its start-split and complete-split transactions by first taking the IN transaction and separating it into two parts: the first part is the IN token packet and the second part is a repeat of the IN token packet and the data and handshake packets (if present).

The first part (the IN token packet) has a start-split extended token packet prepended and that becomes the start-split high-speed transaction. A handshake packet may also be appended for some transfer types.

The second part (the IN, data and handshake packets) has a complete-split extended token packet prepended and that becomes the complete-split high-speed transaction.

Figure 8-6 shows this conceptual form “conversion”. The arrows in the figure simply show which packets of the original transaction form correspond to the packets in the split transaction form.

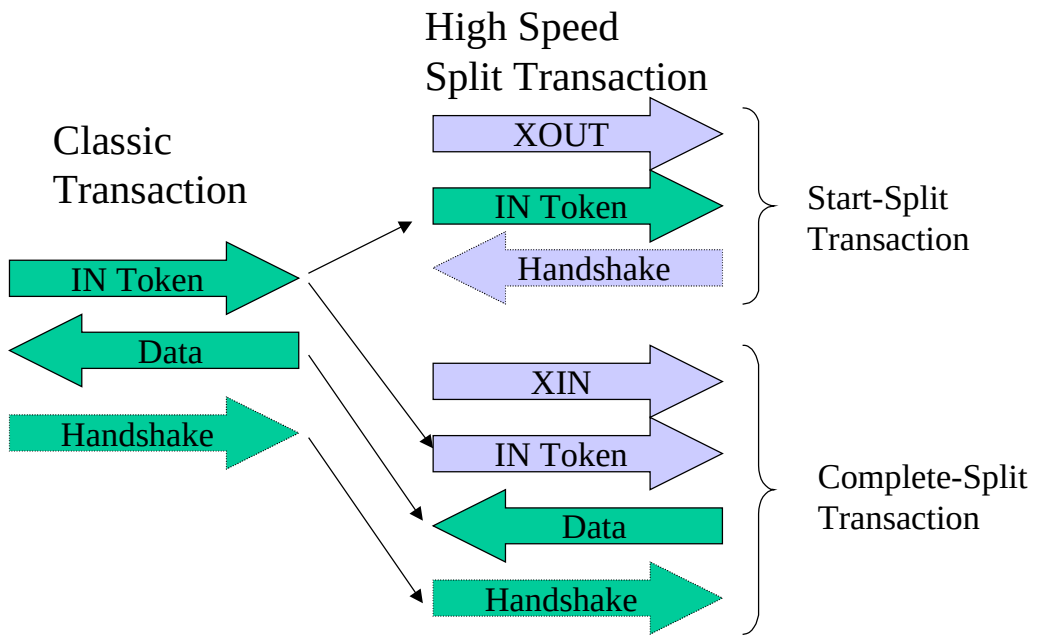

Figure 8-6. Conversion of IN Transaction to High-Speed Split IN Transaction

A normal full-/low-speed OUT transaction form is similarly conceptually "converted" into its start-split and complete-split transactions by first taking the OUT transaction and separating it into two parts: the first part is the OUT token and data packet and the second part is the OUT token packet and the handshake packet (if present).

The first part (the OUT token and data packets) has a start-split extended token packet prepended and that becomes the start-split high-speed transaction. Some transfer types may also have a handshake packet appended.

The second part (the OUT token and the handshake packet) has the complete-split extended token packet prepended and that becomes the complete-split high-speed transaction.

Figure 8-7 shows this conceptual form "conversion". The arrows in the figure simply show which packets of the original transaction form correspond to the packets in the split transaction form.

