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 11
Hub Specification

This chapter describes the architectural requirements for the USB hub. It contains a description of the two principal sub-blocks: the Hub Repeater and the Hub Controller. The chapter also describes the hub's operation for error recovery, reset, and suspend/resume. The second half of the chapter defines hub request behavior and hub descriptors.

The hub specification supplies sufficient additional information to permit an implementer to design a hub that conforms to the USB specification.

11.1 Overview

Hubs provide the electrical interface between USB devices and the host. Hubs are directly responsible for supporting many of the attributes that make USB user friendly and hide its complexity from the user. Listed below are the major aspects of USB functionality that hubs must support:

- Connectivity behavior
- Power management
- Device connect/disconnect detection
- Bus fault detection and recovery
- ~~Full-~~High-, full- and low-speed device support.

A hub consists of ~~two~~three components: the Hub ~~Repeater and the Hub Controller.~~Repeater, the Hub Controller and the Transaction Translator. The Hub Repeater is responsible for connectivity setup and tear-down. It also supports exception handling, such as bus fault detection and recovery and connect/disconnect detect. The Hub Controller provides the mechanism for host-to-hub communication. Hub-specific status and control commands permit the host to configure a hub and to monitor and control its individual downstream ports. The Transaction Translator responds to special high-speed transactions and translates them to full/low-speed transactions with full/low-speed devices attached on downstream facing ports.

11.1.1 Hub Architecture

Figure 11-1 shows a hub and the locations of its upstream and downstream ports. A hub consists of a Hub Repeater ~~section and~~section, a Hub Controller section and a Transaction Translator section. The Hub Repeater is responsible for managing connectivity on a per-packet ~~basis, while the~~basis. The Hub Controller provides status and control and permits host access to the ~~hub.~~

hub. The Transaction Translator takes high-speed split-transactions and translates them to full/low-speed transactions when the hub is operating at high-speed and has full/low-speed devices attached. The operating speed of a device attached on a downstream facing port determines whether the routing logic connects a port to the transaction translator or hub repeater sections.

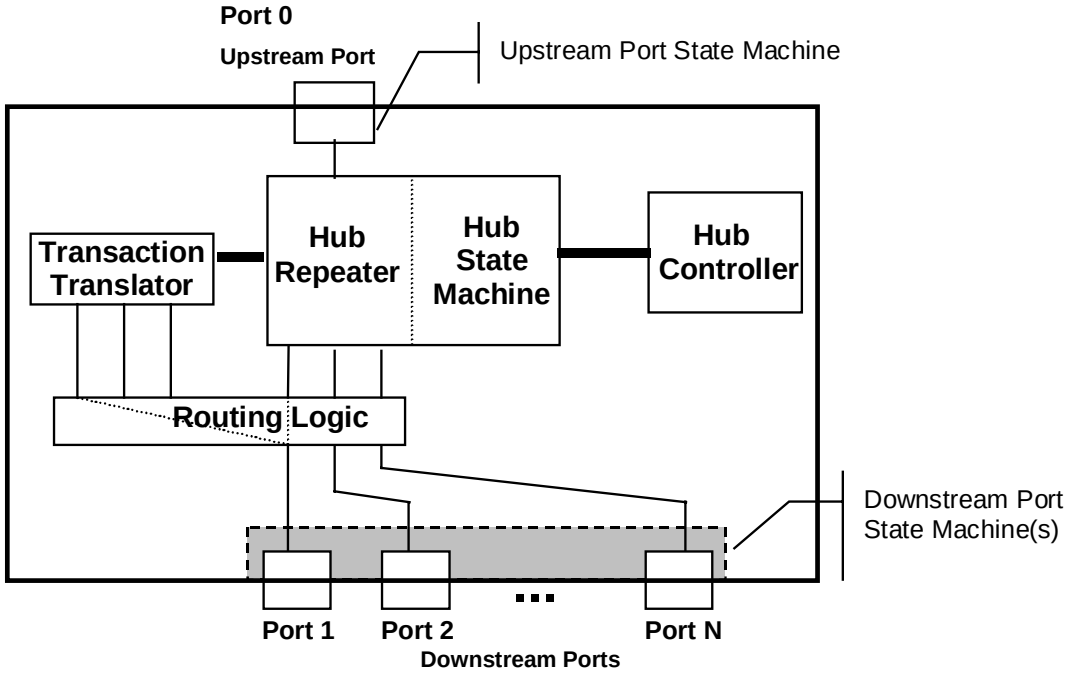

Figure 11-1. Hub Architecture

When a hub upstream facing port is attached to an electrical environment that is operating at full/low speed, the hub's high-speed funcationality is disallowed. This means that the hub will only operate at full/low speed and the transaction translator and high-speed repeater will not operate. In this electrical environment, the hub repeater must operate as a full/low-speed repeater. The routing logic always connects ports to the hub repeater.

When the hub upstream facing port is attached to an electrical environment that is operating at high-speed, the full/low-speed hub repeater is not operational. In this electrical environment when a high-speed device is attached on downstream facing port, the routing logic will connect the port to the hub repeater and the hub repeater must operate as a high-speed repeater. When a full/low-speed device is attached on a downstream facing port, the routing logic must connect the port to the transaction translator.

11.1.2 Hub Connectivity

Hubs display differing connectivity behavior, depending on whether they are propagating packet traffic or resume signaling, or are in the Idle state.

11.1.2.1 Packet Signaling Connectivity

The Hub Repeater contains one port that must always connect in the upstream direction (referred to as the upstream port) and one or more downstream ports. Upstream connectivity is defined as being towards the host, and downstream connectivity is defined as being towards a device. Figure 11-2 shows the packet signaling connectivity behavior for hubs in the upstream and downstream directions. A hub also has an Idle state, during which the hub makes no connectivity. When in the Idle state, all of the hub's ports are in the receive mode waiting for the start of the next packet.

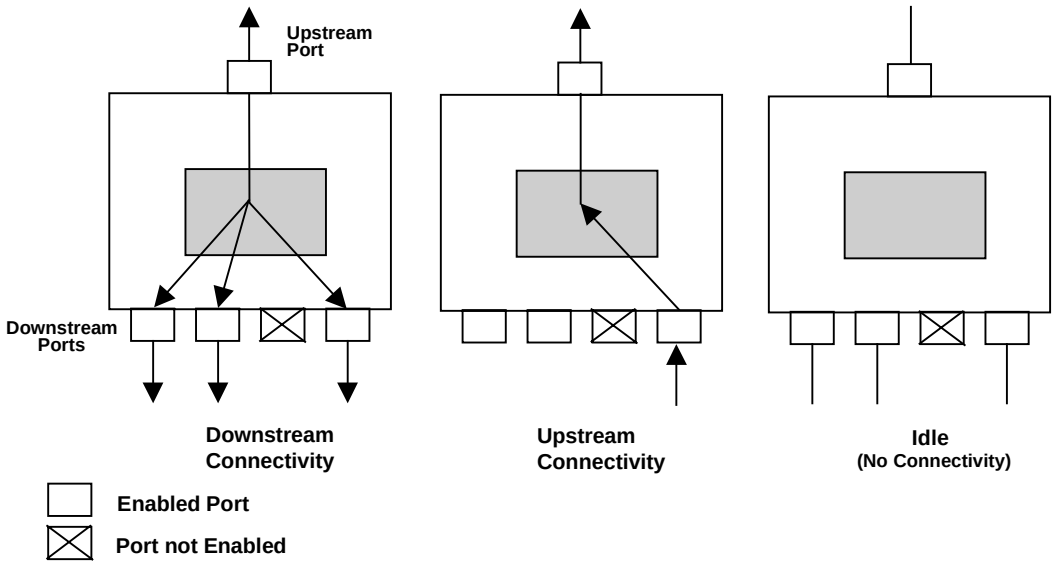

Figure 11-2. Hub Signaling Connectivity

If a downstream hub port is enabled (i.e., in a state where it can propagate signaling through the hub) and the hub detects a Start-of-Packet (SOP) on that port, connectivity is established in an upstream direction to the upstream port of that hub, but not to any other downstream ports. This means that when a device or a hub transmits a packet upstream, only those hubs in line between the transmitting device and the host will see the packet. Refer to Section 11.8.3 for optional behavior when a hub detects simultaneous upstream signaling on more than one port.

In the downstream direction, hubs operate in a broadcast mode. When a hub detects an SOP on its upstream port, it establishes connectivity to all enabled downstream ports. If a port is not enabled, it does not propagate packet signaling downstream.

11.1.2.2 Resume Connectivity

Hubs exhibit different connectivity behaviors for upstream- and downstream-directed resume signaling. A hub that is suspended reflects resume signaling from its upstream port to all of its enabled downstream ports. Figure 11-3 illustrates hub upstream and downstream resume connectivity.

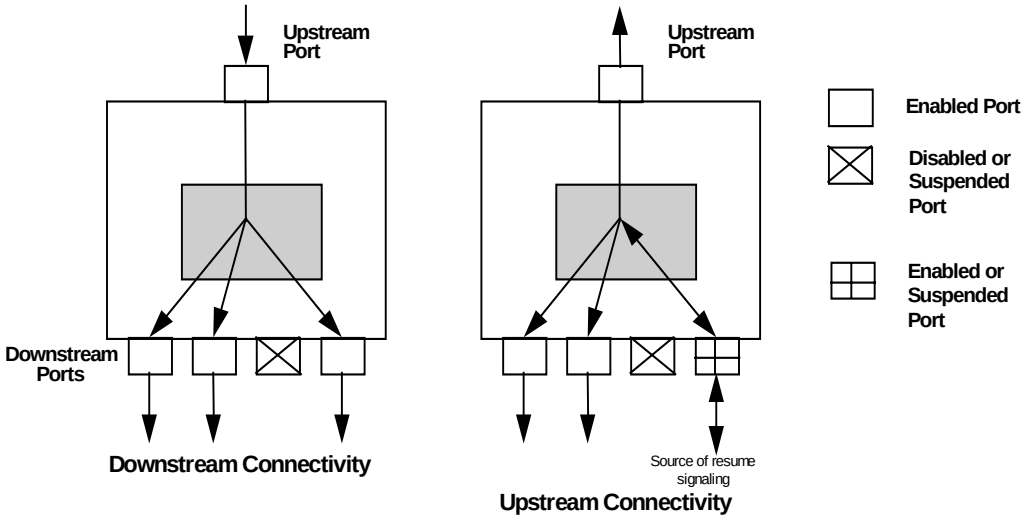

Figure 11-3. Resume Connectivity

If a hub is suspended and detects resume signaling from a selectively suspended or an enabled downstream port, the hub reflects that signaling upstream and to all of its enabled downstream ports, including the port that initiated the resume sequence. Resume signaling is not reflected to disabled or suspended ports. A detailed discussion of resume connectivity appears in Section 11.9.

11.1.2.3 Hub Fault Recovery Mechanisms

Hubs are the essential USB component for establishing connectivity between the host and other devices. It is vital that any connectivity faults, especially those that might result in a deadlock, be detected and prevented from occurring. Hubs need to handle connectivity faults only when they are in the repeater mode.

Hubs must also be able to detect and recover from lost or corrupted packets that are addressed to the Hub Controller. Because the Hub Controller is, in fact, another USB device, it must adhere to the same timeout rules as other USB devices, as described in Chapter 8.

11.2 Hub Frame Timer

<<Update for microframes>>

Each hub has a frame timer whose timing is derived from the hub's local clock and is synchronized to the host frame period by the host-generated Start-of-Frame (SOF). The frame timer provides timing references that are used to allow the hub to detect a babbling device and prevent the hub from being disabled by the upstream hub. The hub frame timer must track the host frame period and be capable of remaining synchronized with the host even if two consecutive SOF tokens are missed by the hub.

The frame timer must lock to the host's frame timing for worst case tolerances and offsets between the host and hub. The offsets have to accommodate the hub oscillator tolerance (≤ 500ppm) and accuracy (≤ 2500ppm) as well as the host's allowed frame tolerance of ≤ 500ppm. The range of the hub frame timer is:

12,000 * 1±(hub accuracy + hub tolerance + host tolerance)

The host tolerance is allowed to be ±500ppm, meaning that a frame time is between 0.9995ms and 1.0005ms, absolute. If the hub's oscillator is at the limits of its accuracy and tolerance, it can be running at between 11,964,000Hz and 12,036,000Hz. If the host is generating an SOF every 1.0005ms and the hub is running at 12,036,000Hz, then the hub's frame timer will count 12,042 times between each SOF. If the host is generating an SOF every 0.9995ms and the hub is running at 11,964,000Hz, then the hub's frame timer will count 11,958 times between each SOF. If the hub accuracy and tolerance are both zero, the hub frame timer range is ±6 bit times.

11.2.1 Frame Timer Synchronization

A hub's frame timer is clocked by the hub's clock source and is synchronized to SOF packets that are derived from the host's frame timer. After a reset or resume, the hub's frame timer is not synchronized. Whenever the hub receives two consecutive SOF packets, its frame timer should be synchronized. Synchronized is synonymous with lock(ed). A example for a method of constructing a timer that properly synchronizes is as follows.

The hub maintains three timer values: frame timer (down counter), current frame (up counter), and next frame (register). After a reset or resume, a flag is set to indicate that the frame timer is not synchronized.

When the first SOF token is detected, the current frame timer resets and starts counting once per hub bit time. On the next SOF, if the timer has not rolled over, the value in the current frame timer is loaded into the next frame register and into the frame timer. The current frame timer is reset to zero and continues to count and the flag is set to indicate that the frame timer is locked. If the current frame timer has rolled over (exceeded 12,043 – a test at 16,383 is adequate), then an SOF was missed and the frame timer and next frame values are not loaded and the flag indicating that the timer is not synchronized remains set.

Whenever the frame timer counts down to zero, the current value of the next frame register is loaded into the frame timer. When an SOF is detected, and the current frame timer has not rolled over, the value of the current frame timer is loaded into the frame timer and the next frame registers. The current frame timer is then reset to zero and continues to count. If the current frame timer has rolled over, then the value in the next frame register is loaded into the frame timer. This process can cause the frame timer to be updated twice in a single frame: once when the frame timer reaches zero and once when the SOF is detected.

The synchronization circuit described above depends on successfully decoding an SOF packet identifier (PID). This means that the frame timer will be synchronized to a time that is at least 16 bit times into the frame. Each implementation will take some time to react to the SOF decode and set the appropriate timer/counter values. (This reaction time is implementation-dependent but is assumed to be less than four full-speed bit times.) Subsequent sections describe the actions that are controlled by the frame timer. These actions are defined at the EOF1, EOF2, and EOF points, which should nominally be the same points in time throughout the bus. EOF1 and EOF2 are defined in later sections. These sections assume that the hub's frame timer will count to zero at the end of the frame (EOF). The circuitry described above will have the frame timer counting to zero 16-20 bit times after the start of a frame (or end of previous frame). The timings and bit offsets in the later sections should be advanced to account for this offset (add 16-20 bit times to the EOF1 and EOF2 points.)

The frame timer provides a indication to the hub Repeater state machine to indicate that the frame timer has synchronized to SOF and that the frame timer is capable of generating the EOF1 and EOF2 timing points. This signal is important after a global resume because of the possibility that a device may have been detached and a different speed device attached while the host was generating a long resume (several seconds) and the disconnect cannot be detected. A different speed device will bias D+ and D- to appear like a K on the hub which would then be treated as an SOP and, unless inhibited, this SOP would propagate

though the resumed hubs. Since the hubs would not have seen any SOF's at this point, the hubs would not be synchronized and, thus, unable to generate the EOF1 and EOF2 timing points. The only recovery from this would be for the host to reset and re-enumerate the section of the bus containing the changed device. This scenario is prevented by inhibiting any downstream port from establishing connectivity until the hub is locked after a resume.

11.2.2 EOF1 and EOF2 Timing Points

The EOF1 and EOF2 are timing points that are derived from the hub's frame timer. These timing points are used to ensure that devices and hubs do not interfere with the proper transmission of the SOF packet from the host. *These timing points have meaning only when the frame timer has been synchronized to the SOF.*

The host and hub frame markers, while all synchronized to the host's SOF, are subject to certain skews that dictate the placement of the EOF points. Figure 11-4 illustrates critical End-of-Frame (EOF) timing points. Table 11-1 summarizes the host and hub EOF timing points.

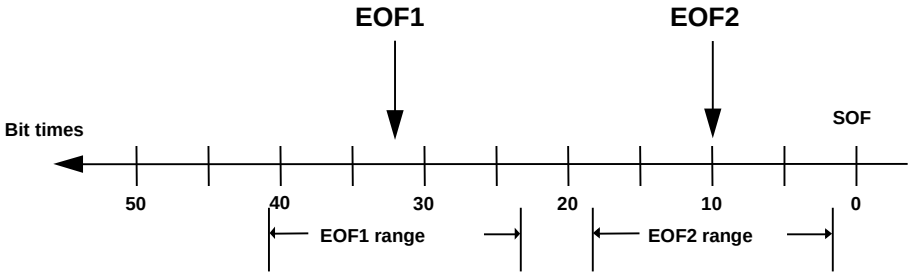

Figure 11-4. EOF Timing Points

At the EOF2 point, any port that has upstream connectivity will be disabled as a babbler. Hubs prevent becoming disabled by sending an End-of-Packet (EOP) to the upstream hub before that hub reaches its EOF2 point (i.e., at EOF1).

Note: a hub is permitted to send the EOP if upstream connectivity is not established at EOF1 time. A hub must send the EOP if connectivity is established from any downstream port at the EOF1 point.

The EOF2 point is defined to occur at least one bit time before the first bit of the SYNC for an SOP. The period allowed for an EOP is four full-speed bit times (the upstream port on a hub is always full-speed.)

Although the hub is synchronized to the SOF, timing skew can accumulate between the host and a hub or between hubs. This timing skew represents the difference between different frame timers on different hubs and the host. The total accumulated skew can be as large as ±9 bit times. This is composed of ±1 bit times per frame of quantization error and ±1 bit per frame of wander. The quantization error occurs when the hub times the interval between SOFs and arrives at a value that is off by a fraction of a bit time but, due to quantization, is rounded to a full bit. Frame wander occurs when the host's frame timer is adjusted by the USB System Software so that the value sampled by the hub in a previous frame differs from the frame interval being used by the host. These values accumulate over multiple frames because SOF packets can be lost and the hub cannot resynchronize its frame timer. This specification allows for the loss of two consecutive SOFs. During this interval the quantization error accumulates to ±3 bit times and the wander accumulates to ±1 ± 2 ± 3 = ±6 for a total of ±9 bit times of accumulated skew in three frames. This skew timing affects the placement of the EOF1 and EOF2 points as follows.

Note: although the USB System Software is not allowed to cause the frame interval to change more than one bit time every six frames, the hub skew timing assumes that the frame interval can change one bit time per frame. This cannot be reduced because it would create interoperability problems with hubs designed to previous versions of this specification.

A hub must reach its EOF2 point one bit time before the end of the frame. In order to ensure this, a 9-bit time guard-band must be added so that the EOF2 point is set to occur when the hub's local frame timer

reaches 10. A hub must complete its EOP before the hub to which it is attached reaches its EOF2 point. A hub may reach its EOF2 point nine bit times before bit time 10 (at bit time 19 before the SOF). To ensure that the EOP is completed by bit time 19, it must start before bit time 23. To ensure that the hub starts at bit time 23 with respect to another hub, a hub must set its EOF1 point nine bit times ahead of bit time 23 (at bit time 32). If a hub sets its timer to generate an EOP at bit time 32, that EOP may start as much as 9 bit times early (at bit time 41).

Table 11-1. Hub and Host EOF Timing Points

Description	Nominal Number of Bits from Start of SOF	Notes
EOF1	32	End-of-Frame point #1
EOF2	10	End-of-Frame point #2

11.3 Host Behavior at End-of-Frame

<<Update for microframes>>

It is the responsibility of the USB host controller (the host) to not provoke a response from a device if the response would cause the device to be sending a packet at the EOF2 point. Furthermore, because a hub will terminate an upstream directed packet when the hub reaches its EOF1 point, the host should not start a transaction if a response from the device (data or handshake) would be pending or in process when a hub reaches its EOF1 point. The implications of these limitations are described in the following sections.

In defining the timing points below, the last bit interval in a frame is designated as bit time zero. Bit times in a frame that occur before the last have values that increase the further they are from bit time zero (earlier bit times have higher numbers). These bit time designations are used for convenience only and are not intended to imply a particular implementation. The only requirement of an implementation is that the relative bit time values be preserved.

11.3.1 Latest Host Packet

Hubs are allowed to send an EOP on their upstream ports at the EOF1 point if there is no downstream-directed traffic in progress at that time. To prevent potential contention, the host is not allowed to start a packet if connectivity will not be established on all connections before a hub reaches its EOF1 point. This means that the host must not start a packet after bit time 42.

Note: although there is as much as a six-bit time delay between the time the host starts a packet and all connections are established, this time need not be added to the packet start time as this phase delay exists for the SOF packet as well, causing all hub frame timers to be phase delayed with respect to the host by the propagation delay. There is only one bit time of phase delay between any two adjacent hubs and this has been accounted for in the skew calculations.

11.3.2 Packet Nullification

If a device is sending a packet (data or handshake) when a hub in the device's upstream path reaches its EOF1 point, the hub will send a full-speed EOP. Any packet that is truncated by a hub must be discarded.

A host implementation may discard any packet that is being received at bit time 41. Alternatively, a host implementation may attempt to maximize bus utilization by accepting a packet if the packet is predicted to start at or before bit time 41.

11.3.3 Transaction Completion Prediction

A device can send two types of packets: data and handshake. A handshake packet is always exactly 16 bit times long (sync byte plus PID byte.) The time from the end of a packet from the host until the first bit of the handshake must be seen at the host is 17 bit times. This gives a total allocation of 35 bit times from the end of data packet from the root (start of EOP) until it is predicted that the handshake will be completed (start of EOP) from the device. Therefore, if the host is sending a data packet for which the device can return a handshake (anything other than an isochronous packet), then if the host completes the data packet and starts sending EOP before bit time 76, then the host can predict that the device will complete the handshake and start the EOP for the handshake on or before bit time 41. For a low-speed device, the 36 bit times from start of EOP from root to start of EOP from the device are low-speed bit times, which convert 1 to eight into full-speed bit times. Therefore, if the host completes the low-speed data packet by bit time 329, then the low-speed device can be predicted to complete the handshake before bit time 41

Note: if the host cannot accept a full-speed EOP as a valid end of a low-speed packet, then the low-speed EOP will need to complete before bit time 41, which will add 13 full-speed bit times to the low-speed handshake time.

As the host approaches the end of the frame, it must ensure that it does not require a device to send a handshake if that handshake can't be completed before bit time 41. The host expects to receive a handshake after any valid, non-isochronous data packet. Therefore, if the host is sending a non-isochronous data packet when it reaches bit time 76 (329 for low-speed), then the host should start an abnormal termination sequence to ensure that the device will not try to respond. This abnormal termination sequence consists of 7 consecutive bits of 1 followed by an EOP. The abnormal termination sequence is sent at the speed of the current packet.

If the host is preparing to send an IN token, it may not send the token if the predicted packet from the device would not complete by bit time 41. The maximum valid length of the response from the device is known by the host and should be used in the prediction calculation. For a full-speed packet, the maximum interval between the start of the IN token and the end of a data packet is:

token_length + (packet_length + header + CRC) * 7/6 + 18

Where *token_length* is 34 bit times, *packet_length* is the maximum number of data bits in the packet, *header* is eight bits of sync and eight bits of PID, and CRC is 16 bits. The 7/6 multiplier accounts for the absolute worst case bit-stuff on the packet and the 18 extra bits allow for worst case turn-around delay. For a low-speed device, the same calculation applies but the result must be multiplied by 8 to convert to full-speed bit times and an additional 20 full-speed bit times must be added to account for the low-speed prefix. This gives the maximum number of bit times between the start of the IN token and the end of the data packet, so the token cannot be sent if this number of bit times does not exist before the earliest EOF1 point (bit time 41). (E.g., take the results of the above calculation and add 41. If the number of bits left in the frame is less than this value, the token may not be sent.)

The host is allowed to use a more conservative algorithm than the one given above for deciding whether or not to start a transaction. The calculation might also include the time required for the host to send the handshake when one is required, as there is no benefit in starting a transfer if the handshake cannot be completed.

11.4 Internal Port

<<Update for high-speed>>

The internal port is the connection between the Hub Controller and the Hub Repeater. Besides conveying the serial data to/from the Hub Controller, the internal port is the source of certain resume signals. Figure 11-5 illustrates the internal port state machine; Table 11-2 defines the internal port signals and events.

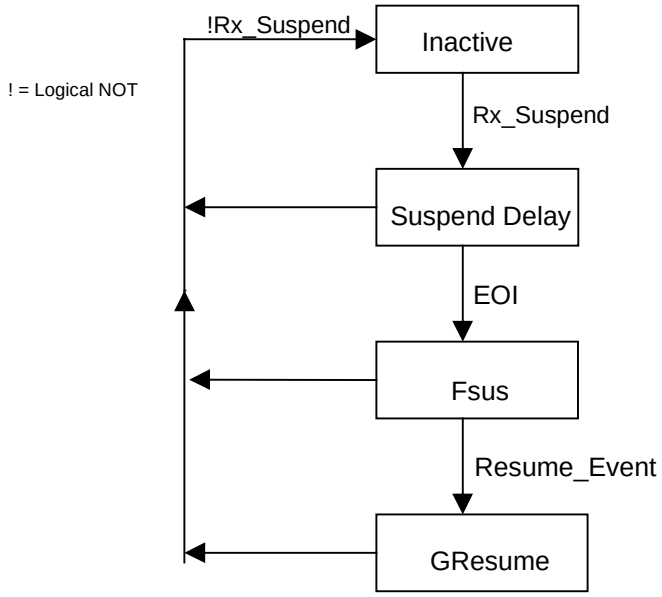

Figure 11-5. Internal Port State Machine

Table 11-2. Internal Port Signal/Event Definitions

Signal/Event Name	Event/Signal Source	Description
EOI	Internal	End of timed interval
Rx_Suspend	Receiver	Receiver is in the Suspend state
Resume_Event	Hub Controller	A resume condition exists in the Hub Controller

11.4.1 Inactive

This state is entered whenever the Receiver is not in the Suspend state.

11.4.2 Suspend Delay

This state is entered from the Inactive state when the Receiver transitions to the Suspend state.

This is a timed state with a 2ms interval.

11.4.3 Full Suspend (Fsus)

This state is entered when the Suspend Delay interval expires.

11.4.4 Generate Resume (GResume)

This state is entered from the Fsus state when a resume condition exists in the Hub Controller. A resume condition exists if the C_PORT_SUSPEND bit is set in any port or if the hub is enabled as a wakeup source and any bit is set in a Port Change field or the Hub Change field (as described in Table 11-15 and Table 11-10, respectively).

In this state, the internal port generates signaling to emulate an SOP_FD to the Hub Repeater.

11.5 Downstream Ports

<<Update for high-speed>>

The following sections provide a functional description of a state machine that exhibits the correct behavior for a downstream port on a hub.

Figure 11-6 is an illustration of the downstream port state machine. The events and signals are defined in Table 11-3. Each of the states is described in Section 11.5.1. In the diagram below, some of the entry conditions into states are shown without origin. These conditions have multiple origin states and the individual transitions lines are not shown so that the diagram can be simplified. The description of the entered state indicates from which states the transition is applicable.

