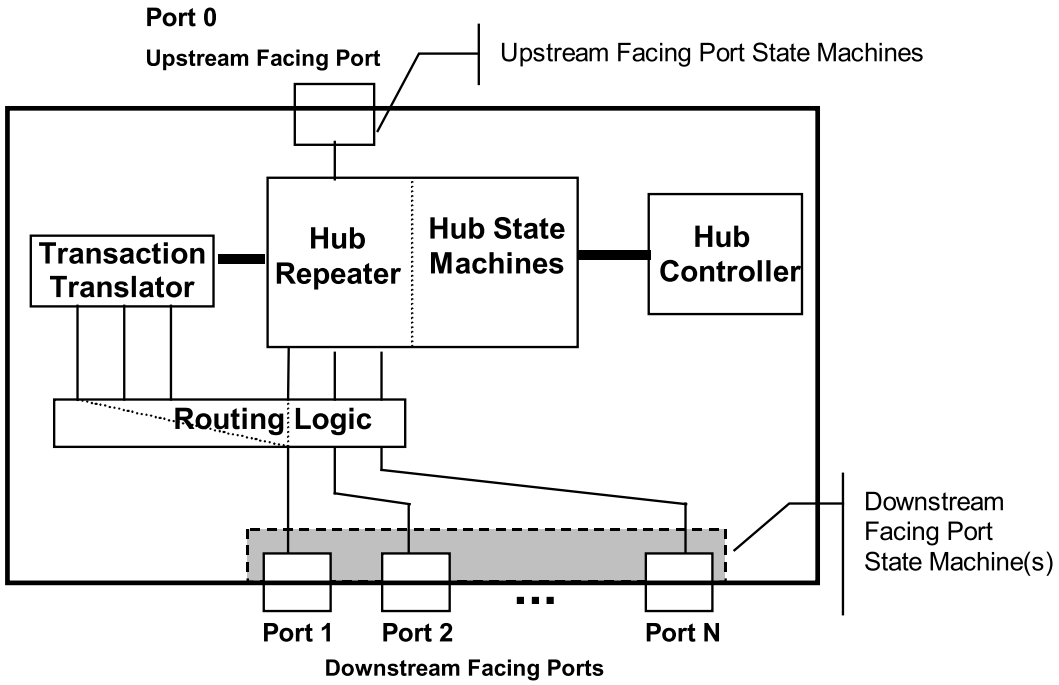



**Figure 11-1. Hub Architecture**

When a hub's upstream facing port is attached to an electrical environment that is operating at full-/low-speed, the hub's high-speed functionality 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 and the routing logic 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. In this case, 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 exhibit different 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 facing port) and one or more downstream facing 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.

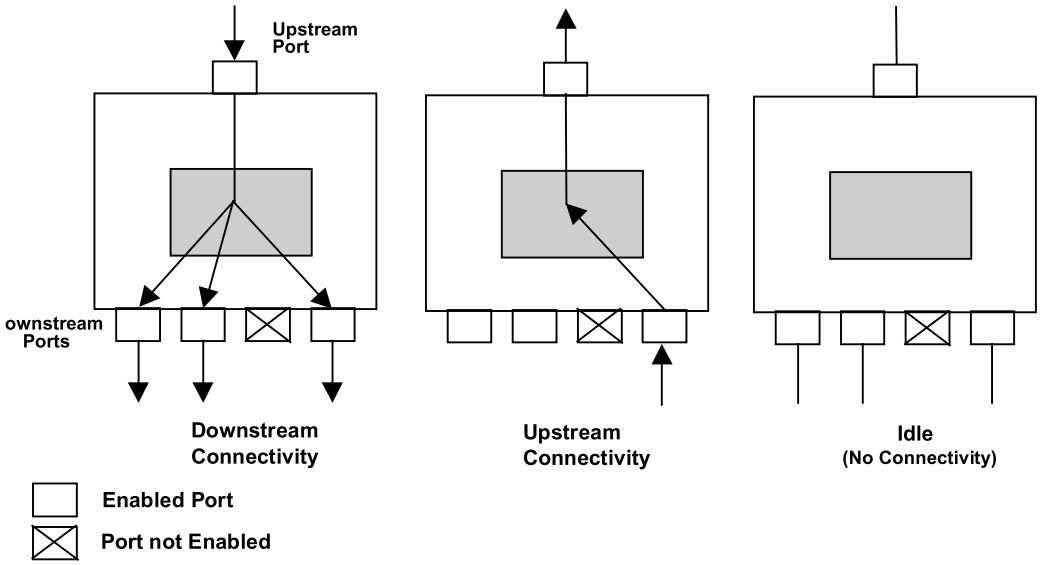



**Figure 11-2. Hub Signaling Connectivity**

If a downstream facing port is enabled (i.e., in a state where it can propagate signaling through the hub), and the hub detects the start of a packet on that port, connectivity is established in an upstream direction to the upstream facing port of that hub, but not to any other downstream facing 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 the start of a packet on its upstream facing port, it establishes connectivity to all enabled downstream facing 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 facing port to all of its enabled downstream facing ports. Figure 11-3 illustrates hub upstream and downstream resume connectivity.

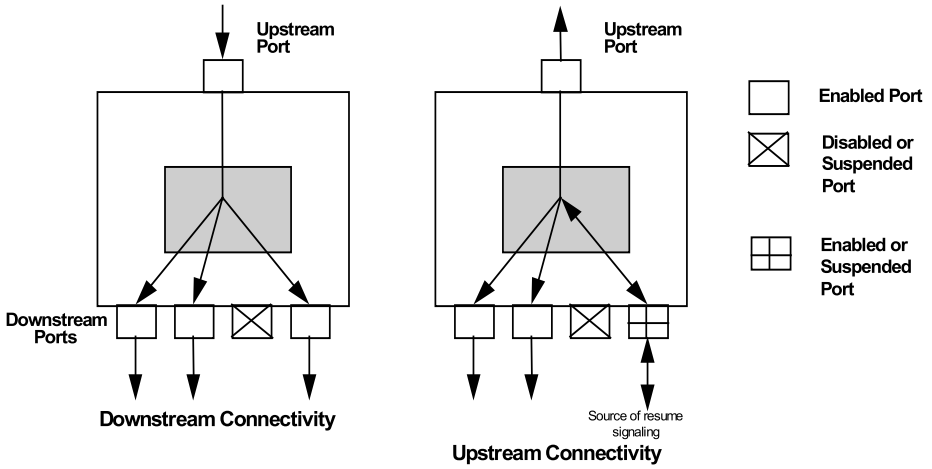



**Figure 11-3. Resume Connectivity**

If a hub is suspended and detects resume signaling from a selectively suspended or an enabled downstream facing port, the hub reflects that signaling upstream and to all of its enabled downstream facing 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/Microframe Timer

Each hub has a (micro)frame timer whose timing is derived from the hub's local clock and is synchronized to the host (micro)frame period by the host-generated Start-of-(micro)frame (SOF). The (micro)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 (micro)frame timer must track the host (micro)frame period and be capable of remaining synchronized with the host even if two consecutive SOF tokens are missed by the hub.

The (micro)frame timer must lock to the host's (micro)frame timing for worst case clock accuracies and timing offsets between the host and hub. There are specific requirements for hubs when their upstream facing port is operating at high-speed and full-speed.

### 11.2.1 High-speed Microframe Timer Range

The range for a microframe timer must be from 59904 to 60096 high-speed bits.

The nominal microframe interval is 60000 high-speed bit times. The hub microframe timer range specified above is 60000 +/- 96 high-speed bit times in order to accommodate host accuracy, hub accuracy, repeater jittter, and hub quantization. The +/-96 full-speed bit time variation is calculated in Table 11-2.

**Table 11-1. High-speed Microframe Timer Range Contributions**

| Source of Variation | Variation (ppm) | Variation (bits) Over One Microframe Interval | Comment |
|---|---|---|---|
| Host accuracy | +/- 500 | +/- 30 | |
| Hub accuracy | +/- 500 | +/- 30 | |
| Host jitter | | +/- 2 | |
| Hub chain jitter | | +/- 20 | Four hubs in series upstream of hub; 0 to 5 bits of jitter per hub |
| Quantization | | +/-14 | Bits need to round total variation to multiple of 16 |

### 11.2.2 Full-speed Frame Timer Range

The range of the frame timer must be from 11958 to 12042 full-speed bits.

The nominal frame interval is 12000 full-speed bit times. The hub frame timer range specified above is 12000 +/- 42 full-speed bit times in order to accommodate host accuracy and hub accuracy. The +/-42 full-speed bit time variation is calculated in Table 11-2.

**Table 11-2. Full-speed Frame Timer Range Contributions**

| Source of Variation | Variation (ppm) | Variation (bits) Over One Frame Interval | Comment |
|---|---|---|---|
| Host accuracy | +/- 500 | +/- 6 | |
| Hub accuracy | +/- 3000 | +/- 36 | +/-6 bits due to hub accuracy (500 ppm)<br><br>+/-30 bits due to 1.x parent hub accuracy (2500 ppm) |

### 11.2.3 Frame/Microframe Timer Synchronization

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

#### 11.2.3.1 Example (Micro)frame Timer Synchronization Method

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

When the first SOF token is detected, the current (micro)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 (micro)frame timer is loaded into the next (micro)frame register and into the (micro)frame timer. The current (micro)frame timer is reset to zero and continues to count and the flag is set to indicate that the (micro)frame timer is locked. The (micro)frame timer rolls over when the count exceeds 60096 for high-speed or 12042 for full-speed (a test at 65535 for high-speed or 16383 for full-speed is adequate). If the current (micro)frame timer has rolled over, then an SOF was missed and the (micro)frame timer and next (micro)frame values are not loaded. When an SOF is missed, the flag indicating that the timer is not synchronized remains set.

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

### 11.2.3.2 EOF Advancement

The hub must advance its EOF points based on its SOF decode time in order to ensure that in the tiered topology, hubs farther away from the host will always have later EOF points than hubs nearer to the host. The magnitude of advance is implementation-dependent; the possible range of advance is derived below.

The synchronization circuit described above depends on successfully decoding an SOF packet identifier (PID). This means that the (micro)frame timer will be synchronized to a time that is later than the synchronization point in the SOF packet: later by at least 40 bit times for high-speed or 16 bit times for full-speed. Each implementation also takes 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 192 bit times for high-speed and four bit times for full-speed. Subsequent sections describe the actions that are controlled by the (micro)frame timer. These actions are defined at the EOF1, EOF2, and EOF. EOF1 and EOF2 are defined in later sections. These sections assume that the hub's (micro)frame timer will count to zero at the end of the (micro)frame (EOF). The circuitry described above will have the (micro)frame timer counting to zero after 40 to 192 for high-speed bit times or 16-20 full-speed bit times after the start of a (micro)frame (or end of previous (micro)frame). The timings and bit offsets in the later sections must be advanced to account for this delay (i.e., add 40-192 for high-speed or 16-20 bit times for full-speed to the EOF1 and EOF2 points).

Advancing the EOF points by the processing delay ensures that the spread between the EOFs is only due to the propagation delay. For example, for high-speed, the maximum spread between 2 EOF points anywhere on the USB is less than 216 bits (144 + 72). 144 bit times are due to 36 bit times of max latency through 4 repeaters. 72 bit times are due to five maximum cable and interconnect delays of 30 ns each. As can be seen in Figure 11-4 without EOF advancement, a hub with a larger tier number could have an EOF occuring earlier than a hub with a smaller tier number. In Figure 11-5 with EOF advancement ensures that in the tiered topology, hubs with larger tier numbers always have later EOF points than hubs with smaller tier numbers. Note: 13 bit times in the figures is an example maximum cable delay (approximately 30 ns).

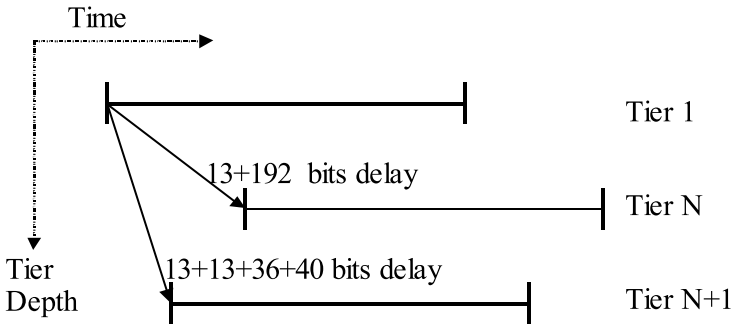



**Figure 11-4. Example High-speed EOF Offsets Due to Propagation Delay Without EOF Advancement**

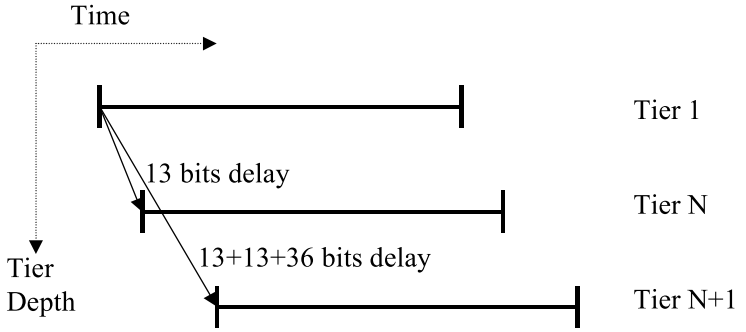



**Figure 11-5. Example High-speed EOF Offsets Due to Propagation Delay With EOF Advancement**

#### 11.2.3.3 Effect of Synchronization on Repeater Behavior

The (micro)frame timer provides an indication to the hub Repeater state machine that the (micro)frame timer has synchronized to SOF and that the (micro)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 full-/low-speed device may have been detached, and a low-/full-speed device attached while the host was generating a long resume (several seconds) and the disconnect cannot be detected. The new 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 SOFs 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 facing port from establishing connectivity until the hub is locked after a resume.

### 11.2.4 Microframe Jitter Related to Frame Jitter

The period between the SOFs from the Transaction Translator must not vary by more than +/- 42 ns. The microframe timer count must be used by the Transaction Translator to generate SOFs to full-speed devices (and keepalives to low-speed devices) connected to it.

The SOF received at the upstream facing port of the hub is repeated with a local clock. The frequency of this clock may be a divided version of the bit rate. This could result in a quantization error and microframe-to-microframe jitter. The microframe-to-microframe jitter of a hub repeater must be between 0 and 5 bit times. This means that the latency through the repeater of consecutive SOFs must differ by less than 5 bits. A hub may register the SOF for internal use, e.g., microframe synchronization. This requires SOF PID detection. The circuitry used for internal registering of the SOF must have a jitter which is less than or equal to 16 bits. This means that the microframe timer count values between consecutive equally spaced SOFs must differ by less than or equal to 16 bits. The host controller frequency may drift over the period of a microframe resulting in microframe period jitter. The host controller source jitter for SOFs must be less than 4 bits. This means that the consecutive periods between SOFs must differ by less than 4 bits. These requirements ensure that the microframe period at the end of five hub tiers will have a jitter of less than 40 bits (4 from host controller + 4*5 from hub repeaters + 16 from the internal SOF registering). This means that the consecutive periods between SOFs as measured at any microframe timer will differ by less than 40 bits (83.3 ns at 480 Mbs). This is less than the +/- 42 ns variation allowed.

### 11.2.5 EOF1 and EOF2 Timing Points

The EOF1 and EOF2 are timing points that are derived from the hub's (micro)frame timer. Table 11-3 specifies the required host and hub EOF timing points for high-speed and full-speed operation.

**Table 11-3. Hub and Host EOF1/EOF2 Timing Points**

| Label | Bit Times Before EOF for High-speed | Bit Times Before EOF for Full-speed | Notes |
|---|---|---|---|
| EOF1 | 560 | 32 | End-of-(micro)frame point #1 |
| EOF2 | 64 | 10 | End-of-(micro)frame point #2 |

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 (micro)frame timer has been synchronized to the SOF.*

The host and hub (micro)frame markers, while all synchronized to the host's SOF, are subject to certain skews that dictate the placement of the EOF points. Figure 11-6 illustrates EOF2 timing point for high-

speed operation. Figure 11-7 illustrates the EOF1 high-speed timing point. The numbers in the figures are in high-speed bit times.

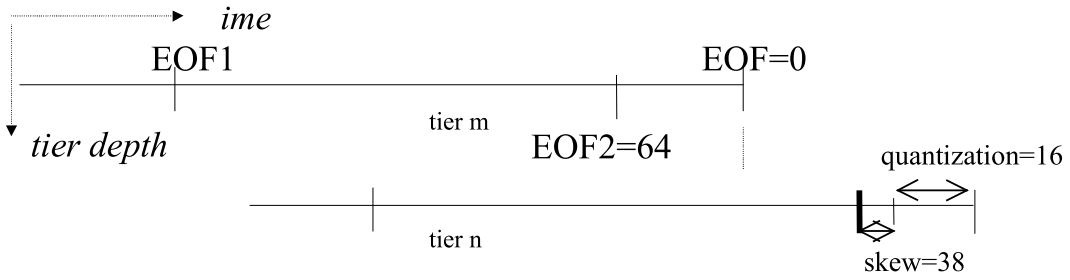



**Figure 11-6. High-speed EOF2 Timing Point**

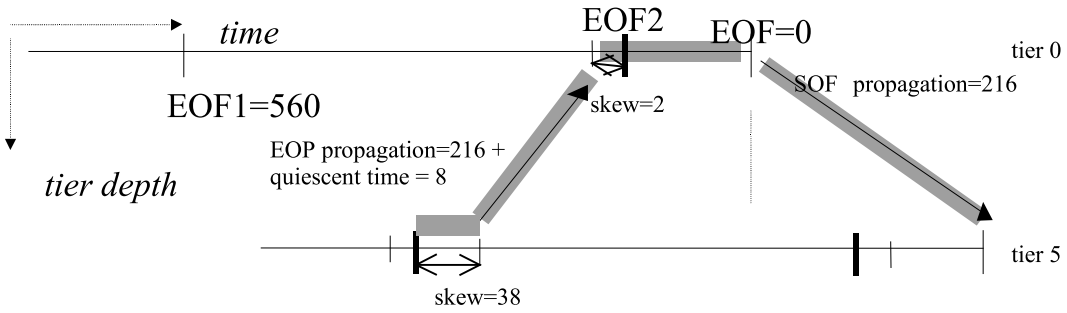



**Figure 11-7. High-speed EOF1 Timing Point**

At the EOF2 point, any port that has upstream connectivity will be disabled as a babbler. Hubs operating as a full-/low-speed repeater prevent becoming disabled by sending an end of packet to the upstream hub before that hub reaches its EOF2 point (i.e., at EOF1).