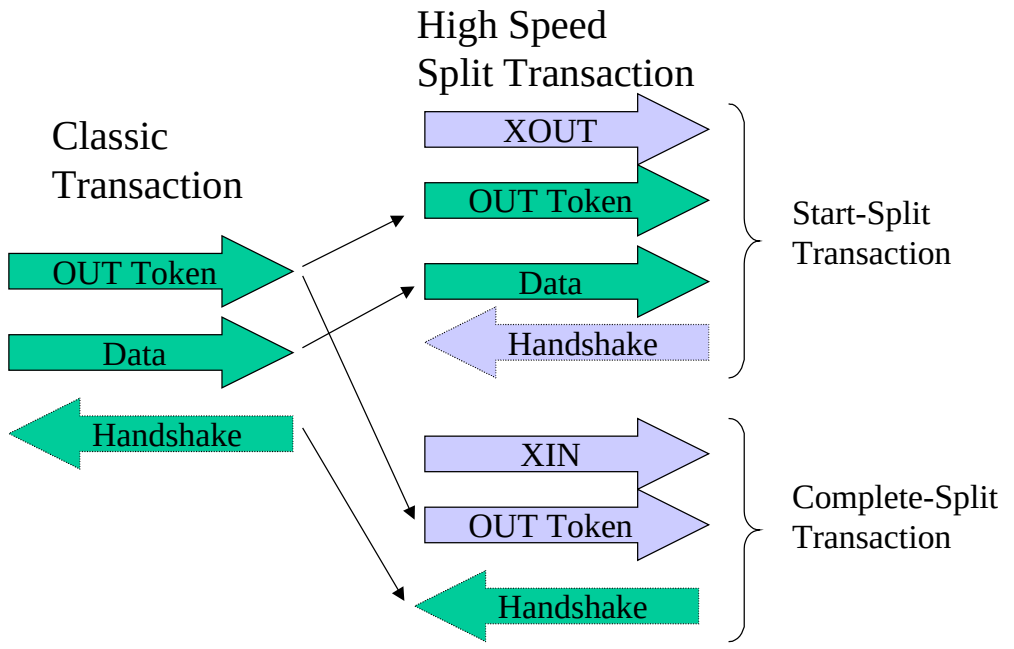

Figure 8-7. Conversion of OUT Transaction to High-Speed Split OUT Transaction

The next two sections describe the fields composing the start- and complete-split extended token packets. Figure 8-8 and Figure 8-9 show the fields in the split-transaction extended token packets. Two extended token PIDs are defined: XIN (1000b) and XOUT (0000b). The extended token follows the general token format and starts with a PID field (after a SYNC) and ends with a CRC5 field (and EOP).

Start-split and complete-split extended token packets are both 4 bytes long.

8.4.2.2 Start-Split Transaction Extended Token

Field	XOUT PID	X	Hub Addr	Bus	S	E	ET	CRC5
Bits	8	1	7	7	1	1	2	5

Figure 8-8. Start-Split (XOUT) Extended Token

The X field must be set to zero to indicate that this is a start-split transaction.

The Hub addr field contains the USB device address of the hub supporting the specified full-/low-speed device for this full-/low-speed transaction.

The Bus field contains a bus number to identify the bus of the target hub for which this full-/low-speed transaction is destined. The relationship of bus number to downstream facing port is described in the Hub Descriptor in Chapter 11. The value must be set to 0 if the hub only supports a single full-/low-speed bus for all downstream facing ports.

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

- 0 – Low speed
- 1 – Full speed

For full-speed isochronous OUT start-splits, the S[1] and E fields specify how the high speed data payload corresponds to data for a full-speed data packet as follows:

- S=1 and E=1: This high-speed data is all of the full-speed data payload.
- S=1 and E=0: This high-speed data is the beginning of the full-speed data payload
- S=0 and E=0: This high-speed data is the middle of the full-speed data payload
- S=0 and E=1: This high-speed data is the end of the full-speed data payload

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

The ET field specifies the endpoint type of the full-/low-speed transaction as follows:

- 00 – isochronous
- 01 – interrupt
- 10 – bulk
- 11 - control

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

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

8.4.2.3 Complete-Split Transaction Extended Token

Field	XIN PID	X	Hub Addr	Bus	S	U	ET	CRC5
Bits	8	1	7	7	1	1	2	5

Figure 8-9. Complete-Split (XIN) Transaction Extended Token

The X field must be set to 0B to indicate that this is a complete-split transaction.

The U bit is reserved/unused and must be set to 0B.

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

8.4.3 Start-of-Frame Packets

Start-of-Frame (SOF) packets are issued by the host at a nominal rate of once every 1.00ms ±0.0005ms for a full-speed bus and 125us ±0.000<<TBD>>ms for a high speed bus. SOF packets consist of a PID indicating packet type followed by an 11-bit frame number field as illustrated in Figure 8-10.

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

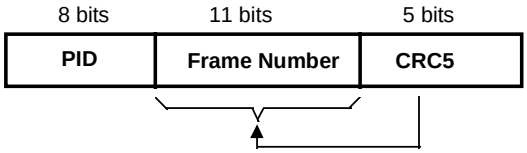

Figure 8-10. SOF Packet

The SOF token comprises the token-only transaction that distributes an SOF marker and accompanying frame number at precisely timed intervals corresponding to the start of each frame. All high-speed and full-speed functions, including hubs, receive the SOF packet. The SOF token does not cause any receiving function to generate a return packet; therefore, SOF delivery to any given function cannot be guaranteed. The SOF packet delivers two pieces of timing information. A function is informed that an SOF has occurred when it detects the SOF PID. Frame timing sensitive functions, which do not need to keep track of frame number (e.g., a hub), need only decode the SOF PID; they can ignore the frame number and its CRC. If a function needs to track frame number, it must comprehend both the PID and the time stamp. Full-speed devices that have no particular need for bus timing information may ignore the SOF packet.

8.4.3.1 USB Frames and Microframes

USB defines a full-speed 1ms frame time indicated by a Start Of Frame (SOF) packet each and every 1ms period with defined jitter tolerances. USB also defines a high speed microframe with a 125µs frame time with related jitter tolerances (See Chapter 7). The host controller generates an SOF packet every 1ms for full speed links. The host controller also generates SOF packets after the next seven 125µs periods for high speed links. Figure 8-11 shows the relationship between microframes and frames.

<<figutre>>

Figure 8-11. Relationshipo between Frames and Microframes

High speed devices see an SOF packet with the same frame number eight times (every 125µs) during each 1ms period. If desired, a high-speed device can locally determine a particular microframe “number” by detecting the SOF that had a different frame number than the previous SOF and treating that as microframe 0. The next seven SOFs with the same frame number can be treated as microframes 1 through 7.

8.4.4 Data Packets

A data packet consists of a PID, a data field containing zero or more bytes of data, and a CRC as shown in Figure 8-12. There are ~~two~~four types of data packets, identified by differing PIDs: ~~DATA0 and DATA1.~~DATA0, DATA1, DATA2 and MDATA. Two data packet PIDs (DATA0 and DATA1) are defined to support data toggle synchronization (refer to Section 8.6~~).~~

). All four data PIDs are used in data PID sequencing for high bandwidth high speed endpoints (refer to Section 5.9). Three data PIDs (MDATA, DATA0, DATA1) are used in split transactions (refer to Section 11.17-11.20).