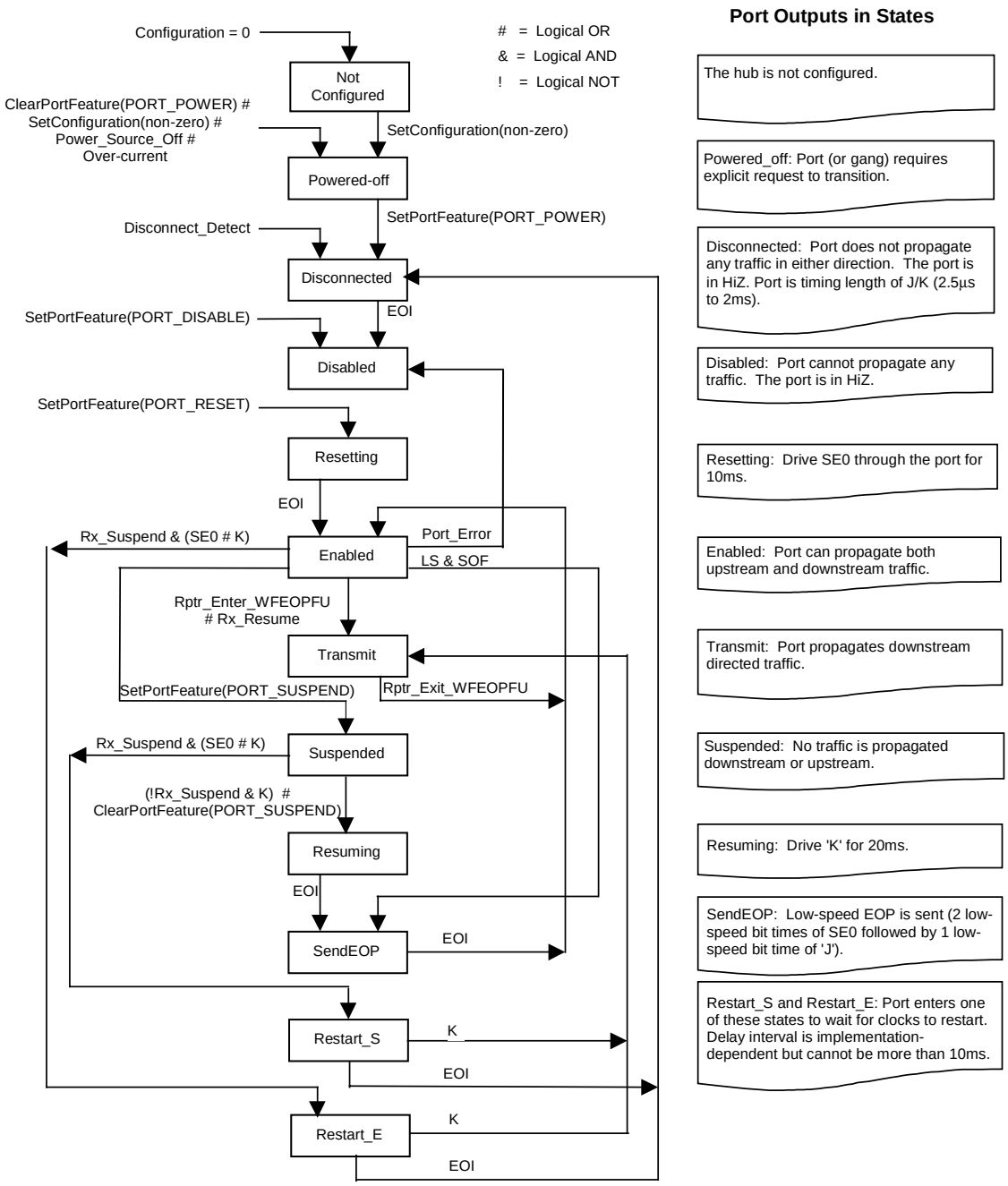

Figure 11-6. Downstream Hub Port State Machine

Table 11-3. Downstream Hub Port Signal/Event Definitions

Signal/Event Name	Event/Signal Source	Description
Power_source_off	Implementation-dependent	Power to the port not available due to over-current or termination of source power (e.g., external power removed)
Over-current	Hub Controller	Over-current condition exists on the hub or the port
EOI	Internal	End of a timed interval or sequence
SE0	Internal	SE0 received on port
Disconnect_Detect	Internal	Long SE0 detected on port (See Section 11.5.2)
LS	Hub Controller	Low-speed device attached to this port
SOF	Hub Controller	SOF token received
J	Internal	'J' received on port
K	Internal	'K' received on port
Rx_Resume	Receiver	Upstream Receiver in Resume state
Rx_Suspend	Receiver	Upstream Receiver in Suspend state
Rptr_Exit_WFEOPFU	Hub Repeater	Hub Repeater exits the WFEOPFU state
Rptr_Enter_WFEOPFU	Hub Repeater	Hub Repeater enters the WFEOPFU state
Port_Error	Internal	Error condition detected (see Section 11.8.1)
Configuration = 0	Hub Controller	Hub controller's configuration value is zero

11.5.1 Downstream Port State Descriptions

11.5.1.1 Not Configured

A port transitions to and remains in this state whenever the value of the hub configuration is zero. While the port is in this state, the hub will drive an SE0 on the port (this behavior is optional on root hubs). No other active signaling takes place on the port when it is in this state.

11.5.1.2 Powered-off

This state is supported for all hubs.

A port transitions to this state in any of the following situations:

- From any state except Not Configured when the hub receives a ClearPortFeature(PORT_POWER) request for this port
- From any state when the hub receives a SetConfiguration() request with a configuration value other than zero
- From any state except Not Configured when power is lost to the port or an over-current condition exists.

A port will enter this state due to an over-current condition on another port if that over-current condition may have caused the power supplied to this port to drop below specified limits for port power (see Section 7.2.1.2.1 and Section 7.2.4.1).

If a hub was configured while the hub was self-powered, then if external power is lost the hub must place all ports in the Powered-off state. If the hub is configured while bus powered, then the hub need not change port status if the hub switched to externally applied power. However, if external power is subsequently lost, the hub must place ports in the Powered-off state.

In this state, the port's differential and single-ended transmitters and receivers are disabled.

Control of power to the port is covered in Section 11.11.

11.5.1.3 Disconnected

A port transitions to this state in any of the following situations:

- from the Powered-off state when the hub receives a SetPortFeature(PORT_POWER) request
- from any state except the Not Configured and Powered-off states when the port's disconnect timer times out
- from the Restart_S or Restart_E state at the end of the restart interval.

In the Disconnected state, the port's differential transmitter and receiver are disabled and only connection detection is possible.

This is a timed state. While in this state, the timer is reset as long as the port’s signal lines are in the SE0 state. If another signaling state is detected, the timer starts. Unless the hub is suspended with clocks stopped, this timer's duration is 2.5µs to 2ms.

If the hub is suspended with its remote wakeup feature enabled then on a transition from the SE0 state on a Disconnected port the hub will start its clocks and time this event. The hub must be able to start its clocks and time this event within 12ms of the transition. If a hub does not have its remote wakeup feature enabled, then transitions on a port that is in the Disconnected state are ignored until the hub is resumed.

11.5.1.4 Disabled

A port transitions to this state in any of the following situations:

- From the Disconnected state when the timer expires indicating a connection is detected on the port
- From any but the Powered-off, Disconnected, or SenseSE0 states on receipt of a ClearPortFeature(PORT_ENABLE) request
- From the Enabled state when an error condition is detected on the port

A port in the Disabled state will not propagate signaling in either the upstream or the downstream direction. While in this state, the duration of any SE0 received on the port is timed.

11.5.1.5 Resetting

Unless it is in the Powered-off or Disconnected states, a port transitions to the Resetting state upon receipt of a SetPortFeature(PORT_RESET) request. The hub drives SE0 on the port during this timed interval. The duration of the Resetting state is nominally 10ms to 20ms (10ms is preferred).

11.5.1.6 Enabled

A port transitions to this state in any of the following situations:

- At the end of the Resetting state
- From the Transmit state when the Hub Repeater exits the WFEOPFU state
- From the Suspended state if the upstream Receiver is in the Suspend state when a 'K' is detected on the port
- At the end of the SendEOP state.

While in this state, the output of the port's differential receiver is available to the Hub Repeater so that 'J'-to-'K' transitions can establish upstream connectivity.

11.5.1.7 Transmit

For full- and low-speed ports this state is entered in either of the following situations:

- from the Enabled state if the upstream Receiver is in the Resume state
- immediately from the Restart_S or Restart_E state if a 'K' is detected on the port.

For a full-speed port, this state is entered from the Enabled state on the transition of the Hub Repeater to the WFEOPFU state. While in this state, the port will transmit the data that is received on the upstream port.

For a low-speed port, this state is entered from the Enabled state if a full-speed PRE PID is received on the upstream port. While in this state, the port will retransmit the data that is received on the upstream port (after proper inversion).

11.5.1.8 Suspended

A port enters the Suspended state from the Enabled state when it receives a SetPortFeature(PORT_SUSPEND) request. While a port is in the Suspended state, the port's differential transmitter is disabled.

An implementation is allowed to have a SE0 ‘noise’ filter for a port that is in the suspended state. This filter can time the length of SE0 and, if the length of the SE0 is shorter than 2.5µs, the port may remain in this state. However, this filter may not be used if the hub is suspended and the clocks are stopped. Rather, if the hub is suspended with its clocks stopped, a transition to SE0 on a suspended port must cause the port to immediately transition to the Restart_S state. This is to insure that the attached device is not reset and placed at the default address without having the hub disable the port.

11.5.1.9 Resuming

A port enters this state from the Suspended state in either of the following situations:

- If a 'K' is detected on the port and the Receiver is not in the Suspend state
- When a ClearPortFeature(PORT_SUSPEND) request is received.

This is a timed state with a nominal duration of 20ms (the interval may be longer under the conditions described in the note below). While in this state, the hub drives a 'K' on the port.

Note: a single timer is allowed to be used to time both the Resetting interval and the Resuming interval and that timer may be shared among multiple ports. When shared, the timer is reset when a port enters the Resuming state or the Resetting state. If shared, it may not be shared among more than ten ports as the cumulative delay could exceed the amount of time required to replace a device and a disconnect could be missed.

11.5.1.10 SendEOP

This state is entered from the Resuming state if the 20ms timer expires. It is also entered from the Enabled state when an SOF (or other FS token) is received and a low-speed devices is attached to this port. In this state, the hub will send a low-speed EOP (two low-speed bits times of SE0 followed by one low-speed bit times of J). At the end of the EOP, the state ends.

Since the transmitted EOP should be of fixed length, the SendEOP timer, if shared, should not be reset. If the hub implementation shares the SendEOP timing circuits between ports, then the Resuming state should not end until an SOF (or other FS token) has been received (see Section 11.8.4.1 for Keep-alive generation rules).

11.5.1.11 Restart_S/Restart_E

A port enters the Restart_S state from the Suspended state or enters the Restart_E state from the Enabled state when an SE0 or 'K' is seen at the port and the Receiver is in the Suspended state.

These states are needed to ensure that a transient SE0, which may be seen at the start of resume signaling, does not cause the port to be disabled.

In these states, the port continuously monitors the bus state and exits to the Transmit state immediately on seeing the K state. In this case, the port completes its transition to the Transmit state within 100µs after entering the Restart_S or Restart_E state. If the bus state is not 'K', the port transitions to the Disconnected state. This transition should happen within 10ms of entering the Restart_S or Restart_E state.

11.5.2 Disconnect Detect Timer

Each port is required to have a disconnect timer. This timer is used to constantly monitor the ports single-ended receivers to detect a disconnect event. The reason for constant monitoring is that a noise event on the bus can cause the attached device to detect a reset condition on the bus after 2.5µs of SE0 on the bus. If the hub does not place the port in the disconnect state before the device resets, then the device can be at in the Default Address state with the port enabled. This can cause systems errors that are very difficult to isolate and correct.

This timer should be reset whenever the D+ and D- lines on the port are not in the SE0 state or when the port is not in the Enabled, Suspended, or Disabled states. This timer may have a timeout that is a short as 1.994µs (2.0µs - 3000ppm) but should not be longer than 2.508µs (+3000 ppm). When this timer expires, it generates the Disconnect_Detect signal to the port state machine.

11.6 Upstream Port

<<Update for high-speed>>

The upstream port has four components: transmitter, transmitter state machine, receiver and receiver state machine. The transmitter and its state machine are the Transmitter, while the receiver and its state machine are the Receiver. Both the transmitter and receiver have differential and single-ended components. The differential transmitter and receiver can send/receive 'J' or 'K' to/from the bus while the single-ended components are used to send/receive SE0, suspend, and resume signaling. In this section, when it is necessary to differentiate the signals sent/received by the differential component of the transmitter/receiver from those of the single-ended components, DJ and DK will be used to denote the differential signal and SJ, SK and SE0 will be used for the single-ended signals.

It is assumed that the differential transmitter and receiver are turned off during suspend to minimize power consumption. The single-ended components are left on at all times, as they will take minimal power.

11.6.1 Receiver

The receiver state machine is responsible for monitoring the signaling state of the upstream connection to detect long-term signaling events such as bus reset, resume, and suspend. Figure 11-7 illustrates the state transition diagram. Table 11-4 defines the signals and events referenced in Figure 11-7.

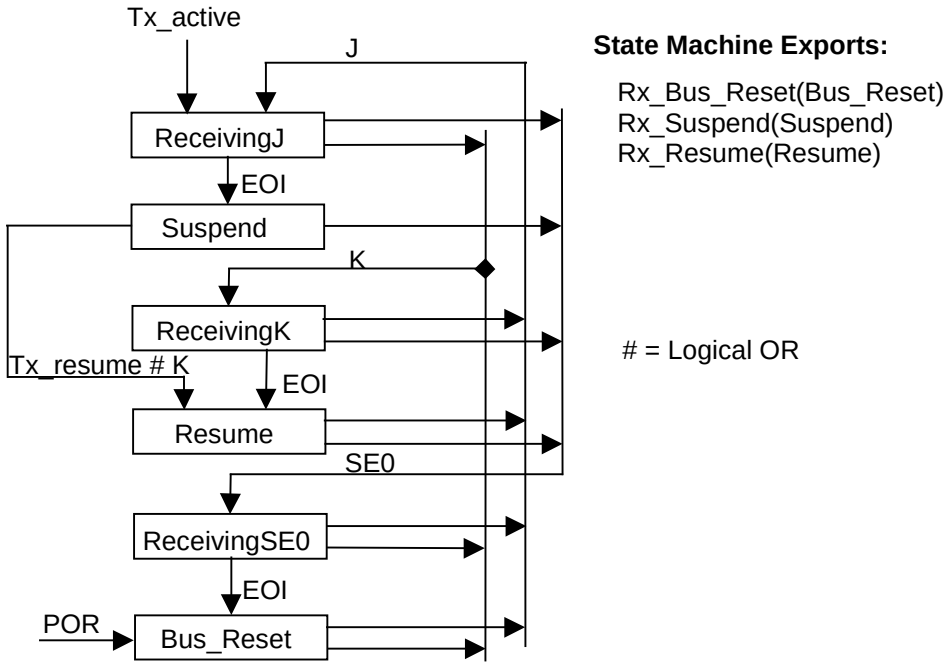

Figure 11-7. Upstream Port Receiver State Machine

Table 11-4. Upstream Hub Port Receiver Signal/Event Definitions

Signal/Event Name	Event/Signal Source	Description
Tx_active	Transmitter	Transmitter in the Active state
J	Internal	Receiving a 'J' (IDLE) on the upstream port
EOI	Internal	End of timed interval
K	Internal	Receiving a 'K' on the upstream port
Tx_resume	Transmitter	Transmitter is in the Sresume state
SE0	Internal	Receiving an SE0 on the upstream port
POR	Implementation-dependent	Power_On_Reset

11.6.1.1 ReceivingJ

This state is entered from any state except the Suspend state if the receiver detects an SJ (or Idle) condition on the bus or while the Transmitter is in the Active state.

This is a timed state with an interval of 3ms. The timer is reset each time this state is entered.

The timer only advances if the Transmitter is in the Inactive state.

11.6.1.2 Suspend

This state is entered if the 3ms timer expires in the ReceivingJ state. When the Receiver enters this state, the Hub Controller starts a 2ms timer. If that timer expires while the Receiver is still in this state, then the Hub Controller is suspended. When the Hub Controller is suspended, it may generate resume signaling.

11.6.1.3 ReceivingK

This state is entered from any state except the Resume state when the receiver detects an SK condition on the bus and the Hub Repeater is in the WFSOP or WFSOPFU state. This is a timed state with a duration of 2.5µs to 100µs. The timer is reset each time this state starts.

11.6.1.4 Resume

This state is entered from the ReceivingK state when the timer expires.

This state is also entered from the Suspend state while the Transmitter is in the Sresume state or if there is a transition to the K state on the upstream port.

If the hub enters this state when its timing reference is not available, the hub may remain in this state until the hub's timing reference becomes stable. If this state is being held pending stabilization of the hub's clock, the Receiver should provide a K to the repeater for propagation to the downstream ports. When clocks are stable, the Receiver should repeat the incoming signals.

Note: constraints on hub behavior after reset require that the hub be able to start clocks and get them stable in less than 10ms.

11.6.1.5 ReceivingSE0

This state is entered from any state except Bus_Reset when the receiver detects an SE0 condition and the Hub Repeater is in the WFSOP or WFSOPFU state. This is a timed state. The minimum interval for this state is 2.5µs. The maximum depends on the hub but this interval must timeout early enough such that if the width of the SE0 on the upstream port is only 10ms, the Receiver will enter the Bus_Reset state with sufficient time remaining in the 10ms interval for the hub to complete its reset processing. Furthermore, if the hub is suspended when the Receiver enters this state, the hub must be able to start its clocks, time this interval, and complete its reset processing within 10ms. It is preferred that this interval be as long as possible given the constraints listed here. This will provide for the maximum immunity to noise on the upstream port and reduce the probability that the device will reset in the presence of noise before the upstream hub disables the port.

The timer is reset each time this state starts.

11.6.1.6 Bus_Reset

This state is entered from the ReceivingSE0 state when the timer expires. As long as the port continues to receive SE0, the Receiver will remain in this state.

This state is also entered while power-on-reset (POR) is being generated by the hub's local circuitry. The state machine cannot exit this state while POR is active.

11.6.2 Transmitter

This state machine is used to monitor the upstream port while the Hub Repeater has connectivity in the upstream direction. The purpose of this monitoring activity is to prevent propagation of erroneous indications in the upstream direction. In particular, this machine prevents babble and disconnect events on the downstream ports of this hub from propagating and causing this hub to be disabled or disconnected by the hub to which it is attached. Figure 11-8 is the transmitter state transition diagram. Table 11-5 defines the signals and events referenced in Figure 11-8.

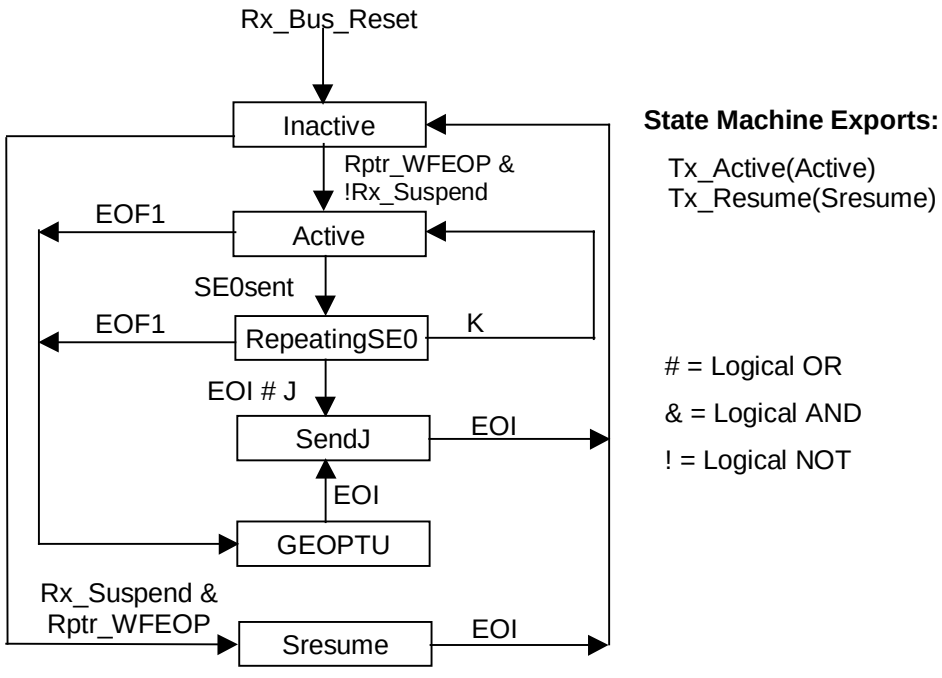

Figure 11-8. Upstream Hub Port Transmitter State Machine

Table 11-5. Upstream Hub Port Transmit Signal/Event Definitions

Signal/Event Name	Event/Signal Source	Description
Rx_Bus_Reset	Receiver	Receiver is in the Bus_Reset state
EOF1	Frame Timer	Hub frame time has reached the EOF1 point or is between EOF1 and the end of the frame
J	Internal	Transmitter transitions to sending a 'J' and transmits a 'J'
Rptr_WFEOP	Hub Repeater	Hub Repeater is in the WFOEP state
K	Internal	Transmitter transmits a 'K'
SE0sent	Internal	At least one bit time of SE0 has been sent through the transmitter
Rx_Suspend	Receiver	Receiver is in Suspend state
EOI	Internal	End of timed interval

11.6.2.1 Inactive

This state is entered at the end of the SendJ state or while the Receiver is in the Bus_Reset state. This state is also entered at the end of the Sresume state. While the transmitter is in this state, both the differential and single-ended transmit circuits are disabled and placed in their high-impedance state.

11.6.2.2 Active

This state is entered from the Inactive state when the Hub Repeater transitions to the WFEOP state. This state is entered from the RepeatingSE0 state if the first transition after the SE0 is not to the J state. In this state, the data from a downstream port is repeated and transmitted on the upstream port.

11.6.2.3 RepeatingSE0

The port enters this state from the Active state when one bit time of SE0 has been sent on the upstream port. While in this state, the transmitter is still active and downstream signaling is repeated on the port. This is a timed state with a duration of 23 full-speed bit times.

11.6.2.4 SendJ

The port enters this state from the RepeatingSE0 state if either the bit timer reaches 23 or the repeated signaling changes from SE0 to 'J'. This state is also entered at the end of the GEOPTU state. This state lasts for one full-speed bit time. During this state, the hub drives an SJ on the port.

11.6.2.5 Generate End of Packet Towards Upstream Port (GEOPTU)

The port enters this state from the Active or RepeatingSEO state if the frame timer reaches the EOF1 point.

In this state, the port transmits SE0 for two full-speed bit times.

11.6.2.6 Send Resume (Sresume)

The port enters this state from the Inactive state if the Receiver is in the Suspend state and the Hub Repeater transitions to the WFEOP state. This indicates that a downstream device (or the port to the Hub Controller) has generated resume signaling, causing upstream connectivity to be established.

On entering this state, the hub will restart clocks if they had been turned off during the Suspend state. While in this state, the Transmitter will drive a 'K' on the upstream port. While the Transmitter is in this state, the Receiver is held in the Resume state. While in the Resume state, all downstream ports that are in the Enable state are placed in the Transmit state and the resume on this port is transmitted to those downstream ports.

The port stays in this state for at least 1ms but for no more than 15ms.

11.7 Hub Repeater

<<Update for high-speed>>

The Hub Repeater provides the following functions:

- Sets up and tears down connectivity on packet boundaries
- Ensures orderly entry into and out of the Suspend state, including proper handling of remote wakeups

The state machine in Figure 11-9 shows the states and transitions needed to implement the Hub Repeater. Table 11-6 defines the Hub Repeater signals and events. The following sections describe the states and the transitions.

Several of the state transitions below will occur when an EOP is detected. When such a transition is indicated, the transition does not occur until after the hub has repeated the SE0-to-'J' transition and has driven 'J' for at least one bit time (bit time is determined by the speed of the port.)

Some of the transitions are triggered by an SOP. Transitions of this type occur as soon as the hub detects the 'J'-to-'K' transition, ensuring that the initial edge of the SYNC field is preserved.

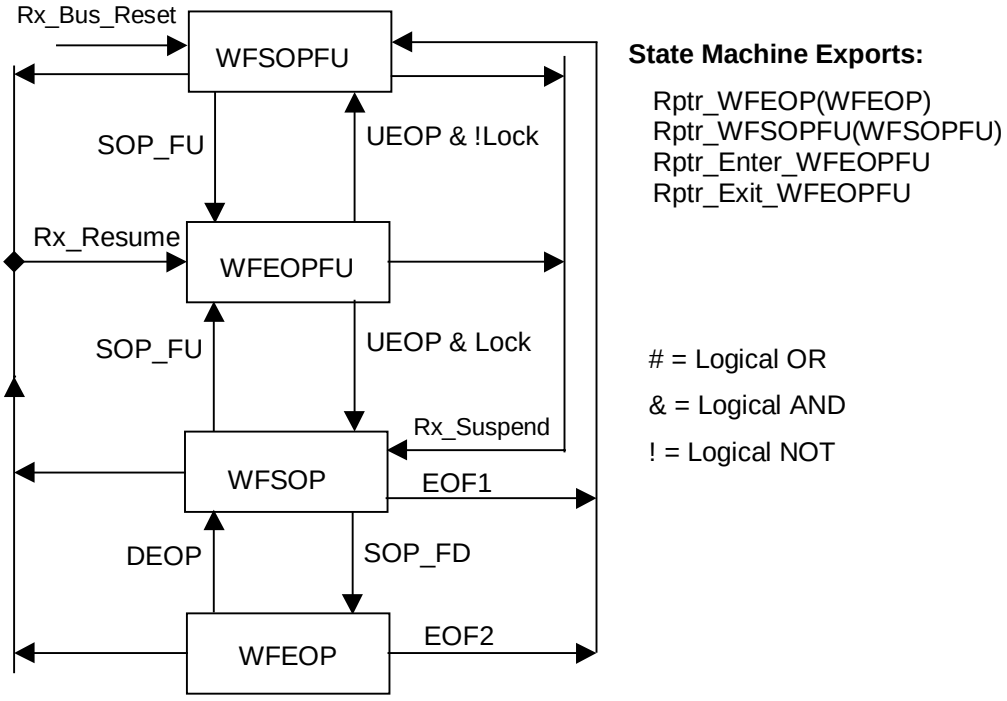

Figure 11-9. Hub Repeater State Machine

Table 11-6. Hub Repeater Signal/Event Definitions

Signal/Event Name	Event/Signal Source	Description
Rx_Bus_Reset	Receiver	Receiver is in the Bus_Reset state
UEOP	Internal	EOP received from the upstream port
DEOP	Internal	Generated when the Transmitter enters the SendJ state
EOF1	Frame Timer	Frame timer is at the EOF1 point or between EOF1 and End-of-Frame
EOF2	Frame Timer	Frame timer is at the EOF2 point or between EOF2 and End-of-Frame
Lock	Frame Timer	Frame timer is locked
Rx_Suspend	Receiver	Receiver is in the Suspend state
Rx_Resume	Receiver	Receiver is in the Resume state
SOP_FD	Internal	SOP received from downstream port or Hub Controller. Generated on the transition from the Idle to K state on a port.
SOP_FU	Internal	SOP received from upstream port. Generated on the transition from the Idle to K state on the upstream port.

11.7.1 Wait for Start of Packet from Upstream Port (WFSOPFU)

This state is entered in either of the following situations:

- From any other state when the upstream Receiver is in the Bus_Reset state
- From the WFSOP state if the frame timer is at or has passed the EOF1 point
- From the WFEOP state at the EOF2 point.
- From the WFEOPFU if the frame timer is not synchronized (locked) when an EOP is received on the upstream port.

In this state, the hub is waiting for an SOP on the upstream port and transitions on downstream ports are ignored by the Hub Repeater. While the Hub Repeater is in this state, connectivity is not established.

This state is used during the End-of-Frame (past the EOF1 point) to ensure that the hub will be able to receive the SOF when it is sent by the host.

11.7.2 Wait for End of Packet from Upstream Port (WFEOPFU)

The hub enters this state if the hub is in the WFSOP or WFSOPFU state and an SOP is detected on the upstream port. The hub also enters this state from the WFSOP, WFSOPFU, or WFEOP states when the Receiver enters the Resume state.

While in this state, connectivity is established from the upstream port to all enabled downstream ports. Downstream ports that are in the Enabled state are placed in the Transmit state on the transition to this state.

11.7.3 Wait for Start of Packet (WFSOP)

This state is entered in any of the following situations:

- From the WFEOPstate when an EOP is detected from the downstream port
- From the WFEOPFU state if the frame timer is synchronized (locked) when an EOP is received from upstream
- From the WFSOPFU or WFEOPFU states when the upstream Receiver transitions to the Suspend state.

A hub in this state is waiting for an SOP on the upstream port or any downstream port that is in the Enabled state. While the Hub Repeater is in this state, connectivity is not established.

11.7.4 Wait for End of Packet (WFEOP)

This state is entered from the WFSOP state when an SOP is received from a downstream port in the Enabled state.

In this state, the hub has connectivity established in the upstream direction and the signaling received on an enabled downstream port is repeated and driven on the upstream port. The upstream Transmitter is placed in the Active state on the transition to this state.

If the Hub Repeater is in this state when the EOF2 point is reached, the downstream port for which connectivity is established is disabled as a babble port.

Note: the Transmitter will send an EOP at EOF1 but the Hub Repeater stays in this state until the device sends an EOP or the EOF2 point is reached.

11.8 Bus State Evaluation

<<Update for high-speed>>

A hub is required to evaluate the state of the connection on a port in order to make appropriate port state transitions. This section describes the appropriate times and means for several of these evaluations.

11.8.1 Port Error

A Port Error can occurs on a port that is in the Enabled state. A Port Error condition exists when::

- The hub is in the WFEOP state with connectivity established upstream from the port when the frame timer reaches the EOF2 point.
- At the EOF2 point the Hub Repeater is in the WFSOPFU state and there is other than an Idle/J state on the port.

If upstream-directed connectivity is established when the frame timer reaches the EOF1 point, the upstream Transmitter will generate a full-speed EOP to prevent the hub from being disabled by the upstream hub. The connected port is then disabled if it has not ended the packet and returned to the Idle state before the frame timer reaches the EOF2 point.

11.8.2 Speed Detection

The speed of an attached device is determined by the placement of a pull-up resistor on the device (see Section 7.1.5). When a device is attached, the hub is expected to detect the speed of the device by sensing the Bus Idle state. Due to connect and start-up transients, the hub may not be able to reliably determine the speed of the device until the transients have ended. The USB System Software is required to "debounce" the connection and provide a delay between the time a connection is detected and the device is used (see Section 7.1.7.1). At the end of the debounce interval, the device is expected to have placed its upstream

port in the Idle state and be able to react to reset signaling. The USB System Software must send a SetPortFeature(PORT_RESET) request to the port to enable the port and make the attached device ready for use. This provides a convenient time for the hub to evaluate the speed of the device attached to the port. Speed detection can be done at the beginning of the port reset as the port leaves the Disabled state or at the end of the port reset between the end of the Resetting state and the start of the Enabled state.

If an implementation chooses to do speed evaluation on entry to the Resetting state from the Disabled state, it will set the PORT_LOW_SPEED status according to the condition of the D+ and D- lines at that time. (Note: if both D+ and D- are high at this time, the hub may stay in the Disabled state and set the C_PORT_ENABLE bit to indicate that the hub could not determine the speed of the device. Otherwise the hub should assume that the device is low-speed.) This determines the speed of the device and the Idle/J state for the port. The hub will then drive an SE0 for the duration of the Resetting state timer. At the end of the Resetting state, the hub will drive the lines to the J state that is appropriate for the speed of the attached device and transition to the Enabled state.

Note: because the SendEOP state also exits to the Enabled state, an implementation might exit the Resetting state to the SendEOP state without driving the 'J' and then let the SendEOP circuit complete the operation.

If an implementation chooses to do speed evaluation on exit from the Resetting state, then it will need an additional state called the Speed_eval state. At the end of the Resetting state, the hub will float the D+ and D- lines and allow the lines to settle to the Idle state appropriate for the attached device. At the end of the Speed_eval state, the hub will set the PORT_LOW_SPEED status as appropriate. The Speed_eval state must last for at least 2.5µs but no longer than 1ms. It is possible that the port will detect a disconnect condition during the speed evaluation. If so, the port transitions to the Disconnected state and will not enter the Enabled state.