Figure 11-8 illustrates EOF timing points for full-/low-speed repeater operation.

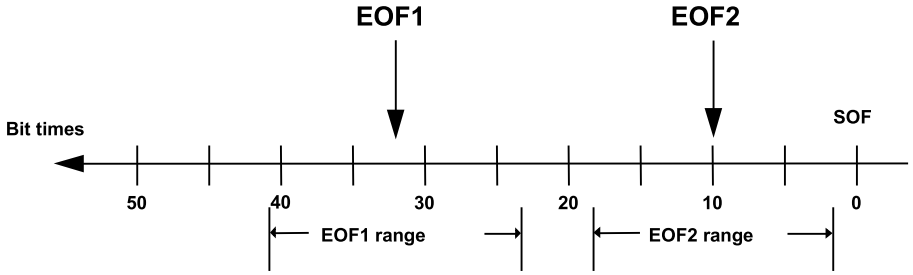



**Figure 11-8. Full-speed EOF Timing Points**

The hub operating as a full-/low-speed repeater is permitted to send the EOP if upstream connectivity is not established at EOF1 time. A full-speed repeater must send the EOP if connectivity is established from any downstream facing port at the EOF1 point.

A high-speed repeater must tear down upstream connectivity at the EOF1 point.

A high-speed repeater must tear down connectivity after the bus returns to the Idle state and the Elasticity buffer is emptied (as described in Section 11.7.2) rather than on decoding an EOP pattern as in full-/low-speed. Therefore, abrupt end of signaling (i.e, without a high-speed EOP) may cause malformed packets, and this must not affect repeater operation. The host controller design must be capable of processing such packets correctly.

#### 11.2.5.1 High-speed EOF1 and EOF2 Timing Points

The EOF2 point is 64 bit times before EOF as shown in Figure 11-6, and the EOF1 point is 560 bit times before EOF as shown in Figure 11-7.

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 microframe timers on different hubs and the host. The total accumulated skew can be as much as 38 bit times. This is composed of ±2 bit times of (micro)frame host source jitter and 0 to 36 bit times of repeater jitter as derived earlier. This skew timing affects the placement of the EOF1 and EOF2 points.

Note: The hub skew timing assumes that the microframe interval will not be changed by the host after the microframe timers have synchronized.

EOF skew can be from –2 to + 38 bits, so all EOFs are within 256 bits (216 bits of EOF propagation delay + 40 bits of EOF skew) of each other.

Note: The EOF2 point is based on 16 bit times for quantization + 38 bit times of skew; therefore, the EOF2 point needs to located at least 54 bit times before EOF. The EOF2 point is set at 64 bit times to allow babble detection to be done with a divided (by 16) version of the bit clock. An upstream-directed packet ending before EOF1 must reach every upstream hub/host before it gets to its EOF2 point. This is achieved if the EOF1 point is located at least 544 bits before any upstream EOF (64 bits of EOF2 offset + 216 bits of EOP propagation delay + 8 bits of idle time + 216 bits of SOF propagation delay + 38 bits of EOF1 skew + 2 bits of EOF2 skew). The EOF1 point is set at 560 bit times to allow using a divided (by 16) version of the bit clock.

#### 11.2.5.2 Full-speed EOF1 and EOF2 Timing Points

When the hub operates as a full-/low-speed repeater, the EOF1 point is 10 bit times before EOF and EOF1 is 32 bit times before EOF as shown in Figure 11-8.

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 facing 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. (Note: Such adjustment was permitted in the USB 1.0 and 1.1 specification but is no longer permitted.) 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.

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).

## 11.3 Host Behavior at End-of-Frame

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.

Note: The above requirements can be met if the host controller ensures that the last transaction will complete by its EOF1. The time consumed by a transaction (and consequently the latest start time of the transaction) can be evaluated by accumulating the various delay components in the transaction. The packet lengths should include all fields and account for bit-stuffing overhead as described in Chapter 7 and Chapter 8. Formulae for calculating transaction times are located in Section 5.11.3.

In defining the timing points below, the last bit interval in a (micro)frame is designated as bit time zero. Bit times in a (micro)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 time relationships be preserved.

Host controllers issuing high-speed transactions on a high-speed bus must meet the above requirements. Host controllers issuing full-/low-speed transactions on a full-/low-speed bus may also use the following three behaviors near EOF.

### 11.3.1 Full-/low-speed Latest Host Packet

Hubs are allowed to send an EOP on their upstream facing 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 Full-/low-speed 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 Full-/low-speed 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 8 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 cannot 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 (non-bitstuffed) bits of 1 followed by an EOP. The abnormal termination sequence is sent at the speed of the current packet. Note: The intent of this sequence is to force a bitstuffing violation (and possibly other errors) at the receiver.

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). (For example, 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

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 11-9 illustrates the internal port state machine; Table 11-4 defines the internal port signals and events.

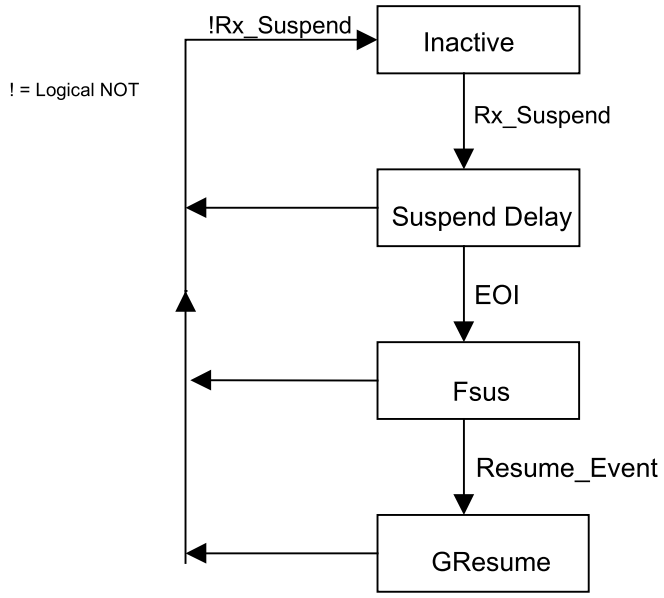



**Figure 11-9. Internal Port State Machine**

**Table 11-4. 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 2 ms 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 Figures 11-22 and 11-20, respectively).

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

## 11.5 Downstream Facing Ports

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

Figure 11-10 is an illustration of the downstream facing port state machine. The events and signals are defined in Table 11-5. 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.

Note: For the root hub, the signals from the upstream facing port state machines are implementation dependent.

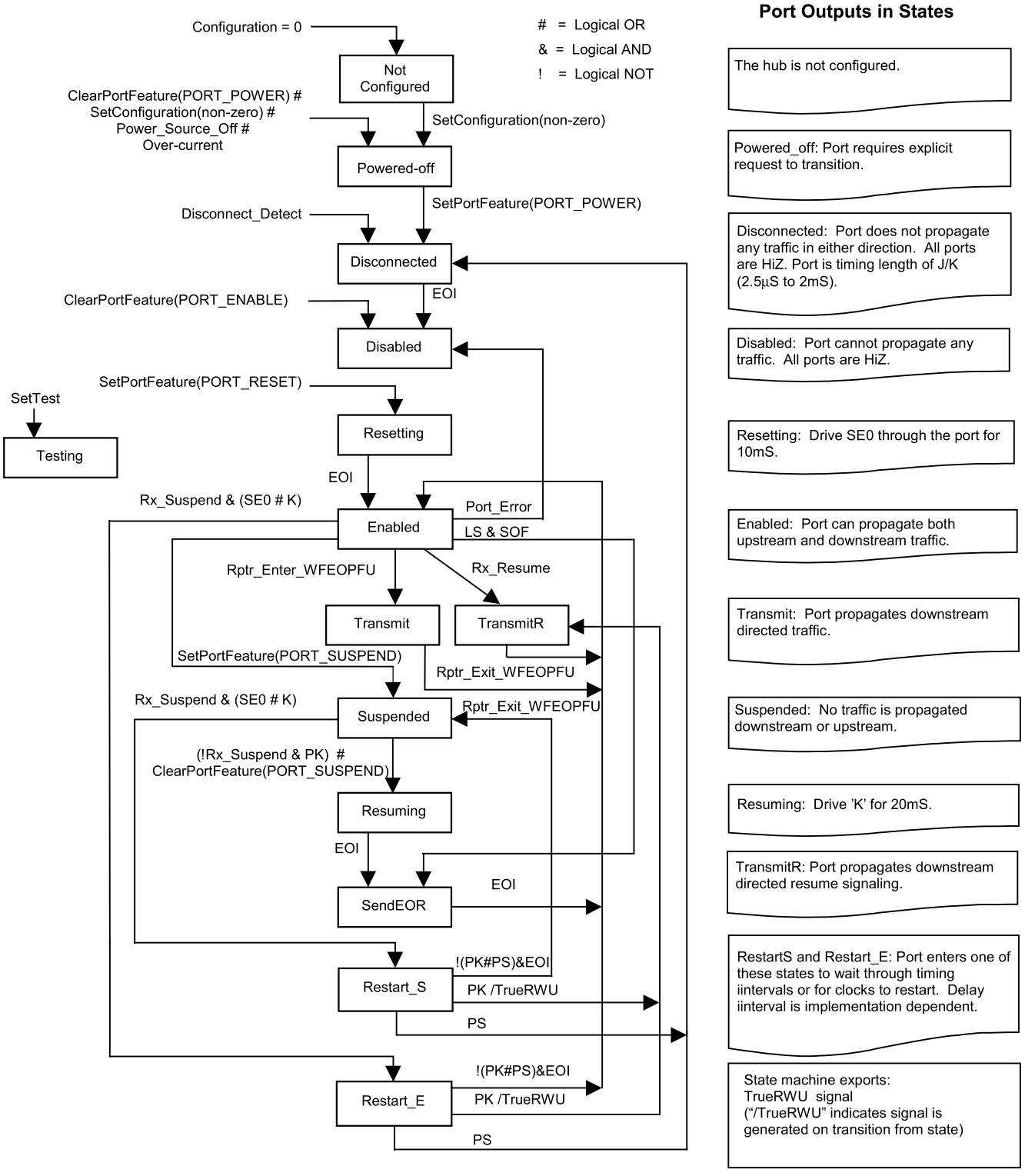



**Figure 11-10. Downstream Facing Hub Port State Machine**

**Table 11-5. Downstream Facing 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 | Disconnect seen at port |
| LS | Hub Controller | Low-speed device attached to this port |
| SOF | Hub Controller | SOF token received |
| TrueRWU | Internal | K lasting for at least TDDIS (see Table 7-13) |
| PK | Internal | K lasting for at least TDDIS |
| PS | Internal | SE0 lasting for at least TDDIS |
| 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) |
| SetTest | Hub Controller | Logical OR of SetPortFeature(Test_SE0_NAK), SetPortFeature(Test_J), SetPortFeature(Test_K), SetPortFeature(Test_PRBS), SetPortFeature(Test_Force_Enable) |
| Configuration = 0 | Hub Controller | Hub controller's configuration value is zero |

## 11.5.1 Downstream Facing 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, and 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 or SE1 state. If another signaling state is detected, the timer starts. Unless the hub is suspended with clocks stopped, this timer's duration is 2.5 μs to 2 ms.

If the hub is suspended with its remote wakeup feature enabled, then on a transition to any state other than the SE0 state or SE1 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 12 ms 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 Not Configured 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. If the port is using high-speed terminations when it enters this state, it switches to full-speed terminations. The port must not perform normal disconnect detection until at least 4 ms after entering this state.

#### 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 10 ms to 20 ms (10 ms is preferred).

A hub in high-speed operation will use the high-speed terminations of the port when in this state.

#### 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 or the TransmitR 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 SendEOR state
- From the Restart_E state when a persistent K or persistent SE0 has not been seen within 900 µs of entering that state

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

A port which is using high-speed terminations in this state switches to full-speed terminations on Rx_Suspend (i.e., when the hub is suspended). The port must not perform normal disconnect detection until at least 1 ms after Rx_Suspend becomes active.

#### 11.5.1.7 Transmit

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 facing 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 facing port. While in this state, the port will retransmit the data that is received on the upstream facing port (after proper inversion).

In high-speed, this state is used for testing for disconnect at the port. The disconnect detection circuit is enabled after 32 bits of the same signaling level ('J' or 'K') have been transmitted down the port.

Note: Because of the timing skew in the repeater path to the downstream facing ports, all downstream facing ports may not be enabled for disconnect detection at the same instant in time.

#### 11.5.1.8 TransmitR

This state is entered in either of the following situations:

- From the Enabled state if the upstream Receiver is in the Resume state
- From the Restart_S or Restart_E state if a PK is detected on the port

When in this state, the port repeats the resume 'K' at the upstream facing port to the downstream facing port. Depending on the speed of the port, two behaviors are possible on the K->SE0 transition at the upstream facing port at the end of the resume.

- Upstream facing port high-speed and downstream facing port full-/low-speed: After the K->SE0 transition, the port drives SE0 for 16 to 18 full-speed bit times followed by driving J for at least one full-speed bit time. Note: The timer in the Resume state of the upstream port receiver state machine which generates EOITR can be used to time this requirement at the downstream facing port(s). The pullup resistor and the latency of the Transaction Translator(TT) results in this Idle state being maintained for at least one low-speed bit time ensuring that a device sees the same end of resume behavior below the TT as it would below a USB 1.x hub.
- Upstream facing port and downstream facing port are the same speed: port continues to repeat the signaling which follows the K->SE0 transition.

A port operating in high-speed reverts to its high-speed terminations within 18 full-speed bit times after the K->SE0 transition as described in Section 7.1.7.7.

#### 11.5.1.9 Suspended

A port enters the Suspended state:

- From the Enabled state when it receives a SetPortFeature(PORT_SUSPEND) request
- From the Restart_S state when a persistent K or persistent SE0 has not been seen within 900 µs of entering that state

While a port is in the Suspended state, the port's differential transmitter is disabled. A high-speed port reverts from high-speed to full-speed terminations but its speed status continues to be high-speed. The port must not perform normal disconnect detection until at least 4 ms after entering this state.

An implementation must have a K/SE0 'noise' filter for a port that is in the suspended state. This filter can time the length of K/SE0 and, if the length of the K/SE0 is shorter than TDDIS, the port must remain in this state. If the hub is suspended with its clocks stopped, a transition to K/SE0 on a suspended port must cause the port to immediately transition to the Restart_S state.

#### 11.5.1.10 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 persists for at least 2.5 µs and the Receiver is not in the Suspended state. The transition from the Suspended state must happen within 900 µs of the J->K transition.
- When a ClearPortFeature(PORT_SUSPEND) request is received.

This is a timed state with a nominal duration of 20 ms (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.

#### 11.5.1.11 SendEOR

This state is entered from the Resuming state if the 20 ms timer expires. It is also entered from the Enabled state when an SOF (or other FS token) is received and a low-speed device is attached to this port.

This is a timed state which lasts for three low-speed bit times.

In this state, if the port is high-speed it will drive the bus to the Idle state for three low-speed bit times and then exit from this state to the Enabled state. It must also revert to its high-speed terminations within 18 full-speed bit times after the K->SE0 transition as described in Section 7.1.7.7.

If the port is full-speed or low-speed, the port must drive two low-speed bit times of SE0 followed by one low-speed bit time of Idle state and then exit from this state to the Enabled state.

Since the driven SE0 period should be of fixed length, the SendEOR timer, if shared, should not be reset. If the hub implementation shares the SendEOR timing circuits between ports, then for a port with a low-speed device attached, 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).

#### 11.5.1.12 Restart_S

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

In this state, the port continuously monitors the bus state. If the bus is in the 'K' state for at least TDDIS, the port sets the C_PORT_SUSPEND bit, exits to the TransmitR, and generates a signal to the repeater called 'TrueRWU'. If the bus is in the 'SE0' state for at least TDDIS, the port exits to the Disconnected state. Either of these transitions must happen within 900 µs after entering the Restart_S state; otherwise, the port must transition back to the Suspended state.

#### 11.5.1.13 Restart_E

A port 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.

In this state, the port continuously monitors the bus state. If the bus is in the 'K' state for at least TDDIS, the port exits to the TransmitR state and generates a signal to the repeater called 'TrueRWU'. If the bus is in the 'SE0' state for at least TDDIS, the port exits to the Disconnected state. Either of these transitions must happen within 900 µs after entering the Restart_E state; otherwise the port must transition back to the Enabled state.

#### 11.5.1.14 Testing

A port transitions to this state from any state when the port sees SetTest.

While in this state, the port executes the host command as decoded by the hub controller. If the command was a SetPortFeature(PORT_TEST, Test_Force_Enable), the port supports packet connectivity in the downstream direction in a manner identical to that when the port is in the Enabled state.

### 11.5.2 Disconnect Detect Timer

#### 11.5.2.1 High-speed Disconnect Detection

High-speed disconnect detection is described in Section 7.1.7.3.

### 11.5.2.2 Full-/low-speed Disconnect Detection

Each port is required to have a timer used for detecting disconnect when a full-/low-speed device is attached to the port. This timer is used to constantly monitor the port's 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 2.5 µs of SE0 or SE1 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 the Default Address state with the port enabled. This can cause systems errors that are very difficult to isolate and correct.

This timer must be reset whenever the D+ and D- lines on the port are not in the SE0 or SE1 state or when the port is not in the Enabled, Suspended, Disabled, Restart-E, or Restart_S states. This timer must be reset for 4ms upon entry to the Suspended and Disabled states. This timer times an interval TDDIS. The range of TDDIS is 2.0 µs to 2.5 as defined in Table 7-13. When this timer expires, it generates the Disconnect_Detect signal to the port state machine.

This timer can also be used for filtering the K/SE0 signal in the Suspended, Restart_E, or Restart_S states as described in Section 11.5.1.

## 11.5.3 Port Indicator

Each downstream facing port of a hub can support an optional status indicator. The presence of indicators for downstream facing ports is specified by bit 7 of the *wHubCharacteristics* field of the hub class descriptor. Each port's indicator must be located in a position that obviously associates the indicator with the port. The indicator provides two colors: green and amber. This can be implemented as physically one LED with two color capability or two separate LEDs. A combination of hardware and software control is used to inform the user of the current status of the port or the device attached to the port and to guide the user through problem resolution. Colors and blinking are used to provide information to the user.