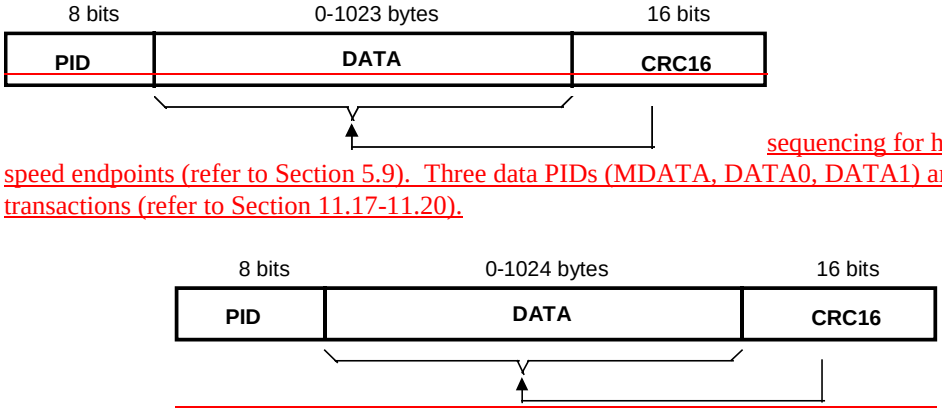

Figure 8-12. Data Packet Format

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

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

8.4.5 Handshake Packets

Handshake packets, as shown in Figure 8-13, consist of only a PID. Handshake packets are used to report the status of a data transaction and can return values indicating successful reception of data, command acceptance or rejection, flow control, and halt conditions. Only transaction types that support flow control can return handshakes. Handshakes are always returned in the handshake phase of a transaction and may be returned, instead of data, in the data phase. Handshake packets are delimited by an EOP after one byte of packet field. If a packet decodes as an otherwise valid handshake but does not terminate with an EOP after one byte, it must be considered invalid and ignored by the receiver.

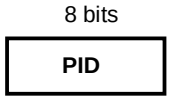

Figure 8-13. Handshake Packet

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

- **STALL** is returned by a function in response to an IN token or after the data phase of an OUT transaction (see Figure 8-16 and Figure 8-20). STALL indicates that a function is unable to transmit or receive data, or that a control pipe request is not supported. The host is not permitted to return a STALL under any condition.

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

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

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

8.4.6 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. Note that if an error occurs during the transmission of the token to the function, the function will not respond with any packets until the next token is received and successfully decoded.

8.4.6.1 Function Response to IN Transactions

Table 8-2 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 halt or a flow control condition, it issues a STALL or NAK handshake, respectively. If the function is able to issue data, it does so. If the received token is corrupted, the function returns no response.

Table 8-2. Function Responses to IN Transactions

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

8.4.6.2 Host Response to IN Transactions

Table 8-3 shows the host response to an IN transaction. The host is able to return only one type of handshake: ACK. If the host receives a corrupted data packet, it discards the data and issues no response. If the host cannot accept data from a function, (due to problems such as internal buffer overrun) this condition is considered to be an error and the host returns no response. If the host is able to accept data and the data packet is received error-free, the host accepts the data and issues an ACK handshake.

Table 8-3. 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

8.4.6.3 Function Response to an OUT Transaction

Handshake responses for an OUT transaction are shown in Table 8-4. Assuming successful token decode, a function, upon receiving a data packet, may return any one of the three handshake types. If the data packet was corrupted, the function returns no handshake. If the data packet was received error-free and the function's receiving endpoint is halted, the function returns STALL. If the transaction is maintaining sequence bit synchronization and a mismatch is detected (refer to Section 8.6 for details), then the function returns ACK and discards the data. If the function can accept the data and has received the data error-free, it returns ACK. If the function cannot accept the data packet due to flow control reasons, it returns NAK.

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

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

8.4.6.4 Function Response to a SETUP Transaction

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

8.5 Transaction Formats

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

8.5.1 NAK Limiting via Ping Flow Control

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

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

The host controller queries the high speed device endpoint with a PING token. The PING token packet is a normal token packet as shown in Figure 8-5 with a PID value of 0100b. The endpoint either responds to the PING with a NAK or an ACK handshake.

A NAK handshake indicates that the endpoint does not have space for a MAXPACKET data payload. The host controller will retry the PING at some future time to query the endpoint again.

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

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

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

The endpoint may also respond to the OUT/DATA transaction with a NAK handshake. This means that the endpoint did not accept the data and does not have space for a data payload at this time. The host controller will also return to using a PING token until the endpoint indicates it has space. A NAK response is expected to be an unusual occurrence. It suggests that the endpoint responded to a previous OUT or PING with an incorrect handshake, or that the endpoint transitioned into a state where it (temporarily) could not accept data.

Figure 8-14 shows the host controller state machine for the interactions and transitions between PING and OUT/DATA tokens and the allowed ACK, NAK and NYET handshakes for the PING mechanism.