11.8.3 Collision

If the Hub Repeater is in the WFEOP state and an SOP is detected on another enabled port, a Collision condition exists. There are two allowed behaviors for the hub in this instance.

The first, and preferred, behavior is to ‘garble’ the message so that the host can detect the problem. The hub garbles the message by transmitting a 'K' on the upstream port. This 'K' should persist until packet traffic from all downstream ports ends. The hub should use the last EOP to terminate the garbled packet. babble detection is enabled during this garbled message.

A second behavior is to block the second packet and, when the first message ends, return the hub to the WFSOPFU or WFSOP state as appropriate. If the second stream is still active, the hub may reestablish connectivity upstream. This method is not preferred, as it does not convey the problem to the host. Additionally, if the second stream causes the hub to reestablish upstream connectivity as the host is trying to establish downstream connectivity, additional packets can be lost and the host cannot properly associate the problem.

11.8.4 Full- versus Low-speed Behavior

The upstream connection of a hub must always be a full-speed connection. All downstream ports of a hub that are attached to USB connectors must be able to support both full-speed and low-speed devices. When low-speed data is sent or received through a hub's upstream connection, the signaling is full-speed even though the bit times are low-speed.

Full-speed signaling must not be transmitted to low-speed ports.

If a port is detected to be attached to a low-speed device, the hub port’s output buffers are configured to operate at the slow slew rate (75-300ns), and the port will not propagate downstream-directed packets unless they are prefaced with a PRE PID. When a hub receives a PRE PID, it must enable the drivers on the enabled, low-speed ports within four bit times of receiving the last bit of the PID.

Note: when the driver is turned on, the upstream port will be in the 'J' state and the downstream ports should be driven to the same state.

Low-speed data follows the PID and is propagated to both low and full-speed devices. Hubs continue to propagate downstream signaling to all enabled ports until a downstream EOP is detected, at which time all output drivers are turned off.

Full-speed devices will not misinterpret low-speed traffic because no low-speed data pattern can generate a valid full-speed PID.

When a low-speed device transmits, it does not preface its data packet with a PRE PID. Hubs will propagate upstream-directed packets of any speed using full-speed signaling polarity and edge rates.

For both upstream and downstream low-speed data, the hub is responsible for inverting the polarity of the data before transmitting to/from a low-speed port.

Although a low-speed device will send a low-speed EOP to properly terminate a packet, a hub may truncate a low-speed packet at the EOF1 point with a full-speed EOP. Thus, hubs must always be able to tear down connectivity in response to a full-speed EOP regardless of the data rate of the packet.

Because of the slow transitions on low-speed ports, when the D+ and D- signal lines are switching between the 'J' and 'K', they may both be below 2.0V for a period of time that is longer than a full-speed bit time. A hub must ensure that these slow transitions do not result in termination of connectivity and must not result in an SE0 being sent upstream.

11.8.4.1 Low-speed Keep-alive

All hub ports to which low-speed devices are connected must generate a low-speed keep-alive strobe, generated at the beginning of the frame, which consists of a valid low-speed EOP (described in Section 7.1.13.2). The strobe must be generated at least once in each frame in which an SOF is received from the host. This strobe is used to prevent low-speed devices from suspending if there is no other low-speed traffic on the bus. The hub can generate the keep-alive on any valid full-speed token packet. The following rules for generation of a low-speed keep-alive must be adhered to:

- A keep-alive must minimally be derived from each SOF. It is recommended that a keep-alive be generated on any valid full-speed token.
- The keep-alive must start by the eighth bit after the PID of the full-speed token.

11.9 Suspend and Resume

<<Update for high-speed>>

Hubs must support suspend and resume both as a USB device and in terms of propagating suspend and resume signaling. Hubs support both global and selective suspend and resume. Global and selective suspend are defined in Section 7.1.7.4. Global suspend/resume refers to the entire bus being suspended or resumed without affecting any hub's downstream port states; selective suspend/resume refers to a downstream port of a hub being suspended or resumed without affecting the hub state. Global suspend/resume is implemented through the root port(s) at the host. Selective suspend/resume is implemented via requests to a hub. Device-initiated resume is called remote-wakeup (see Section 7.1.7.5).

Figure 11-10 shows the timing relationships for an example remote-wakeup sequence. This example illustrates a device initiating resume signaling through a suspended hub ('B') to an awake hub ('A'). Hub 'A' in this example times and completes the resume sequence and is the "Controlling Hub". The timings and events are defined in Section 7.1.7.5.

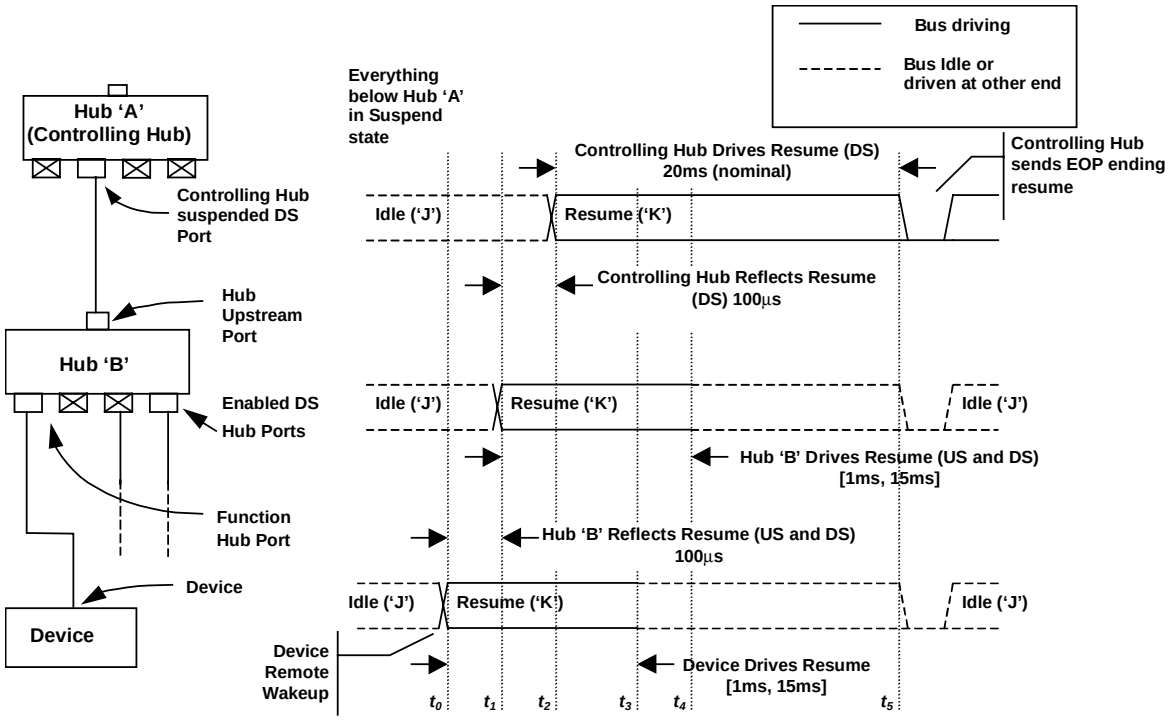

Figure 11-10. Example Remote-Wakeup Resume Signaling

Here is an explanation of what happens at each t_n:

t0 Suspended device initiates remote-wakeup by driving a 'K' on the data lines.

t1 Suspended hub 'B' detects the 'K' on its downstream port, wakes up enough within 100µs to reflect the resume upstream and down through all enabled ports.

t2 Hub 'A' is not suspended (implication is that the port at which 'B' is attached is selectively suspended), detects the 'K' on the selectively suspended port where 'B' is attached, and reflects the resume signal back to 'B' within 100µs.

t3 Device ceases driving 'K' upstream.

t4 Hub 'B' ceases driving 'K' upstream and down all enabled ports and begins repeating upstream signaling to all enabled downstream ports.

t5 Hub 'A' completes resume sequence, after appropriate timing interval, by driving a low-speed EOP downstream.

The hub reflection time is much smaller than the minimum duration a USB device will drive resume upstream. This relationship guarantees that resume will be propagated upstream and downstream without any gaps.

11.10 Hub Reset Behavior

<<Update for high-speed>>

The following sections describe hub reset behavior and its interactions with resume, attach detect, and power-on.

11.10.1 Hub Receiving Reset on Upstream Port

Reset signaling to a hub is defined only in the downstream direction, which is at the hub's upstream port. A

hub may start its reset sequence if it detects 2.5µs or more of continuous SE0 signaling and must complete its reset sequence by the end of the reset signaling.

Note: the 2.5µs lower limit is set by a need to prevent low-speed EOP strobes from being interpreted as reset.

A suspended hub must interpret the start of reset as a wakeup event; it must be awake and have completed its reset sequence by the end of reset signaling.

After completion of the reset sequence, a hub is in the following state:

- Hub Controller default address is 0
- Hub status change bits are set to zero
- Hub Repeater is in the WFSOPFU state
- Transmitter is in the Inactive state
- Downstream ports are in the Not Configured state and SE0 driven on all downstream ports.

11.11 Hub Port Power Control

Self-powered hubs may have power switches that control delivery of power downstream ports but it is not required. Bus-powered hubs are required to have power switches. A hub with power switches can switch power to all ports as a group/gang, to each port individually, or have an arbitrary number of gangs of one or more ports .

A hub indicates whether or not it supports power switching by the setting of the Logical Power Switching Mode field in *wHubCharacteristics*.If a hub supports per-port power switching, then the power to a port is turned on when a SetPortFeature(PORT_POWER) request is received for the port. Port power is turned off when the port is in the Powered-off or Not Configured states. If a hub supports ganged power switching, then the power to all ports in a gang is turned on when any port in a gang receives a SetPortFeature(PORT_POWER) request. The power to a gang is not turned off unless all ports in a gang are in the Powered-off or Not Configured states. Note, the power to a port is not turned on by a SetPortFeature(PORT_POWER) if both C_HUB_LOCAL_POWER and Local Power Status (in *wHubStatus*) are set to 1B at the time when the request is executed and the PORT_POWER feature would be turned on.

Although a self-powered hub is not required to implement power switching, the hub must support the Powered-off state for all ports. Additionally, the hub must implement the *PortPwrCtrlMask* (all bits set to 1b) even though the hub has no power switches that can be controlled by the USB System Software.

Note: to ensure compatibility with previous versions of USB software, hubs must implement the Logical Power Switching Mode field in *wHubCharacteristics*. This is because some versions of SW will not use the SetPortFeature() request if the hub indicates in *wHubCharacteristics* that the port does not support port power switching. Otherwise, the Logical Power Switching Mode field in *wHubCharacteristics* would have become redundant as of this version of the specification.

The setting of the Logical Power Switching Mode for hubs with no power switches should reflect the manner in which over-current is reported. For example, if the hub reports over-current conditions on a per-port basis, then the Logical Power Switching Mode should be set to indicate that power switching is controlled on a per-port basis.

For a hub with no power switches, *bPwrOn2PwrGood* should be set to zero.

11.11.1 Multiple Gangs

A hub may implement any number of power and/or over-current gangs. A hub that implements more than one over-current and/or power switching gang must set both the Logical Power Switching Mode and the

Over-current Reporting Mode to indicate that power switching and over-current reporting are on a per port basis (these fields are in *wHubCharacteristics*.) Also, all bits in *PortPwrCtrlMask* must be set to 1b.

When an over-current condition occurs on an over-current protection device, the over-current is signaled on all ports that are protected by that device. When the over-current is signaled, all the ports in the group are placed in the Powered-off state, and the C_PORT_OVER-CURRENT field is set to 1B on all the ports. When port status is read from any port in the group, the PORT_OVER-CURRENT field will be set to 1b as long as the over-current condition exists. The C_PORT_OVER-CURRENT field must be cleared in each port individually.

When multiple ports share a power switch, setting PORT_POWER on any port in the group will cause the power to all ports in the group to turn on. It will not, however, cause the other ports in that group to leave the Powered-off state. When all the ports in a group are in the Powered-off state or the hub is not configured, the power to the ports is turned off.

If a hub implements both power switching and over-current, it is not necessary for the over-current groups to be the same as the power switching groups.

If an over-current condition occurs and power switches are present, then all power switches associated with an over-current protection circuit must be turned off. If multiple over-current protection devices are associate with a single power switch then that switch will be turned off when any of the over-current protection circuits indicates an over-current condition.

11.12 Hub I/O Buffer Requirements

<<Update for high-speed>>

All hub ports must be able to detect and generate all the bus signaling states described in Table 7-1. This requires that hub be able to independently drive and monitor the D+ and D- outputs on each of its ports. Each hub port must have single-ended receivers and transmitters on the D+ and D- lines as well as a differential receiver and transmitter. Details on voltage levels and drive requirements appear in Chapter 7.

11.12.1 Pull-up and Pull-down Resistors

Hubs, and the devices to which they connect, use a combination of pull-up and pull-down resistors to control D+ and D- in the absence of their being actively driven. These resistors establish voltage levels used to signal connect and disconnect and maintain the data lines at their idle values when not being actively driven. Each hub downstream port requires a pull-down resistor (R_{pd}) on each data line; the hub upstream port requires a pull-up resistor (R_{pu}) on its D+ line. Values for R_{pu} and R_{pd} appear in Chapter 7.

11.12.2 Edge Rate Control

Downstream hub ports must support transmission and reception of both low-speed and full-speed edge rates. The respective signaling specifications are given in Chapter 7. Edge rate on a downstream port must be selectable, based upon whether a downstream device was detected as being full-speed or low-speed. The hub upstream port always uses full-speed signaling, and its output buffers always operate with full-speed edge rates and signal polarities.

11.13 Hub Controller

The Hub Controller is logically organized as shown in Figure 11-11.

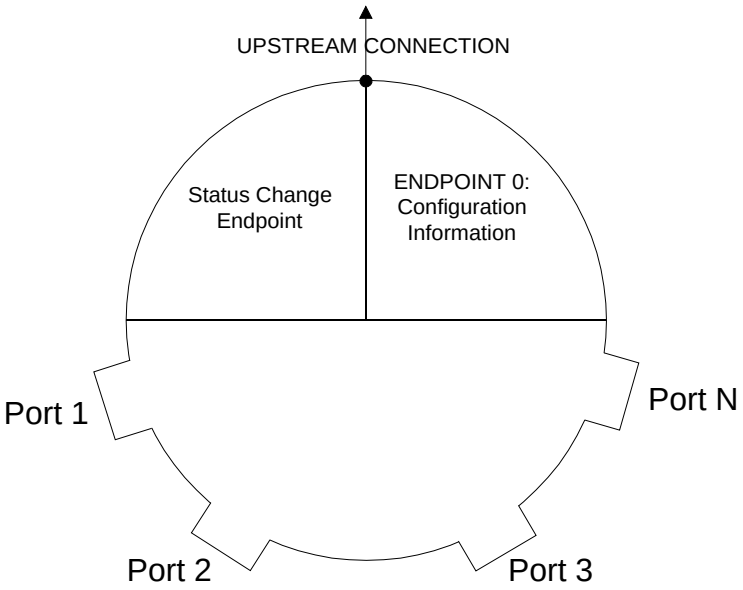

Figure 11-11. Example Hub Controller Organization

11.13.1 Endpoint Organization

The Hub Class defines one additional endpoint beyond Default Control Pipe, which is required for all ~~devices:~~hubs: the Status Change endpoint. The host system receives port and hub status change notifications through the Status Change endpoint. The Status Change endpoint is an interrupt endpoint. If no hub or port status change bits are set, then the hub returns an NAK when the Status Change endpoint is polled. When a status change bit is set, the hub responds with data, as shown in Section 11.13.4, indicating the entity (hub or port) with a change bit set. The USB System Software can use this data to determine which status registers to access in order to determine the exact cause of the status change interrupt.

11.13.2 Hub Information Architecture and Operation

Figure 11-12 shows how status, status change, and control information relate to device states. Hub descriptors and Hub/Port Status and Control are accessible through the Default Control Pipe. The Hub descriptors may be read at any time. When a hub detects a change on a port or when the hub changes its own state, the Status Change endpoint transfers data to the host in the form specified in Section 11.13.4.

Hub or port status change bits can be set because of hardware or software events. When set, these bits remain set until cleared directly by the USB System Software through a ClearPortFeature() request or by a hub reset. While a change bit is set, the hub continues to report a status change when polled until all change bits have been cleared by the USB System Software.

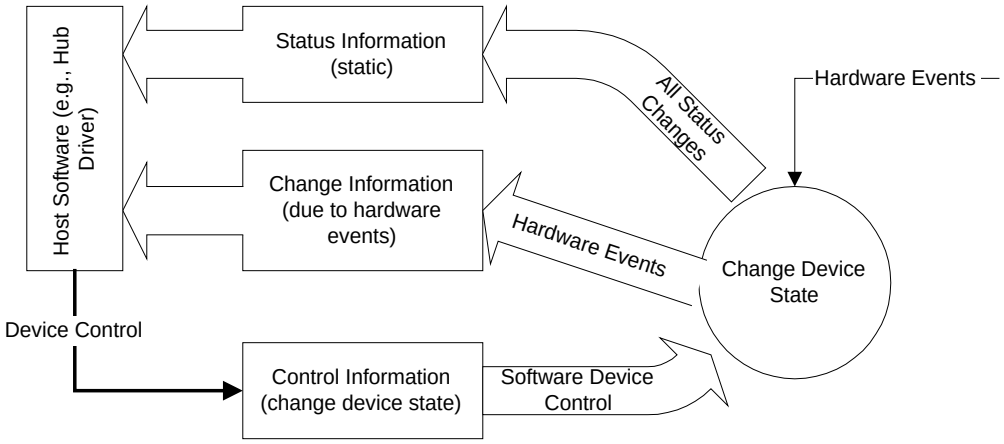

Figure 11-12. Relationship of Status, Status Change, and Control Information to Device States

The USB System Software uses the interrupt pipe associated with the Status Change endpoint to detect changes in hub and port status.

11.13.3 Port Change Information Processing

Hubs report a port's status through port commands on a per-port basis. The USB System Software acknowledges a port change by clearing the change state corresponding to the status change reported by the hub. The acknowledgment clears the change state for that port so future data transfers to the Status Change endpoint do not report the previous event. This allows the process to repeat for further changes (see Figure 11-13.)

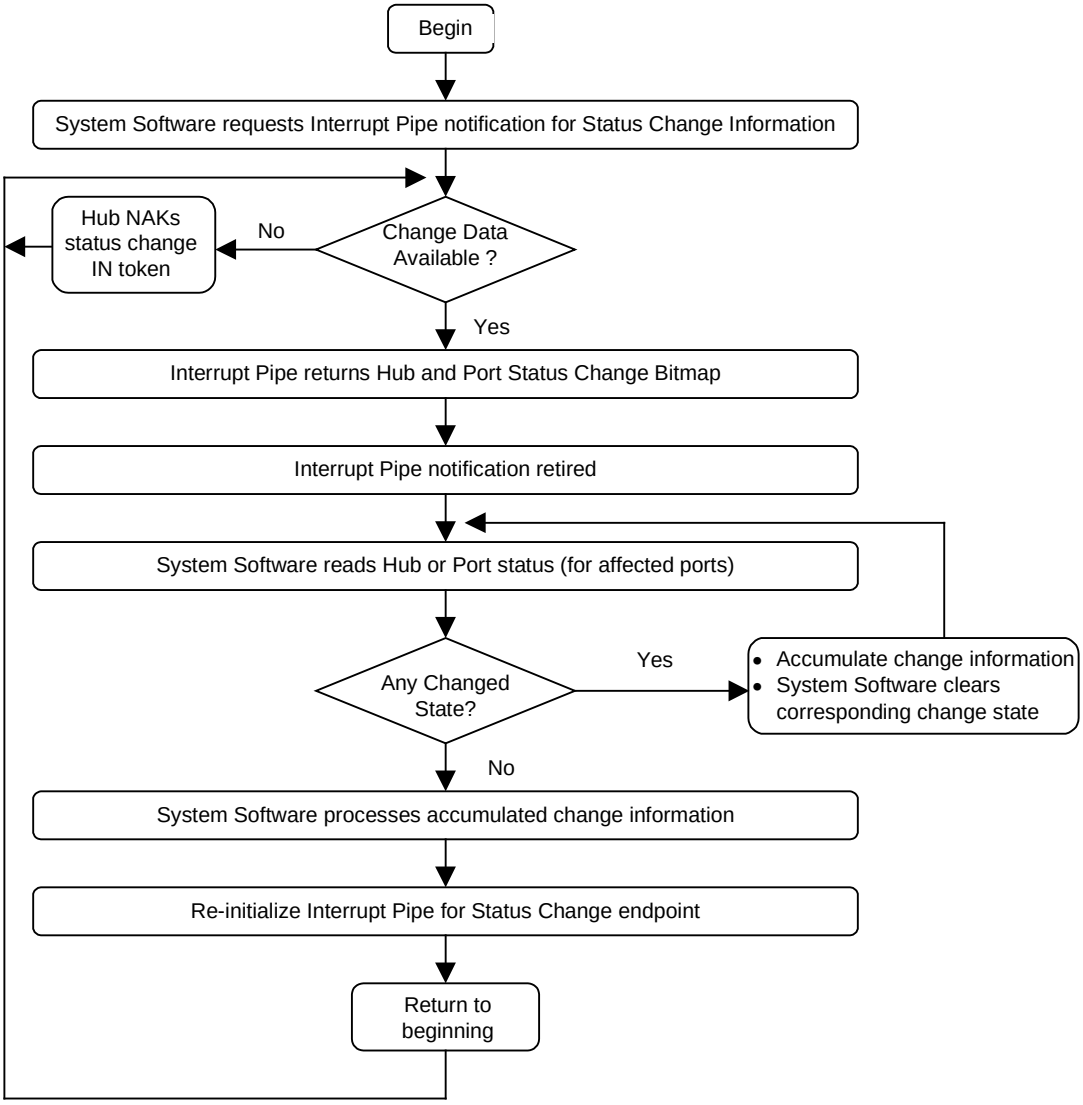

Figure 11-13. Port Status Handling Method

11.13.4 Hub and Port Status Change Bitmap

The Hub and Port Status Change Bitmap, shown in Figure 11-14, indicates whether the hub or a port has experienced a status change. This bitmap also indicates which port(s) have had a change in status. The hub returns this value on the Status Change endpoint. Hubs report this value in byte-increments. That is, if a hub has six ports, it returns a byte quantity and reports a zero in the invalid port number field locations. the USB System Software is aware of the number of ports on a hub (this is reported in the hub descriptor) and

decodes the Hub and Port Status Change Bitmap accordingly. The hub reports any changes in hub status in bit zero of the Hub and Port Status Change Bitmap.

The Hub and Port Status Change Bitmap size varies from a minimum size of one byte. Hubs report only as many bits as there are ports on the hub, subject to the byte-granularity requirement (i.e., round up to the nearest byte).

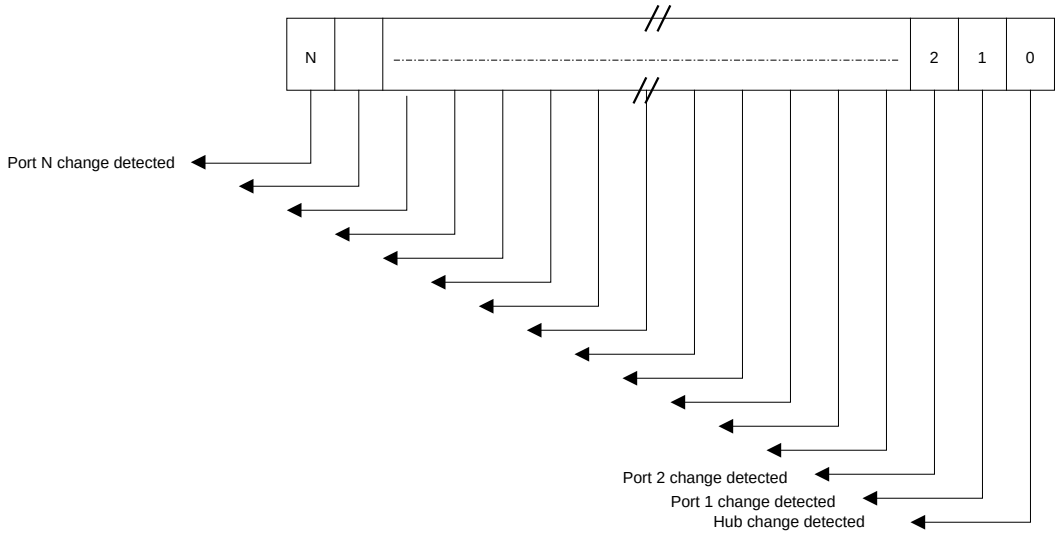

Figure 11-14. Hub and Port Status Change Bitmap

Any time the Status Change endpoint is polled by the host controller and any of the Status Changed bits are non-zero, the Hub and Port Status Change Bitmap is returned. Figure 11-15 shows an example creation mechanism for hub and port change bits.

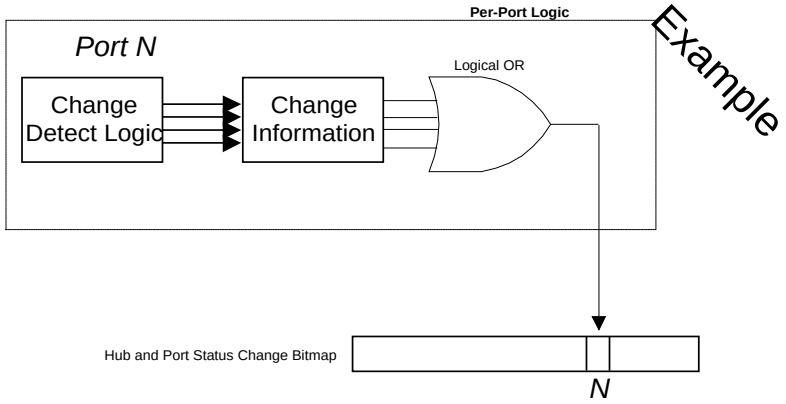

Figure 11-15. Example Hub and Port Change Bit Sampling

11.13.5 Over-current Reporting and Recovery

USB devices must be designed to meet applicable safety standards. Usually, this will mean that a self-powered hub implement current limiting on its downstream ports. If an over-current condition occurs, it causes a status and state change in one or more ports. This change is reported to the USB System Software so that it can take corrective action.

A hub may be designed to report over-current as either a port or a hub event. The hub descriptor field *wHubCharacteristics* is used to indicate the reporting capabilities of a particular hub (see Section 11.23.2). The over-current status bit in the hub or port status field indicates the state of the over-current detection when the status is returned. The over-current status change bit in the Hub or Port Change field indicates if the over-current status has changed.

When a hub experiences an over-current condition, it must place all affected ports in the Powered-off state. If a hub has per-port power switching and per-port current limiting, an over-current on one port may still cause the power on another port to fall below specified minimums. In this case, the affected port is placed in the Powered-off state and C_PORT_OVER_CURRENT is set for the port, but PORT_OVER_CURRENT is not set. If the hub has over-current detection on a hub basis, then an over-current condition on the hub will cause all ports to enter the Powered-off state. However, in this case, neither C_PORT_OVER_CURRENT nor PORT_OVER_CURRENT is set for the affected ports.

Host recovery actions for an over-current event should include the following:

1. Host gets change notification from hub with over-current event.
2. Host extracts appropriate hub or port change information (depending on the information in the change bitmap).
3. Host waits for over-current status bit to be cleared to 0.
4. Host cycles power on to all of the necessary ports (e.g., issues a SetPortFeature(PORT_POWER) request for each port).
5. Host re-enumerates all affected ports.

11.13.6 High Speed Enumeration Handling

The hub device class commands are used to manipulate its downstream port state. When a device is attached, the device attach event is detected by the hub and reported on the status change interrupt. The host will accept the status change report and request a SetPortFeature(PORT_RESET) on the port. As part of the bus reset sequence, a speed detect is performed by the hub's port hardware. If the attached device is high speed, the port is connected to the high-speed repeater. The Get_Status(PORT) request invoked by the host will return a "not PORT_LOW_SPEED and PORT_HIGH_SPEED" indication.

If the attached device is not high speed (e.g. the high speed detect fails), the port is routed to the transaction translator(TT) of the hub (or hub port). The Get_Status(PORT) will report "PORT_LOW_SPEED" or "not PORT_LOW_SPEED and not PORT_HIGH_SPEED" for low- or full-speed respectively.

When the device is detached from the port, the port reports the status change through the status change endpoint and the port will be reconnected to the high-speed repeater. Then the process is ready to be repeated on the next device attach detect.

11.14 Hub Configuration

Hubs are configured through the standard USB device configuration commands. A hub that is not configured behaves like any other device that is not configured with respect to power requirements and addressing. If a hub implements power switching, no power is provided to the downstream ports while the hub is not configured. Configuring a hub enables the Status Change endpoint. The USB System Software may then issue commands to the hub to switch port power on and off at appropriate times.

The USB System Software examines hub descriptor information to determine the hub's characteristics. By examining the hub's characteristics, the USB System Software ensures that illegal power topologies are not allowed by not powering on the hub's ports if doing so would violate the USB power topology. The device status and configuration information can be used to determine whether the hub should be used as a bus or self-powered device. Table 11-7 summarizes the information and how it can be used to determine the current power requirements of the hub.

Table 11-7. Hub Power Operating Mode Summary

Configuration Descriptor		Hub Device Status (Self Power)	Explanation
MaxPower	bmAttributes (Self Powered)		
0	0	N/A	N/A This is an illegal set of information.
0	1	0	N/A A device which is only self-powered, but does not have local power cannot connect to the Bus and communicate.