An external hub must automatically control the color of the indicator as specified in Figure 11-11. Automatic port indicator setting support for root hubs may be implemented with either hardware or software. The port indicator color selector value is zero (indicating automatic control) when the hub transitions to the configured device state. When the hub is suspended or not configured, port indicators must be off.

Table 11-6 identifies the mapping of color to port state when the port indicators are automatically controlled.

**Table 11-6. Automatic Port State to Port Indicator Color Mapping**

| Power Switching | Downstream Facing Hub Port State | | | |
|---|---|---|---|---|
| | **Powered-off** | **Disconnected, Disabled, Not Configured, Resetting, Testing** | **Enabled, Transmit, or TransmitR** | **Suspended, Resuming, SendEOR, Restart_E, or Restart_S** |
| **With** | Off or amber if due to an over-current condition | Off | Green | Off |
| **Without** | Off | Off or amber if due to an over-current condition | Green | Off |

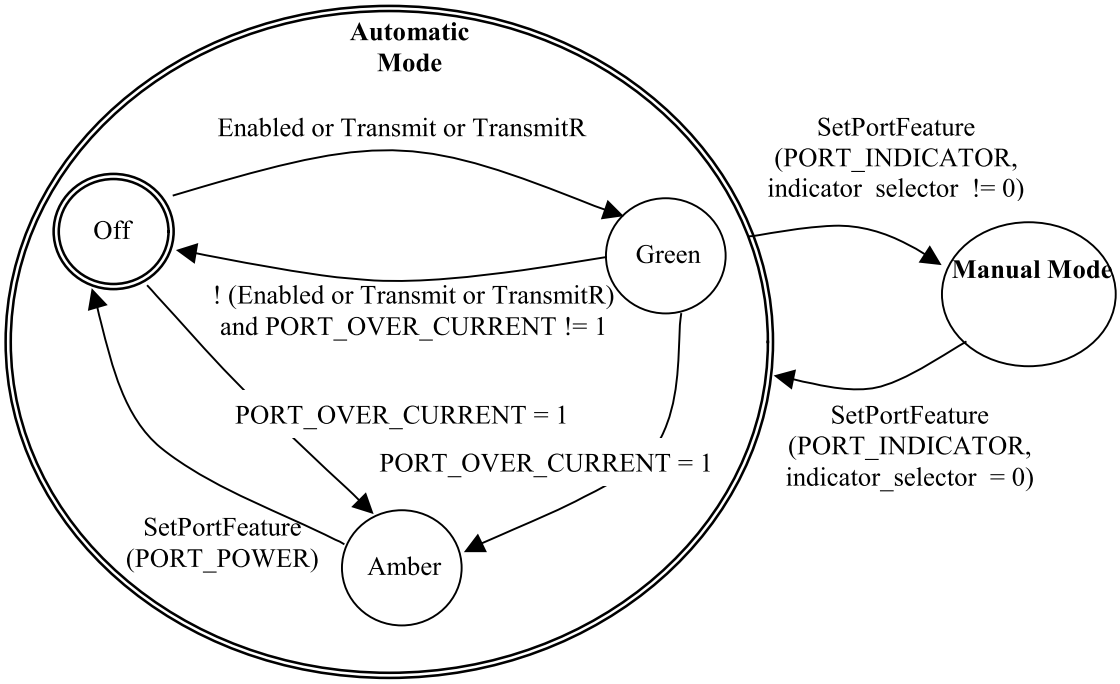



**Figure 11-11. Port Indicator State Diagram**

In **Manual Mode** the color of a port indicator (Amber, Green, or Off) is set by a system software USB Hub class request. In **Automatic Mode** the color of a port indicator is set by the port state information.

Table 11-7 defines port state as understood by the user.

**Table 11-7. Port Indicator Color Definitions**

| Color | Definition |
|---|---|
| Off | Not operational |
| Amber | Error condition |
| Green | Fully operational |
| Blinking Off/Green | Software attention |
| Blinking Off/Amber | Hardware attention |
| Blinking Green/Amber | Reserved |

Note that the indicators reflect the status of the port, not necessarily the device attached to it. Blinking of the indicator is used to draw the user's attention to the port, irrespective of its color.

Port indicators allow control by software. Host software forces the state of the indicator to draw attention to the port or to indicate the current state of the port.

See Section 11.24.2.7.1.10 for the specification of indicator requests.

#### 11.5.3.1 Labeling

USB system software uses port numbers to reference an individual port with a ClearPortFeature or SetPortFeature request. If a vendor provides a labeling to identify individual downstream facing ports, then each port connector must be labeled with their respective port number.

## 11.6 Upstream Facing Port

The upstream facing 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. The Transmitter and Receiver operate in high-speed and full-speed depending on the current hub configuration.

### 11.6.1 Full-speed

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. The single-ended components are also used to receive SE1. 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, while SJ, SK, SE0, and SE1 will be used for the single-ended signals.

When the Hub Repeater has connectivity in the upstream direction, the transmitter must not send or propagate SE1 signaling. Instead, the SE1 must be propagated as a DJ.

### 11.6.2 High-speed

Both the transmitter and receiver have differential components only. These signals are called HJ and HK. The HS_Idle state is the idle state of the bus in high-speed.

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.

### 11.6.3 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. This state machine details the operation of the device state diagram shown in Figure 9-1 in the Default, Address, Configured, and Suspended state. The Suspend, Resume, and ReceivingSE0 states are only used when the upstream facing port is operating in full-speed mode with full-speed terminations. The ReceivingIS, ReceivingHJ, and ReceivingHK states are only used when the upstream facing port is operating in high-speed mode with high-speed terminations; so these states are categorized as the HS (high-speed) states, and all other states are categorized as nonHS in the description below.

Figure 11-12 illustrates the state transition diagram.

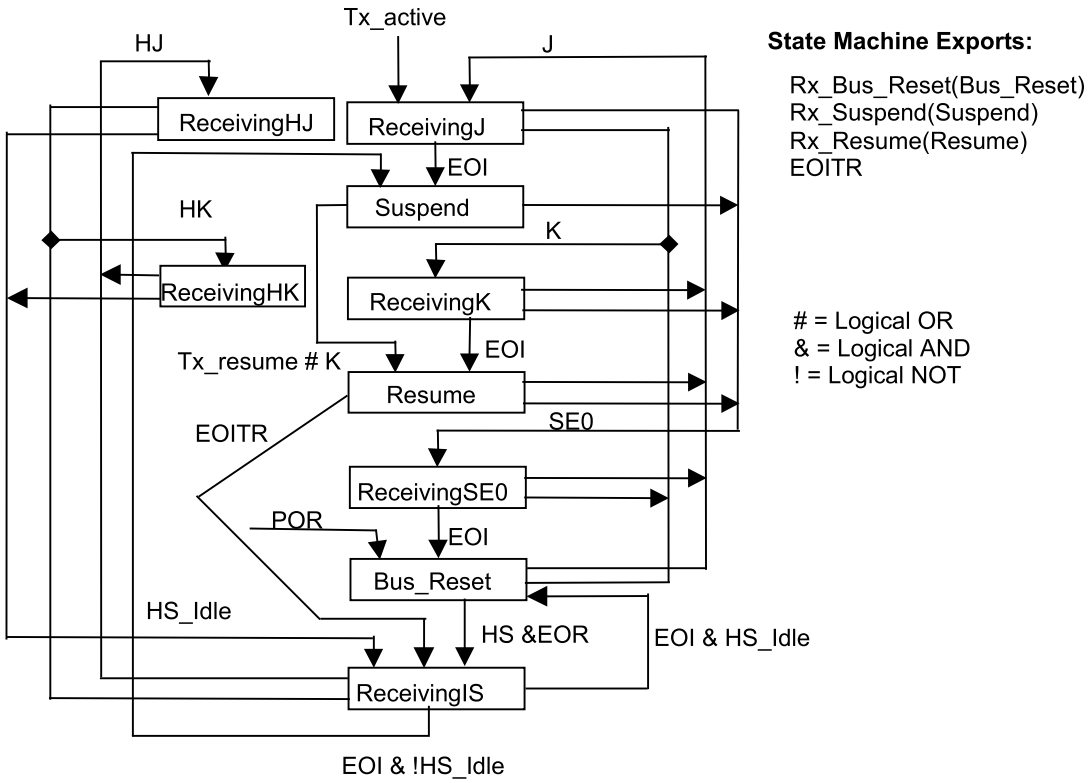



**Figure 11-12. Upstream Facing Port Receiver State Machine**

Table 11-8 defines the signals and events referenced in the figures.

**Table 11-8. Upstream Facing Port Receiver Signal/Event Definitions**

| Signal/Event Name | Event/Signal Source | Description |
|---|---|---|
| HS | Internal | Port is operating in high-speed |
| Tx_active | Transmitter | Transmitter in the Active state |
| J | Internal | Receiving a 'J' (IDLE) or an ‘SE1’ on the upstream facing port |
| HJ | Internal | Receiving an HJ on the upstream facing port |
| EOI | Internal | End of timed interval |
| EOITR | Internal | Generated 24 full-speed bit times after the K->SE0 transition at the end of resume |
| HK, K | Internal | Receiving an HK, 'K' on the upstream facing port |
| Tx_resume | Transmitter | Transmitter is in the Sresume state |
| HS_Idle | Internal | Receiving an Idle state on the high-speed upstream facing port |
| SE0 | Internal | Receiving an SE0 on the full-speed upstream facing port |
| EOR | Internal | End of Reset signaling from upstream |
| POR | Implementation-dependent | Power_On_Reset |

#### 11.6.3.1 ReceivingIS

This state is entered

- From the ReceivingHJ or ReceivingHK state when a SE0 is seen at the port and the port is in high-speed operation
- From the Resume state when a EOITR is seen and the port is in high-speed operation
- From the Bus Reset state at the End of Reset signaling from upstream when the port is in high-speed operation

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

#### 11.6.3.2 ReceivingHJ

This state is entered from an HS state when a HJ is seen on the bus.

#### 11.6.3.3 ReceivingJ

This state is entered from a nonHS state except the Suspend state if the receiver detects an SJ (or Idle) or SE1 condition on the bus or while the Transmitter is in the Active state.

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

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

#### 11.6.3.4 Suspend

This state is entered when:

- The 3 ms timer expires in the ReceivingJ
- The 3 ms timer expires in the ReceivingIS state and the port has removed its high-speed terminations and connected its D+ pull-up resistor and the resulting bus state is not SE0.

When the Receiver enters this state, the Hub Controller starts a 2 ms 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.

#### 11.6.3.5 ReceivingHK

This state is entered from an HS state when a HK is seen on the bus.

#### 11.6.3.6 ReceivingK

This state is entered from any nonHS 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 2.5 μs to 100 μs. The timer is reset each time this state starts.

#### 11.6.3.7 Resume

This state is entered:

- From the ReceivingK state when the timer expires
- 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 facing 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 (timing references must stabilize in less than 10 ms). If this state is being held pending stabilization of the hub's clock, the Receiver must provide a K to the repeater for propagation to the downstream facing ports. When clocks are stable, the Receiver must repeat the incoming signals.

Note: Hub timing references will be stable in less than 10 ms since reset requirements already specify that they be stable in less than 10 ms and a hub must support reset from suspend.

#### 11.6.3.8 ReceivingSE0

This state is entered from any nonHS 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 2.5 μs. The maximum depends on the hub but this interval must timeout early enough such that if the width of the SE0 on the upstream facing port is only 10 ms, the Receiver will enter the Bus_Reset state with sufficient time remaining in the 10 ms 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 (chirp) protocol and processing in the Bus_Reset state within 10 ms. 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 facing 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.

### 11.6.3.9 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.
- The 3 ms timer expires in the ReceivingIS state and the port has removed its high-speed terminations and connected its D+ pull-up resistor and the resulting bus state is still SE0.

In this state, a high-speed capable port will implement the chirp signaling, handshake, and timing protocol as described in Section 7.1.7.5.

## 11.6.4 Transmitter

This state machine is used to monitor the upstream facing 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 facing ports of this hub from propagating and causing this hub to be disabled or disconnected by the hub to which it is attached. Figure 11-13 is the transmitter state transition diagram. Table 11-9 defines the signals and events referenced in Figure 11-13.

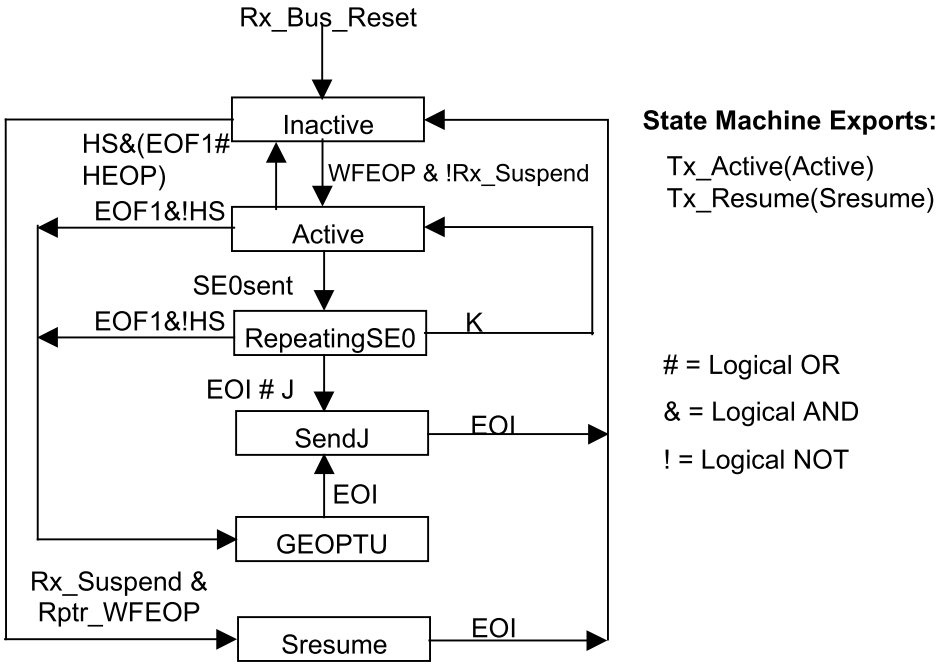



**Figure 11-13. Upstream Facing Port Transmitter State Machine**

**Table 11-9. Upstream Facing Port Transmit Signal/Event Definitions**

| Signal/Event Name | Event/Signal Source | Description |
|---|---|---|
| Rx_Bus_Reset | Receiver | Receiver is in the Bus_Reset state |
| EOF1 | (micro)frame Timer | Hub (micro)frame time has reached the EOF1 point or is between EOF1 and the end of the (micro)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 |
| HEOP | Repeater | Completion of packet transmission in upstream direction |
| HS | Internal | Upstream facing port is operating as high-speed port |
| EOI | Internal | End of timed interval |

#### 11.6.4.1 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.

When port is operating as a high-speed port, this state is entered from the Active state at EOF1 or after an HEOP from downstream.

#### 11.6.4.2 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 facing port is repeated and transmitted on the upstream facing port.

#### 11.6.4.3 RepeatingSE0

The port enters this state from the Active state when one bit time of SE0 has been sent on the upstream facing 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.

#### 11.6.4.4 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' or 'SE1'. 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.

#### 11.6.4.5 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.

#### 11.6.4.6 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 facing port. While the Transmitter is in this state, the Receiver is held in the Resume state. While the Receiver is in the Resume state, all downstream facing ports that are in the Enabled state are placed in the TransmitR state and the resume on this port is transmitted to those downstream facing ports.

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

### 11.7 Hub Repeater

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

### 11.7.1 High-speed Packet Connectivity

High-speed packet repeaters must reclock the packets in both directions. Reclocking means that the repeater extracts the data from the received stream and retransmits the stream using its own local clock. This is necessary in order to keep the jitter seen at a receiver within acceptable limits (see Chapter 7 for definition and limits on jitter).

Reclocking creates several requirements which can be best understood with the example repeater signal path shown in Figure 11-14.

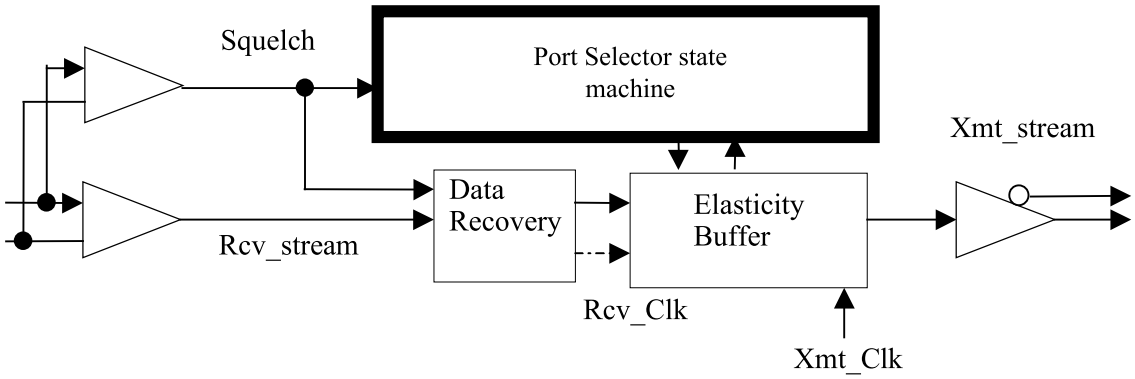



**Figure 11-14. Example Hub Repeater Organization**

#### 11.7.1.1 Squelch Circuit

Because of squelch detection, the initial bits of the SYNC field may not be seen in the rest of the repeater. At most, 4 bits of the SYNC field may be sacrificed in the entire repeater path.

The squelch circuit may take at most 4 bit times to disable the repeater after the bus returns to the Idle state. This results in bits being added after the end of the packet. This is also known as EOP dribble and up to 4 random bits may get added after the packet by the entire repeater path.

#### 11.7.1.2 Data Recovery Unit

The data recovery unit extracts the receive clock and receive data from this stream. Note that this is a conceptual model only; actual implementations (e.g., DLL) may achieve the reclocking by the local clock without separation of the receive clock and data.

#### 11.7.1.3 Elasticity Buffer

The half-depth of the elasticity buffer in the repeater must be at least 12 bits.