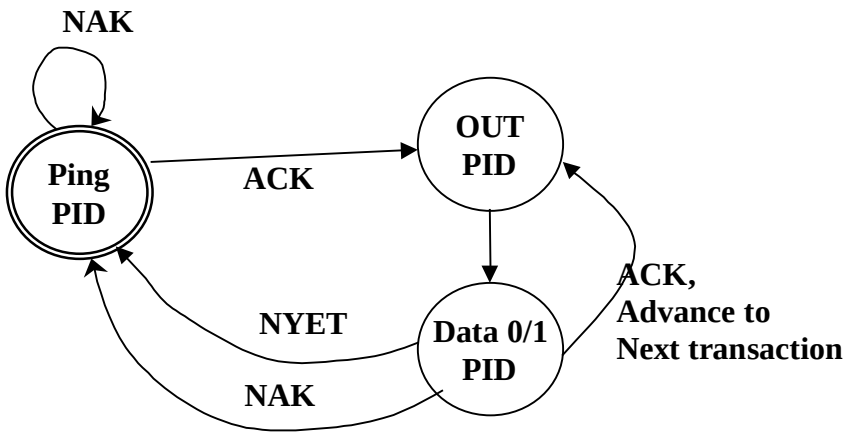

Figure 8-14. Host Bulk/Control OUT Ping State Machine

Figure 8-15 shows the device endpoint state machine for PING based on the buffer space the endpoint has available. In each figure, the state with the double line indicates the reset or initial condition.

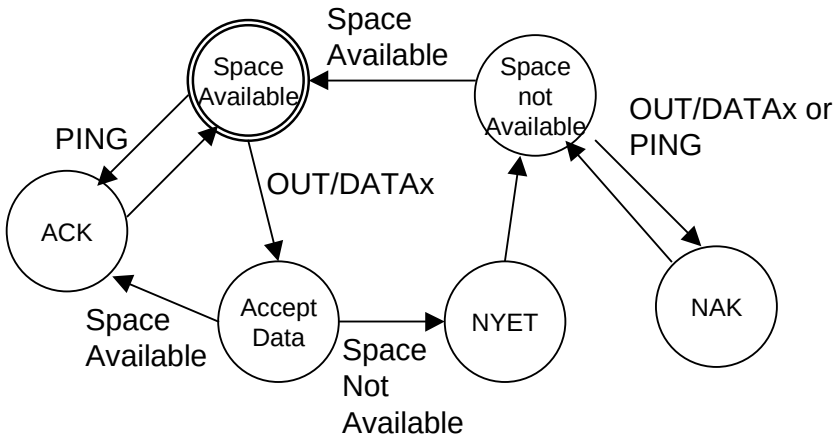

Figure 8-15. Device Bulk/Control OUT Ping State Machine

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

Note: The PING protocol is also not included as part of the split-transaction protocol definition. Split transactions have their own efficient flow control protocol and therefore don't require PING. Hubs must support PING on their control endpoint, but PING is not defined for the split-transactions that are used to communicate with full/low-speed devices supported by a hub.

8.5.2 Bulk Transactions

Bulk transaction types are characterized by the ability to guarantee error-free delivery of data between the host and a function by means of error detection and retry. Bulk transactions use a three-phase transaction consisting of token, data, and handshake packets as shown in Figure 8-16. Under certain flow control and halt conditions, the data phase may be replaced with a handshake resulting in a two-phase transaction in which no data is transmitted.

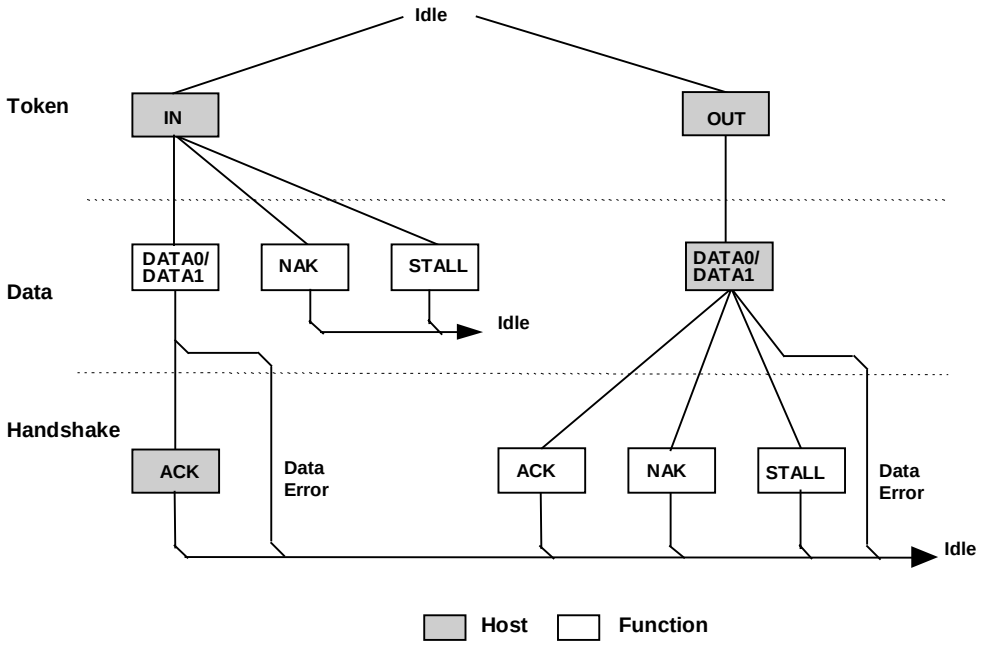

Figure 8-16. Bulk Transaction Format

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

When the host is ready to transmit bulk data, it first issues an OUT token packet followed by a data packet. If the data is received without error by the function it will return 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 (e.g., buffer full).
- If the endpoint was halted, STALL is returned to indicate that the host should not retry the transmission because there is an error condition on the function.