Table 11-7. Hub Power Operating Mode Summary (Continued)

Configuration Descriptor		Hub Device Status (Self Power)	Explanation
MaxPower	bmAttributes (Self Powered)		
~~0~~	~~1~~	~~1~~	~~Self-powered only hub and local power supply is good. Hub status also indicates local power good, see Section 11.16.2.5. Hub functionality is valid anywhere depth restriction is not violated.~~
0	1	1	Self-powered only hub and local power supply is good. Hub status also indicates local power good, see Section 11.24.2.6. Hub functionality is valid anywhere depth restriction is not violated.
> 0	0	N/A	Bus-powered only hub. Downstream ports may not be powered unless allowed in current topology. Hub device status reporting Self Powered is meaningless in combination of a zeroed *bmAttributes.Self-Powered*.
> 0	1	0	This hub is capable of both self- and bus-powered operating modes. It is currently only available as a bus-powered hub.
> 0	1	1	This hub is capable of both self- and bus-powered operating modes. It is currently available as a self-powered hub.

A self-powered hub has a local power supply, but may optionally draw one unit load from its upstream connection. This allows the interface to function when local power is not available (see Section 7.2.1.2). When local power is removed (either a hub-wide over-current condition or local supply is off), a hub of this type remains in the Configured state but transitions all ports (whether removable or non-removable) to the Powered-off state. While local power is off, all port status and change information read as zero and all SetPortFeature() requests are ignored (request is treated as a no-operation). The hub will use the Status Change endpoint to notify the USB System Software of the hub event (see Section 11.24.2.6 for details on hub status).

The *MaxPower* field in the configuration descriptor is used to report to the system the maximum power the hub will draw from VBUS when the configuration is selected. For bus-powered hubs, the reported value

must not include the power for any of external downstream ports. The external devices attaching to the hub will report their individual power requirements.

A compound device may power both the hub electronics and the permanently attached devices from VBUS. The entire load may be reported in the hubs' configuration descriptor with the permanently attached devices each reporting self-powered, with zero *MaxPower* in their respective configuration descriptors.

11.15 Transaction Translator

A hub has a special responsibility when it is operating in high-speed and has full/low-speed devices connected on downstream facing ports. In this case, the hub must isolate the high-speed signaling environment from the full/low-speed signaling environment. This function is performed by the Transaction Translator (TT) portion of the hub.

This section defines the required behavior of the transaction translator.

11.15.1 Overview

Figure 11-16 shows an overview of the Transaction Translator. The TT has buffers and tracks transaction state for the four USB transfer types. The high-speed handler accepts high-speed start-split transactions or responds to high-speed complete-split transactions. The high-speed handler places the start-split transactions in local buffers for the full/low-speed handler's use.

The buffered start-split transactions provide the full/low speed handler with the information that allows it to issue corresponding full/low speed transactions to full/low speed devices attached on downstream facing ports. The full/low speed handler buffers the results of these full/low speed transactions so that they can be returned with a corresponding complete-split transaction on high-speed.

The conversion between full/low speed transactions and corresponding high-speed split-transaction protocol is described in Section 8.4.2.

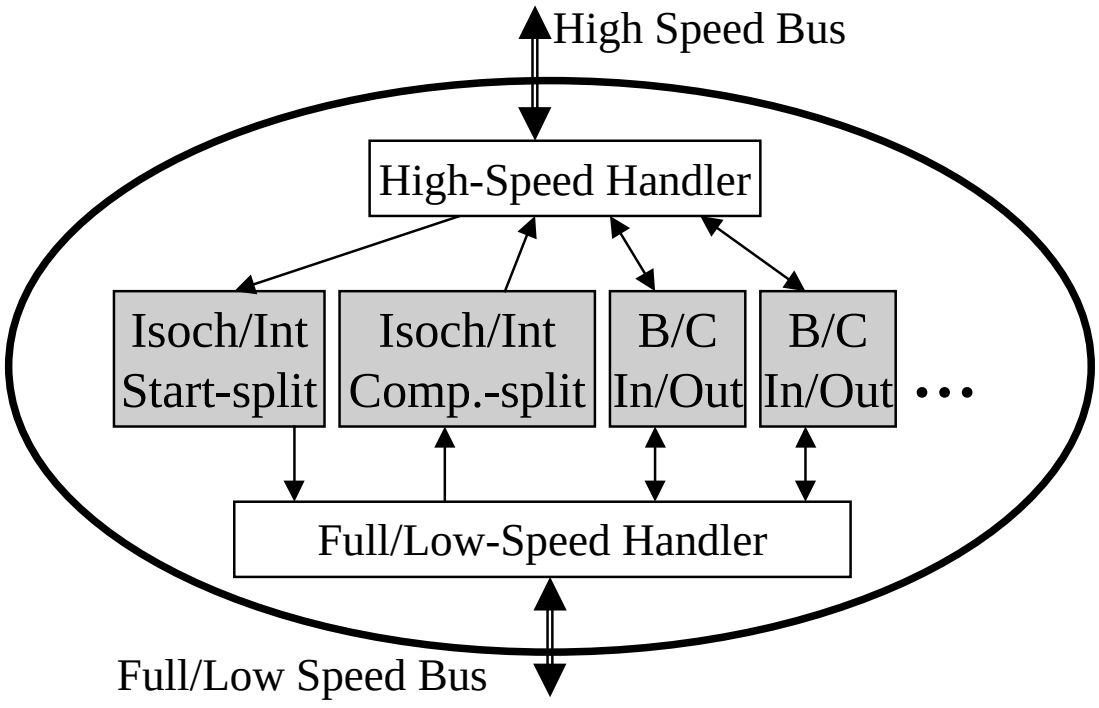

Figure 11-16 Transaction Translator Overview

The high-speed handler of the TT operates independently of the full/low-speed handler. They use the local buffers to exchange information where required.

The TT has two buffer and state tracking sections (shown in gray in Figure 11-16): periodic (for isochronous/interrupt full/low-speed transactions) and non-periodic (for bulk/control full/low-speed transactions). The requirements on the TT for these two buffer and state tracking sections are different. Each will be described in turn later in this chapter.

11.15.1.1 Data Handling Between High Speed and Full/Low Speed

The host converts full/low-speed transactions to corresponding high-speed split-transactions. Low speed Preamble(PRE) packets are never used on the high-speed bus to indicate a low-speed transaction. Instead, a low-speed transaction is encoded in the split transaction extended token. The split transactions appear in the host controller high-speed transaction schedule along with other high-speed transactions.

11.15.1.2 Host Controller and TT Split Transactions

The host controller uses the split transaction protocol for initiating full/low-speed transactions via the TT and then determining the completion status of the full/low-speed transaction. This approach allows the host controller to start a full/low-speed transaction and then continue with other high-speed activities while avoiding having to wait for the slower transaction to proceed/complete at its speed. A high-speed split transaction has two parts: a start-split and a complete-split. Split transactions are only used between the host controller and a hub. No other high-speed or full/low-speed devices ever use split transactions.

When the host controller sends a start-split transaction at high-speed, the TT for that device will accept the transaction and buffer it locally. The high-speed handler responds with an appropriate handshake to inform the host controller that the transaction has been accepted. Not all split transactions have a handshake phase to the start-split. The start-split transactions are kept temporarily in a TT local buffer.

The full/low-speed handler processes start-split periodic transactions in the buffer (in order) as the downstream bus is ready for the “next” transaction. The full/low-speed handler accepts any response information from the downstream bus (in response to the full/low-speed transaction) and accumulates them in a local buffer for later transmission to the host controller.

At an appropriate future time, the host controller sends a high-speed complete-split transaction to retrieve the status/data for appropriate full/low-speed transactions. The high-speed handler checks this high-speed complete-split transaction with the response at the head of the local response buffer and responds accordingly. The specific split transaction sequences are defined for each USB transfer type in later sections.

11.15.1.3 Multiple Transaction Translators

A hub has two choices for organizing transaction translators. A hub can have one transaction translator for all downstream facing ports that have full/low-speed devices attached or the hub can have one TT for each downstream facing port. The hub must report which organization is supports in the hub class descriptor.

11.15.2 Transaction Translator Scheduling

As the high-speed handler accepts start-splits, the full/low-speed transaction information and data for OUTs or the transaction information for INs accumulate in buffers awaiting their service on the downstream bus. The host manages the periodic TT buffers differently than the non-periodic buffers.

11.15.2.1 TT Isochronous/Interrupt (Periodic) Buffering

Periodic transactions have strict timing requirements to meet on a full/low-speed bus (as defined by the specific endpoint and transfer type). Therefore, transactions must move through across the high-speed bus, through the TT, across the full/low-speed bus, back through the TT and onto the high-speed bus in a timely

fashion. The TT implements a traditional pipeline of transactions with its periodic buffers. There is separate buffer space for start-splits and complete-splits. The host is responsible for filling and draining the pipeline correctly. The host software manages the host controller to cause high-speed split transactions to occur at the correct times to avoid over/under runs in the TT periodic pipeline. The host controller sends data "just in time" for full/low-speed OUTs and retrieves response data from full/low-speed INs to ensure that the periodic buffering required in the TT is the minimum possible. See Section 11.18 for more detailed information.

USB strictly defines the timing requirements of periodic transfers and the isochronous transport capabilities of the high-speed and full/low-speed buses. This allows the host to accurately predict when data for periodic transfers must be moved on both the full/low-speed and high-speed buses, whenever a client requests a data transfer with a full/low-speed periodic endpoint. Therefore the host can "pipeline" data to/from the hub so that it moves in a timely manner with its target endpoint. Once the configuration of a full/low-speed device with periodic endpoints is set, the host streams data to/from the hub to keep the device's endpoints operating normally.

11.15.2.2 TT Bulk/Control (Non-Periodic) Buffering

Non-periodic transactions have no timing requirements, but the TT should support the maximum full/low-speed throughput allowed. A TT provides a few transaction buffers for bulk/control transactions. The host and TT use simple flow control (NAK) mechanisms to manage the bulk/control non-periodic buffers. The host issues a start-split transaction and if there is available buffer space, the TT accepts the transaction. The full/low-speed handler uses the buffered information to issue the downstream full/low-speed transaction and then uses the same buffer to hold any results (e.g. handshake or data or timeout). The buffer is then emptied with a corresponding high-speed complete-split and the process continues.

11.15.2.3 Full/Low-Speed Handler Scheduling

The full/low-speed handler uses a simple, scheduled, priority scheme to service pending transactions on the downstream bus. Each microframe, the full/low-speed handler processes any accumulated start-split transactions in the isochronous/interrupt periodic buffers first. If there are start-split transactions pending in the bulk/control buffer(s) and there is sufficient time left in the full/low-speed 1ms frame to complete the transaction, the full/low-speed handler issues one of the transactions (in round robin order). Figure 11-17 shows pseudo code for the full/low-speed handler start-split transaction scheduling algorithm.

The TT also sequences the transaction state pipeline based on the high-speed microframe clock to ensure that it doesn't start full/low-speed transactions too early or too late. The "Advance_pipeline" procedure in the pseudo code illustrates how the TT keeps the microframe "pipeline" advancing. This procedure is described in more detail later in Figure 11-35.

```
While (1) loop
   While (not end of microframe) loop
      -- process next start-split transaction
      If pending periodic start-split transaction then
         Process full/low-speed transaction
      Else if (ready bulk/control transaction) and
              (fits in full/low-speed 1ms frame) then
         Process one transaction
      End if
   End loop

   Advance_Pipeline();  -- see description in Figure 11-35(below)
End loop
```

Figure 11-17 Example Full/Low-speed Handler Scheduling for Start-splits

<<expand, rewrite, move>> Given the strict sequencing of microframes and the derivation of the 1ms SOF from the 0th microframe, this eliminates a need to have any timing information carried in the periodic data stream sent to the Hub. See Section 11.18 for more information.

11.16 Split Transaction Notation Information

The following sections describe the details of the transaction phases and flow sequences of split-transactions for the different USB transfer types: bulk/control, interrupt and isochronous. Each description also shows detailed example host and TT state machines to achieve the required transaction definitions. Appendix <<TBD>> includes example transactions with different high-speed and full/low-speed results to clarify the relationships between the host controller, the hub's TT and a full/low-speed endpoint.

Low-speed is not discussed in detail since beyond the handling of PRE packets (which is defined in chapter 8), there are no packet sequencing differences between low- and full-speed.

For each transfer direction, reference figures showing the possible flow sequences for the start-split and the complete-split portion of the split transaction are shown in Appendix <<TBD>>.

<<<<Include flow sequences in chapter or move to appendix???? Currently, in chapter, but intro text says in appendix.>>>>

The transitions on the flow sequence figures have labels that correspond to the transitions in the host and TT state machines. These labels are also included in the appendix examples. The three character labels are of the form: < S | C >< T | D | H | E ><number>. S indicates that this is a start-split label. C indicates that this is a complete-split label. T indicates token phase, D indicates data phase, H indicates handshake phase, E indicates an error case. The number simply distinguishes different labels of the same case/phase in the same split-transaction part.

The flow sequence figures further identify the visibility of transitions according to the legend in Figure 11-18. The flow sequences also include some indication of states required in the host or TT or actions taken. The legend indicates how these are identified.

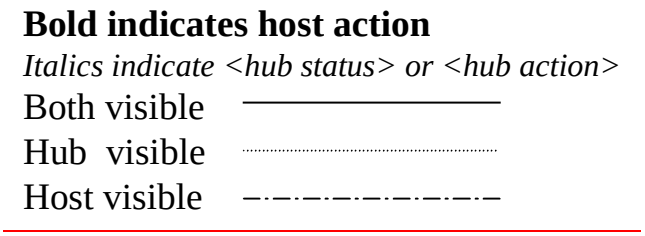

Figure 11-18 Flow Sequence Legend

Figure 11-19 shows the legend for the state machine diagrams.

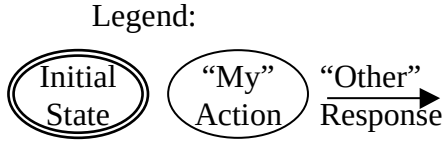

Figure 11-19 Legend for State Machines

The descriptions of the split-transactions for the four transfer types refer to the status of the full/low-speed transaction. This status is used by the high-speed handler to determine its response to a complete-split transaction. The status is only visible within a TT implementation and is used in the specification purely for ease of explanation. The defined status values are:

- Ready – The transaction has completed on the downstream facing full/low-speed bus with the result as follows:
  - Ready/NAK – A NAK handshake was received.
  - Ready /timeout or Old/t.o. – The full/low-speed transaction experienced a timeout between phases of the transaction.

- Ready /ACK – An ACK handshake was received.
- Ready /Stall – A STALL handshake was received.
- Ready /Data – A data packet was received and the CRC check passed. (bulk/control IN)
- Ready /badcrc – A data packet should have been received, but the CRC check failed.
- Ready /last – A data packet was finished being received (isochronous/interrupt IN)
- Ready /more – A data packet was being received when the microframe clock occurred. (isochronous/interrupt IN)

- Old – A complete-split has been received by the high-speed handler for a transaction that previously had a "ready" status. The possible status results are the same as for the Old status.
- Pending – The transaction is waiting to be completed on the downstream facing full/low-speed bus.

The figures use "old/*", "old/x", "ready/*", and "ready/x" to indicate any of the old or ready status respectively.

11.17 Bulk/Control Transaction Translation Overview

Each TT has at least two bulk/control transaction buffers. Each buffer holds the information for a start- or complete-split transaction and represents a single full/low-speed transaction that is awaiting transfer on the downstream bus. The buffer is used to hold the transaction information from the start-split (and data for an OUT) and then the status result of the full/low-speed transaction (and data for an IN). This buffer is filled and emptied by split-transactions via the high-speed handler. It is also updated by the full/low-speed handler while the transaction is in progress on the downstream bus.

The high-speed handler accepts a start-split transaction from the host controller for a bulk/control endpoint whenever the high-speed handler has space in a bulk/control buffer.

The host controller attempts a start-split transaction according to its bulk/control schedule. As soon as the high-speed handler buffer has space, it can accept the next start-split for some (possibly other) full/low-speed endpoint.

There is no method to control what start-split transaction is accepted next by the high-speed handler. Sequencing of start-split transactions is simply determined by available buffer space and the current state of the host controller schedule (e.g. what the next start-split transaction is that it tries as a normal part of processing high-speed transactions).

The host controller doesn't segregate split transaction bulk (or control) transactions from high-speed bulk (control) transactions when building its schedule. The host controller is required to track whether a transaction is a normal high-speed or a split- ransaction.

The following sections describe the details of the transaction phases and flow sequences and state machines for split-transactions used to support full/low-speed bulk and control OUT and IN transactions. There are only minor differences between bulk and control. In the figures, some areas are shaded to indicate that they don't apply for control transactions.

11.17.1 Bulk/Control Split Transaction Sequences

<<<More words to explain these figures…..>>>

The state machine figures show the transitions required for high-speed split transactions for full/low-speed bulk/control transfer types for a single endpoint. These figures must not be interpreted as showing any particular specific timing. In particular, other high-speed or split transactions can be "interleaved" with these transaction sequences. Specific details are described as appropriate.

Figure 11-20 shows a sample code algorithm that clarifies the requirements for the sh1/sh2/sh3 transitions for both Bulk/Control IN and OUT start-split transactions.

```
If all buffers are in the old buffer state:
  If a buffer has the same dev/endpoint/dir as specified in the start-split
          use that buffer to accept (and hold) the start-split and ACK
          -- This ensures that status information for an endpoint is only
          -- in a single TT buffer (otherwise response is ambiguous)
  Else
          use any buffer with old transaction status
  Endif
Else if some buffer has the same dev/endpoint/dir as specified in the start-split:
  if that buffer transaction status is pending or ready
          ACK the start-split but don't buffer the start-split
  Else
          use that buffer to accept (and hold) the start-split and ACK
  Endif
Else if some buffer transaction status is old:
  use that buffer to accept (and hold) the start-split and ACK
Else
  NAK
```

Figure 11-20 Sample Algorithm for Start-Split and Buffer State

Figure 11-21 shows the sequence of packets for a start-split transaction for the full/low-speed bulk OUT transfer type. The block labeled XOUT represents an extended token packet as described in Chapter 8. It is followed by an OUT token packet (or SETUP token packet for a control setup transaction). If the high-speed handler times out after the XOUT or OUT token packets, and doesn't see the following DATAx packet, it won't respond with a handshake as indicated by the dotted line transition labeled "se3". This causes the host to subsequently see a timeout (labeled "se2" and indicated with a dashed line). If the high-speed handler receives the DATAx packet and it fails the CRC check, it takes the transition "se1" which causes the host to timeout and take the "se2" transition.

If the high-speed handler doesn't have space to hold the start-split, it responds with a NAK via transition "sh3". This will cause the host to retry this start-split at some future time based on its normal schedule.

<<Require the HC to not issue another start-split after a NAK until a complete-split completes?>>

If the high-speed handler has space for the start-split, it takes transition "sh1" and responds with an ACK. This tells the host it must try a complete-split the next time it attempts to process a transaction for this full/low-speed endpoint. After receiving an ACK handshake, the host must not issue a further start-split for this endpoint until the corresponding complete-split has been completed.

If the high-speed handler already has a start-split for this full/low-speed endpoint pending, it follows transition "sh2" and also responds with an ACK, but ignores the data. This handles the case where an ACK handshake was smashed and missed by the host controller and now the host controller is retrying the start-split, e.g. a high-speed handler transition of "sh1" but a host transition of "se2".

In the host controller error cases, the host controller implements the "three strikes and you're out" mechanism. I.e. it increments an error count and if the error count is less than three (transition "se4"), it will retry the transaction. If the error count is greater or equal to three (transition "se5"), it does endpoint halt processing and doesn't retry the transaction.

The high-speed handler has no immediate knowledge of what the host sees, so the "se2", se3", "se4" and "se5" transitions show only host visibility.

This packet flow sequence showing the interactions between the host and hub is also represented by host and high-speed handler state machine diagrams in the next section. Those state machine diagrams use the same labels to correlate transitions between the two representations of the split-transaction rules.

Figure 11-22 shows the corresponding flow sequence for the complete-split transaction for the full/low-speed bulk/control OUT transfer type. The notation "ready/x" or "old/x" indicates that the TT transaction status of the split transaction is any of the ready or old states. After a full/low-speed transaction is run on the downstream facing bus, the split transaction status is updated to reflect the result of the transaction. The

possible result status is: nak, stall, ack. The “x” means any of the NAK, ACK, STALL full/low-speed transaction status results. Each status result reflects the handshake response from the full/low-speed transaction.

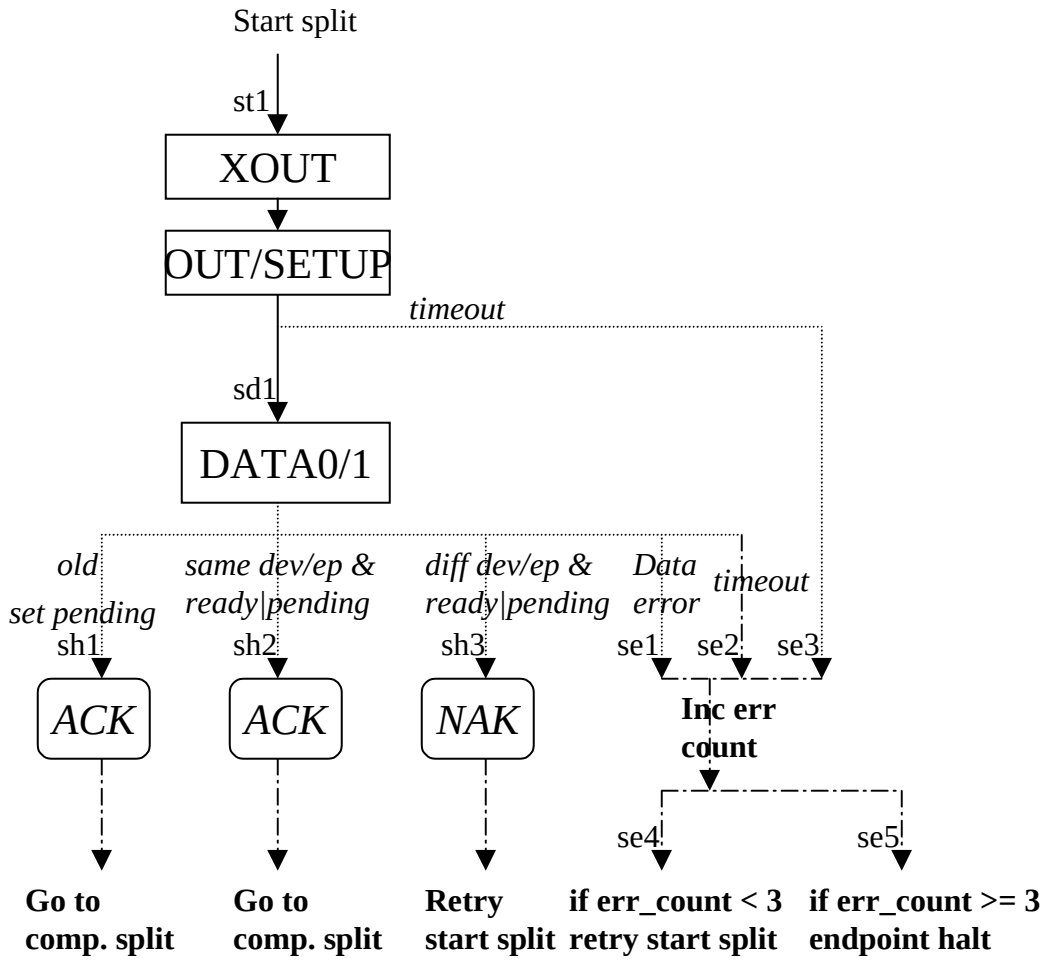

Figure 11- 21 Bulk/Control OUT Start-Split Transaction Sequence

There is no timeout response transaction status because the full/low-speed handler must perform a local retry of a full/low-speed bulk or control transaction that times out. It locally implements a “three strikes and you’re out” mechanism. This means that the full/low-speed transaction will resolve to one of a NAK, STALL or ACK handshake results. If the transaction times out three times, the full/low-speed handler will reflect this as a stall status result.

If the high-speed handler receives the complete-split extended token packet (and the token packet) and the full/low-speed transaction has not been completed (e.g. the transaction status is “pending”), the high-speed handler responds with a NYET handshake. This causes the host to retry the complete-split for this endpoint some time in the future.

If the high-speed handler recieves a complete-split extended token packet (and the token packet) and finds no local buffer with a corresponding transaction, the TT responds with a STALL to indicate a protocol violation.

Once the full/low-speed handler has finished a full/low-speed transaction it changes the transaction status from pending to ready and remembers the transaction result. This allows the high-speed handler to respond to the complete-split transaction with something besides NYET. Once the high-speed handler has seen a

complete-split, it changes the transaction status from ready/x to old/x. This allows the high-speed handler to reuse its local buffer for some other bulk/control transaction after this complete-split is finished.

If the host times out the transaction or doesn't receive a valid handshake, it immediately retries the complete-split before going on to any other bulk/control transactions for this high-speed handler. The normal "three strikes" mechanism applies here also for the host.

If the host receives a STALL handshake, it performs endpoint halt processing and won't issue any more split-transactions for this full/low-speed endpoint until the halt condition is changed.

If the host receives an ACK, it records the results of the full/low-speed transaction and advances to the next split transaction for this endpoint. The next transaction will be issued at some time in the future according to normal scheduling rules.

If host receives a NAK, it will retry the start-split transaction for this endpoint at some time in the future according to normal scheduling rules.

After the host receives a NAK, ACK or STALL handshake in response to a complete-split transaction, it may subsequently issue a start-split transaction for the same endpoint. The host may choose to instead issue a start-split transaction for a different endpoint that is not awaiting a complete-split response.

The shaded case shown in the figure indicates that a control setup transaction should never encounter a NAK response since that is not allowed for full/low-speed transactions.

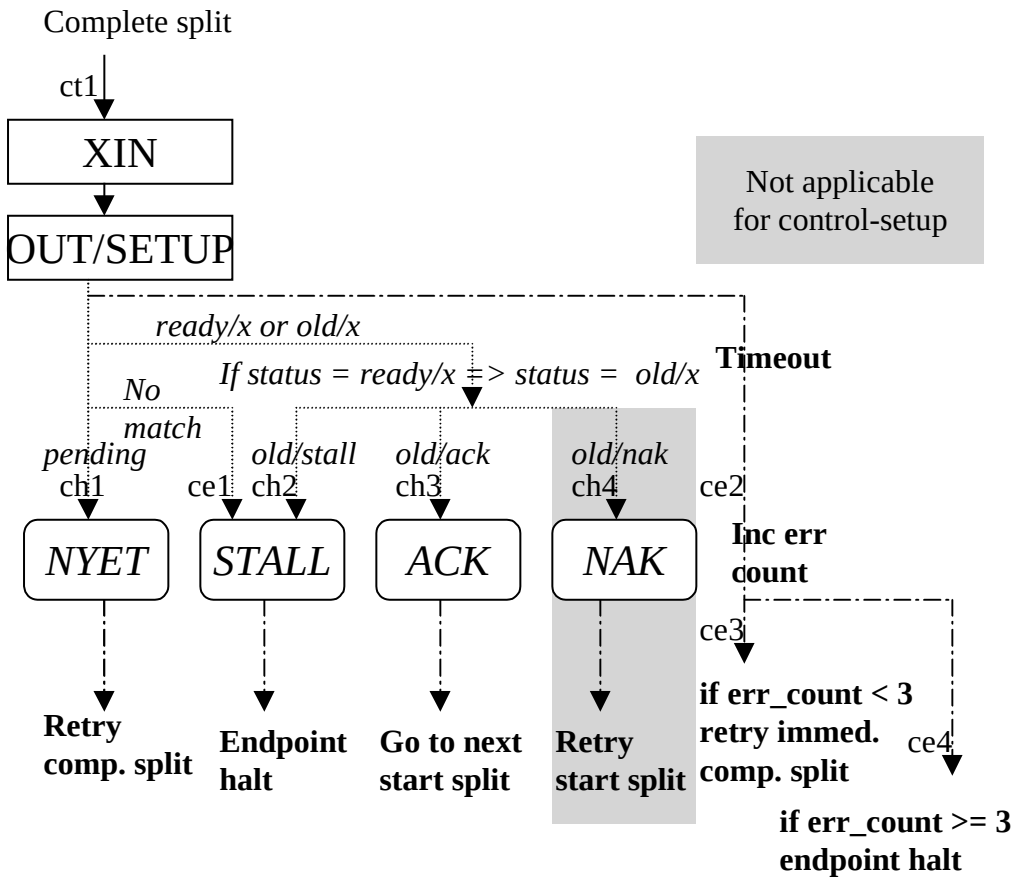

Figure 11-22 Bulk/Control OUT Complete-Split Transaction Sequence

Figure 11-23 and Figure 11-24 shows the corresponding flow sequences for bulk/control IN split-transactions.