The total latency of a packet through a repeater must be less than 36 bit times. This includes the latency through the elasticity buffer.

The elasticity buffer is used to handle the difference in frequency between the receive clock and the local clock and works as follows. The elasticity buffer is primed (filled with at least 12 bits) by the receive clock before the data is clocked out of it by the transmit clock. If the transmit clock is faster than the receive clock, the buffer will get emptied more quickly than it gets filled. If the transmit clock is slower, the buffer will get emptied slower than it gets filled. If the half-depth of the buffer is chosen to be equal to the maximum difference in clock rate over the length of a packet, bits will not be lost or added to the packet. The half-depth is calculated as follows.

The clock tolerance allowed is 500 ppm. This takes into account the effect of voltage, temperature, aging, etc. So the received clock and the local clock could be different by 1000 ppm. The longest packet has a data payload of 1 Kbytes. The maximum length of a packet is computed by adding the length of all the fields and assuming maximum bit-stuffing. This maximum length is 9644 bits (9624 bits of packet + 20 bits of EOP dribble). This means that when the repeater is clocking out a packet with its local clock, it could get ahead of or fall behind the receive clock by 9.644 bits (1000 ppm*9644). This calculation yields 10 bits. The half-depth of the elasticity buffer in the repeater must be at least 12 bits to provide system timing margin.

#### 11.7.1.4 High-Speed Port Selector State Machine

This state machine is used to establish connectivity on a valid packet and to keep the repeater from establishing connectivity from a port which is seeing noise. This state machine must implement the behavior shown in Figure 11-15. (Note: This state machine may be implemented on a per-port or per-hub basis.)

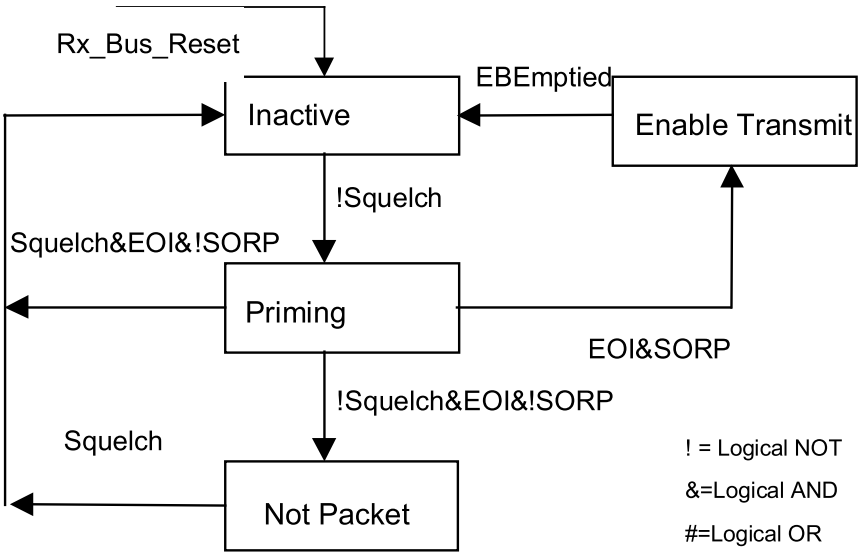



**Figure 11-15. High-speed Port Selector State Machine**

**Table 11-10. High-speed Port Selector Signal/Event Definitions**

| Signal/Event Name | Event/Signal Source | Description |
|---|---|---|
| Rx_Bus_Reset | Internal | Receiver is in the Bus_reset state. |
| EBEmptied | Internal | All bits accumulated in the elasticity buffer have been transmitted. |
| EOI | Internal | End of interval of time needed for priming elasticity buffer |
| Squelch | Internal | Bus is in squelch state |
| SORP | Internal | Start Of Repeating Pattern; a 'JKJK' or 'KJKJ' pattern has been seen in data in elasticity buffer. |

#### 11.7.1.4.1 Inactive

This state is entered

- From the Enable Transmit state when all the bits accumulated in the elasticity buffer have been transmitted
- From the Priming state if squelch is seen and the elasticity buffer is primed without a SORP being seen
- From the Not Packet state when the squelch circuit indicates a squelch state on the port
- From on any state on Rx_Bus_Reset

#### 11.7.1.4.2 Priming

This state is entered from the Inactive state when the squelch circuit indicates that valid signal levels have been observed at the port. This is a timed state and the priming interval is the time needed for the implementation to fill the elasticity buffer with at least 12 bits.

#### 11.7.1.4.3 Enable Transmit

This state is entered from the Priming state when the Elasticity buffer priming interval has elapsed and the bits in the elasticity buffer include the SORP pattern.

In this state, the state machine generates a signal “start of high-speed packet” (SOHP) to the repeater state machine which allows the repeater to establish connectivity from this port to the upstream facing port (or downstream facing ports).

#### 11.7.1.4.4 Not Packet

This state is entered from the Priming state when the Elasticity buffer priming interval has elapsed, and the bits in the elasticity buffer do not include the SORP pattern, and the squelch signal is not active.

### 11.7.2 Hub Repeater State Machine

The Hub repeater state machine in Figure 11-16 shows the states and transitions needed to implement the Hub Repeater. Table 11-11 defines the Hub Repeater signals and events. The following sections describe the states and the transitions.

#### 11.7.2.1 High-speed Repeater Operation

Connectivity is setup on SOHP and torn down on HEOP. (HEOP is either the EBemptied signal from the port selector state machine ‘OR’ the EOI signal which causes the transition out of the SendEOR state in downstream facing port state machine.) Several of the state transitions below will occur when the HEOP is seen. When such a transition is indicated, the transition does not occur until after the hub has repeated the last bit in the elasticity buffer. Some of the transitions are triggered by an SOHP. Transitions of this type occur as soon as the hub detects the SOHP from the port selector state machine ensuring that a valid packet start has been seen.

#### 11.7.2.2 Full-/low-speed Repeater Operation

Connectivity is setup on SOP and torn down on EOP. Several of the state transitions below will occur when the EOP is seen. 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.

### 11.7.2.3 Repeater State Machine

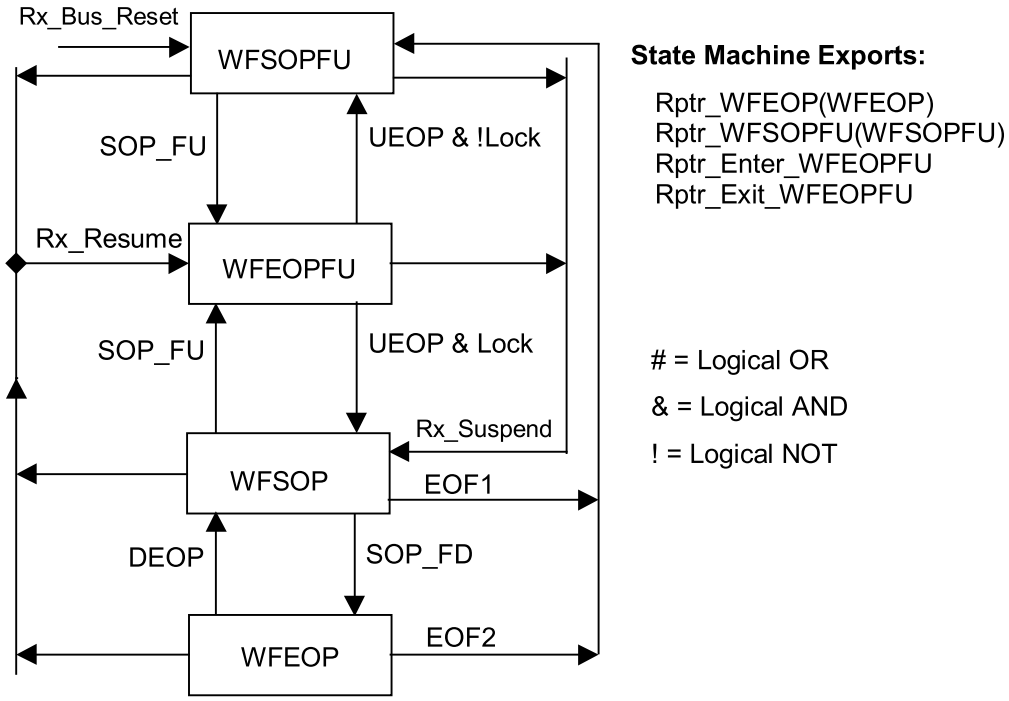



**Figure 11-16. Hub Repeater State Machine**

**Table 11-11. Hub Repeater Signal/Event Definitions**

| Signal/Event Name | Event/Signal Source | Description |
|---|---|---|
| Rx_Bus_Reset | Receiver | Receiver is in the Bus_Reset state |
| HEOP | Internal (Port selector, Downstream port, Upstream port receiver) | Three sources of HEOP:<br>EBEmptied signal from port selector state machine OR<br>transition at EOI from SendEOR state in downstream facing port state machine OR<br>EOITR from upstream facing port receiver state machine |
| UEOP | Internal | (HEOP)EOP received from the upstream facing port |
| DEOP | Internal | Generated when the Transmitter enters the (Inactive) SendJ state |
| EOF1 | (Micro)frame Timer | (micro)frame timer is at the EOF1 point or between EOF1 and End-of-(micro)frame |
| EOF2 | (Micro)frame Timer | (micro)frame timer is at the EOF2 point or between EOF2 and End-of-(micro)frame |
| Lock | (Micro)frame Timer | (micro)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 | (SOHP)SOP received from downstream facing port or Hub Controller. Generated (after SOHP identified) on the transition from the Idle to K state on a port. |
| SOP_FU | Internal | (SOHP)SOP received from upstream facing port. Generated (after SOHP identified) on the transition from the Idle to K state on the upstream facing port. |

### 11.7.3 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 (micro)frame timer is at or has passed the EOF1 point
- From the WFEOP state at the EOF2 point
- From the WFEOPFU if the (micro)frame timer is not synchronized (locked) when an (HEOP)EOP is received on the upstream facing port

In this state, the hub is waiting for an (SOHP)SOP on the upstream facing port, and transitions on downstream facing 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-(micro)frame (past the EOF1 point) to ensure that the hub will be able to receive the SOF when it is sent by the host.

### 11.7.4 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 (SOHP)SOP is detected on the upstream facing 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 facing port to all enabled downstream facing ports. Downstream facing ports that are in the Enabled state are placed in the Transmit state on the transition to this state.

### 11.7.5 Wait for Start of Packet (WFSOP)

This state is entered in any of the following situations:

- From the WFEOP state when an (HEOP)EOP is detected from the downstream facing port
- From the WFEOPFU state if the (micro)frame timer is synchronized (locked) when an (HEOP)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 (SOHP)SOP on the upstream facing port or any downstream facing port that is in the Enabled state. While the Hub Repeater is in this state, connectivity is not established.

### 11.7.6 Wait for End of Packet (WFEOP)

This state is entered from the WFSOP state when an (SOHP)SOP is received from a downstream facing 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 facing port is repeated and driven on the upstream facing 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 facing port for which connectivity is established is disabled as a babble port.

Note: The full-speed Transmitter will send an EOP at EOF1, but the Repeater stays in this state until the device sends an (HEOP)EOP or the EOF2 point is reached.

## 11.8 Bus State Evaluation

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 occur on a downstream facing 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 (micro)frame timer reaches the EOF2 point.
- At the EOF2 point, the Hub Repeater is in the WFSOPFU state, and there is other than Idle state on the port.

If upstream-directed connectivity is established when the (micro)frame timer reaches the EOF1 point, the upstream Transmitter will (return to Inactive state) 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 (micro)frame timer reaches the EOF2 point.

### 11.8.2 Speed Detection

At the end of reset, the bus is in the Idle state for the speed recorded in the port status register. Speed detection is described in Section 7.1.7.5.

If the device connected at the downstream facing port is high-speed, the repeater (rather than the Transaction Translator) is used to signal between this port and the upstream facing port.

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 7.1.7.3). At the end of the debounce interval, the device is expected to have placed its upstream facing 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.

The downstream facing port monitors the state of the D+ and D- lines to determine if the connected device is low-speed. If so, the PORT_LOW_SPEED status bit is set to one to indicate a low-speed device. If not, the PORT_LOW_SPEED status bit is set to zero to indicate a full-/high-speed device. Upon exit from the reset process, the hub must set the PORT_HIGH_SPEED status bit according to the detected speed. The downstream facing port performs the required reset processing as defined in Section 7.1.7.5. At the end of the Resetting state, the hub will return the bus to the Idle state that is appropriate for the speed of the attached device and transition to the Enabled state.

### 11.8.3 Collision

If the Hub Repeater is in the WFEOP state and an (SOHP)SOP is detected on another enabled port, a Collision condition exists. There are two allowed behaviors for the hub in this instance. In either case, connectivity teardown at EOF1 and babble detection at EOF2 is required.

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 (‘J’ or) 'K' on the upstream facing port. This (‘J’ or) 'K' should persist until packet traffic from all downstream facing ports ends. The hub should use the last (‘J’ or ‘K’) 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.

Note: In high-speed repeaters, use of the SOHP to detect collisions would need replication of the datapath shown in Figure 11-14 at every port. The unsquelch signal at a port can be used instead of the SOHP to detect collisions; in this case, the second behavior (blocking) described above must be used.

### 11.8.4 Low-speed Port Behavior

When a hub is configured for full-/low-speed operation, low-speed data is sent or received through the hub’s upstream facing port at full-speed signaling 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 (75-300 ns), and the port will not propagate downstream-directed packets unless they are prefaced with a PRE PID. When a PRE PID is received, the 'J' state must be driven on enabled low-speed ports within four bit times of receiving the last bit of the PRE PID.

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 full-/low-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 2.0 V 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. 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

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 7.1.7.6. Global suspend/resume refers to the entire bus being suspended or resumed without affecting any hub's downstream facing port states; selective suspend/resume refers to a downstream facing 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 7.1.7.7).

If the hub upstream facing port is in (high-speed) full-speed, the required behavior is the same as that for a function with upstream facing port in (high-speed) full-speed and is described in Chapter 7.

When a downstream facing port operating at high-speed goes into the Suspended state, it switches to full-speed terminations but continues to have high-speed port status. In response to a remote wakeup or selective resume, this port will drive full-speed 'K' throughout its Resuming state. The requirements and timings are the same as for full-speed ports and described below. At the end of this signaling, the bus will

be returned to the high-speed Idle state (using the SendEOR state). After this, the port will return to the Enabled state. The high-speed status of the port is maintained throughout the suspend-resume cycle.

Figure 11-17 and Figure 11-18 show 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 7.1.7.7.

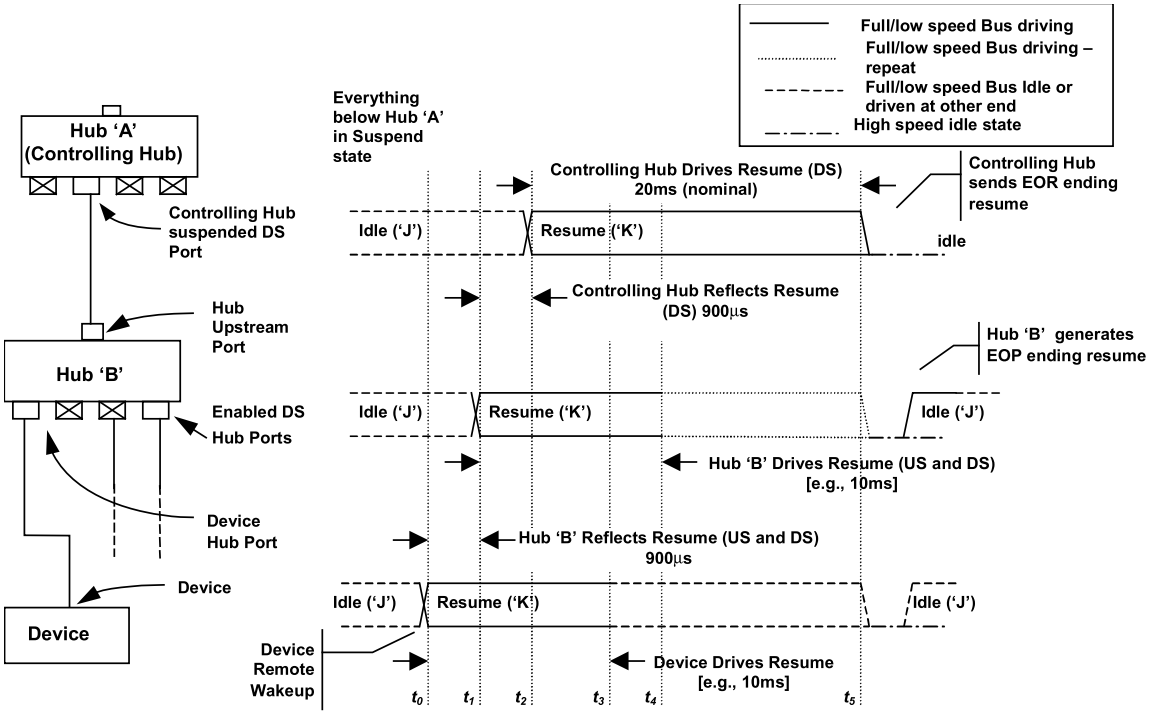



**Figure 11-17. Example Remote-wakeup Resume Signaling With Full-/low-speed Device**

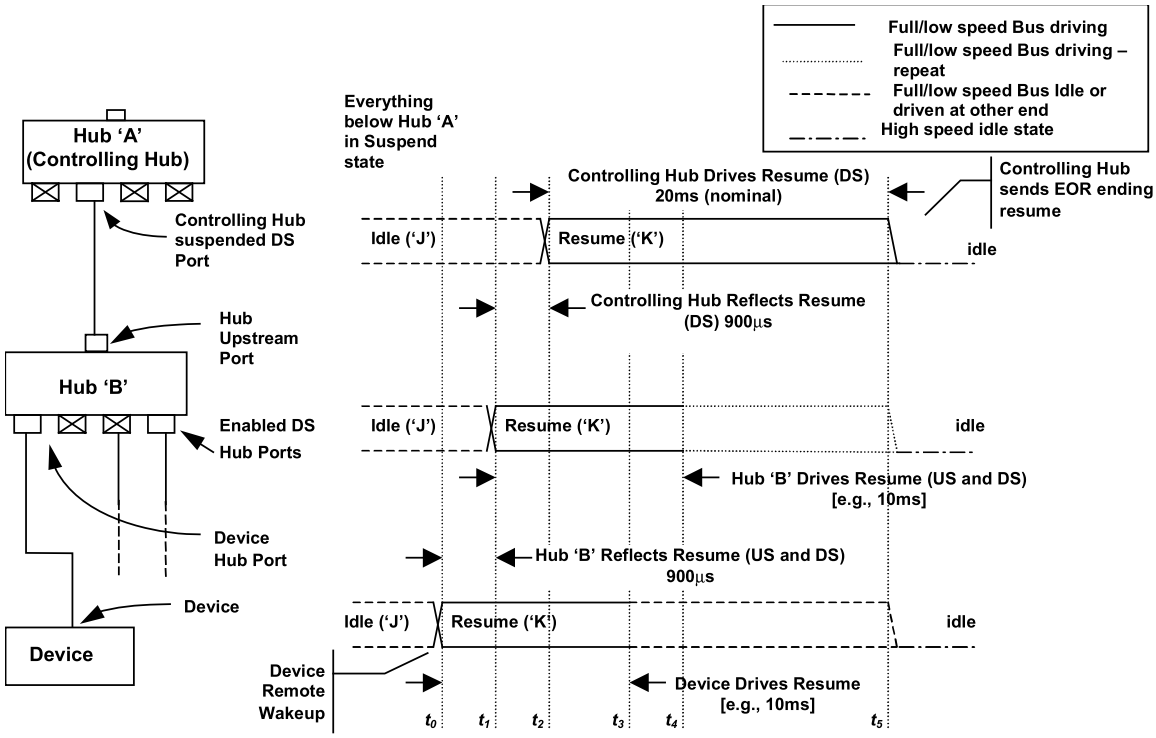



**Figure 11-18. Example Remote-wakeup Resume Signaling With High-speed Device**

Here is an explanation of what happens at each $t_n$:

$t_0$ Suspended device initiates remote-wakeup by driving a 'K' on the data lines.