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

Figure 8-17 shows the sequence bit and data PID usage for bulk reads and writes. Data packet synchronization is achieved via use of the data sequence toggle bits and the DATA0/DATA1 PIDs. A bulk endpoint's toggle sequence is initialized to DATA0 when the endpoint experiences any configuration event (configuration events are explained in Sections 9.1.1.5 and 9.4.5). Data toggle on an endpoint is NOT initialized as the direct result of a short packet transfer or the retirement of an IRP.

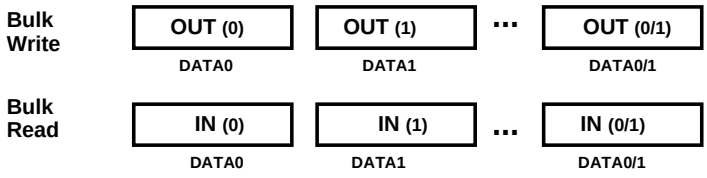

Figure 8-17. Bulk Reads and Writes

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

8.5.3 Control Transfers

Control transfers minimally have two transaction stages: Setup and Status. A control transfer may optionally contain a Data stage between the Setup and Status stages. During the Setup stage, a SETUP transaction is used to transmit information to the control endpoint of a function. SETUP transactions are similar in format to an OUT, but use a SETUP rather than an OUT PID. Figure 8-18 shows the SETUP transaction format. A SETUP always uses a DATA0 PID for the data field of the SETUP transaction. The function receiving a SETUP must accept the SETUP data and respond with ACK, if the data is corrupted, discard the data and return no handshake.

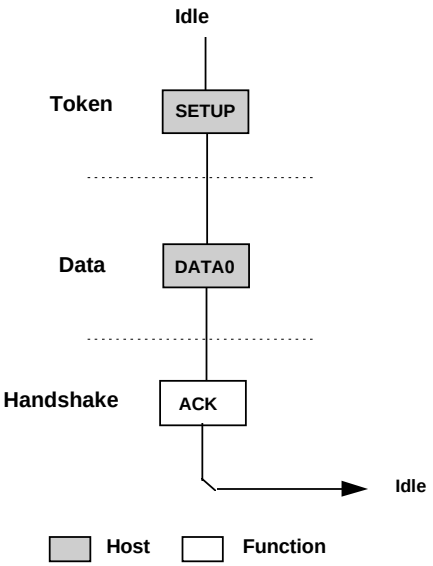

Figure 8-18. Control SETUP Transaction

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) that carry the maximum packet size. Any remaining data is sent as a residual in the last transaction.

The Status stage of a control transfer is the last 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 8-19 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.

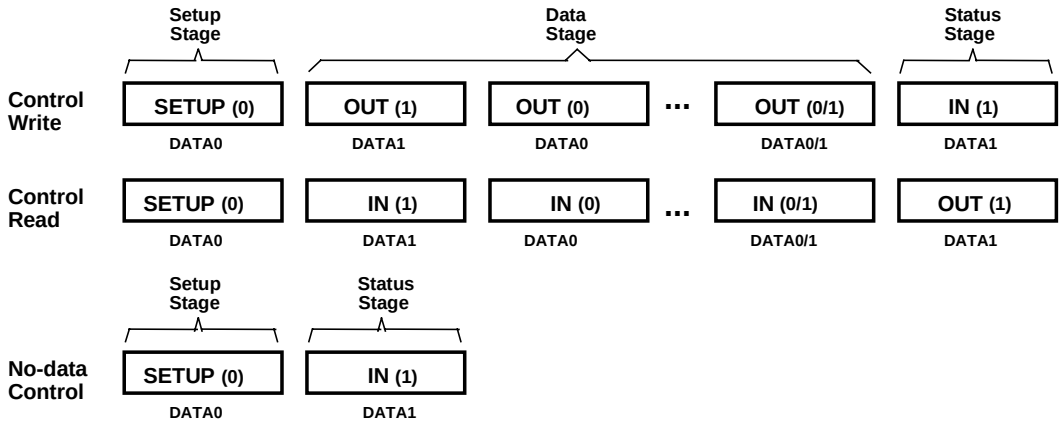

Figure 8-19. Control Read and Write Sequences

When a STALL handshake is sent by a control endpoint in either the Data or Status stages of a control transfer, a STALL handshake must be returned on all succeeding accesses to that endpoint until a SETUP PID is received. The endpoint is not required to return a STALL handshake after it receives a subsequent SETUP PID.

8.5.3.1 Reporting Status Results

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

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

Status reporting is always in the function-to-host direction. The Table 8-5 summarizes the type of responses required for each. Control write transfers return status information in the data phase of the Status stage transaction. Control read transfers return status information in the handshake phase of a Status stage transaction, after the host has issued a zero-length data packet during the previous data phase.