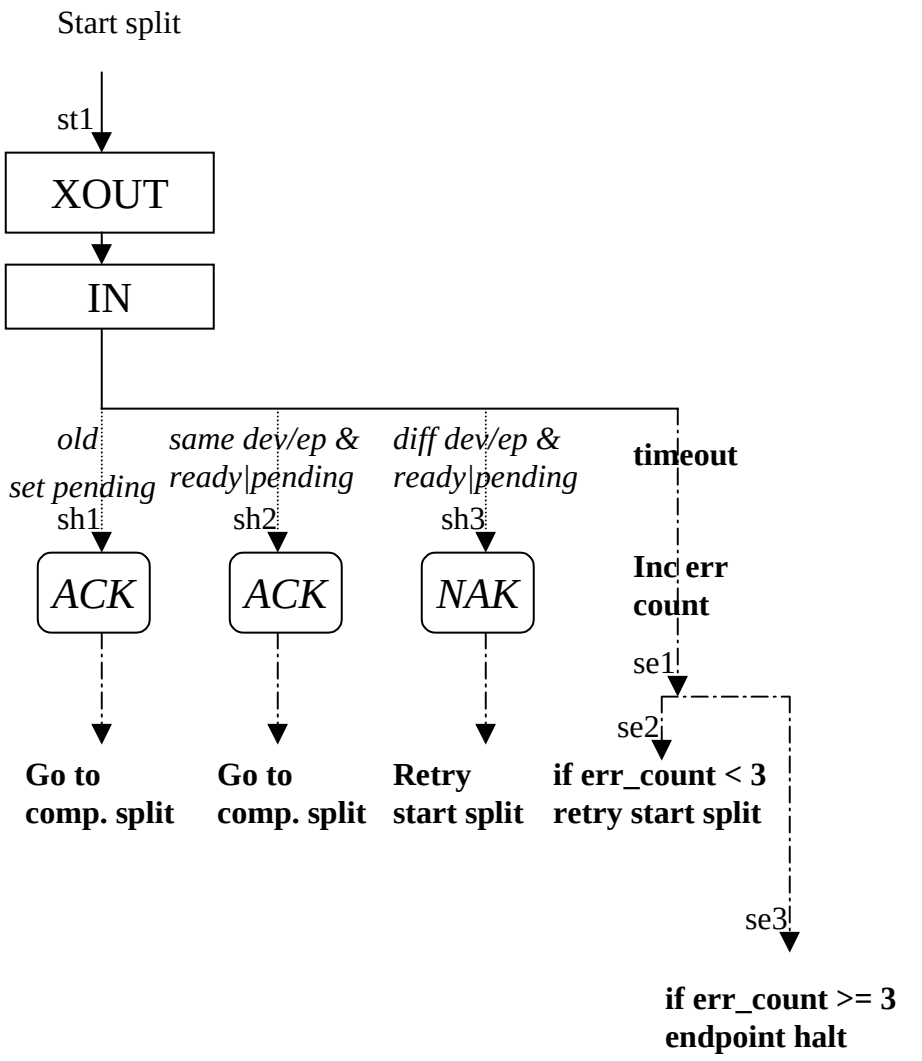

Figure 11-23 Bulk IN Start-Split Transaction Sequence

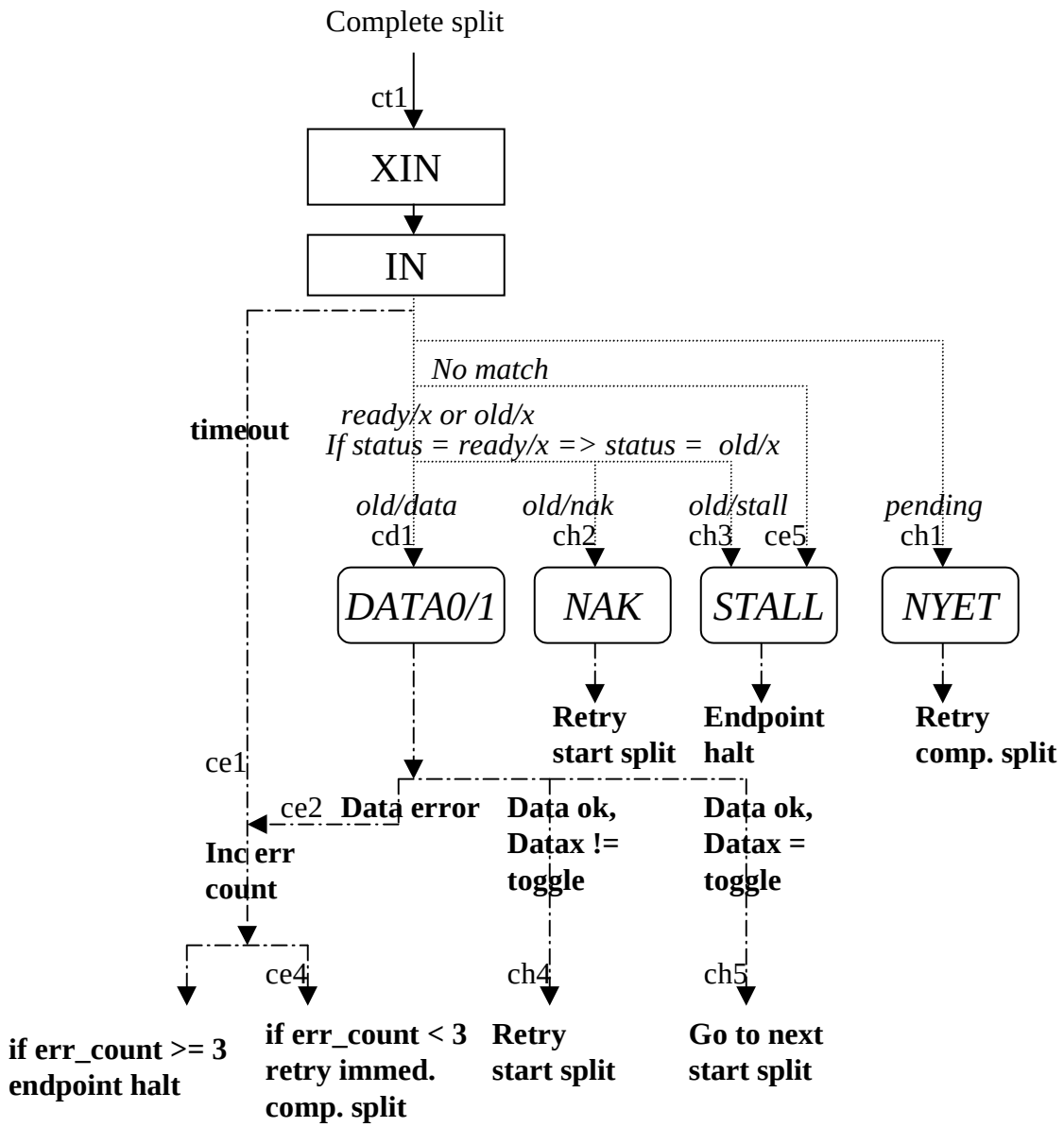

Figure 11-24 Bulk IN Complete-Split Transaction Sequence

11.17.2 Bulk/Control Split Transaction State Machines

The host and TT state machines for bulk/control IN and OUT split-transactions are show in the following figures. The transitions for these state machines are labeled the same as in the flow sequence figures.

The notation of "XIN+OUT", "XOUT+IN", etc. in the state machines indicates the extended token and the token packets.

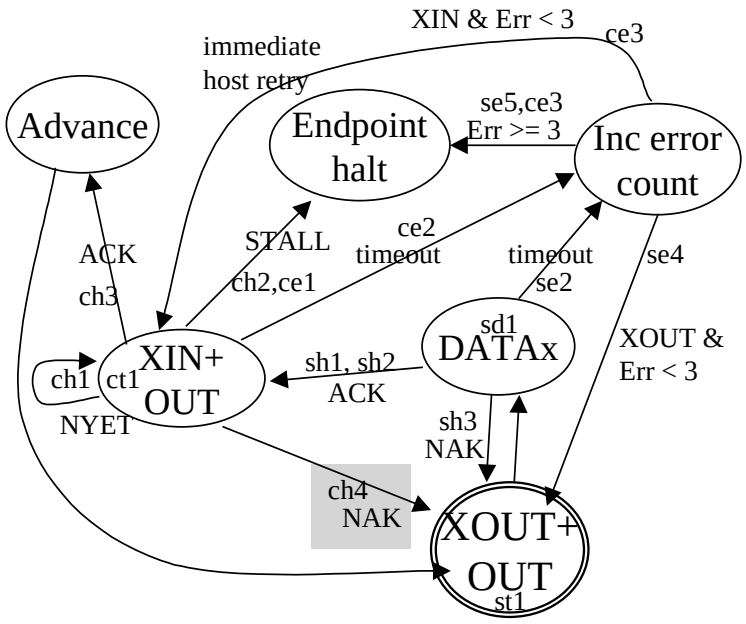

Figure 11-25 Bulk/Control OUT Split Transaction Host State Machine

<<missing SETUP cases to correspond with flow sequences>>

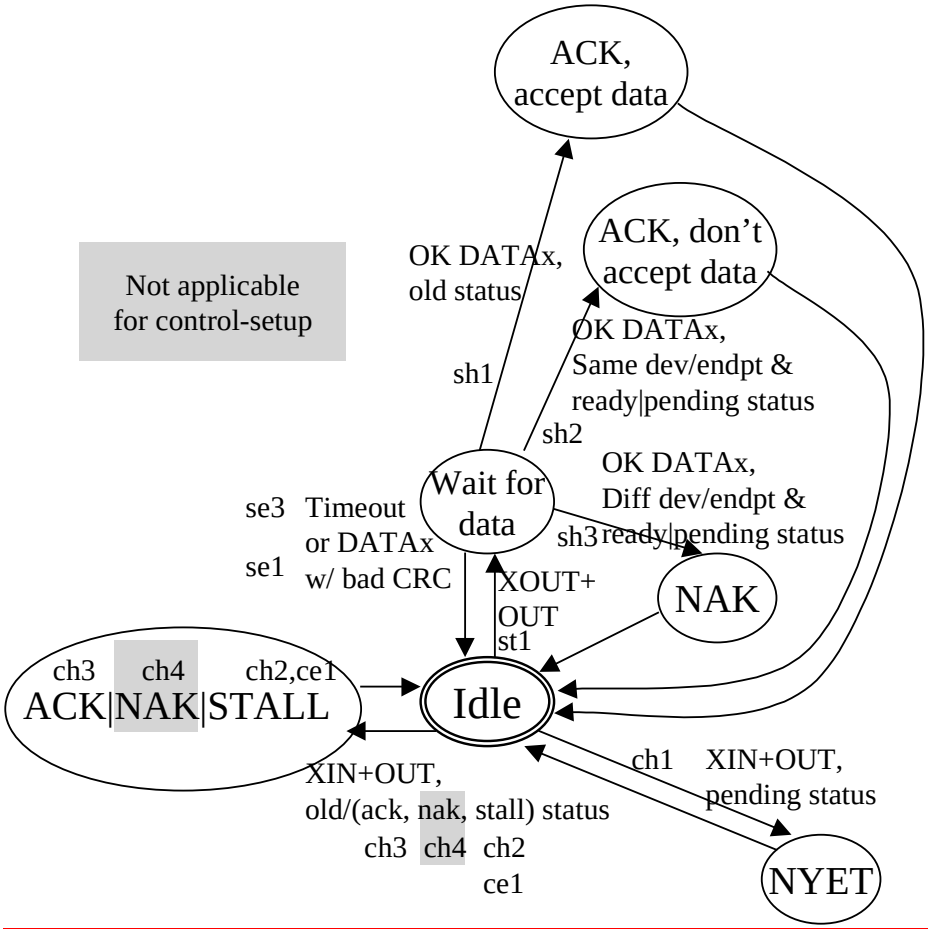

Figure 11-26 Bulk/Control OUT Split Transaction TT State Machine

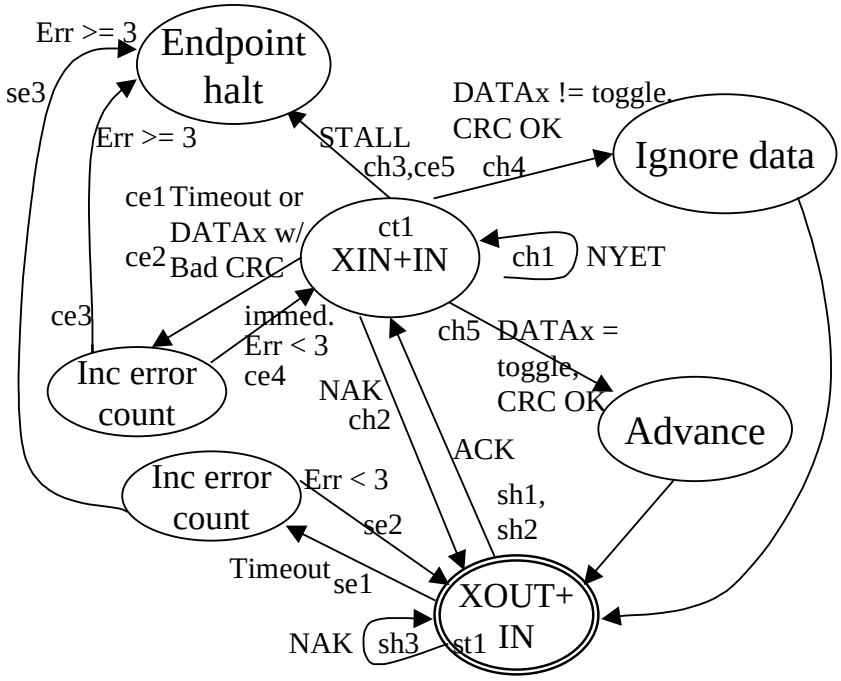

Figure 11-27 Bulk/Control IN Split Transaction Host State Machine

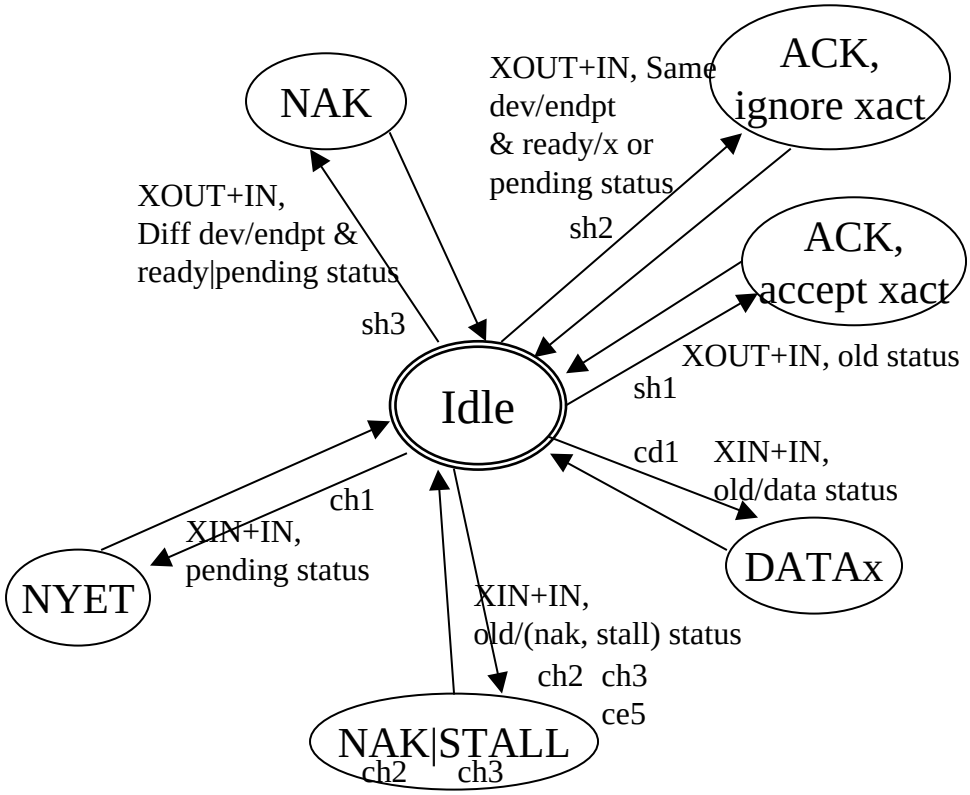

Figure 11-28 Bulk/Control IN Split Transaction TT State Machine

11.17.3 Bulk/Control Sequencing

Once the high-speed handler has received a start-split for an endpoint and saved it in a local buffer, it responds with an ACK split-transaction handshake. This tells the host controller to do a complete-split transaction next time this endpoint is polled.

As soon as possible (subject to scheduling rules described previously), the full/low-speed handler does the full/low-speed transaction and saves the handshake status (for OUT) or data/handshake status (for IN) in the same buffer.

Some time later (according to the host controller schedule), this endpoint will be polled for the complete-split transaction. The high-speed handler responds to the complete-split to return the full/low-speed endpoint status for this transaction (as recorded in the buffer). If the host controller polls for the complete-split transaction for this endpoint before the full/low-speed handler has finished processing this transaction on the downstream bus, the high-speed handler responds with a NYET handshake. This tells the host controller that the transaction is not yet complete. In this case, the host controller will retry the complete-split again at some later time.

When the full/low-speed handler finally finishes the full/low-speed transaction, it saves the data/status in the buffer to be ready for the next host controller complete-split transaction for this endpoint. At that time, the high-speed handler responds with the indicated data/status as recorded in the buffer. The buffer transaction status is updated from ready to old so the high-speed handler is ready for either a retry or a new start-split transaction for this (or some other) full/low-speed endpoint.

If there is an error on the complete-split transaction, the host controller will retry the complete-split transaction for this bulk/control endpoint “immediately” before proceeding to some other bulk/control split transaction. The host controller may issue other periodic split-transactions or other high-speed transactions before doing this complete-split transaction retry.

11.17.4 Bulk/Control Buffering Requirements

The TT must provide at least 2 transactions of non-periodic buffering to allow the hub to deliver maximum full/low-speed throughput on a downstream bus when the high-speed bus is idle.

As the high-speed bus becomes busier, the throughput possible on downstream full/low-speed buses will decrease.

A TT may provide more than 2 transactions of non-periodic buffering and this can improve throughput for downstream buses for specific combinations of device configurations.

11.17.5 Other Bulk/Control Details

When a bulk/control split transaction fails, it can leave the TT buffer in a busy (ready) state. This buffer state won’t allow the buffer to be reused for other bulk/control split transactions. Therefore as part of endpoint halt processing for full/low-speed endpoints connected via a TT, the host software must use the Clear_TT_Buffer request to the TT to ensure that the buffer is not in the busy state.

The Appendix shows examples of packet sequences for full/low-speed bulk/control transactions and their relationship with start-splits and complete-splits in various normal and error conditions.

11.18 Periodic Split Transaction Pipelining and Buffer Management

Interrupt and isochronous transfers require that the host controller carefully schedule split transactions with a TT for full/low-speed endpoints connected on a downstream bus. This section describes details of the TT pipeline that affect both isochronous and interrupt transactions. Then the split-transaction rules for interrupt and isochronous are described.

Bulk/control transactions are not scheduled with this mechanism. They are handled as described in the previous section.

11.18.1 Budgeted Full-Speed Wire Time

A microframe of time allows at most 187.5 raw bytes of signaling on a full-speed bus. For full-speed wire budgeting purposes, this is considered 188 bytes per microframe. However, due to maximum bit stuffing of 7 bits on the wire for 6 transmitted bits (e.g. 16.667% additional bits), 188 bytes of data can take 219.33 bytes of signaling time on a full-speed bus. At most 1157 byte times can be allocated to full/low-speed isochronous and interrupt endpoints to fit within the limit of 90%, e.g. 12Mb/s has at most 1500 bytes per 1ms frame, with 90% gives 1157 maximum periodic bytes per frame.

Figure 11-29 shows the maximum allocatable byte times in a 1ms frame and how that budgeted allocation relates to microframes.

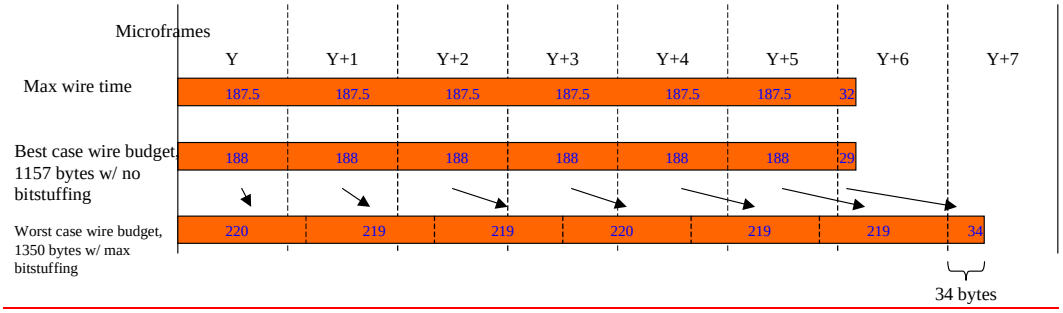

Figure 11-29 Full-speed wire time with budgeted minimum and maximum bit stuffing

The scheduling rules for a TT allow a full/low-speed bulk or control transaction to be "reclaimed" whenever there are no currently pending start-splits in a microframe. This can cause the end of the periodic allocation to be delayed some additional time.

A low-speed control transaction can delay the budgeted maximum allocation as shown in Figure 11-30. The low-speed control transaction takes 161 bytes of time on the bus with maximum bit stuffing and protocol overhead. The 161 bytes of time are calculated as:

1. 8 bytes of data at low-speed (64 full -peed byte times)
2. 9 bytes of low-speed protocol overhead and 4 bytes of full-speed bus turnaround timing (from Chapter 5) giving 76 full-speed byte times
3. 2 bytes of low-speed bit stuffing from 1/6 * 8 data bytes and 3 protocol bytes that can be bit stuffed (16 full-speed byte times)
4. 5 bytes of PRE packets (2 PREs at 2 bytes + 4 bits post-PRE idle each)

A bulk transaction can only be issued on the full/low-speed bus when the full/low-speed bus is otherwise idle. For the bus to become idle, the start-splits issued that were assumed to take up to 220 byte times worst case must instead have completed in 187 byte times or less (e.g. better than "best" case). This is approximately 33 bytes less bus time compared to the previously calculated baseline bitstuffed worst case. This means that the addition of a "reclaimed" bulk/control transaction results in a delta of 128 (=161 – (220-187)) bytes compared to the maximally bit stuffed worst case, e.g. this extends the end of the periodic allocation by 128 byte times.

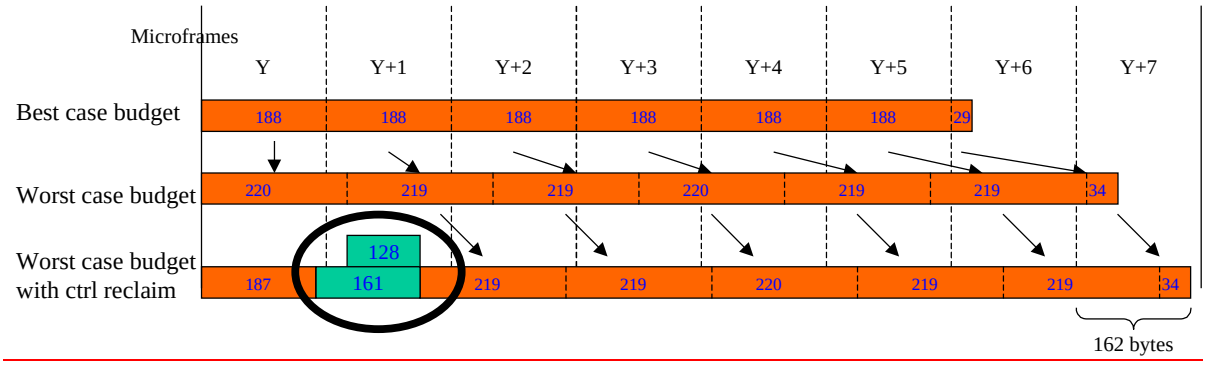

Figure 11-30 Low Speed Control Transaction Impact on schedule

This worst case budget includes a low-speed control packet artificially included in the first microframe to account for delay effects caused by "random" bandwidth reclamation.

11.18.2 TT Microframe Pipeline

The TT implements a microframe pipeline of split transactions in support of a full/low-speed bus. Start-split transactions are scheduled a microframe before the earliest time that their corresponding full/low-speed transaction is expected to start. Complete-split transactions are scheduled in microframes that the full/low-speed transaction can finish.

When a full/low-speed endpoint is attached to the bus and configured, the host schedules some time on the full/low-speed bus at some budgeted time based on the endpoint requirements of the configured device.

The effects of bit stuffing can delay when the full/low-speed transaction actually runs. The results of other previous full/low-speed transactions can cause the transaction to run earlier or later on the full/low-speed bus.

This leads to the host constructing two budgets for how the full/low-speed downstream bus will actually run: one best case budget assuming minimum bit stuffing and one worst case budget assuming maximum bit stuffing. The host always uses the maximum data payload size for a full/low-speed endpoint in doing its budgeting. It doesn't attempt to schedule the actual data payloads that may be used in specific transactions to full/low-speed endpoints.

Figure 11-31 shows an example of a new endpoint that is assigned some portion of a full/low-speed bus and where its start and complete splits are generally scheduled. More precise rules for scheduling are presented later. The start-split for this example transaction is scheduled in microframe Y-1, the transaction is budgeted to run in microframe Y and the complete-split is scheduled for microframe Y+1.

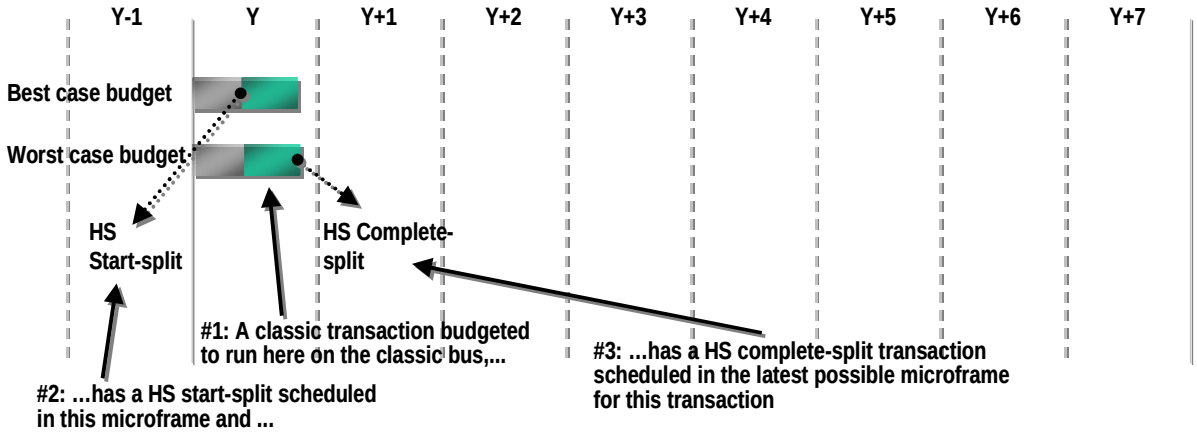

Figure 11-31 TT Microframe Pipeline

The host must determine precisely when start- and complete- splits are scheduled to avoid overruns or underruns in the TT.

11.18.3 Generation of Full-speed Frames

The TT must generate SOFs on the full-speed bus to establish the 1ms frame clock within the defined jitter tolerances. The TT has its own frame clock that is synchronized to the microframe SOFs on the high-speed bus. The SOF that reflects a change in the frame number it carries is identified as the zero-th micro-SOF. That high-speed SOF corresponds to the full-speed SOF on the TT's downstream facing bus. The TT must adhere to all timing/jitter requirements of a host controller related to SOF as defined in other parts of this specification.

The time delay from the occurrence of the high-speed SOF to the generation of the full-speed SOF by the TT on the downstream facing bus must be less than <<TBD; 16?>> full-speed bit times.

11.18.4 Start-Split Scheduling

When the host schedules start-splits for isochronous IN or interrupt full/low-speed transactions, it determines the earliest start for the full/low-speed transaction based on the best case schedule. It "overlays" a microframe "template" on the 1ms frame schedule to determine in which microframe the full/low-speed transaction can start. Then it schedules a start-split on the high-speed bus one microframe earlier. Figure 11-32 shows some example full/low-speed transactions. The first transaction in the 1ms frame would have a start-split scheduled in microframe Y-1. Isochronous IN and interrupt IN/OUT split transactions never require more than a single start-split transaction.

When the host is scheduling an isochronous OUT transaction, if the transaction is shorter than 188 byte times, it is scheduled as above. If it is longer than 188 byte times, multiple start-splits are scheduled. The first start-split is scheduled as described above. Another start-split is scheduled for each additional 188 byte times of budgeted full/low-speed bus time required. The fourth transaction in the figure shows two start-splits: one for the first 188 byte times of the full/low-speed transaction and the second for the remainder. At most one start-split for a full/low-speed transaction is scheduled in each microframe. An isochronous OUT split transaction requires multiple start-split transactions, one for each 188 bytes of data payload.

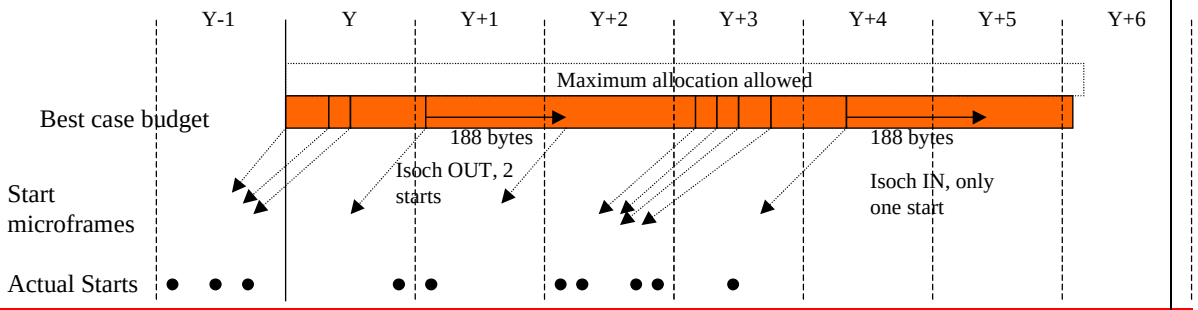

Figure 11-32 Start-Split Scheduling

The "pattern" of start-splits that need to be scheduled for a full/low-speed transaction can be computed once when the endpoint is configured. Then the pattern doesn't change unless some change occurs to the collection of currently configured full/low-speed endpoints.

11.18.5 Complete-Split Scheduling

The host schedules complete-split transactions for a full/low-speed transaction to retrieve the results of the transaction from the TT. Isochronous OUT transactions do not have complete-split transactions, so the host must not schedule any.

For interrupt transactions, the host uses the best case and worst case budgets and determines the microframes in which the full/low-speed transaction can complete. Once the microframe(s) in which the full/low-speed transaction can finish is determined, the host schedules complete-split(s) on the high-speed bus in the next microframe(s). Figure 11-33 shows an example of best/worst budgeted full/low-speed transactions and their corresponding scheduled complete-splits. The first full/low-speed transaction has a complete-split scheduled for microframe Y+1.