$t_1$ Suspended hub 'B' detects the 'K' on its downstream facing port and wakes up enough within 900 µs to filter and then reflect the resume upstream and down through all enabled ports.

$t_2$ 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 filters and then reflects the resume signal back to 'B' within 900 µs.

$t_3$ Device ceases driving 'K' upstream.

$t_4$ Hub 'B' ceases driving 'K' upstream and down all enabled ports and begins repeating upstream signaling to all enabled downstream facing ports.

$t_5$ Hub 'A' completes resume sequence, after appropriate timing interval, by driving a speed-appropriate end of resume downstream. (End of resume will be an Idle state for a high-speed device or a low-speed EOP for a full-/low-speed device.)

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

Reset signaling to a hub is defined only in the downstream direction, which is at the hub's upstream facing port. Reset signaling required of the hub is described in Section 7.1.7.5.

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 facing ports are in the Not Configured state and SE0 driven on all downstream facing ports.

## 11.11 Hub Port Power Control

Self-powered hubs may have power switches that control delivery of power downstream facing 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* must 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 associated 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 Controller

The Hub Controller is logically organized as shown in Figure 11-19.

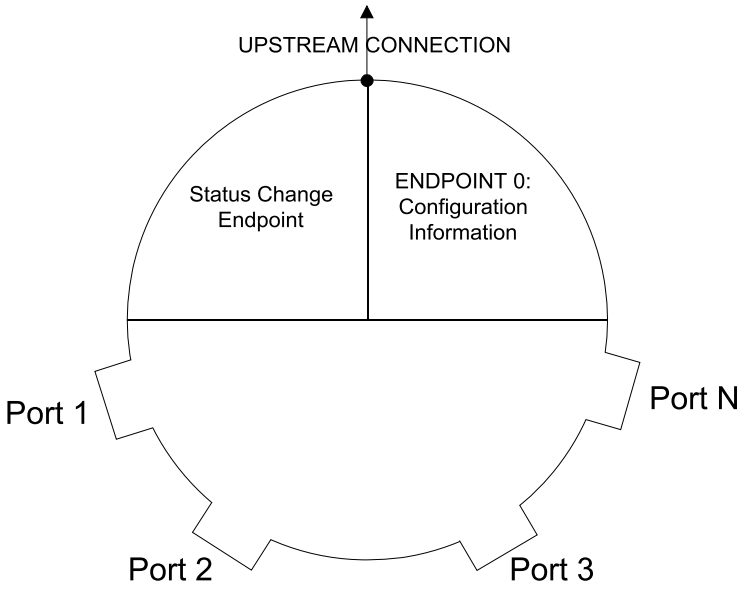



**Figure 11-19. Example Hub Controller Organization**

### 11.12.1 Endpoint Organization

The Hub Class defines one additional endpoint beyond Default Control Pipe, which is required for all 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 11.12.4, 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.

## 11.12.2 Hub Information Architecture and Operation

Figure 11-20 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 11.12.4.

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.

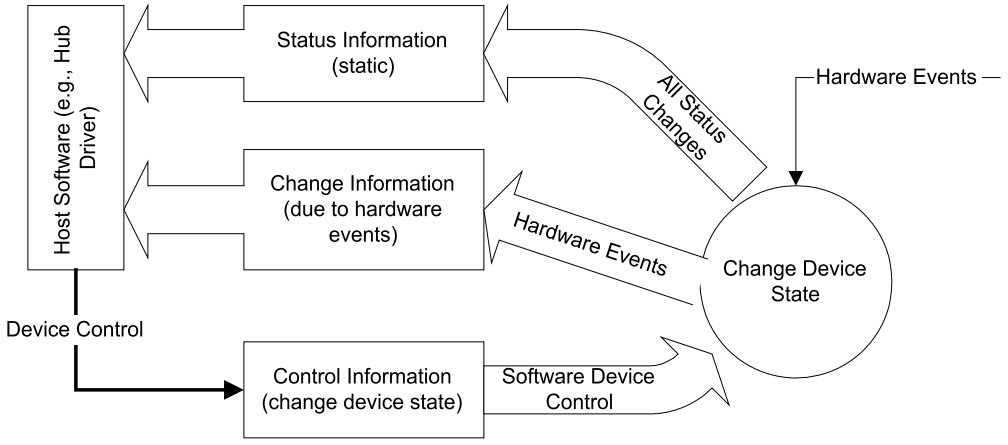



**Figure 11-20. Relationship of Status, Status Change, and Control Information to Device States**

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

## 11.12.3 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 11-21).

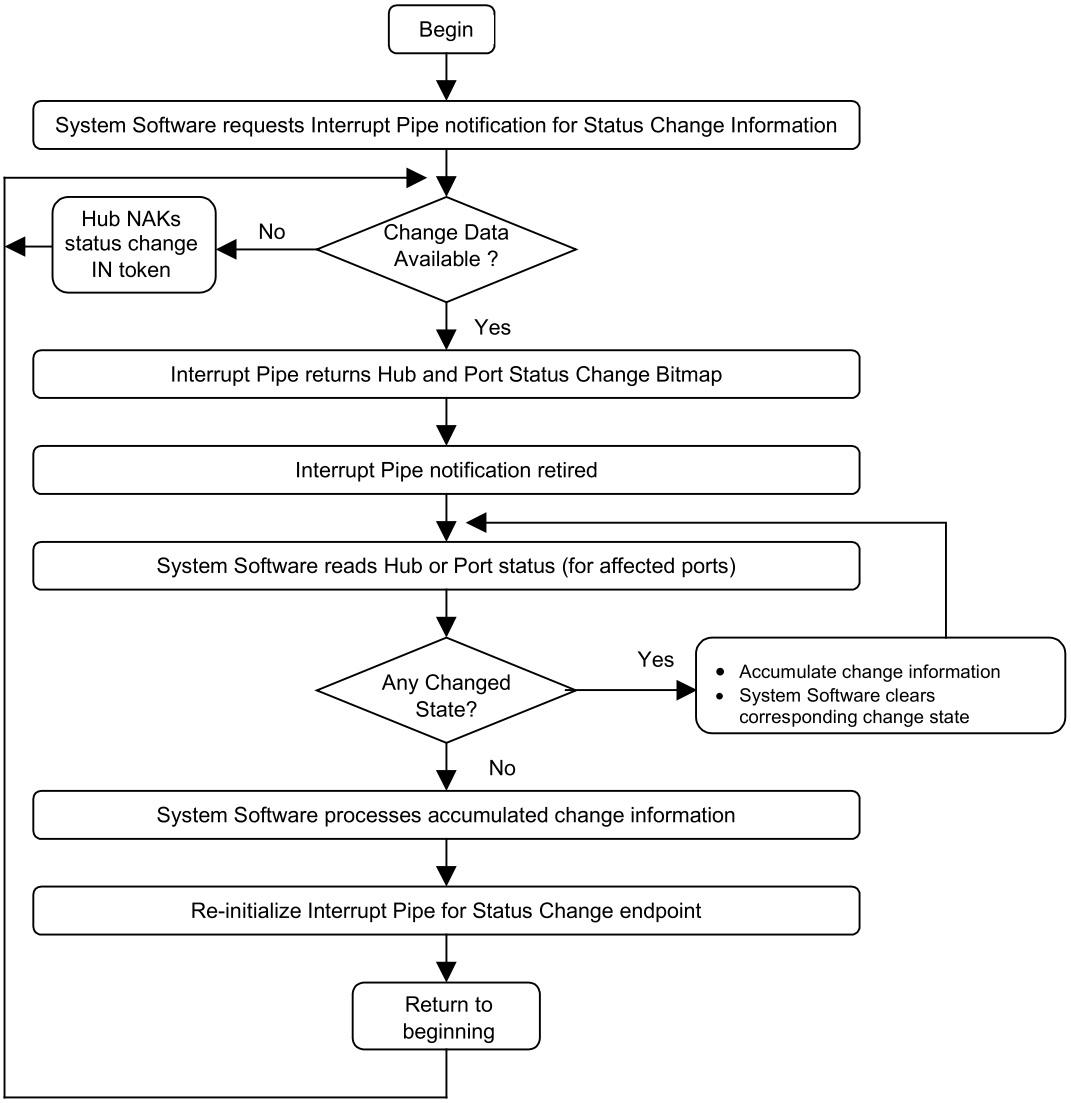



**Figure 11-21. Port Status Handling Method**

## 11.12.4 Hub and Port Status Change Bitmap

The Hub and Port Status Change Bitmap, shown in Figure 11-22, indicates whether the hub or a port has experienced a status change. This bitmap also indicates which port(s) has 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).

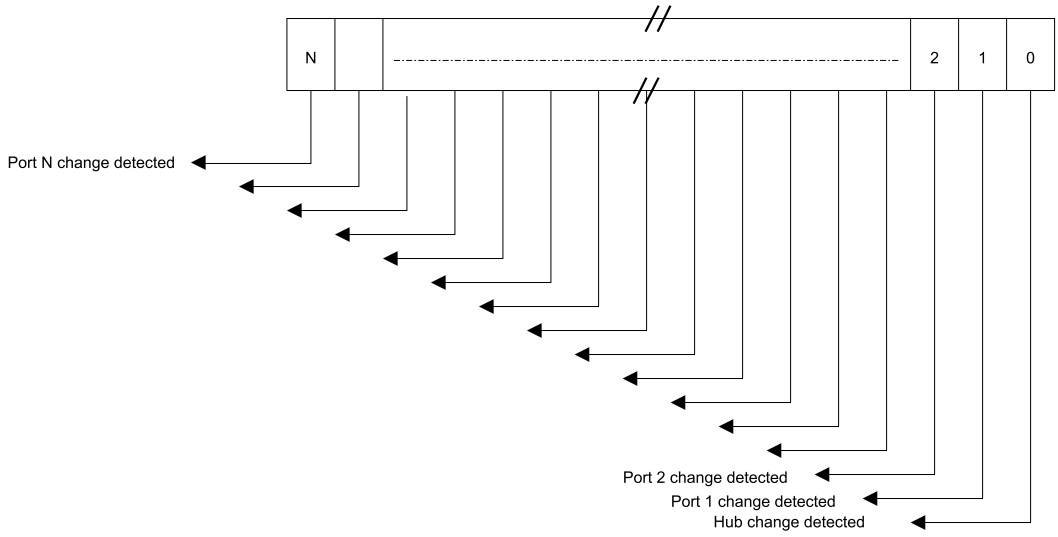



**Figure 11-22. Hub and Port Status Change Bitmap**

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 11-23 shows an example creation mechanism for hub and port change bits.

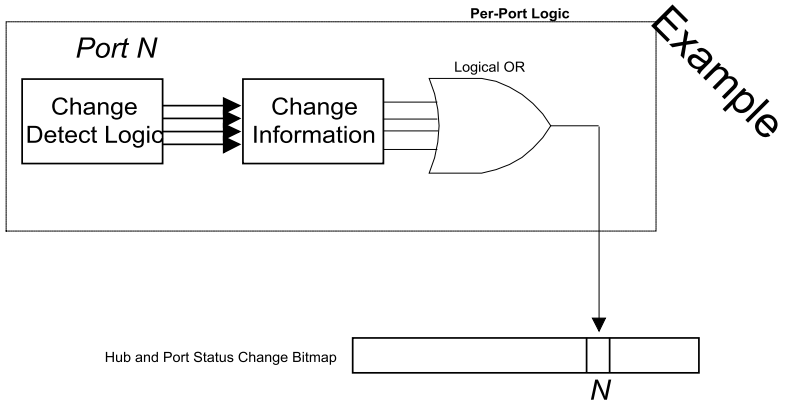



**Figure 11-23. Example Hub and Port Change Bit Sampling**

## 11.12.5 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 facing 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.

### 11.12.6 Enumeration Handling

The hub device class commands are used to manipulate its downstream facing 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.

The Get_Status(PORT) request invoked by the host will return a "not PORT_LOW_SPEED and PORT_HIGH_SPEED" indication for a downstream facing port operating at high-speed. The Get_Status(PORT) will report "PORT_LOW_SPEED" for a downstream facing port operating at low-speed. The Get_Status(PORT) will report "not PORT_LOW_SPEED and not PORT_HIGH_SPEED" for a downstream facing port operating at full-speed.

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.

## 11.13 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 facing 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 11-12 summarizes the information and how it can be used to determine the current power requirements of the hub.

**Table 11-12. Hub Power Operating Mode Summary**

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

## 11.14 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.

### 11.14.1 Overview

Figure 11-24 shows an overview of the Transaction Translator. The TT is responsible for participating in high-speed split transactions on the high-speed bus via its upstream facing port and issuing corresponding full-/low-speed transactions on its downstream facing ports that are operating at full-/low-speed. The TT acts as a high-speed function on the high-speed bus and performs the role of a host controller for its downstream facing ports that are operating at full-/low-speed. The TT includes a high-speed handler to deal with high-speed transactions. The TT also includes a full-/low-speed handler that performs the role of a host controller on the downstream facing ports that are operating at full-/low-speed.

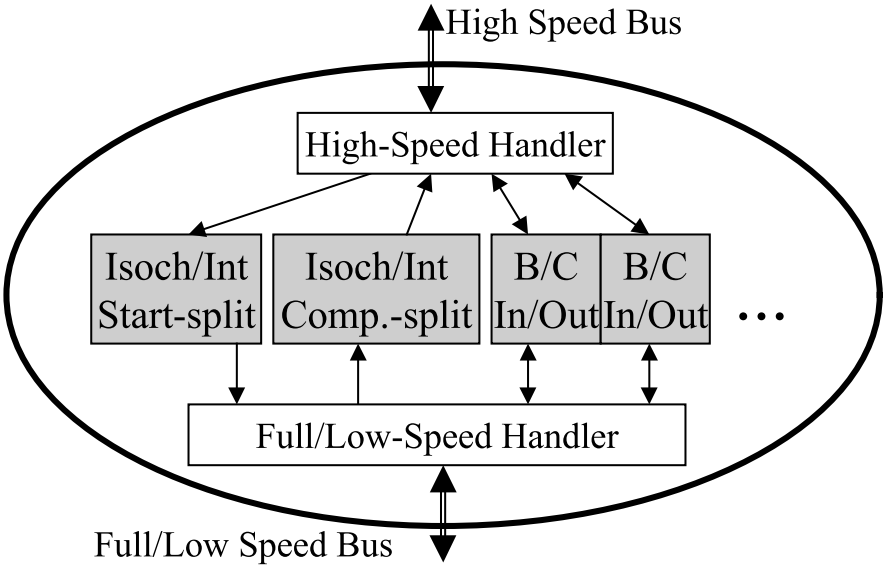



**Figure 11-24. Transaction Translator Overview**

The TT has buffers (shown in gray in the figure) to hold transactions that are in progress and tracks the state of each buffered transaction as it is processed by the TT. The buffers provide the connection between the high-speed and full-/low-speed handlers. The state tracking the TT does for each transaction depends on the specific USB transfer type of the transaction (i.e., bulk, control, interrupt, isochronous). 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 the high-speed bus.

The general conversion between full-/low-speed transactions and the corresponding high-speed split transaction protocol is described in Section 8.4.2. More details about the specific transfer types for split transactions are described later in this chapter.

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

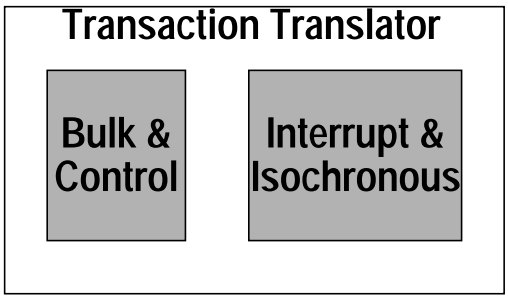



**Figure 11-25. Periodic and Non-periodic Buffer Sections of TT**

The TT has two buffer and state tracking sections (shown in gray in Figure 11-24 and Figure 11-25): 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.

#### 11.14.1.1 Data Handling Between High-speed and Full-/low-speed

The host converts transfer requests involving a full-/low-speed device into corresponding high-speed split transactions to the TT to which the device is attached.

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 token.

The host can have a single schedule of the transactions that need to be issued to devices. This single schedule can be used to hold both high-speed transactions and high-speed split transactions used for communicating with full-/low-speed devices.

#### 11.14.1.2 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 transactions 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-/full-/low-speed devices ever participate in split transactions.

When the host controller sends a start-split transaction at high-speed, the split transaction is addressed to the TT for that device. That TT 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 transaction buffer.