Table 8-5. Status Stage Responses

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

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

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

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

8.5.3.2 Variable-length Data Stage

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

8.5.3.3 Error Handling on the Last Data Transaction

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

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

8.5.3.4 STALL Handshakes Returned by Control Pipes

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

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

8.5.4 Interrupt Transactions

Interrupt transactions may consist of IN or OUT transfers. Upon receipt of an IN token, a function may return data, NAK, or STALL. If the endpoint has no new interrupt information to return (i.e., no interrupt is pending), the function returns a NAK handshake during the data phase. If the *Halt* feature is set for the interrupt endpoint, the function will return a STALL handshake. If an interrupt is pending, the function returns the interrupt information as a data packet. The host, in response to receipt of the data packet, issues either an ACK handshake if data was received error-free or returns no handshake if the data packet was received corrupted. Figure 8-20 shows the interrupt transaction format.

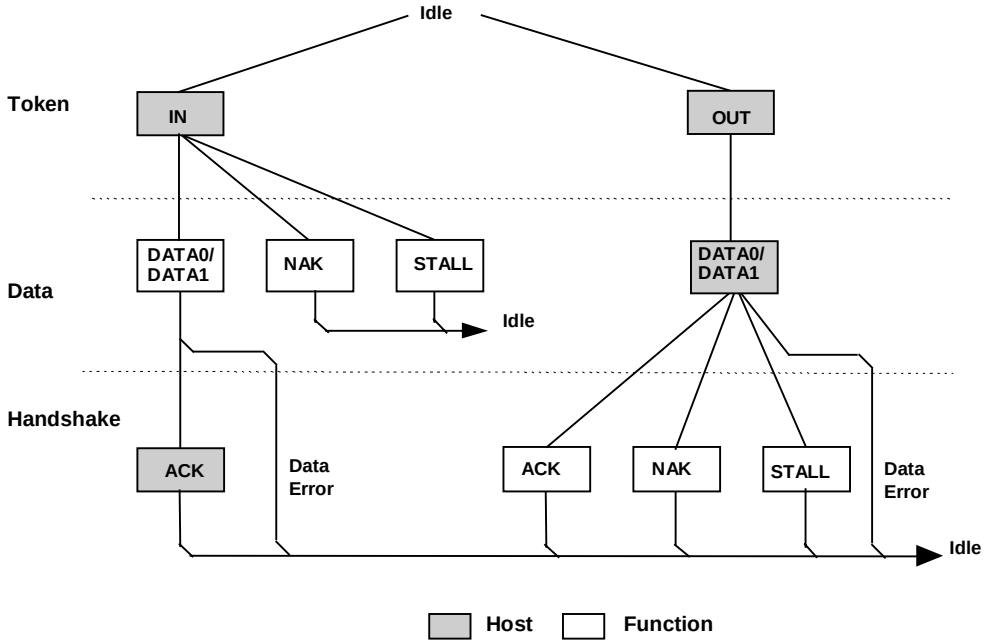

Figure 8-20. Interrupt Transaction Format

When an endpoint is using the interrupt transfer mechanism for actual interrupt data, the data toggle protocol must be followed. This allows the function to know that the data has been received by the host and the event condition may be cleared. This “guaranteed” delivery of events allows the function to only send the interrupt information until it has been received by the host rather than having to send the interrupt data every time the function is polled and until the USB System Software clears the interrupt condition. When

used in the toggle mode, an interrupt endpoint is initialized to the DATA0 PID by any configuration event on the endpoint and behaves the same as the bulk transactions shown in Figure 8-17.

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. This capability is supported only for interrupt IN endpoints.

8.5.5 Isochronous Transactions

Isochronous (ISO) transactions have a token and data phase, but no handshake phase, as shown in Figure 8-21. 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.

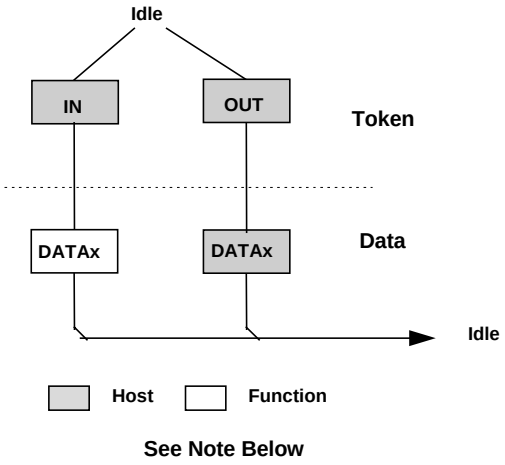

Figure 8-21. Isochronous Transaction Format

Note: a device or Host Controller should be able to accept either DATA0 or DATA1. A device or Host Controller should only send DATA0.

Full-speed ISO transactions do not support toggle sequencing. High bandwidth, high-speed ISO transactions support data PID sequencing (see Chapter 5 for more details).

8.6 Data Toggle Synchronization and Retry

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

8.6.1 Initialization via SETUP Token

Control transfers use the SETUP token for initializing host and function sequence bits. Figure 8-22 shows the host issuing a SETUP packet to a function followed by an OUT transaction. The numbers in the circles represent the transmitter and receiver sequence bits. The function must accept the data and return ACK. When the function accepts the transaction, it must set its sequence bit so that both the host's and function's sequence bits are equal to one at the end of the SETUP transaction.