For isochronous IN full-speed transactions, the host also schedules multiple complete-splits when the transaction is budgeted for multiple microframes according to the best and worst case budgets. A complete-split transaction is scheduled by the host for each microframe that the transaction can receive data from the full-speed bus. In the figure according to the worst case budget, the last transaction has its last complete-split scheduled in microframe Y+8. It also has complete-splits scheduled in microframes Y+7 and Y+6. A complete-split is also scheduled in microframe Y+5 based on the best case budget.

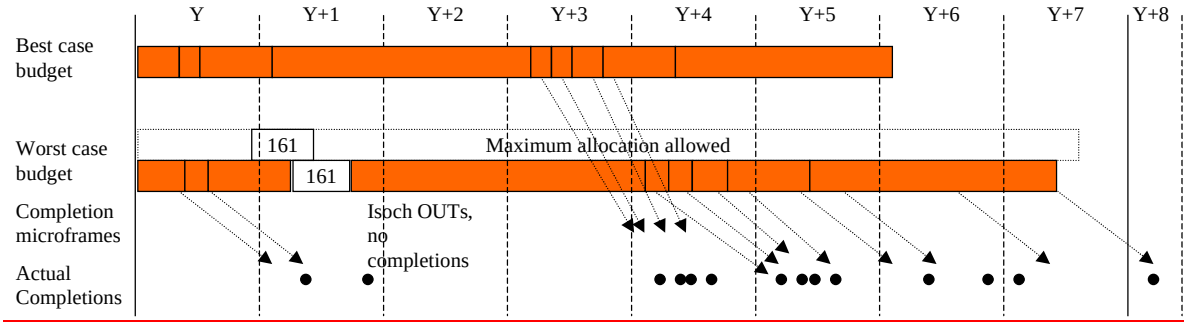

Figure 11-33 Complete Split Scheduling

The "pattern" of complete-splits that need to be scheduled for a full/low-speed transaction can be computed once when each endpoint is configured. Then the pattern doesn't change unless some change occurs to the collection of currently configured full/low-speed endpoints attached via a TT.

The approach used for long full-speed isochronous INs and interrupt INs/OUTs ensures that there is always an opportunity for the TT to return data/results whenever it has something from the full/low-speed transaction. Then whenever the full/low-speed handler starts the full/low-speed transaction, it simply accumulates the results in each microframe and then returns it in response to a complete-split from the host. The TT acts similar to an isochronous device in that it uses the microframe clock to "carve up" the full/low-speed data to be returned to the host. The TT doesn't do any computation on how much data to return when. It simply returns whatever data it got from full/low-speed bus in a microframe in response to the "next" complete-split.

So in this example, the host schedules complete splits for Y+5 through Y+8. The complete-split in Y+5 is scheduled due to the best case schedule for this transaction (as shown in Figure 11-32). The full-speed transaction can run no earlier than the beginning of Y+4 (if the previous transactions were shorter than budgeted). If the full-speed IN started at the very beginning of Y+4, it could generate at most approximately 188 bytes (minus protocol overhead) of data during that microframe. Whenever the microframe clock ticks, that amount of data will be returned to the host in response to the complete-split in microframe Y+5. And so on.

Whenever the TT has data to return in response to a complete-split for an interrupt full/low-speed or isochronous full-speed transaction, it uses either a DATA0/1 or MDATA PID for the data packet. If it has completed the full-speed isochronous transaction during the microframe before the complete-split, it uses the DATAx PID for the data packet of the complete-split transaction to indicate that this is the last data of the full-speed transaction. A DATA0 PID is always used for isochronous transactions. For interrupt transactions, a DATA0/1 PID is used corresponding to the full/low-speed data packet PID. If it is still receiving data on full-speed at the microframe clock, the high-speed handler uses the MDATA PID in the complete-split transaction to indicate to the host that more data is being received and another complete-split transaction is required.

After the full-speed transaction finishes, the high-speed handler responds with a DATAx (instead of MDATA) for that microframe's complete-split (which would likely be in Y+7 or earlier depending on how much full-speed data was provided by the full-speed device). When the host controller receives the DATAx PID, it stops issuing any remaining complete-splits that might be scheduled (for example those scheduled in Y+8 or earlier).

Now, what happens if a full/low-speed transaction isn't run on the full/low-speed bus as soon as possible, but as late as possible? In this case, it won't start until Y+5 and therefore there can't be any data to return to the complete split in Y+5. So when the complete-split is sent by the host, the high-speed handler searches for a corresponding status tracking entry in the complete-split buffer and doesn't find one in the ready or old state. The state machines and flow sequences show that in this case, the high-speed handler responds with an NYET. The host controller will issue additional complete-splits in the next microframe(s) until the last complete-split that was scheduled. The processing from here on is the same as before.

So the schedule is computed to ensure at most one complete-split in those microframes where the TT could have data for the host. However, the high-speed handler simply returns whatever data it has accumulated between microframe marks for that full/low-speed endpoint. The host simply advances its data pointer based on the data it receives from the TT.

11.18.6 TT Transaction Tracking

Figure 11-34 shows the TT microframe pipeline of transactions. The 8 high-speed microframes that compose a full/low-speed frame are labeled with NOW-7, NOW-6, etc. assuming the microframe “clock” has occurred at the point in time shown by the arrow (e.g. time “NOW”).

As shown in the figure, a start-split high-speed transaction that the high-speed handler receives in microframe NOW-7 (e.g. a start-split “B”) can run on the full/low-speed bus during microframe time NOW-6 or NOW-5 or NOW-4. This variation in starting on the full/low-speed bus is due to bit stuffing and bulk/control reclamation that can occur on the full/low-speed bus. Once the full/low-speed transaction finishes, its complete-split transaction (if one is required) will run on the high-speed bus during the microframe after the latest that the full/low-speed transaction can complete. Only full/low-speed transactions that are budgeted early in microframe NOW-6 can have their complete-split scheduled in microframe NOW-5 or NOW-4.

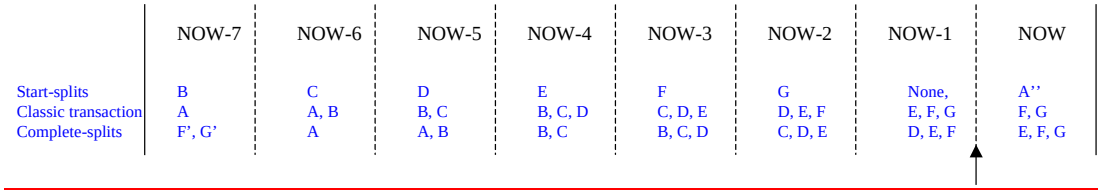

Figure 11-34 Microframe Pipeline

When the microframe clock for NOW occurs, the high-speed handler must mark any start-splits it received in microframe NOW-1 as “pending” so that they can be processed on the full/low-speed bus as appropriate. This prevents the full/low-speed transactions from running too early.

Also, the high-speed handler must change any start-split transactions that are still pending from microframe NOW-4 to “ready/timeout”. If the transaction is in progress on the downstream facing bus, the transaction must be aborted (in methods as defined in Chapter 8). This ensures that even if the full/low-speed bus has encountered a babble condition on the bus (or other delay condition), the TT keeps its transaction pipeline running on time (e.g. transactions don’t run too late). This also ensures that when the last scheduled complete-split transaction is received by the TT, the full/low-speed transaction has been completed (either successfully or by being aborted). <<rewrite to describe some microframes require NOW-3 for this case>>

Finally, the high-speed handler must change any complete-split transaction responses in the ready state from microframe NOW-4 to the free state so that their space can be reused in the next microframe.

This algorithm is shown in pseudo code in Figure 11-35. This pseudo-code corresponds to the Advance_pipeline procedure identified previously.

```
-- Clean up start-split state in case full/low-speed bus fell behind
while start-splits in pending state received by TT before microframe-4 loop
   Set to ready/timeout response status
End loop

-- Clean up complete-split pipeline in case no complete-splits were received
While ready/x complete-split transaction states from (previous_microframe) loop
   Free response transaction entry
End loop

-- Enable full/low-speed transactions received in previous microframe
While start-split transactions from (previous_microframe) loop
   Set to pending status
End loop
```

Figure 11-35 Advance_Pipeline Pseudocode

The full/low-speed handler creates a complete-split entry whenever a full/low-speed transaction completes. The full/low-speed handler also creates a "more data" entry for IN transactions when the microframe clock "ticks"and a full/low-speed transaction is in progress. A complete-split entry for a completed full/low-speed IN transaction (with no errors) will use a DATAx PID for the split-transaction response data packet. A complete-split entry for a "more data" entry will use the MDATA PID for the split-transaction response data packet.

11.18.7 TT Complete-Split Transaction State Searching

A host is required to issue complete-split transactions for a set of full/low-speed endpoints in the same order as the start-splits were issued for this TT. However, errors on start- or complete-splits can cause the high-speed handler to receive a complete-split transaction that doesn't "match" the expected next transaction according to the TT's transaction pipeline.

The TT has a pipeline of complete-split transaction state that it is expecting to use to respond to complete-split transactions. Normally the host will issue the complete-split that the high-speed handler is expecting next and the complete-split will correspond to the entry at the front of the complete-split pipeline.

However, when errors occur, the complete-split transaction that the high-speed handler receives might not match the entry at the front of the complete-split pipeline. This can happen for example, when a start-split is damaged on the high-speed bus and the high-speed handler doesn't receive it successfully. Or the high-speed handler might have a match, but the matching entry is located after the state for other expected complete-splits that the high-speed handler didn't receive (due to complete-split errors on the high-speed bus).

The high-speed handler is required to respond to a complete-split transaction with the results of a full/low-speed transaction that it has completed. This means that the high-speed handler must search to find the correct state tracking entry among several possible complete-split response entries. This searching takes time.

The split-transaction protocol is defined to allow the high-speed handler to timeout the first high-speed complete-split transaction while it is searching for the correct response. This allows the high-speed handler time to complete its search and respond correctly to the next (retried) complete-split.

The following interrupt and isochronous flow sequence figures show these cases with the transitions labeled "Search not complete in time" and "No split response found".

11.19 Approximate TT Buffer Space Required

A transaction translator requires buffer and state tracking space for its periodic and non-periodic portions.

The periodic TT pipeline requires less than:

- 752 bytes for the start-split stage
- 2x 188 bytes for the complete-split stage
- 16x 4x transaction status (<4 bytes for each transaction) for start-split stage
- 16x 2x transaction status (<4 bytes for each transaction) for complete-split stage

There are at most 4 microframes of buffering required for the start-split stage of the pipeline. At most 2 microframes of buffering for the complete-split stage of the pipeline. There are at most 16 full-speed (minimum sized) transactions possible in any microframe.

The non-periodic portion of the TT requires at least:

- 2x (64 data + 4 transaction status) bytes

Different implementations may require more or less buffering and state tracking space.

11.20 Interrupt Transaction Translation Overview

The flow sequence and state machine figures show the transitions required for high-speed split transactions for full/low-speed interrupt transfer types for a single endpoint. These figures must not be interpreted as showing any particular specific timing. In particular, other high-speed or full/low-speed transactions may be "interleaved" with these transaction sequences. Specific details are described as appropriate.

In contrast to bulk/control processing, the full/low-speed handler must not do local retry processing on the full/low-speed bus in response to a timeout for full/low-speed interrupt transactions.

11.20.1.1 Interrupt Split Transaction Sequences

The interrupt IN and OUT flow sequence figures use the same notation and have descriptions similar to the bulk/control figures.

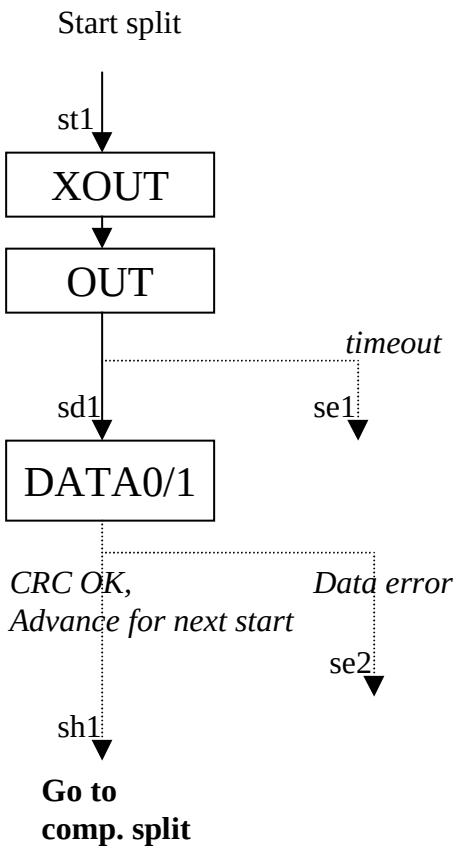

Figure 11-36 Interrupt OUT Start-Split Transaction Sequence

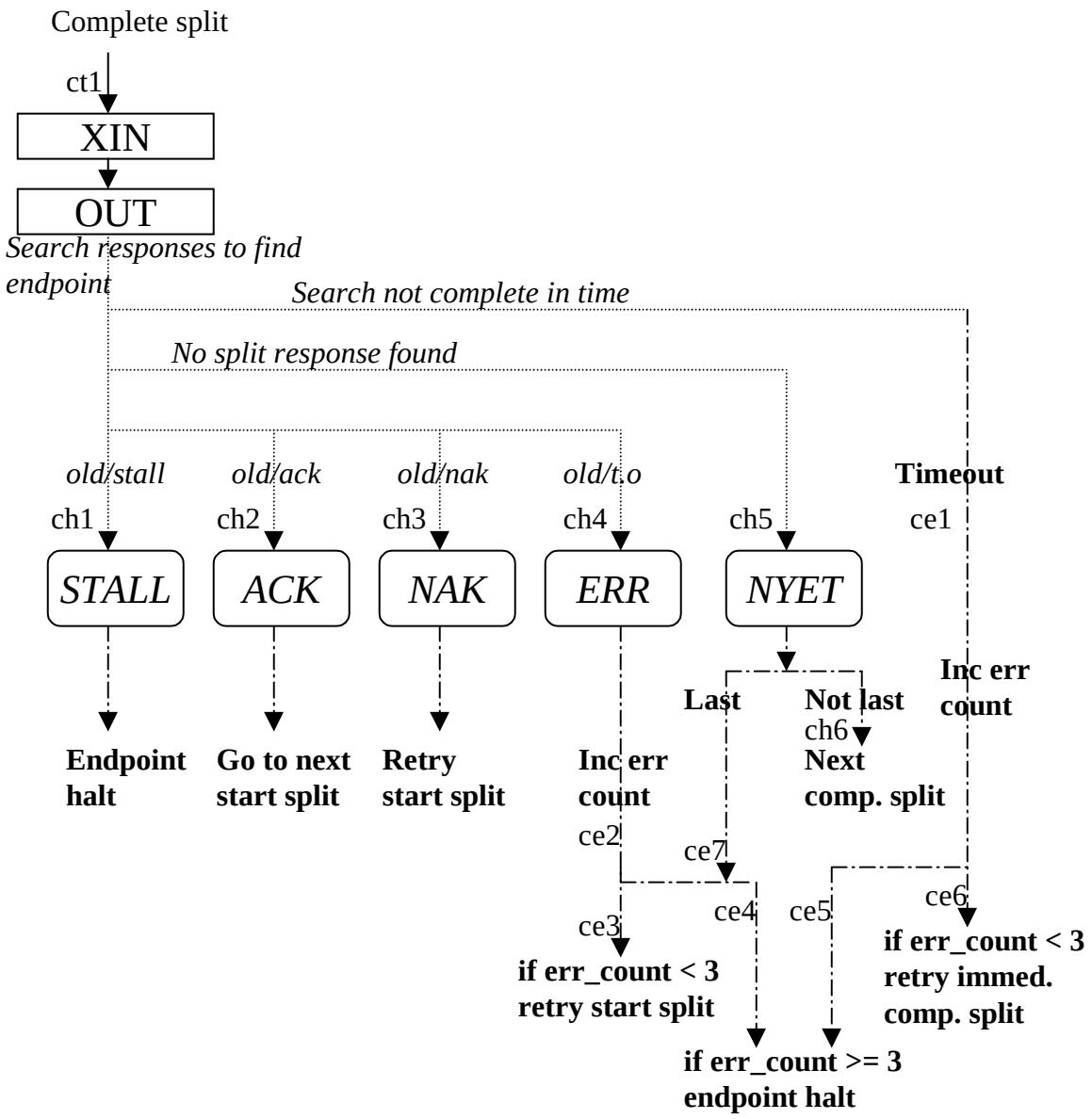

Figure 11-37 Interrupt OUT Complete-Split Transaction Sequence

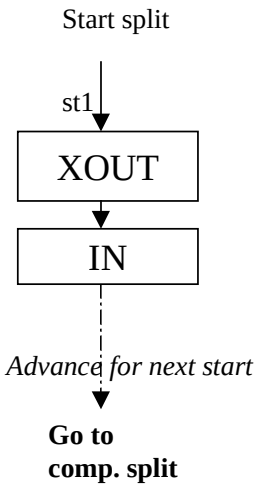

Figure 11-38 Interrupt IN Start-Split Transaction Sequence

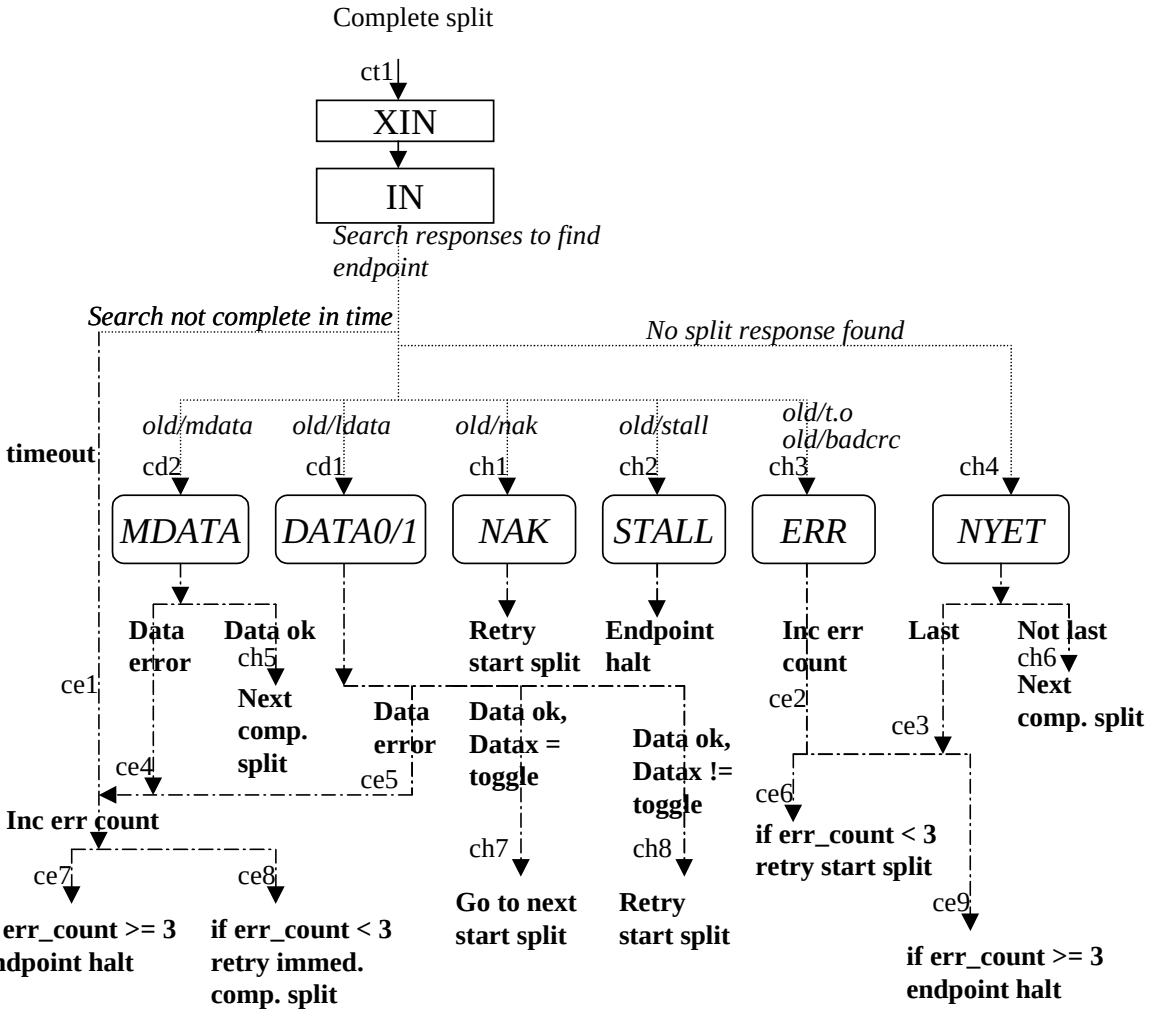

Figure 11-39 Interrupt IN Complete-Split Transaction Sequence

11.20.1.2 Interrupt Split Transaction State Machines

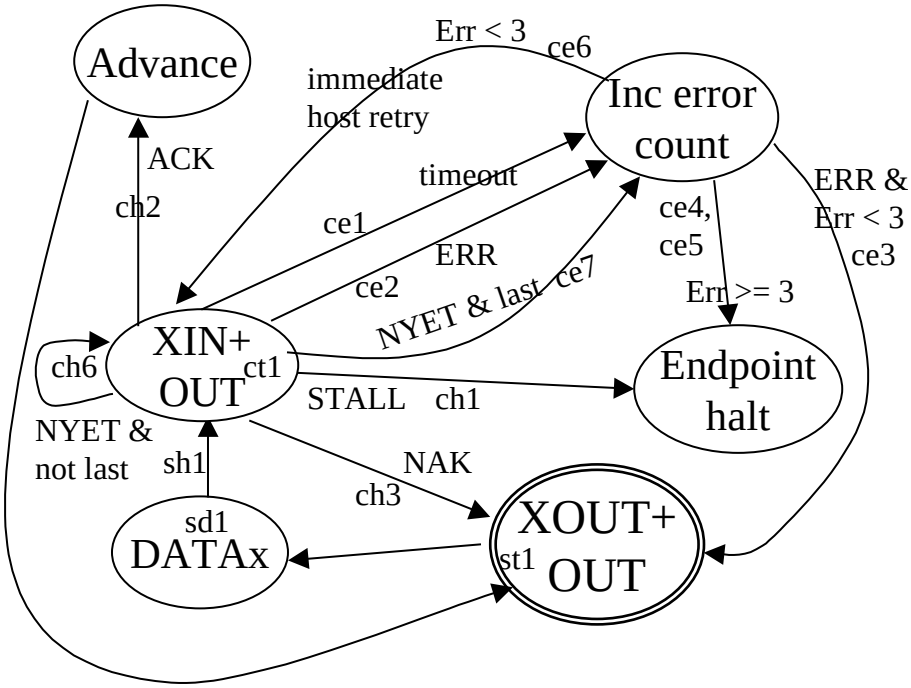

Figure 11-40 Interrupt OUT Split Transaction Host State Machine

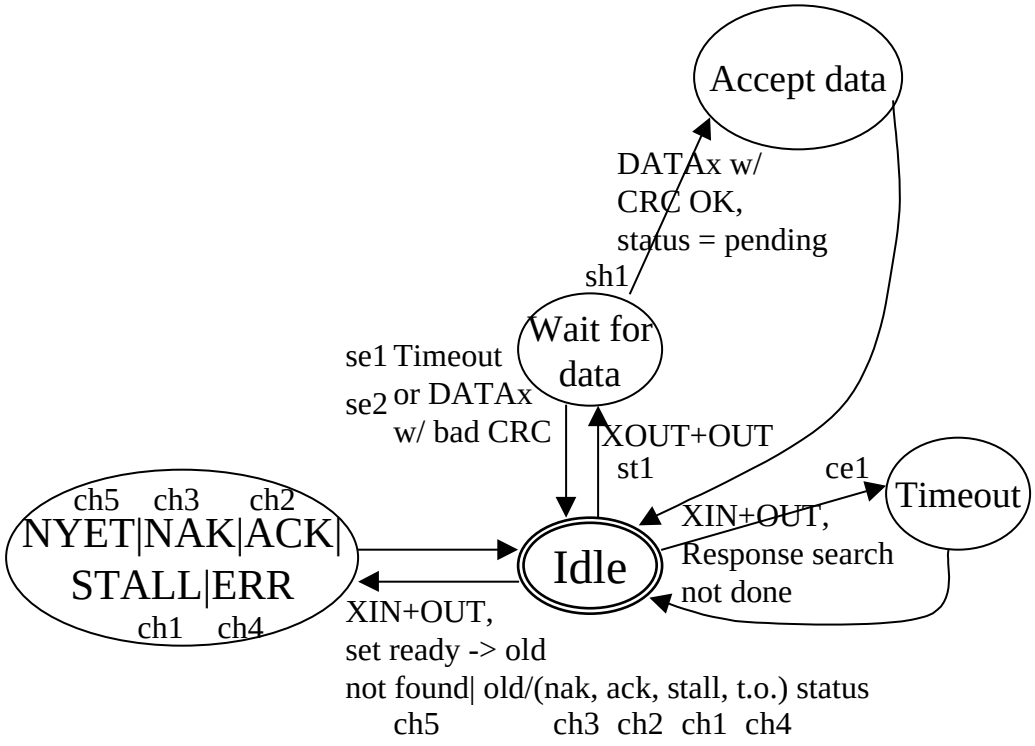

Figure 11-41 Interrupt OUT Split transaction TT State Machine

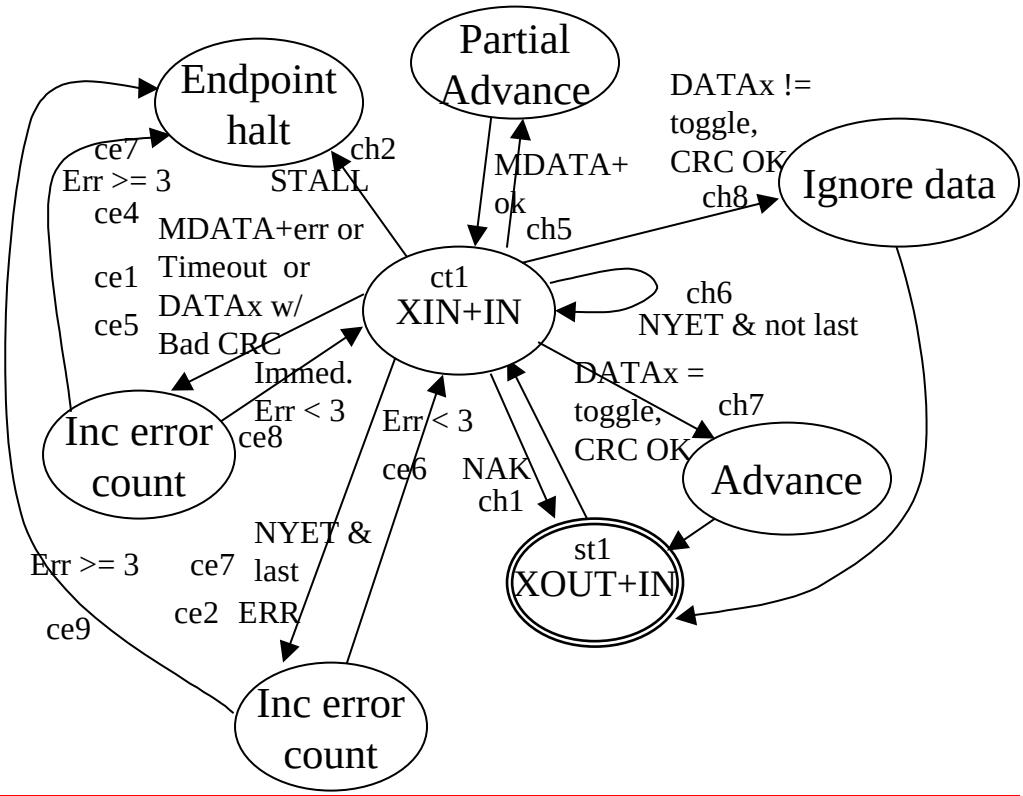

Figure 11-42 Interrupt IN Split Transaction Host State Machine

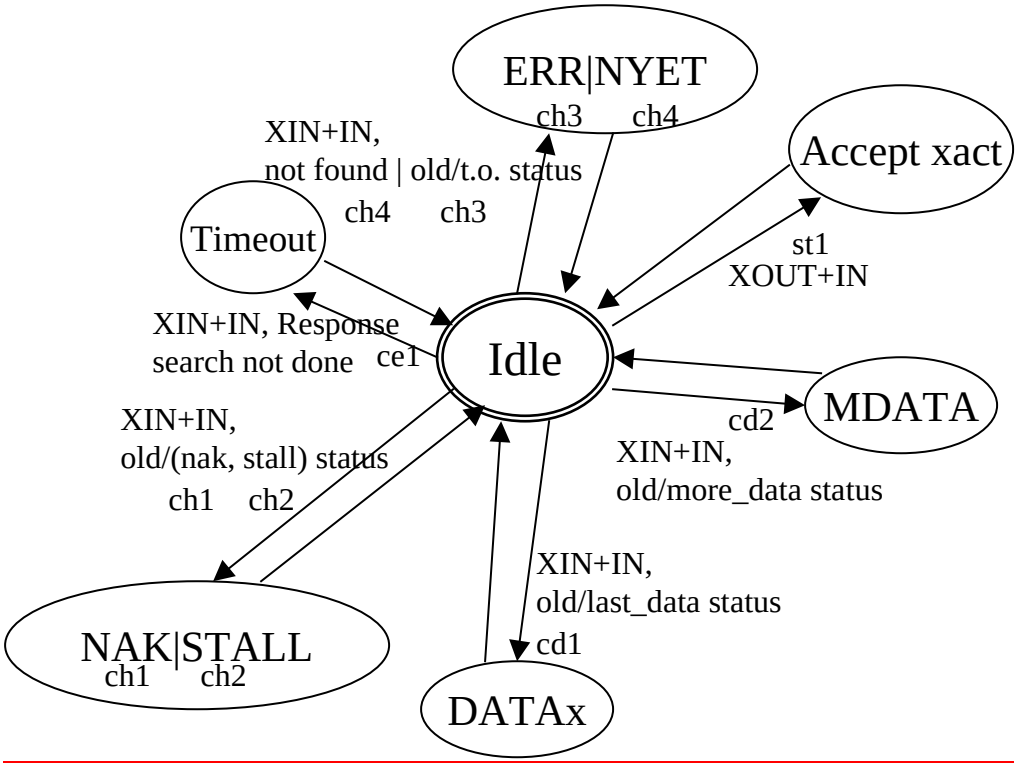

Figure 11-43 Interrupt IN Split Transaction TT State Machine

11.20.2 Interrupt OUT Sequencing

Interrupt OUT split transactions are scheduled by the host controller as normal high-speed transactions with the start- and complete- splits scheduled as described previously.