The full-/low-speed handler processes start-split periodic transactions stored in the periodic transaction buffer (in order) as the downstream full-/low-speed bus is ready for the “next” transaction. The full-/low-speed handler accepts any result information from the downstream bus (in response to the full-/low-speed transaction) and accumulates it 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/result 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 appropriate local transaction buffer and responds accordingly. The specific split transaction sequences are defined for each USB transfer type in later sections.

#### 11.14.1.3 Multiple Transaction Translators

A hub has two choices for organizing transaction translators (TTs). A hub can have one TT 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 its organization in the hub class descriptor.

### 11.14.2 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 transaction buffers differently than the non-periodic transaction buffers.

#### 11.14.2.1 TT Isochronous/Interrupt (Periodic) Transaction 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 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. An overview of the microframe pipeline of buffering in the TT is shown in Figure 11-26. A transaction begins as a start-split on the high-speed bus, is accepted by the high-speed handler, and is stored in the start-split transaction buffer. The full-/low-speed handler uses the next start-split transaction at the head of the start-split transaction buffer when it is time to issue the next periodic full-/low-speed transaction on the downstream bus. The results of the transaction are accumulated in the complete-split transaction buffer. The TT responds to a complete-split from the host and extracts the appropriate response from the complete-split transaction buffer. This completes the flow for a periodic transaction through the TT. This is called the periodic transaction pipeline.

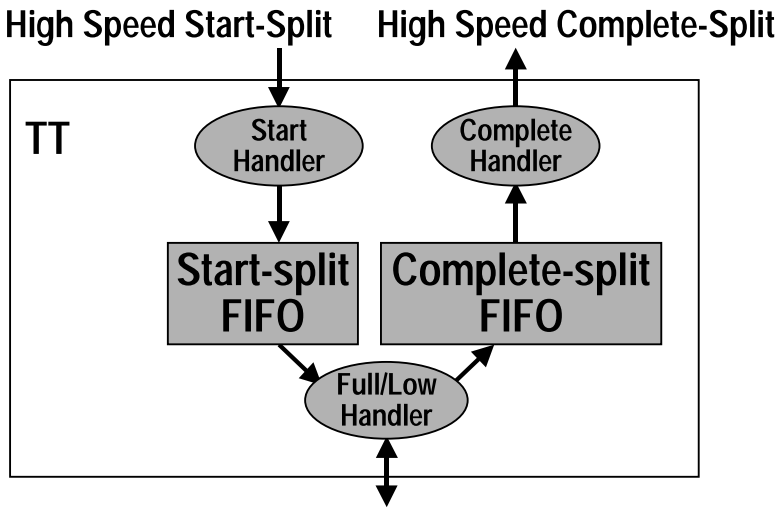



**Figure 11-26. TT Microframe Pipeline for Periodic Split Transactions**

The TT implements a traditional pipeline of transactions with its periodic transaction buffers. There is separate buffer space for start-splits and complete-splits. The host is responsible for filling the start-split transaction buffer and draining the complete-split transaction buffer. The host software manages the host controller to cause high-speed split transactions at the correct times to avoid over/under runs in the TT periodic transaction buffers. 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 transaction buffer space required in the TT is the minimum possible. See Section 11.18 for more detailed information.

USB strictly defines the timing requirements of periodic transactions 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 transactions 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 TT 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 TT to keep the device's endpoints operating normally.

### 11.14.2.2 TT Bulk/Control (Non-Periodic) Transaction Buffering

Non-periodic transactions have no timing requirements, but the TT supports the maximum full-/low-speed throughput allowed. A TT provides a few transaction buffers for bulk/control full-/low-speed transactions. The host and TT use simple flow control (NAK) mechanisms to manage the bulk/control non-periodic transaction 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. Figure 11-27 shows an example overview of a TT that has two bulk/control buffers.

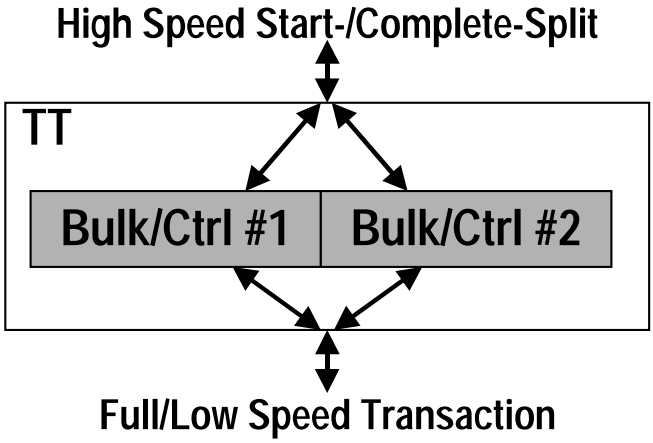



**Figure 11-27. TT Nonperiodic Buffering**

### 11.14.2.3 Full-/low-speed Handler Transaction Scheduling

The full-/low-speed handler uses a simple, scheduled priority scheme to service pending transactions on the downstream bus. Whenever the full-/low-speed handler finishes a transaction on the downstream bus, it takes the next start-split transaction from the start-split periodic transaction buffer (if any). If there are no available start-split periodic transactions in the buffer, the full-/low-speed handler may attempt a bulk/control transaction. If there are start-split transactions pending in the bulk/control buffer(s) and there is sufficient time left in the full-/low-speed 1 ms frame to complete the transaction, the full-/low-speed handler issues one of the bulk/control transactions (in round robin order). Figure 11-28 shows pseudo code for the full-/low-speed handler start-split transaction scheduling algorithm.

The TT also sequences the transaction pipeline based on the high-speed microframe timer to ensure that it does not start full-/low-speed periodic transactions too early or too late. The "Advance_pipeline" procedure in the pseudo code is used to keep the TT advancing the microframe "pipeline". This procedure is described in more detail later in Figure 11-67.