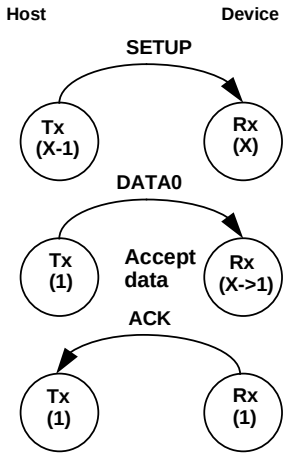

Figure 8-22. SETUP Initialization

8.6.2 Successful Data Transactions

Figure 8-23 shows the case where two successful transactions have occurred. For the data transmitter, this means that it toggles its sequence bit upon receipt of ACK. The receiver toggles its sequence bit only if it receives a valid data packet and the packet's data PID matches the current value of its sequence bit. The transmitter only toggles its sequence bit after it receives and ACK to a data packet.

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

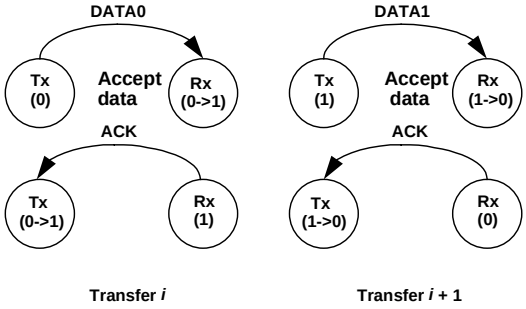

Figure 8-23. Consecutive Transactions

8.6.3 Data Corrupted or Not Accepted

If data cannot be accepted or the received data packet is corrupted, the receiver will issue a NAK or STALL handshake, or timeout, depending on the circumstances, and the receiver will not toggle its sequence bit. Figure 8-24 shows the case where a transaction is NAKed and then retried. Any non-ACK handshake or timeout will generate similar retry behavior. The transmitter, having not received an ACK handshake, will not toggle its sequence bit. As a result, a failed data packet transaction leaves the transmitter's and receiver's sequence bits synchronized and untoggled. The transaction will then be retried and, if successful, will cause both transmitter and receiver sequence bits to toggle.

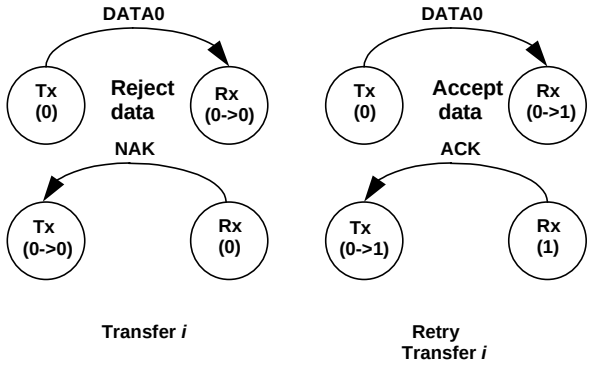

Figure 8-24. NAKed Transaction with Retry

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 8-25. Here the transmitter issues a valid data packet, which is successfully acquired by the receiver; however, the ACK handshake is corrupted.

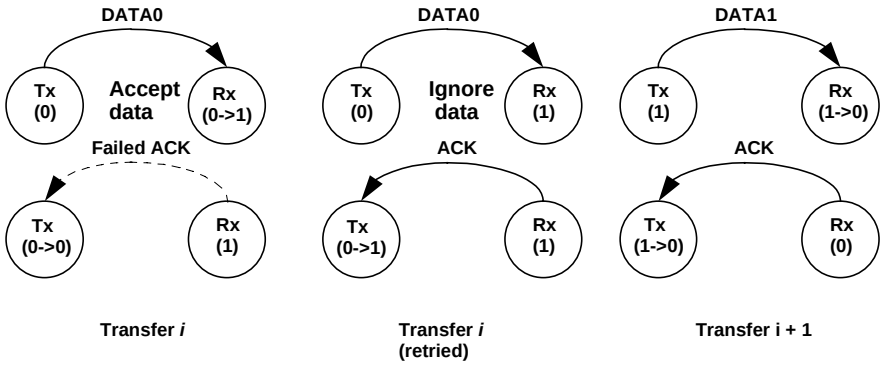

Figure 8-25. Corrupted ACK Handshake with Retry

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

transmitter to toggle its sequence bit. At the beginning of transaction *i*+1, the sequence bits have toggled and are again synchronized.

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

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

Within a full/low-speed signaling environment, hubs disable downstream bus traffic to all ports to which low-speed devices are attached during full-speed downstream signaling. This is required both for EMI reasons and to prevent any possibility that a low-speed device might misinterpret downstream a full-speed packet as being addressed to it.
Figure 8-26 shows an IN low-speed transaction in which the host issues a token and handshake and receives a data packet.