When there are several full/low-speed transactions allocated for a given microframe, they are saved by the high-speed handler in order in the start-split pipeline stage until the end of the microframe. At the end of the microframe, these transactions are ready to be issued by the full/low-speed handler on the full/low-speed bus in the order they were received.

In a following microframe (as described previously), the full/low-speed handler issues the transactions, that had been saved in the start-split pipeline stage, on the downstream facing full/low-speed bus. Some transactions could be leftover from an older microframe since the high-speed schedule was built assuming best case bit stuffing and the full/low-speed transactions could be operating according to the worst case budget. As the full/low-speed handler issues downstream transactions, it saves the results in the periodic complete-split pipeline stage and advances to the next transaction in the start-split pipeline.

In a following microframe (as described previously), the host controller issues its high-speed complete-split transaction. If the transaction status in the complete-split pipeline indicates that this endpoint has a completed status, the high-speed handler responds with the indicated status.

If the full/low-speed bus is running according to the worst case budget, the high-speed handler can receive the complete-split before the full-speed handler has started the transaction on the downstream facing bus. In this case, the high-speed handler responds with a NYET handshake to indicate that there is no information ready yet for this complete-split.

If the high-speed handler receives the complete-split and it has no status matching for the endpoint indicated by the complete split, it also responds with a NYET handshake. When the host issues the last scheduled complete-split for this endpoint for this frame, it must interpret the NYET as an error condition. This stimulates the normal “three strikes” error handling. If there have been more than three errors, the host halts this endpoint. If there have been less than three errors, the host continues processing the scheduled transactions of this endpoint (e.g. a start-split will be issued as the next transaction for this endpoint at the next scheduled time for this endpoint).

The start-split transaction for an interrupt OUT transaction must not include the CRC16 field for the full speed data packet, e.g. there is only a single CRC16 field in the start-split transaction. The TT high-speed handler must check the CRC on the start-split and ignore the start-split if there is a failure in the CRC check. The TT full-speed handler must locally generate the CRC16 value for the full-speed data packet. If the first start-split has a CRC check failure, the full-speed transaction must not be started on the downstream bus.

<<figures for above>>

<<<words about other and error cases??>>>

11.20.3 Interrupt IN Sequencing

When the high-speed handler receives an interrupt start-split transaction, it saves the packet in the start-split pipeline stage. In this fashion, it accumulates some number of start-split transactions for a following microframe.

At the beginning of a following microframe (as described previously), these transactions are ready to be issued by the full/low-speed handler on the downstream full/low-speed bus in the order they were received. The full/low-speed handler saves the results in the complete-split pipeline stage. The full/low-speed handler responds to the full/low-speed transaction with an appropriate handshake.

During a following microframe, the host controller issues a high-speed complete-split transaction to retrieve the data/handshake from the high-speed handler. The host may schedule one or two complete-split transactions based on the best/worst schedules. The TT will return whatever data it has received during a microframe. If the full/low-speed transaction spans a microframe, the TT will require two complete-splits

(in two subsequent microframes) to return all the data for the full/low-speed transaction. The operation of the TT for interrupt IN is similar to isochronous IN (as described above).

The high-speed handler matches the complete-split transaction with the correct entry in the complete-split pipeline stage and advances the pipeline appropriately. If the token and response match, the high-speed handler responds with data/status. If the complete-split transaction is for the next entry in the complete-split pipeline, the high-speed handler advances the complete-split pipeline (e.g. frees the current response information). The host controller is required to issue the complete-split transactions in the same order as the original start-split transactions.

If the complete-split matches the current entry in the complete-split pipeline, the high-speed handler responds with that information. This is the case when the host controller didn't get the first response to the complete-split and retries the complete-split transaction. In such a case, the host controller is required to retry immediately before proceeding to the next periodic split transaction for this endpoint.

If the first entry doesn't match, the high-speed handler also needs to check if the complete-transaction matches the other entries in the complete-split pipeline. This approach handles the case where the host controller was unsuccessful in issuing a complete-split transaction to the high-speed handler and has done endpoint halt processing for that previous endpoint. This leaves a "stale" entry in the complete-split pipeline.

The high-speed handler can also receive a complete-split before it has started a full/low-speed transaction. If there is not an entry in the complete-split pipeline, the high-speed handler responds with a NYET handshake to inform the host that it has no status information. When the host issues the last scheduled complete-split for this endpoint for this frame, it must interpret the NYET as an error condition. This stimulates the normal "three strikes" error handling. If there have been more than three errors, the host halts this endpoint. If there have been less than three errors, the host continues processing the scheduled transactions of this endpoint (e.g. a start-split will be issued as the next transaction for this endpoint at the next scheduled time for this endpoint).

The high-speed handler can timeout its first high-speed complete-split transaction while it is searching for a match. However, the high-speed handler must respond correctly to the subsequent complete-split transaction. If the high-speed handler didn't respond correctly for an interrupt IN after it had acknowledged the full/low-speed transaction, the endpoint software and the device would lose data synchronization and more catastrophic errors could occur.

The complete-split transaction for an interrupt IN transaction must not include the CRC16 field for the full speed data packet (e.g. only a high-speed CRC16 field is used in split-transactions). The TT must not pass the full-speed value received from the device and instead only use a high-speed CRC16 on the last complete-split transaction. If the full-speed handler detects a failed CRC check, it uses an ERR handshake response to reflect that error to the high-speed host controller. The host controller must check the CRC16 on each returned complete-split. A CRC failure (or ERR handshake) on any (partial) complete-split is reflected as a CRC failure on the total full-speed transaction.

<<figures for above>>

<<describe more error cases?>>

11.21 Isochronous Transaction Translation Overview

Isochronous split transactions are handled by the host by scheduling start- and complete-split transactions as described previously. Split IN transactions have two or more schedule entries. One entry for the start-split transaction in the microframe before the earliest the full-speed transaction can occur. One entry for the last complete-split in the microframe after the latest expected data that can occur on the full-speed bus (similar to interrupt IN scheduling).

Furthermore, isochronous transactions are split into microframe sized pieces, e.g. a 300 byte full-speed transaction is budgeted up to 3 high-speed split transactions to move data to/from the TT. This allows any alignment of the data on microframes.

Isochronous OUT transactions don't have complete-split transactions. They must only have start-split transaction(s).

The host controller must preserve the same order for the complete-split transactions (as for the start-split transactions) for IN handling.

Isochronous INs have start- and complete- split transactions. The "first" high-speed split transaction for a full-speed endpoint is always a start-split transaction and the second (and others as required) is always a complete-split no matter what the high-speed handler responds.

The full/low-speed handler recombines OUT data in its local buffers to recreate the single full-speed data transaction and handle the microframe error cases. The full/low-speed handler splits IN response data on microframe boundaries.

Microframe buffers always advance no matter what the interactions with the host controller or the full-speed handler.

11.21.1 Isochronous Split Transaction Sequences

The flow sequence and state machine figures show the transitions required for high-speed split transactions for full-speed isochronous transfer types for a single endpoint. These figures must not be interpreted as showing any particular specific timing. In particular, other high-speed or full-speed transactions may be "interleaved" with these transaction sequences. Specific details are described as appropriate.

In contrast to bulk/control processing, the full-speed handler must not do local retry processing on the full-speed bus in response to a timeout for isochronous transactions.

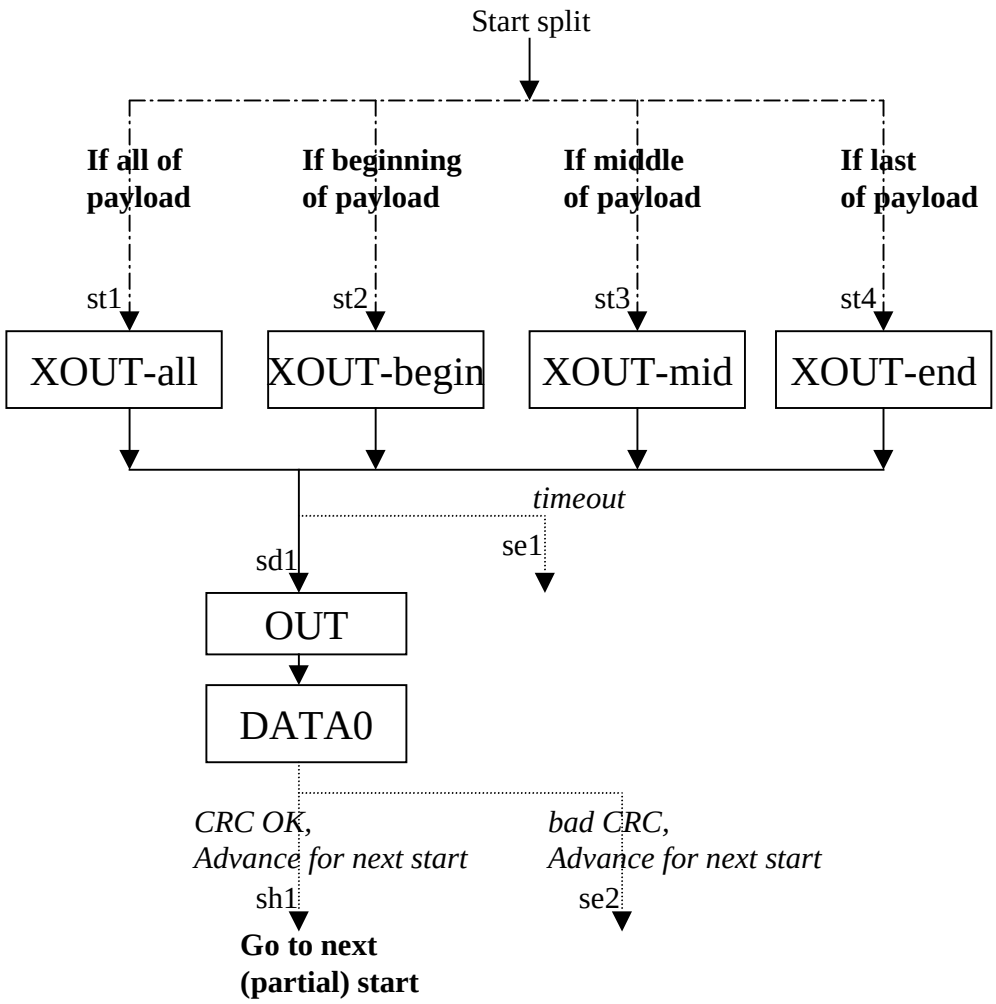

Figure 11-44 Isochronous OUT Start-Split Transaction Sequence

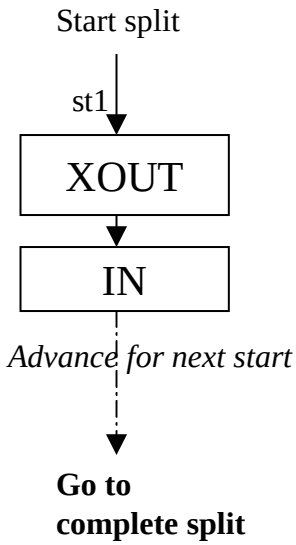

Figure 11-45 Isochronous IN Start-Split Transaction Sequence

In Figure 11-46, the high-speed handler returns an ERR handshake for a “transaction timeout” or CRC check error for the full-speed transaction.

The high-speed handler returns a NYET handshake when it can’t find a matching entry in the complete-split pipeline stage. This handles the case where the host controller issued the first high-speed complete-split transaction, but the full/low-speed handler had not started the transaction yet or has not yet received data back from the full-speed device. This can be due to a delay from bit stuffing for previous full-speed transactions.

The transition labeled “TAdvance" indicates that the host advances to the next transaction for this full-speed endpoint.

The transition labeled “DAdvance" indicates that the host advances to the next data area of the current transaction for the current full-speed endpoint.

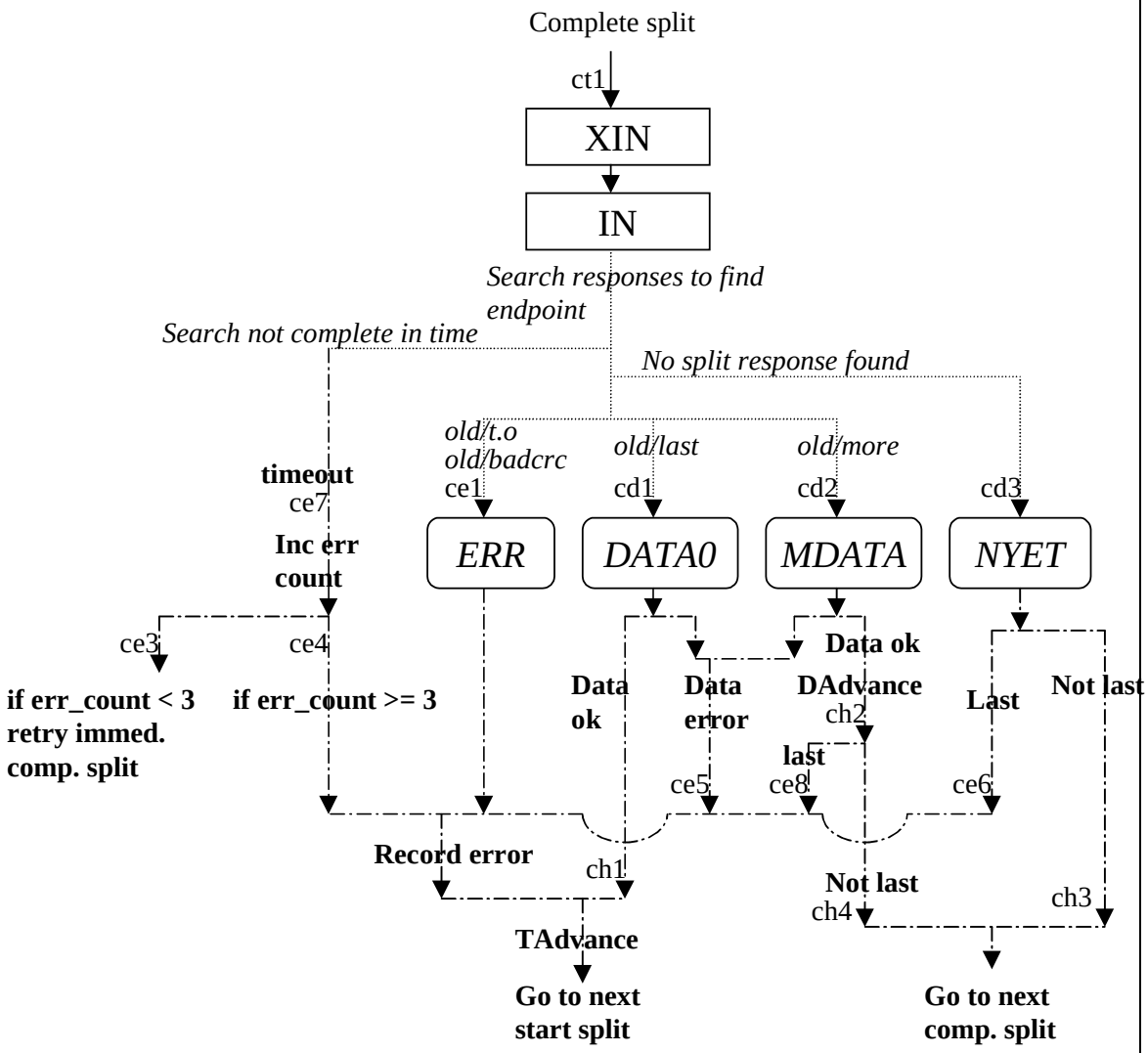

Figure 11-46 Isochronous IN Complete-Split Transaction Sequence

11.21.2 Isochronous Split Transaction State Machines

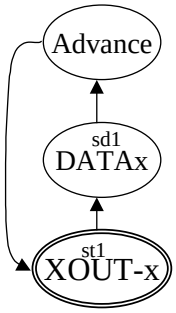

Figure 11-47 Isochronous OUT Split Transaction Host State Machine

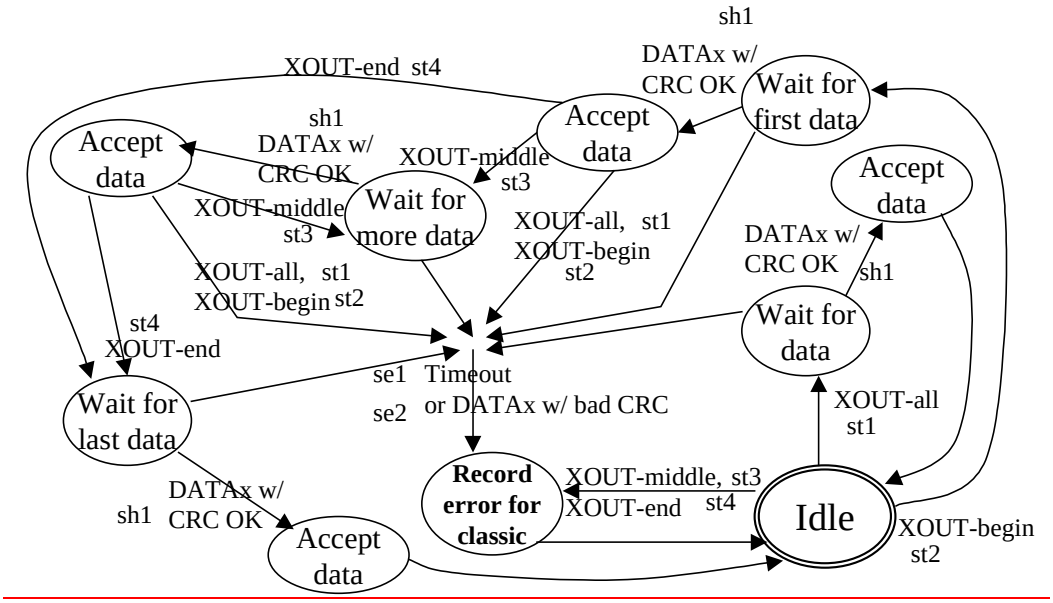

Figure 11-48 Isochronous OUT Split Transaction TT State Machine

<<more words…>>

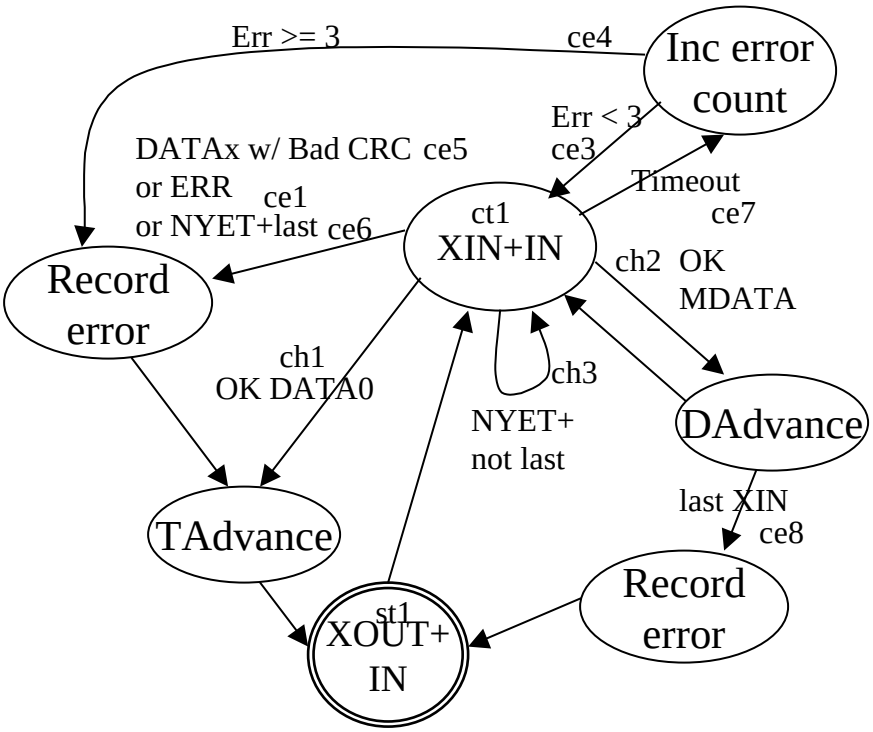

Figure 11-49 Isochronous IN Split Transaction Host State Machine

In Figure 11-49, the transition "No more scheduled XINs" should occur when the high-speed handler responds with a DATA0 to indicate this is the last complete-split data for the full-speed transaction. If a DATA0 response from the high-speed handler is not received, this indicates an error and must be so recorded by the host controller. <<Add to flow sequence?>>

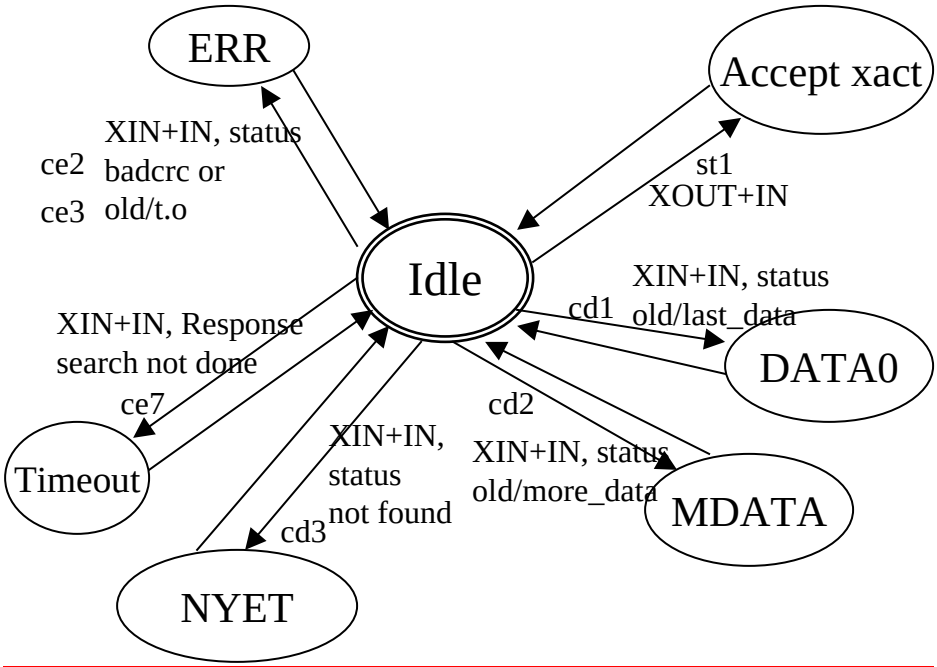

Figure 11-50 Isochronous IN Split Transaction TT State Machine

<<<more words>>>

11.21.3 Isochronous OUT Sequencing

The host controller and TT must ensure that errors that can occur in split transactions of an isochronous full-speed transaction translate into a detectable error. For isochronous OUT split transactions, once the high-speed handler has received an "XOUT-begin"extended token packet, the high-speed handler must ensure that a start-split transaction is received each and every microframe until the end of the full-speed data payload. The host indicates the end of the long isochronous OUT split transaction sequence by sending an XOUT-end extended token packet. If a microframe passes without the high-speed handler receiving a start-split for this full-speed endpoint, it must ensure that the full-speed handler will force a CRC/bitstuff error on the full-speed transaction.

The start-split transaction for an isochronous OUT transaction must not include the CRC16 field for the full speed data packet. The TT high-speed handler must check the CRC on the start-split and indicate to the full-speed handler if there is a failure in the CRC check. Additional start-split transactions for the same endpoint in the same frame must be ignored after a CRC check fails. The TT full-speed handler must locally generate the CRC16 value for the full-speed data packet. If the transaction has been indicated as having a CRC failure, the full-speed handler uses the defined mechanism for forcing a downstream corrupted packet. If the first start-split has a CRC check failure, the full-speed transaction must not be started on the downstream bus.

<<figures for above>>

<<words>>

11.21.4 Isochronous IN Sequencing

The complete-split transaction for an isochronous IN transaction must not include the CRC16 field for the full speed data packet (e.g. only a high-speed CRC16 field is used in split-transactions). The TT must not pass the full-speed value received from the device and instead only use a high-speed CRC16 on the last complete-split transaction. If the full-speed handler detects a failed CRC check, it uses an ERR handshake response to reflect that error to the high-speed host controller. The host controller must check the CRC16 on each returned complete-split. A CRC failure (or ERR handshake) on any (partial) complete-split is reflected by the host controller as a CRC failure on the total full-speed transaction.

<<figures for above>>

<<words>>

11.22 TT Error Handling

The TT has the same requirements for handling errors as a host controller or hub. In particular:

- If the TT is receiving a packet at EOF of the downstream facing bus, it must disable the downstream facing port that is currently transmitting.
- If the TT is transmitting a packet at EOF of the downstream facing bus, it must force a CRC/bitstuff error and stop transmitting.
- If the TT is going to transmit a non-periodic full/low-speed transaction, it must determine that there is sufficient time remaining before EOF to complete the transaction. This determination is based on normal sequencing of the packets in the transaction. Since the TT has no information about data payload size for INs, it must use the maximum allowed size in its determination. Periodic transactions don't' need to be included in this test since the periodic pipeline is maintained separately.

11.23 Descriptors

Hub descriptors are derived from the general USB device framework. Hub descriptors define a hub device and the ports on that hub. The host accesses hub descriptors through the hub's default pipe.

The USB specification (refer to Chapter 9) defines the following descriptors:

- Device
- Configuration
- Interface
- Endpoint
- String (optional).

The hub class defines additional descriptors (refer to Section 11.23.2). In addition, vendor-specific descriptors are allowed in the USB device framework. Hubs support standard USB device commands as defined in Chapter 9.

11.23.1 Standard Descriptors

The hub class pre-defines certain fields in standard USB descriptors. Other fields are either implementation-dependent or not applicable to this class.

Note: for the descriptors and fields shown below, the bits in a field are organized in a little-endian fashion; that is, bit location 0 is the least significant bit and bit location 7 is the most significant bit of a byte value.

Device Descriptor

bDeviceClass	=	HUB_CLASSCODE (09H)
bDeviceSubClass	=	0

Interface Descriptor

bNumEndpoints	=	1
bInterfaceClass	=	HUB_CLASSCODE (09H)
bInterfaceSubClass	=	0
bInterfaceProtocol	=	0

Configuration Descriptor

MaxPower	=	The maximum amount of bus power the hub will consume in this configuration

Endpoint Descriptor (for Status Change Endpoint)

bEndpointAddress	=	Implementation-dependent; Bit 7: Direction = In(1)
wMaxPacketSize	=	Implementation-dependent
bmAttributes	=	Transfer Type = Interrupt (00000111B)
bInterval	=	FFH (Maximum allowable interval)

The hub class driver retrieves a device configuration from the USB System Software using the GetDescriptor() device request. The only endpoint descriptor that is returned by the GetDescriptor() request is the Status Change endpoint descriptor.

11.23.2 Class-specific Descriptors

11.23.2.1 Hub Descriptor

Table 11-8 outlines the various fields contained by the hub descriptor.

Table 11-8. Hub Descriptor

Offset	Field	Size	Description
0	*bDescLength*	1	Number of bytes in this descriptor, including this byte
1	*bDescriptorType*	1	Descriptor Type, value: 29H for hub descriptor
2	*bNbrPorts*	1	Number of downstream ports that this hub supports
~~3~~	~~*wHubCharacteristics*~~	~~__~~	~~D1...D0: Logical Power Switching Mode~~ ~~00: Ganged power switching (all ports' power at once)~~ ~~01: Individual port power switching~~ ~~1X: Reserved. Used only on 1.0 compliant hubs that implement no power switching.~~ ~~D2: Identifies a Compound Device~~ ~~0: Hub is not part of a compound device~~ ~~1: Hub is part of a compound device~~ ~~D4...D3: Over-current Protection Mode~~ ~~00: Global Over-current Protection. The hub reports over-current as a summation of all ports' current draw, without a breakdown of individual port over-current status.~~ ~~01: Individual Port Over-current Protection. The hub reports over-current on a per-port basis. Each port has an over-current indicator.~~ ~~1X: No Over-current Protection. This option is allowed only for bus-powered hubs that do not implement over-current protection.~~ ~~D15...D5: Reserved~~

Offset	Field	Size	Description
3	*wHubCharacteristics*	___	D1...D0: Logical Power Switching Mode 00: Ganged power switching (all ports' power at once) 01: Individual port power switching 1X: Reserved. Used only on 1.0 compliant hubs that implement no power switching. D2: Identifies a Compound Device 0: Hub is not part of a compound device 1: Hub is part of a compound device D4...D3: Over-current Protection Mode 00: Global Over-current Protection. The hub reports over-current as a summation of all ports' current draw, without a breakdown of individual port over-current status. 01: Individual Port Over-current Protection. The hub reports over-current on a per-port basis. Each port has an over-current indicator. 1X: No Over-current Protection. This option is allowed only for bus-powered hubs that do not implement over-current protection. D5: Transaction Translator(TT) Organization 0: Hub has one TT for all downstream facing ports 1: Hub has one TT per downstream facing port D7...D6: TT Think TIme 00: TT requires at most 8 FS bit times of inter transaction gap on a full/low-speed downstream bus 01: TT requires at most 16 FS bit times 10: TT requires at most 24 FS bit times 11: TT requires at most 32 FS bit times D15...D8: Reserved
5	*bPwrOn2PwrGood*	1	Time (in 2ms intervals) from the time the power-on sequence begins on a port until power is good on that port. The USB System Software uses this value to determine how long to wait before accessing a powered-on port.
6	*bHubContrCurrent*	1	Maximum current requirements of the Hub Controller electronics in mA.