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

   Advance_Pipeline();  -- see description in Figure 11-67(below)
End loop
```

**Figure 11-28. Example Full-/low-speed Handler Scheduling for Start-splits**

As described earlier in this chapter, the TT derives the downstream bus's 1 ms SOF timer from the high-speed 125 μs microframe. This means that the host and the TT have the same 1 ms frame time for all TTs. Given the strict relationship between frames and the zeroth microframe, there is no need to have any explicit timing information carried in the periodic split transactions sent to the TT. See Section 11.18 for more information.

## 11.15 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. The diagrams should not be taken as a required implementation, but to specify the required behavior. Appendix A includes example high-speed and full-speed transaction sequences with different results to clarify the relationships between the host controller, the TT, and a full-speed endpoint.

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

For each data transfer direction, reference figures also show the possible flow sequences for the start-split and the complete-split portion of each split transaction transfer type.

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 examples in Appendix A. 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 11-29. The flow sequences also include some indication of states required in the host or TT or actions taken. The legend shown in Figure 11-29 indicates how these are identified.

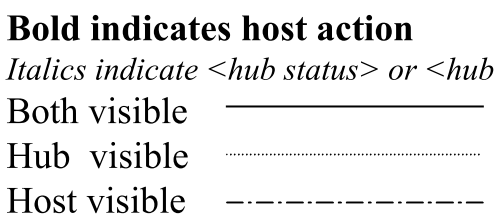



**Figure 11-29. Flow Sequence Legend**

Figure 11-30 shows the legend for the state machine diagrams. A circle with a three line border indicates a reference to another (hierarchical) state machine. A circle with a two line border indicates an initial state. A circle with a single line border is a simple state.

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

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

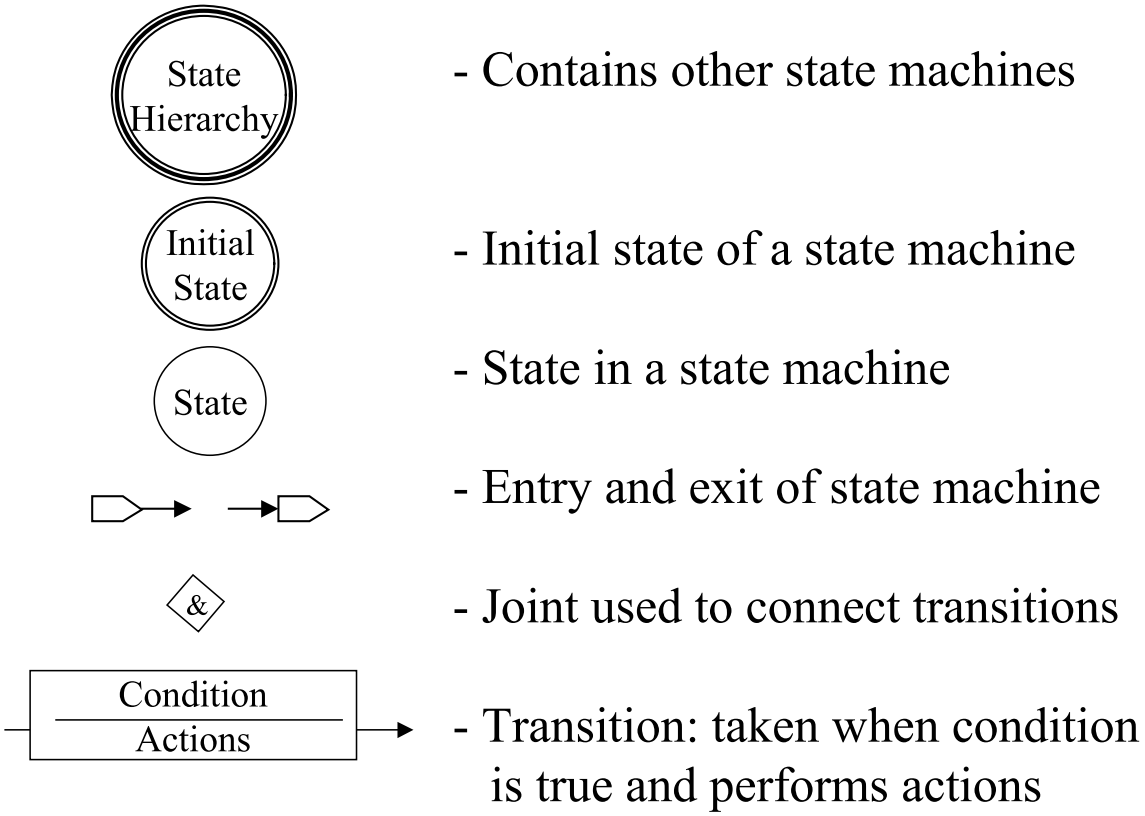



**Figure 11-30. Legend for State Machines**

The descriptions of the split transactions for the four transfer types refer to the status of the full-/low-speed transaction on the bus downstream of the TT. 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 /trans_err – The full-/low-speed transaction experienced a error in the transaction. Possible errors are: PID to PID_invert bits check failure, CRC5 check failure, incorrect PID, timeout, CRC16 check failure, incorrect packet length, bitstuffing error, false EOP.
  - 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 /lastdata – A data packet was finished being received. (isochronous/interrupt IN).
- Ready /moredata – A data packet was being received when the microframe timer 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 Ready status. This is the initial state for a buffer before it has been used for a transaction.
- Pending – The transaction is waiting to be completed on the downstream facing full-/low-speed bus.

The figures use “old/x” and “ready/x” to indicate any of the old or ready status respectively.

The split transaction state machines in the remainder of this chapter are presented in the context of Figure 11-31. The host controller state machines are located in the host controller. The host controller causes packets to be issued downstream (labeled as HSD1) and it receives upstream packets (labeled as HSU2).

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

The host controller has commands that tell it what split transaction to issue next for an endpoint. The host controller tracks transactions for several endpoints. The TT has state in buffers that track transactions for several endpoints.

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

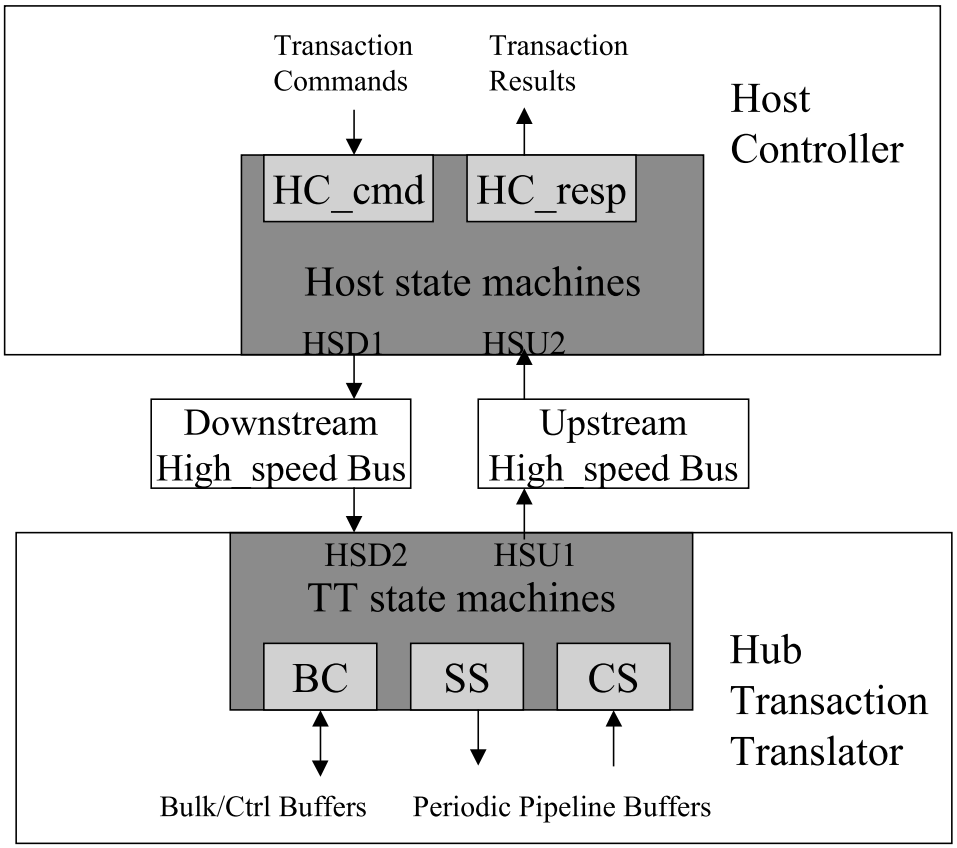



**Figure 11-31. State Machine Context Overview**

## 11.16 Common Split Transaction State Machines

There are several state machines common to all the specific split transaction types. These state machines are used in the host controller and transaction translator to determine the specific split transaction type (e.g., interrupt OUT start-split vs. bulk IN complete-split). An overview of the host controller state machine hierarchy is shown in Figure 11-32. The overview of the transaction translator state machine hierarchy is shown in Figure 11-33. Each of the labeled boxes in the figures show an individual state machine. Boxes contained in another box indicate a state machine contained within another state machine. All the state machines except the lowest level ones are shown in the remaining figures in this section. The lowest level state machines are shown in later sections describing the specific split transaction type.

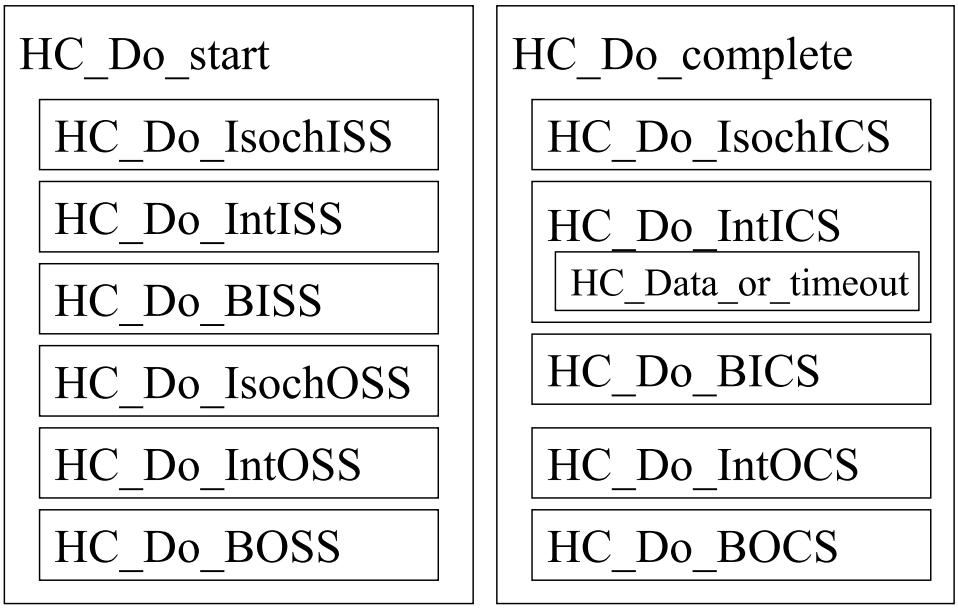



**Figure 11-32. Host Controller Split Transaction State Machine Hierarchy Overview**

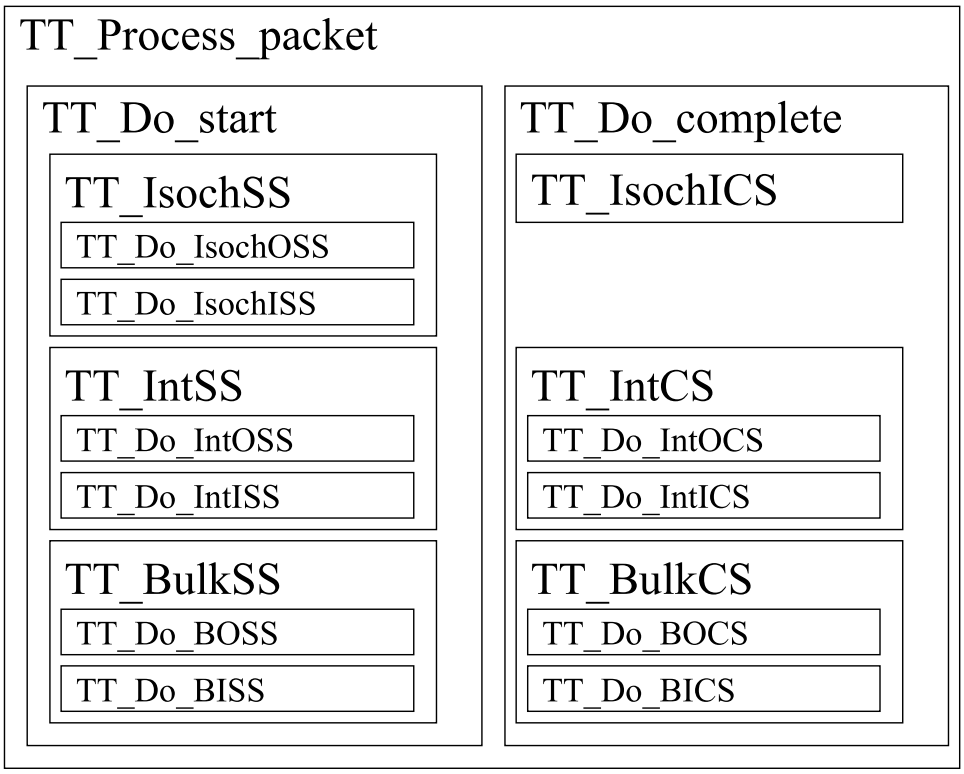



**Figure 11-33. Transaction Translator State Machine Hierarchy Overview**

## 11.16.1 Host Controller State Machine

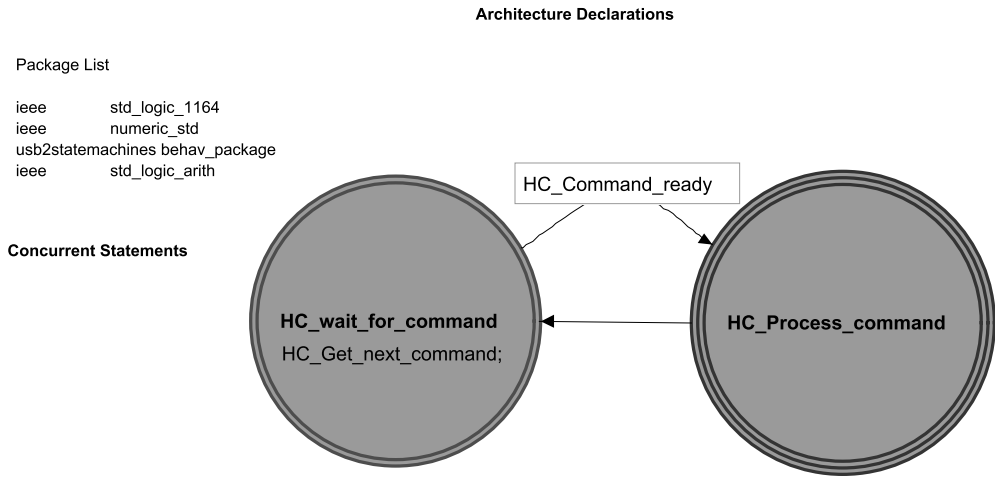



**Figure 11-34. Host Controller**

### 11.16.1.1 HC_Process_command State Machine

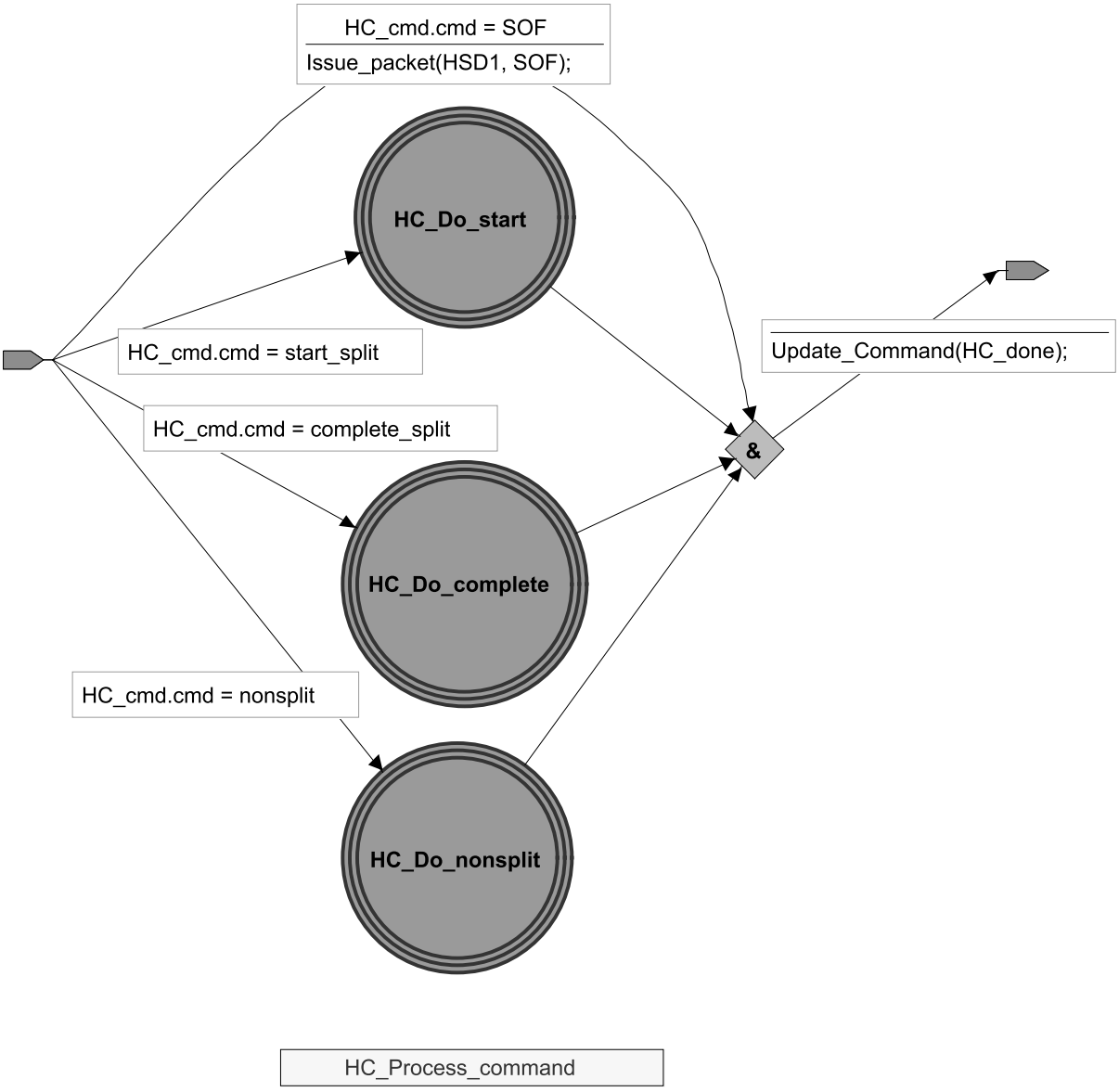



**Figure 11-35. HC_Process_Command**

#### 11.16.1.1.1 HC_Do_start State Machine

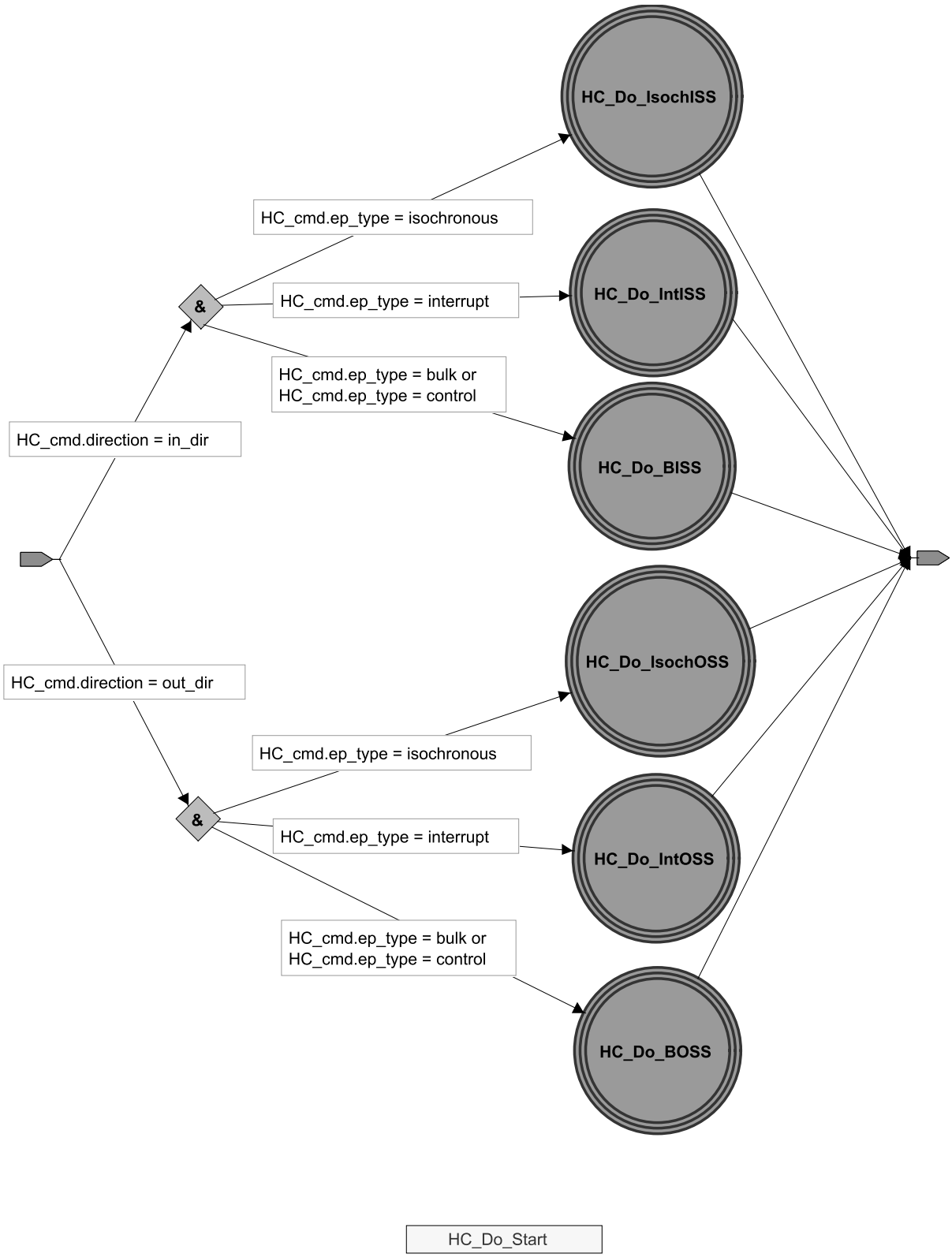



**Figure 11-36. HC_Do_Start**

#### 11.16.1.1.2 HC_Do_complete State Machine

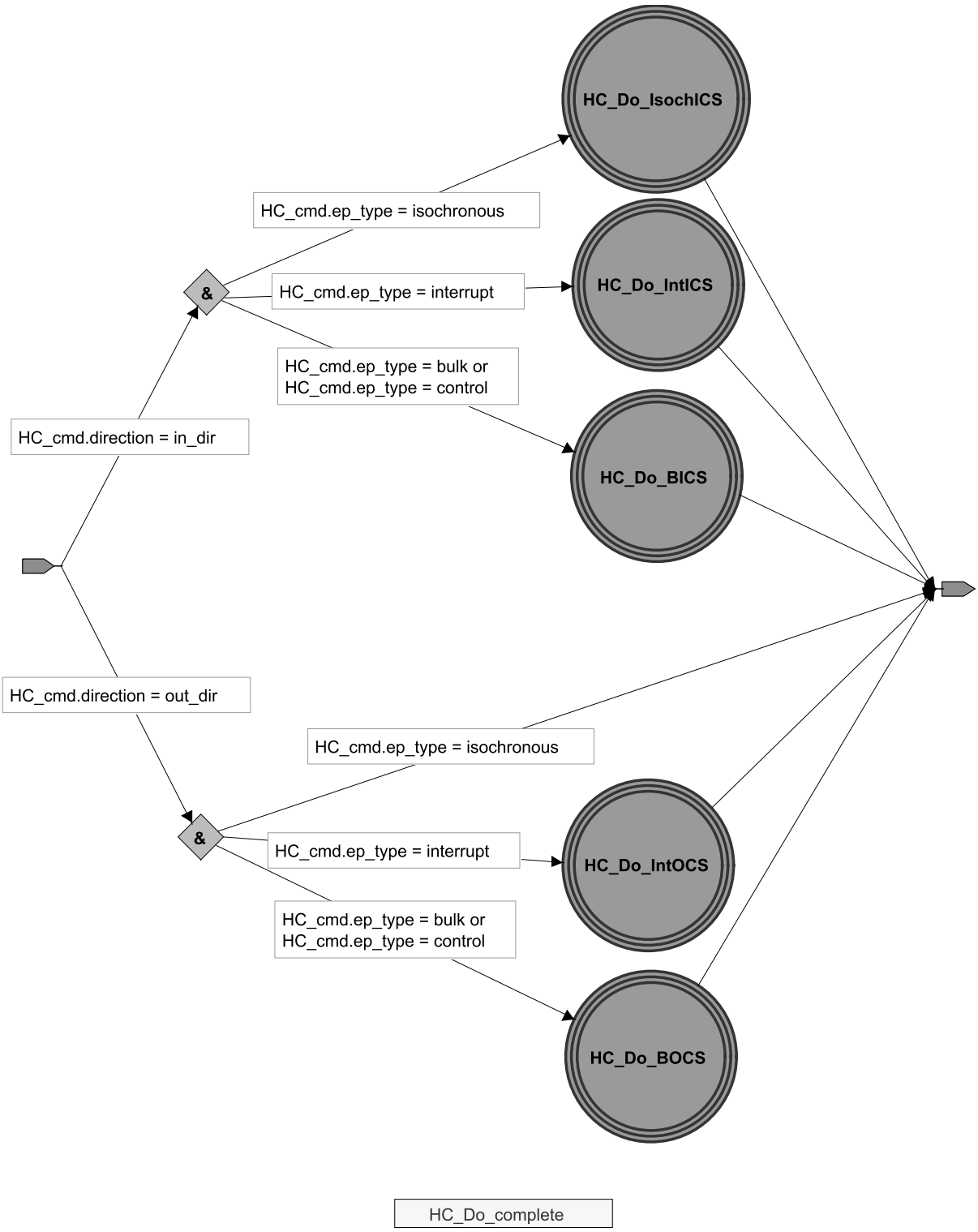



**Figure 11-37. HC_Do_Complete**

## 11.16.2 Transaction Translator State Machine

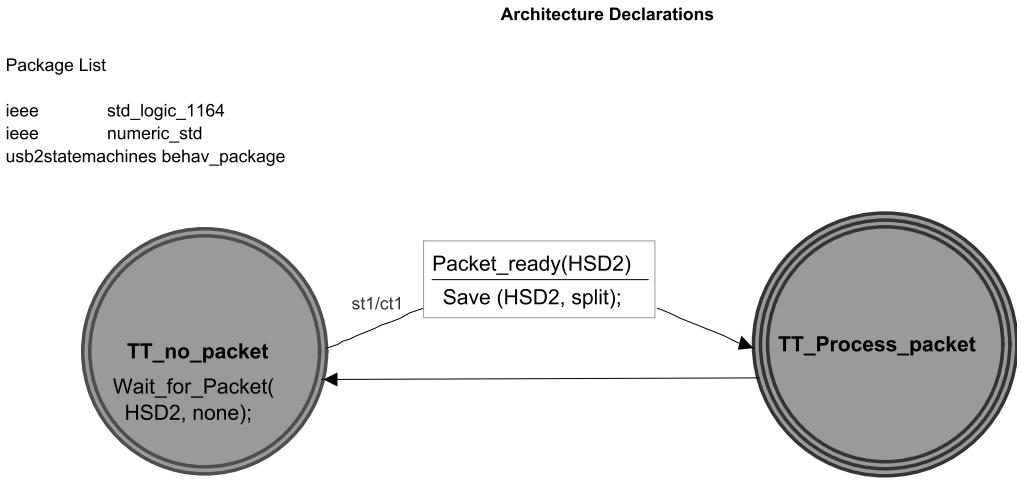



**Figure 11-38. Transaction Translator**

### 11.16.2.1 TT_Process_packet State Machine

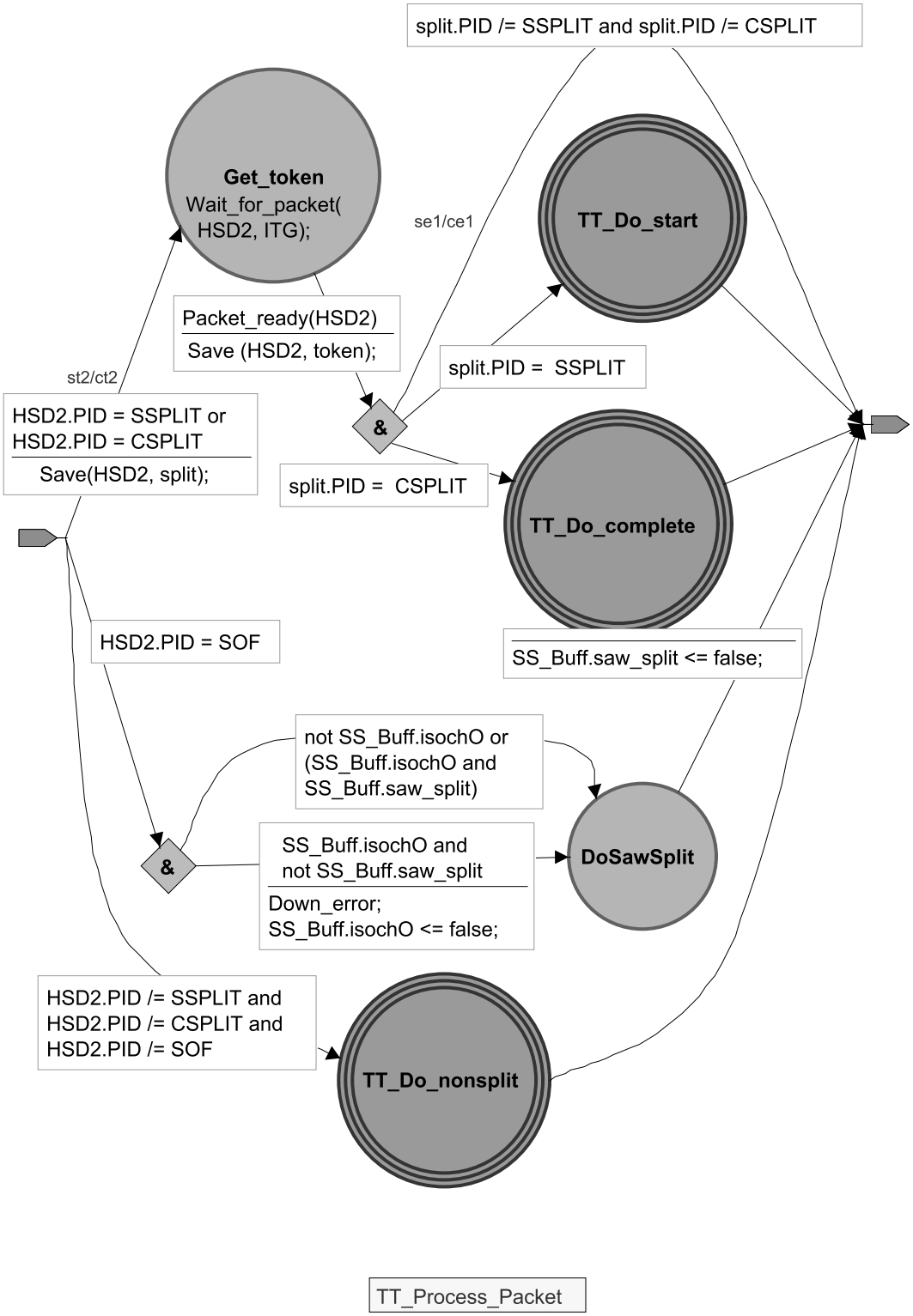



**Figure 11-39. TT_Process_Packet**

#### 11.16.2.1.1 TT_Do_Start State Machine

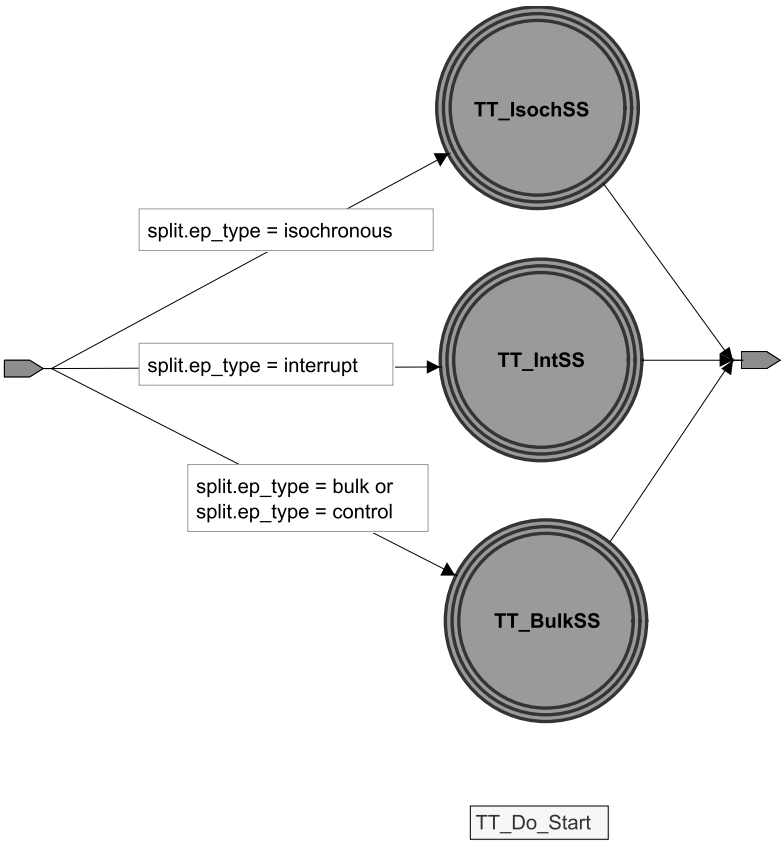



**Figure 11-40. TT_Do_Start**

#### 11.16.2.1.2 TT_Do_Complete State Machine

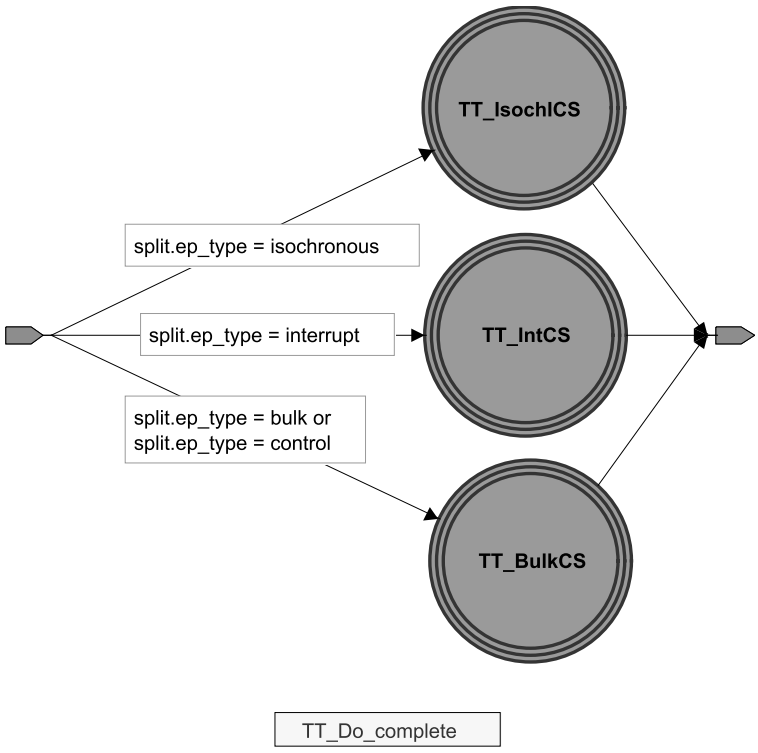



**Figure 11-41. TT_Do_Complete**

#### 11.16.2.1.3 TT_BulkSS State Machine

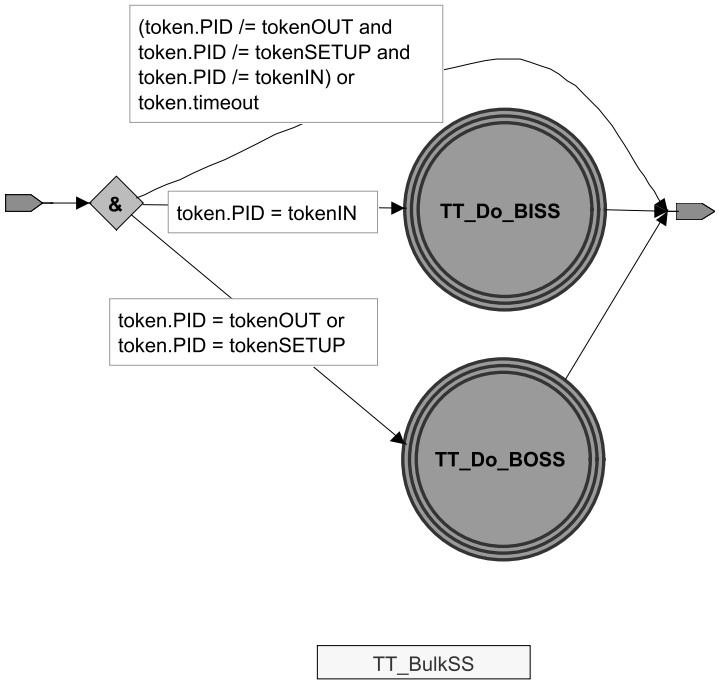



**Figure 11-42. TT_BulkSS**

#### 11.16.2.1.4 TT_BulkCS State Machine

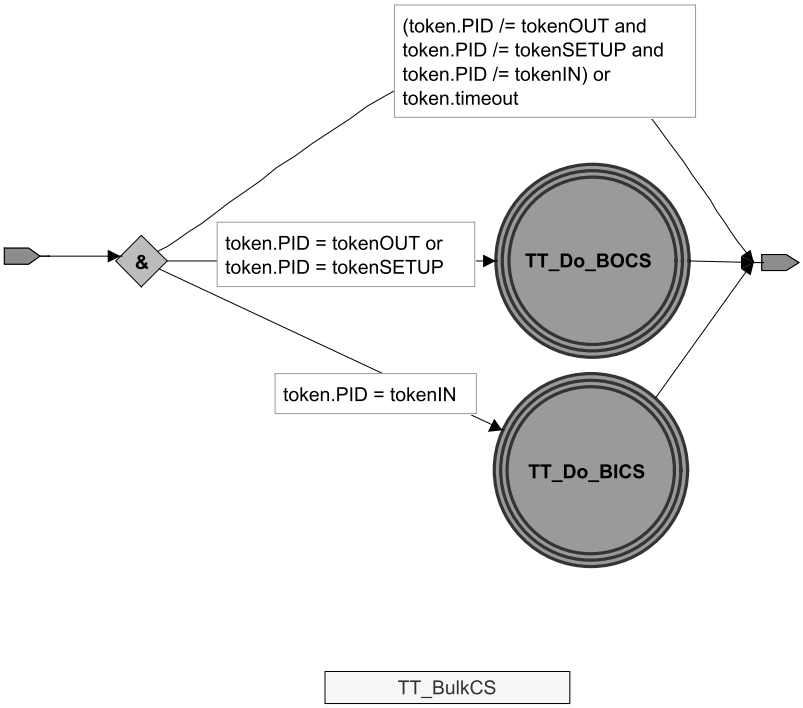



**Figure 11-43. TT_BulkCS**

#### 11.16.2.1.5 TT_IntSS State Machine

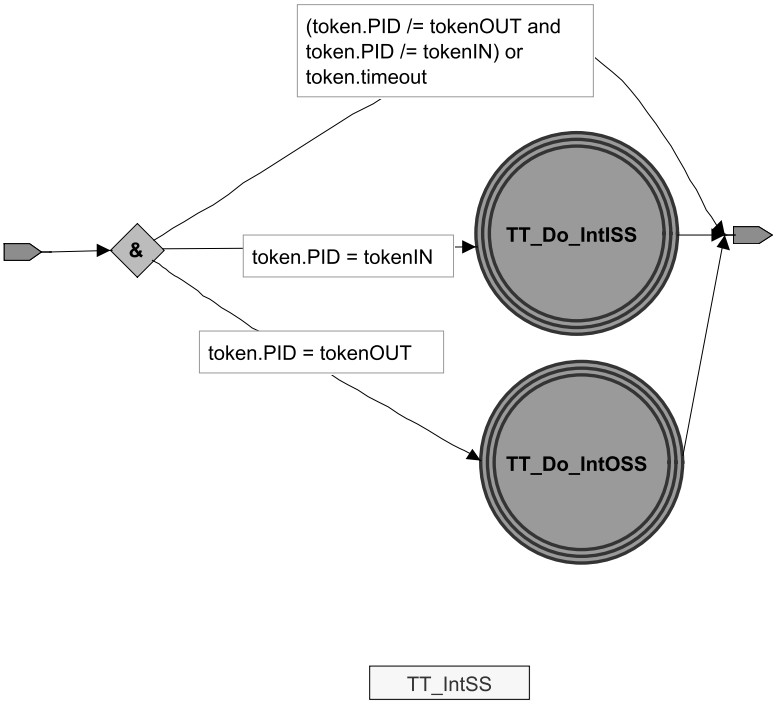



**Figure 11-44. TT_IntSS**

#### 11.16.2.1.6 TT_IntCS State Machine

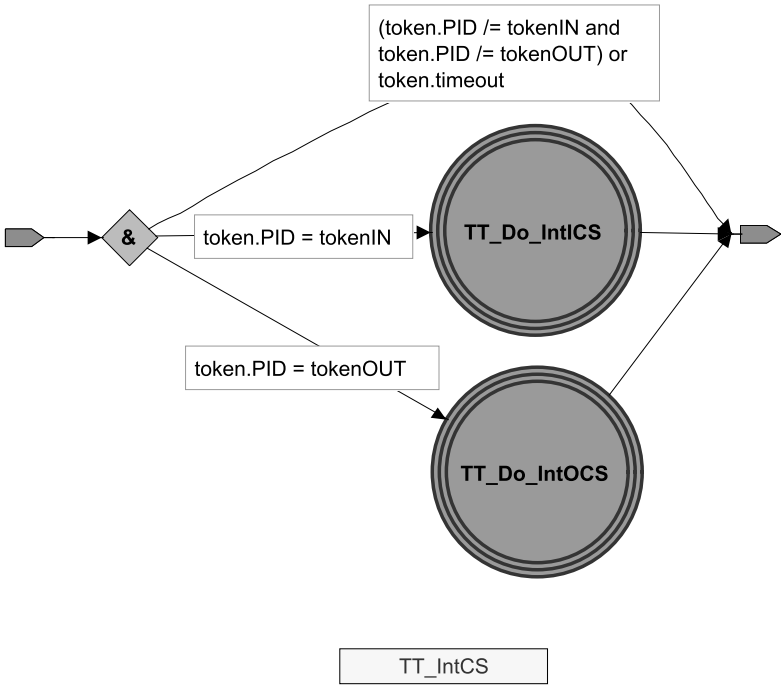



**Figure 11-45. TT_IntCS**

#### 11.16.2.1.7 TT_IsochSS State Machine

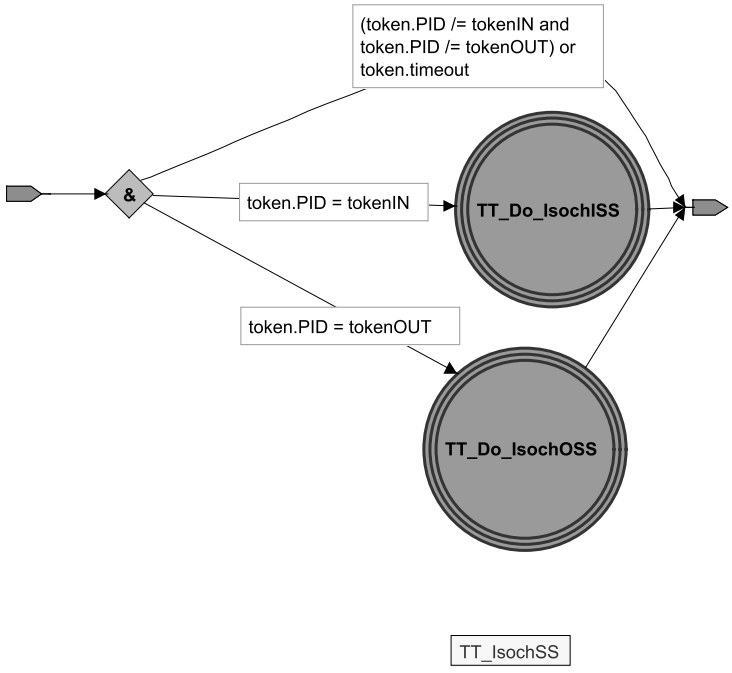



**Figure 11-46. TT_IsochSS**

## 11.17 Bulk/Control Transaction Translation Overview

Each TT must have 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 (or has completed) 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 handshake/result of the full-/low-speed transaction (and data for an IN). This buffer is filled and emptied by split transactions from the high-speed bus via the high-speed handler. The buffer is also updated by the full-/low-speed handler while the transaction is in progress on the downstream bus.

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

The host controller attempts a start-split transaction according to its bulk/control high-speed transaction schedule. As soon as the high-speed handler responds to a complete-split transaction with the results from the corresponding buffer, the next start-split for some (possibly other) full-/low-speed endpoint can be saved in the buffer.

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

The host controller does not need to 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 transaction or a high-speed split transaction.

The following sections describe the details of the transaction phases, 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 split transactions. In the figures, some areas are shaded to indicate that they do not apply for control transactions.

### 11.17.1 Bulk/Control Split Transaction Sequences

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. They define the required sequencing behavior of different packets of a bulk/control split transaction. In particular, other high-speed or split transactions for other endpoints occur before or after these split transaction sequences.

Figure 11-47 shows a sample code algorithm that describes the behavior of the transitions labeled with *Is_new_SS*, *Is_old_SS* and *Is_no_space* shown in the figures for both bulk/control IN and OUT start-split transactions buffered in the TT for any endpoint. This algorithm ensures that the TT only buffers a single bulk/control split transaction for any endpoint. The complete-split protocol definition requires an endpoint has only a single result buffered in the TT at any time. Note that the “buffer match” test is different for bulk and control endpoints. A buffer match test for a bulk transaction must include the direction of the transaction in the test since bulk endpoints are unidirectional. A control transaction must not use direction as part of the match test.