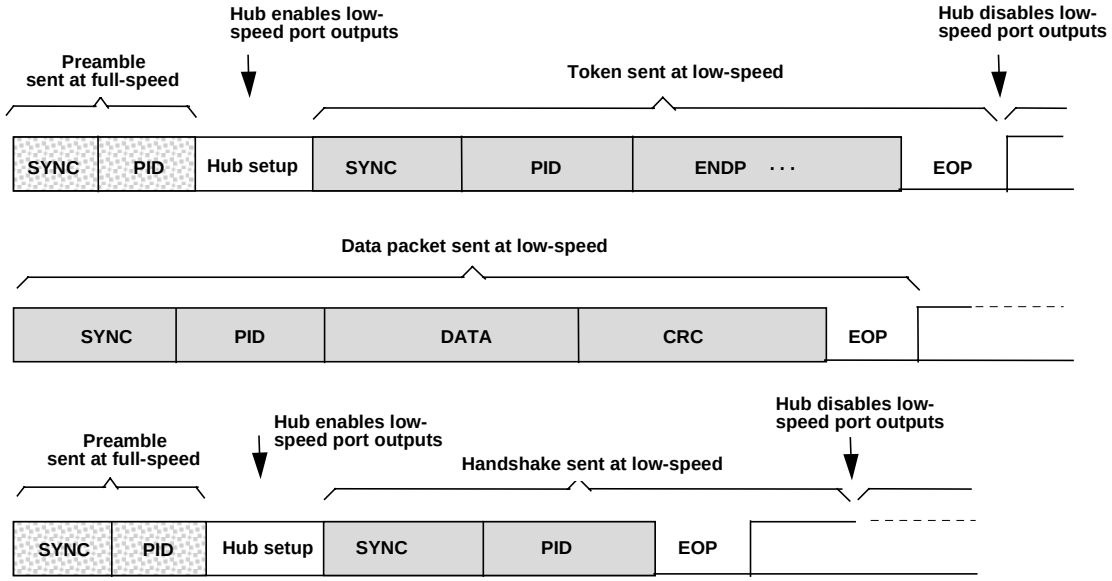

Figure 8-26. Low-speed Transaction

All downstream packets transmitted to low-speed devices within a full/low-speed signaling environment require a preamble. Preambles are never used in a high-speed signaling environment. The preamble consists of a SYNC followed by a PRE PID, both sent at full-speed. Hubs must comprehend the PRE PID; all other USB devices may ignore it and treat it as undefined. 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 enabling the repeater function on ports that are connected to low-speed devices. During this hub setup interval, hubs must drive their full-speed and low-speed ports to their respective Idle states. Hubs must be ready to repeat low-speed signaling on low-speed ports before the end of the hub setup interval. Low-speed connectivity rules are summarized below:

1. Low-speed devices are identified during the connection process and the hub ports to which they are connected are identified as low-speed.

2. All downstream low-speed packets must be prefaced with a preamble (sent at full-speed), which turns on the output buffers on low-speed hub ports.
3. Low-speed hub port output buffers are turned off upon receipt of EOP and are not turned on again until a preamble PID is detected.
4. Upstream connectivity is not affected by whether a hub port is full- or low-speed.

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

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

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

8.7 Error Detection and Recovery

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

8.7.1 Packet Error Categories

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

With the exception of the SOF token, any packet that is received corrupted causes the receiver to ignore it and discard any data or other field information that came with the packet. Table 8-6 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 Turn-around Timing

Neither the device nor the host will send an indication that a received packet had an error. This absence of positive acknowledgement is considered to be the indication that there was an error. As a consequence of this method of error reporting, the host and USB function need to keep track of how much time has elapsed

from when the transmitter completes sending a packet until it begins to receive a response. This time is referred to as the bus turn-around time. The timer starts counting on the SE0-to-'J' transition of the EOP strobe and stops counting when the Idle-to-'K' SOP transition is detected. Both devices and the host require turn-around timers. The device bus turn-around time is defined by the worst case round trip delay plus the maximum device response delay (refer to Section 7.1.18). If a response is not received within this worst case timeout, then the transmitter considers that the packet transmission has failed. Full-/low-speed USB devices timeout no sooner than 16 bit times and no latter than 18 bit times after the end of the previous EOP. 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 ensure that all downstream devices have timed out. High-speed devices timeout no sooner than <<<TBD>>> bit times and no later than <<<TBD>>> bit times after the end of the previous EOP. The host can indicate a high-speed error condition via a timeout by waiting at least <<<TBD>>> bit times before issuing the next token to ensure that all downstream high-speed devices have timed out.

As shown in Figure 8-27, the device uses its bus turn-around timer between token and data or data and handshake phases. The host uses its timer between data and handshake or token and data phases.

If the host receives a corrupted data packet, it 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.

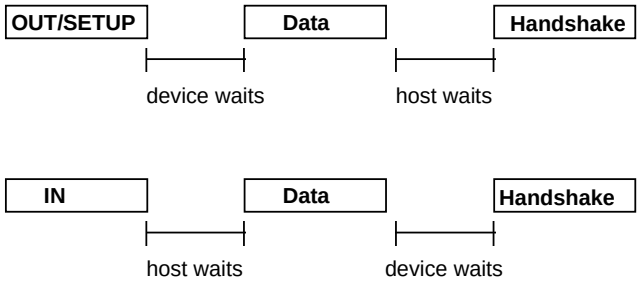

Figure 8-27. Bus Turn-around Timer Usage

8.7.3 False EOPs

<<<Update section on False EOPs for high speed with correct false EOP waiting bit times.>>>

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 timeout.

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 timeout interval (with no bus transitions) greater than the worst case six-bit time bit stuff interval.

- The second condition is that the transmitting device's bus turn-around timer must be guaranteed to expire.

Note that the timeout interval is transaction speed sensitive. For full-speed transactions, the host must wait 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

The USB must be able to detect and recover from conditions which leave it waiting indefinitely for an EOP 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 driven to a 'J' or 'K') 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 11.8.1.