Table 11-8. Hub Descriptor (Continued)

Offset	Field	Size	Description
7	*DeviceRemovable*	Variable, depending on number of ports on hub	Indicates if a port has a removable device attached. This field is reported on byte-granularity. Within a byte, if no port exists for a given location, the field representing the port characteristics returns 0. Bit value definition: 0B - Device is removable 1B - Device is non-removable This is a bitmap corresponding to the individual ports on the hub: Bit 0: Reserved for future use Bit 1: Port 1 Bit 2: Port 2 .... Bit *n*: Port *n* (implementation-dependent, up to a maximum of 255 ports).
Variable	*PortPwrCtrlMask*	Variable, depending on number of ports on hub	This field exists for reasons of compatibility with software written for 1.0 compliant devices. All bits in this field should be set to 1B. This field has one bit for each port on the hub with additional pad bits, if necessary, to make the number of bits in the field an integer multiple of 8.

11.24 Requests

11.24.1 Standard Requests

Hubs have tighter constraints on request processing timing than specified in Section 9.2.6 for standard devices because they are crucial to the 'time to availability' of all devices attached to USB. The worst case request timing requirements are listed below (apply to both Standard and Hub Class requests):.

1. Completion time for requests with no data stage: 50 ms
2. Completion times for standard requests with data stage(s)
   - Time from setup packet to first data stage: 50 ms
   - Time between each subsequent data stage: 50 ms
   - Time between last data stage and status stage: 50 ms

As hubs play such a crucial role in bus enumeration, it is recommended that hubs average response times be less than 5ms for all requests.

Table 11-9 outlines the various standard device requests.

Table 11-9. Hub Responses to Standard Device Requests

bRequest	Hub Response
CLEAR_FEATURE	Standard
GET_CONFIGURATION	Standard
GET_DESCRIPTOR	Standard
GET_INTERFACE	Undefined. Hubs are allowed to support only one interface
GET_STATUS	Standard
SET_ADDRESS	Standard
SET_CONFIGURATION	Standard
SET_DESCRIPTOR	Optional
SET_FEATURE	Standard
SET_INTERFACE	Undefined. Hubs are allowed to support only one interface
SYNCH_FRAME	Undefined. Hubs are not allowed to have isochronous endpoints

Optional requests that are not implemented shall return a STALL in the Data stage or Status stage of the request.

11.24.2 Class-specific Requests

The hub class defines requests to which hubs respond, as outlined in Table 11-10. Table 11-11 defines the hub class request codes. All requests in the table below except for GetBusState() and SetHubDescriptor() are mandatory.

Table 11-10. Hub Class Requests

Request	bmRequestType	bRequest	wValue	wIndex	wLength	Data
ClearHubFeature	00100000B	CLEAR_ FEATURE	Feature Selector	Zero	Zero	None
ClearPortFeature	00100011B	CLEAR_ FEATURE	Feature Selector	Port	Zero	None
ClearTTBuffer	00100011B	CLEAR_TT_BUFFER	Dev_Addr, EP_Num	TT_id	Zero	None
GetBusState	10100011B	GET_ STATE	Zero	Port	One	Per-Port Bus State
GetHubDescriptor	10100000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
GetHubStatus	10100000B	GET_ STATUS	Zero	Zero	Four	Hub Status and Change Indicators
GetPortStatus	10100011B	GET_ STATUS	Zero	Port	Four	Port Status and Change Indicators
SetHubDescriptor	00100000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
SetHubFeature	00100000B	SET_ FEATURE	Feature Selector	Zero	Zero	None
SetPortFeature	00100011B	SET_ FEATURE	Feature Selector	Port	Zero	None

Table 11-11. Hub Class Request Codes

bRequest	Value
GET_ STATUS	0
CLEAR_ FEATURE	1
GET_STATE	2
SET_ FEATURE	3
Reserved for future use	*4-5*
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7

Table 11-12 gives the valid feature selectors for the hub class. See Section 11.24.2.6 and Section 11.24.2.7 for a description of the features.

Table 11-12. Hub Class Feature Selectors

	Recipient	Value
C_HUB_LOCAL_POWER	Hub	0
C_HUB_OVER_CURRENT	Hub	1
PORT_CONNECTION	Port	0
PORT_ENABLE	Port	1
PORT_SUSPEND	Port	2
PORT_OVER_CURRENT	Port	3
PORT_RESET	Port	4
PORT_POWER	Port	8
PORT_LOW_SPEED	Port	9
C_PORT_CONNECTION	Port	16
C_PORT_ENABLE	Port	17
C_PORT_SUSPEND	Port	18
C_PORT_OVER_CURRENT	Port	19
C_PORT_RESET	Port	20
PORT_TEST	Port	21

11.24.2.1 Clear Hub Feature

This request resets a value reported in the hub status.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100000B	CLEAR_ FEATURE	Feature Selector	Zero	Zero	None

Clearing a feature disables that feature; refer to Table 11-12 for the feature selector definitions that apply to the hub as a recipient. If the feature selector is associated with a change indicator, clearing that indicator acknowledges the change. This request format is used to clear either the C_HUB_LOCAL_POWER or C_HUB_OVER_CURRENT features.

It is a Request Error if *wValue* is not a feature selector listed in Table 11-12 or if *wIndex* or *wLength* are not as specified above.

If the hub is not configured, the hub's response to this request is undefined.

11.24.2.2 Clear Port Feature

This request resets a value reported in the port status.

bmRequestType	Brequest	wValue	wIndex	wLength	Data
00100011B	CLEAR_ FEATURE	Feature Selector	Port	Zero	None

The port number must be a valid port number for that hub, greater than zero.

Clearing a feature disables that feature or starts a process associated with the feature; refer to Table 11-12 for the feature selector definitions. If the feature selector is associated with a change indicator, clearing that indicator acknowledges the change. This request format is used to clear the following features:

- PORT_ENABLE
- PORT_SUSPEND
- PORT_POWER
- PORT_TEST
- C_PORT_CONNECTION
- C_PORT_RESET
- C_PORT_ENABLE
- C_PORT_SUSPEND
- C_PORT_OVER_CURRENT.

Clearing the PORT_SUSPEND feature causes a host-initiated resume on the specified port. If the port is not in the Suspended state, the hub should treat this request as a functional no-operation.

Clearing the PORT_ENABLE feature causes the port to beplaced in the Disabled state. If the port is in the Powered-off state, the hub should treat this request as a functional no-operation.

Clearing the PORT_POWER feature causes the port to be placed in the Powered-off state and may, subject to the constraints due to the hub’s method of power switching, result in power being removed from the port.

Refer to Section 11.11 on rules for how this request is used with ports that are gang-powered.

Clearing the PORT_TEST port feature causes the port to be placed in the Disabled state. If the port is not in the test mode, the hub should treat this request as a functional no-operation.

It is a Request Error if *wValue* is not a feature selector listed in Table 11-12, if *wIndex* specifies a port that doesn't exist, or if *wLength* is not as specified above. It is not an error for this request to try to clear a feature that is already cleared (hub should treat as a function no-operation).

If the hub is not configured, the hub's response to this request is undefined.

11.24.2.3 Clear TT Buffer

This request clears the state of a Transaction Translator(TT) bulk/control buffer after it has been left in a busy state due to high-speed errors. After successful completion of this request, the buffer can again be used by the TT with high-speed split-transactions for full/low- speed transactions to attached full/low-speed devices.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100011B	CLEAR_TT_BUFFER	Device_Address, Endpoint_Number	TT_id	Zero	None

If the hub provides TTs per port, then wIndex must specify the number of the TT that encountered the high-speed errors (e.g. with the busy TT buffer). If the hub provides only a single TT, then wIndex must be set to zero.

The device_address and endpoint_number of the full/low-speed endpoint that may have a busy TT buffer must be specified in the wValue field. The specific bit fields used are shown in Table 11-13.

Table 11-13. wIndex Field for Clear_TT_Buffer

Bits	Field
3..0	Endpoint Number
10..4	Device Address
14..11	Reserved, must be zero
15	Direction, 1 = IN, 0 = OUT.

11.24.2.4 Get Bus State

This is an optional per-port diagnostic request that returns the bus state value, as sampled at the last EOF2 point.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100011B	GET_ STATE	Zero	Port	One	Per-Port Bus State

The port number must be a valid port number for that hub, greater than zero. If an invalid port number is specified or if *wValue* or *wLength* are not as specified above, then the hub shall return a STALL in the Data stage of the request (aborting the Status stage).

Hubs may implement an optional diagnostic aid to facilitate system debug. Hubs implement this aid through this optional request. This diagnostic feature provides a glimpse of the USB bus state as sampled at the last EOF2 sample point.

Hubs that implement this diagnostic feature should store the bus state at each EOF2 state in preparation for a potential request in the following USB frame.

The data returned is bitmapped in the following manner:

Bit 0: The value of the D- signal

Bit 1: The value of the D+ signal

Bits 2-7: Reserved for future use and are reset to zero.

The hub must be able to return the bus state in the Data stage transaction within the frame in which the request was received. If the hub does not receive ACK for the data packet, the device is not required to return the same data packet if the host continues with the Data stage. Rather, the hub will always return the bus state at the immediately prior EOF2 sample point along with the DATA0 PID.

Hubs that do not implement this request shall return a STALL in the Data stage of the request (aborting the Status stage).

If the hub is not configured, the hub's response to this request is undefined.

11.24.2.5 Get Hub Descriptor

This request returns the hub descriptor.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero	Descriptor Length	Descriptor

The GetDescriptor() request for the hub class descriptor follows the same usage model as that of the standard GetDescriptor() request (refer to Chapter 9). The standard hub descriptor is denoted by using the value *bDescriptorType* defined in Section 11.23.2.1. All hubs are required to implement one hub descriptor, with descriptor index zero.

If *wLength* is larger than the actual length of the descriptor, then only the actual length is returned. If *wLength* is less than the actual length of the descriptor, then only the first *wLength* bytes of the descriptor are returned; this is not considered an error even if *wLength* is zero.

It is a Request Error if *wValue* or *wIndex* are other than as specified above.

If the hub is not configured, the hub's response to this request is undefined.

11.24.2.6 Get Hub Status

This request returns the current hub status and the states that have changed since the previous acknowledgment.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100000B	GET_ STATUS	Zero	Zero	Four	Hub Status and Change Indicators

The first word of data contains *wHubStatus* (refer to Table 11-14). The second word of data contains *wHubChange* (refer to Table 11-15).

It is a Request Error if *wValue*, *wIndex*, or *wLength* are other than as specified above.

If the hub is not configured, the hub's response to this request is undefined.

Table 11-14. Hub Status Field, *wHubStatus*

Bit	Description
0	**Local Power Source:** This is the source of the local power supply. This field indicates whether hub power (for other than the SIE) is being provided by an external source or from the USB. . This field allows the USB System Software to determine the amount of power available from a hub to downstream devices. 0 = Local power supply good 1 = Local power supply lost (inactive)
1	**Over-current Indicator:** If the hub supports over-current reporting on a hub basis, this field indicates that the sum of all the ports' current has exceeded the specified maximum and all ports have been places in the Powered-off state. If the hub reports over-current on a per-port basis or has no over-current detection capabilities, this field is always zero. For more details on over-current protection, see Section 7.2.1.2.1. 0 = No over-current condition currently exists 1 = A hub over-current condition exists
2-15	**Reserved** These bits return 0 when read.

There are no defined feature selector values for these status bits and they can neither be set nor cleared by the USB System Software.

Table 11-15. Hub Change Field, *wHubChange*

Bit	Description
0	**Local Power Status Change:** (C_HUB_LOCAL_POWER) This field indicates that a change has occurred in the hub's Local Power Source field in *wHubStatus*. This field is initialized to zero when the hub receives a bus reset. 0 = No change has occurred to Local Power Status 1 = Local Power Status has changed
1	**Over-Current Indicator Change**: (C_HUB_OVER_CURRENT) This field indicates if a change has occurred in the Over-Current field in *wHubStatus*. This field is initialized to zero when the hub receives a bus reset. 0 = No change has occurred to the Over-Current Indicator 1 = Over-Current Indicator has changed
2-15	**Reserved** These bits return 0 when read.

Hubs may allow setting of these change bits with SetHubFeature() requests in order to support diagnostics. If the hub does not support setting of these bits, it should either treat the SetHubFeature() request as a Request Error or as a functional no-operation. When set, these bits may be cleared by a ClearHubFeature() request. A request to set a feature that is already set or to clear a feature that is already clear has no effect and the hub will not fail the request.

11.24.2.7 Get Port Status

This request returns the current port status and the current value of the port status change bits.

bmRequestType	BRequest	wValue	wIndex	wLength	Data
10100011B	GET_STATUS	Zero	Port	Four	Port Status and Change Indicators

The port number must be a valid port number for that hub, greater than zero.

The first word of data contains *wPortStatus* (refer to Table 11-16). The second word of data contains *wPortChange* (refer to Table 11-15).

The bit locations in the *wPortStatus* and *wPortChange* fields correspond in a one-to-one fashion where applicable.

It is a Request Error if *wValue* or *wLength* are other than as specified above or if *wIndex* specifies a port that does not exist.

If the hub is not configured, the behavior of the hub in response to this request is undefined.

11.24.2.7.1 Port Status Bits

Table 11-16. Port Status Field, *wPortStatus*

Bit	Description
0	**Current Connect Status:** (PORT_CONNECTION) This field reflects whether or not a device is currently connected to this port. 0 = No device is present 1 = A device is present on this port
1	**Port Enabled/Disabled:** (PORT_ENABLE) Ports can be enabled by the USB System Software only. Ports can be disabled by either a fault condition (disconnect event or other fault condition) or by the USB System Software. 0 = Port is disabled 1 = Port is enabled
2	**Suspend:** (PORT_SUSPEND) This field indicates whether or not the device on this port is suspended. Setting this field causes the device to suspend by not propagating bus traffic downstreamThis field my be reset by a request or by resume signaling from the device attached to the port. 0 = Not suspended 1 = Suspended or resuming
3	**Over-current Indicator:** (PORT_OVER_CURRENT) If the hub reports over-current conditions on a per-port basis, this field will indicate that that the current drain on the port exceeds the specified maximum.For more details, see Section 7.2.1.2.1. 0 = All no over-current condition exists on this port 1 = An over-current condition exists on this port.
4	**Reset**: (PORT_RESET) This field is set when the host wishes to reset the attached device. It remains set until the reset signaling is turned off by the hub. 0 = Reset signaling not asserted 1 = Reset signaling asserted
5-7	**Reserved** These bits return 0 when read.
8	**Port Power:** (PORT_POWER) This field reflects a port's logical, power control state. Because hubs can implement different methods of port power switching, this field may or may not represent whether power is applied to the port. The device descriptor reports the type of power switching implemented by the hub. 0 = This port is in the Powered-off state 1 = This port is not in the Powered-off state
~~9~~	~~**Low Speed Device Attached:** (PORT_LOW_SPEED) This is relevant only if a device is attached.~~ ~~0 = Full-speed device attached to this port~~ ~~1 = Low-speed device attached to this port~~
9	**Low Speed Device Attached:** (PORT_LOW_SPEED) This is relevant only if a device is attached. 0 = Full-speed or High-speed device attached to this port (determined by bit 10) 1 = Low-speed device attached to this port
~~10-15~~	~~**Reserved**~~ ~~These bits return 0 when read.~~
10	**High Speed Device Attached:** (PORT_HIGH_SPEED) This is relevant only if a device is attached. 0 = Full-speed device attached to this port 1 = High-speed device attached to this port
11	**Port Test Mode** : (PORT_TEST) This field reflects the status of the port's test mode. Software uses the SetPortFeature() and ClearPortFeature() requests to manipulate the port test mode. 0B = This port is not in the Port Test Mode 1B = This port is in Port Test Mode.
12-15	**Reserved** These bits return 0 when read.

11.24.2.7.1.1 PORT_CONNECTION

When the Port Power bit is one, this bit indicates whether or not a device is attached. This field reads as one when a device is attached; it reads as zero when no device is attached. This bit is reset to zero when the port is in the Powered-off state or the Disconnected states. It is set to one when the port is in the Powered state, a device attach is detected (see Section 7.1.7.1) and the port transitions from the Disconnected state to the Disabled state.

SetPortFeature(PORT_CONNECTION) and ClearPortFeature(PORT_CONNECTION) requests shall not be used by the USB System Software and must be treated as no-operation requests by hubs.

11.24.2.7.1.2 PORT_ENABLE

This bit is set when the port is allowed to send or receive packet data or resume signaling.

This bit may be set only as a result of a SetPortFeature(PORT_RESET) request. When the hub exits the Resetting state or, if present, the Speed_eval state, this bit is set and bus traffic may be transmitted to the port. This bit may be cleared as the result of any of the following:

- The port being in the Powered-off state
- Receipt of a ClearPortFeature(PORT_ENABLE) request
- Port_Error detection
- Disconnect detection
- When the port enters the Resetting state as a result of receiving the SetPortFeature(PORT_RESET) request.

The hub response to a SetPortFeature(PORT_ENABLE) request is not specified. The preferred behavior is that the hub respond with a Request Error. This may not be used by the USB System Software. The ClearPortFeature(PORT_ENABLE) request is supported as specified in Section 11.5.1.4.

11.24.2.7.1.3 PORT_SUSPEND

This bit is set to one when the port is selectively suspended by the USB System Software. While this bit is set, the hub does not propagate downstream-directed traffic to this port, but the hub will respond to resume signaling from the port. This bit can be set only if the port's PORT_ENABLE bit is set and the hub receives a SetPortFeature(PORT_SUSPEND) request. This bit is cleared to zero on the transition from the SendEOP state to the Enabled state, or on the transition from the Restart_S state to the Transmit state, or on any event that causes the PORT_ENABLE bit to be cleared while the PORT_SUSPEND bit is set.

The SetPortFeature(PORT_SUSPEND) request may be issued by the USB System Software at any time but will have an effect only as specified in Section 11.5.

11.24.2.7.1.4 PORT_OVER-CURRENT

This bit is set to one while an over-current condition exists on the port. This bit is cleared when an over-current condition does not exist on the port.

If the voltage on this port is affected by an over-current condition on another port then this bit is set and remains set until the over-current condition on the affecting port is removed. When the over-current condition on the affecting port is removed, this bit is reset to zero if an over-current condition does not exist on this port.

Over-current protection is required on self-powered hubs (it is optional on bus-powered hubs) as outlined in Section 7.2.1.2.1.

The SetPortFeature(PORT_OVER_CURRENT) and ClearPortFeature(PORT_OVER_CURRENT) requests shall not be used by the USB System Software and may be treated as no-operation requests by hubs.

11.24.2.7.1.5 PORT_RESET

This bit is set while the port is in the Resetting state. A SetPortFeature(PORT_RESET) request will initiate the Resetting state if the conditions in Section 11.5.1.5 are met. This bit is set to zero while the port is in the Powered-off state.

The ClearPortFeature(PORT_RESET) request shall not be used by the USB System Software and may be treated as a no-operation request by hubs.

11.24.2.7.1.6 PORT_POWER

This bit reflects the current power state of a port. This bit is implemented on all ports whether or not actual port power switching devices are present.

While this bit is zero, the port is in the Powered-off state. Similarly, anything that causes this port to go to the Power-off state will cause this bit to be set to zero.

A SetPortFeature(PORT_POWER) will set this bit to one unless both C_HUB_LOCAL_POWER and Local Power Status (in *wHubStatus*) are set to one in which case the request is treated as a functional no-operation.

This bit may be cleared under the following curcumstances:

- Hub receives a ClearPortFeature(PORT_POWER).
- An over-current condition exists on the port.
- An over-current condition on another port causes the power on this port to be shut off.

The SetPortFeature(PORT_POWER) and ClearPortFeature(PORT_POWER) requests may be issued by the USB System Software whenever the port is not in the Not Configured state, but will have an effect only as specified in Section 11.11.

11.24.2.7.1.7 PORT_LOW_SPEED

This bit has meaning only when the PORT_ENABLE bit is set. This bit is set to one if the attached device is low-speed. If this bit is set to zero, the attached device is either full- or high-speed as determined by bit 10 (PORT_HIGH_SPEED, see below).

The SetPortFeature(PORT_LOW_SPEED) and ClearPortFeature(PORT_LOW_SPEED) requests shall not be used by the USB System Software and may be treated as no-operation requests by hubs.

11.24.2.7.1.8 PORT_HIGH_SPEED

This bit has meaning only when the PORT_ENABLE bit is set and the PORT_LOW_SPEED bit is set to zero. This bit is set to one if the attached device is high-speed. The bit is set to zero if the attached device is full-speed.

The SetPortFeature(PORT_HIGH_SPEED) and ClearPortFeature(PORT_HIGH_SPEED) requests shall not be used by the USB System Software and may be treated as no-operation requests by hubs.

11.24.2.7.1.9 PORT_TEST

When the Port Test Mode bit is set to a one (1B), the port is in test mode. The specific test mode is specified in the SetPortFeature(PORT_TEST) request by the test selector. The hub provides no standard

mechanism to report the specific test mode, therefore system software must track which test selector was used. Refer to Section 7.<<<TBD>>> for details on each test mode.

This field may only be set as a result of a SetPortFeature(PORT_TEST) request. A port PORT_TEST request is only valid to a port that is in the Disabled state. If the port is not in the Disabled state, the hub should respond with a request error.

This field may be cleared as a result of any of the following:

- The port being in the Powered-off state
- Receipt of a ClearPortFeature(PORT_TEST) request

11.24.2.7.2 Port Status Change Bits

Port status change bits are used to indicate changes in port status bits that are not the direct result of requests. Port status change bits can be cleared with a ClearPortFeature() request or by a hub reset. Hubs may allow setting of the status change bits with a SetPortFeature() request for diagnostic purposes. If a hub does not support setting of the status change bits, it may either treat the request as a Request Error or as a functional no-operation. Table 11-17 describes the various bits in the *wPortChange* field.

Table 11-17. Port Change Field, *wPortChange*

Bit	Description
~~0~~	~~**Connect Status Change:** (C_PORT_CONNECTION) Indicates a change has occurred in the port's Current Connect Status. The hub device sets this field as described in Section 11.16.2.6.2.1.~~ ~~0 = No change has occurred to Current Connect status~~ ~~1 = Current Connect status has changed~~
0	**Connect Status Change:** (C_PORT_CONNECTION) Indicates a change has occurred in the port's Current Connect Status. The hub device sets this field as described in Section 11.24.2.7.2.1. 0 = No change has occurred to Current Connect status 1 = Current Connect status has changed
1	**Port Enable/Disable Change:** (C_PORT_ENABLE) This field is set to one when a port is disabled because of a Port_Error condition (see Section 11.8.1).
2	**Suspend Change:** (C_PORT_SUSPEND) This field indicates a change in the host-visible suspend state of the attached device. It indicates the device has transitioned out of the Suspend stateThis field is set only when the entire resume process has completed. That is, the hub has ceased signaling resume on this port. 0 = No change 1 = Resume complete
3	**Over-Current Indicator Change:** (C_PORT_OVER_CURRENT) This field applies only to hubs that report over-current conditions on a per-port basis (as reported in the hub descriptor). 0 = No change has occurred to Over-Current Indicator 1 = Over-Current Indicator has changed If the hub does not report over-current on a per-port basis, then this field is always zero.
4	**Reset Change:** (C_PORT_RESET) This field is set when reset processing on this port is complete. 0 = No change 1 = Reset complete
5-15	**Reserved** These bits return 0 when read.

11.24.2.7.2.1 C_PORT_CONNECTION

This bit is set when the PORT_CONNECTION bit changes because of an attach or detach detect event (see Section 7.1.7.1). This bit will be cleared to zero by a ClearPortFeature(C_PORT_CONNECTION) request or while the port is in the Powered-off state.

11.24.2.7.2.2 C_PORT_ENABLE.

This bit is set when the PORT_ENABLE bit changes from one to zero as a result of a Port Error condition (see Section 11.8.1.). This bit is not set on any other changes to PORT_ENABLE.

This bit may be set if, on a SetPortFeature(PORT_RESET) the port stays in the Disabled state because an invalid idle state exists on the bus (see Section 11.8.2).

This bit will be cleared by a ClearPortFeature(C_PORT_ENABLE) request or while the port is in the Powered-off state.

11.24.2.7.2.3 C_PORT_SUSPEND

This bit is set on the following transitions:

- on transition from the Resuming state to the SendEOP state
- on transition from the Restart_S state to the Transmit state.

This bit will be cleared by a ClearPortFeature(C_PORT_SUSPEND) request, or while the port is in the Powered-off state.

11.24.2.7.2.4 C_PORT_OVER-CURRENT

This bit is set when the PORT_OVER_CURRENT bit changes from zero to one or from one to zero. This bit is also set if the port is placed in the Powered-off state due to an over-current condition on another port.

This bit will be cleared when the port is in the Not Configured state or by a ClearPortFeature(C_PORT_OVER-CURRENT) request.

11.24.2.7.2.5 C_PORT_RESET

This bit is set when the port transitions from the Resetting state (or, if present, the Speed_eval state) to the Enabled state.

This bit will be cleared by a ClearPortFeature(C_PORT_RESET) request, or while the port is in the Powered-off state.

11.24.2.8 Set Hub Descriptor

This request overwrites the hub descriptor.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero	Descriptor Length	Descriptor

The SetDescriptor request for the hub class descriptor follows the same usage model as that of the standard SetDescriptor request (refer to Chapter 9). The standard hub descriptor is denoted by using the value *bDescriptorType* defined in Section 11.23.2.1. All hubs are required to implement one hub descriptor, with descriptor index zero.

This request is optional. This request writes data to a class-specific descriptor. The host provides the data that is to be transferred to the hub during the data transfer phase of the control transaction. This request writes the entire hub descriptor at once.

Hubs must buffer all the bytes received from this request to ensure that the entire descriptor has been successfully transmitted from the host. Upon successful completion of the bus transfer, the hub updates the contents of the specified descriptor.

It is a Request Error if *wIndex* is not zero or if *wLength* does not match the amount of data sent by the host. Hubs that do not support this request respond with a STALL during the Data stage of the request.

If the hub is not configured, the hub's response to this request is undefined.

11.24.2.9 Set Hub Feature

This request sets a value reported in the hub status.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100000B	SET_ FEATURE	Feature Selector	Zero	Zero	None

Setting a feature enables that feature; refer to Table 11-12for the feature selector definitions that apply to the hub as recipient. Change indicators may not be acknowledged using this request.

It is a Request Error if *wValue* is not a feature selector listed in Table 11-12 or if *wIndex* or *wLength* are not as specified above.

If the hub is not configured, the hub's response to this request is undefined.

11.24.2.10 Set Port Feature

This request sets a value reported in the port status.

bmRequestType	bRequest	wValue	wIndex		wLength	Data
~~00100011B~~	~~SET_ FEATURE~~	~~Feature Selector~~	~~Port~~		~~Zero~~	~~None~~
00100011B	SET_ FEATURE	Feature Selector	Test Selector	Port	Zero	None

The port number must be a valid port number for that hub, greater than zero. The port number is in the least significant byte of the wIndex field. The most significant byte of wIndex is zero, except when the feature selector is PORT_TEST.

Setting a feature enables that feature or starts a process associated with that feature; see Table 11-12 for the feature selector definitions that apply to a port as a recipient. Change indicators may not be acknowledged using this request. Features that can be set with this request are:

- PORT_RESET
- PORT_SUSPEND
- PORT_POWER
- PORT_TEST
- C_PORT_CONNECTION*
- C_PORT_RESET*
- C_PORT_ENABLE*
- C_PORT_SUSPEND*
- C_PORT_OVER_CURRENT*.

*denotes features that are not required to be set by this request.

Setting the PORT_SUSPEND feature causes bus traffic to cease on that port and, consequently, the device to suspend. Setting the reset feature PORT_RESET causes the hub to signal reset on that port. When the

reset signaling is complete, the hub sets the C_PORT_RESET change indicator and immediately enables the port. Refer to Section 11.10 for a complete discussion of host-initiated reset behavior. Also see Section 11.24.2.7.1 for further details.

When the feature selector is PORT_TEST, the most significant byte of the wIndex field is the selector identifying the specific test mode. Table 11-18 lists the test selector definitions. Refer to Section 7.<<<TBD>>> for test details. The hub will respond with a request error if the request contains an invalid test selector.

Table 11-18. Test Mode Selector Codes

Value	Test Mode Description
0H	Reserved
1H	Test_SE0_NAK
2H	Test_J
3H	Test_K
4H	Test_PRBS (Psuedo-random bit sequence)
5H	Test_Force_enable
06H-3FH	Reserved for Standard Test modes
40H-BFH	Reserved
C0H-FFH	Reserved for Vendor-Unique test modes

The hub must meet the following requirements:

- If the port is in the Powered-off state, the hub must treat a SetPortFeature(PORT_RESET) request as a functional no-operation.
- If the port is not in the Enabled or Transmitting state, the hub must treat a SetPortFeature(PORT_SUSPEND) request as a functional no-operation.
- If the port is not in the Powered-off state, the hub must treat a SetPortFeature(PORT_POWER) request as a functional no-operation.

It is a Request Error if *wValue* is not a feature selector listed in Table 11-12, if *wIndex* specifies a port that doen't exist, or if *wLength* is not as specified above.

If the hub is not configured, the hub's response to this request is undefined.