```
procedure Compare_buffs IS
      variable match:boolean:=FALSE;
begin
--
--  Is_new_SS is true when BC_buff.status == NEW_SS
--  Is_old_SS is true when BC_buff.status == OLD_SS
--  Is_no_space is true when BC_buff.status == NO_SPACE
--
       -- Assume nospace and intialize index to 0.
       BC_buff.status := NO_SPACE;
       BC_buff.index  := 0;

       FOR i IN 0 to num_buffs-1 LOOP
           IF NOT match THEN
               -- Re-use buffer with same Device Address/End point.
               IF (token.endpt = cam(i).store.endpt AND
                   token.dev_addr = cam(i).store.dev_addr AND
                   ((token.direction = cam(i).store.direction AND
                     split.ep_type /= CONTROL) OR
                    split.ep_type = CONTROL)) THEN

                 -- If The buffer is already pending/ready this must be a retry.
                 IF (cam(i).match.state = READY OR cam(i).match.state = PENDING) THEN
                     BC_buff.status := OLD_SS;
                 ELSE
                     BC_buff.status := NEW_SS;
                 END IF;
                 BC_buff.index := i;
                 match := TRUE;

               -- Otherwise use the buffer if it's old.
               ELSIF (cam(i).match.state = OLD) THEN
                 BC_buff.status := NEW_SS;
                 BC_buff.index := i;
               END IF;
           END IF;
       END LOOP;

    end Compare_buffs;
```

**Figure 11-47. Sample Algorithm for Compare_buffs**

Figure 11-48 shows the sequence of packets for a start-split transaction for the full-/low-speed bulk OUT transfer type. The block labeled SSPLIT represents a split transaction 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 SSPLIT or OUT token packets, and does not receive the following OUT/SETUP or DATA0/1 packets, it will not respond with a handshake as indicated by the dotted line transitions labeled "se1" or "se2". This causes the host to subsequently see a transaction error (timeout) (labeled "se2" and indicated with a dashed line). If the high-speed handler receives the DATA0/1 packet and it fails the CRC check, it takes the transition "se2" which causes the host to timeout and follow the "se2" transition.

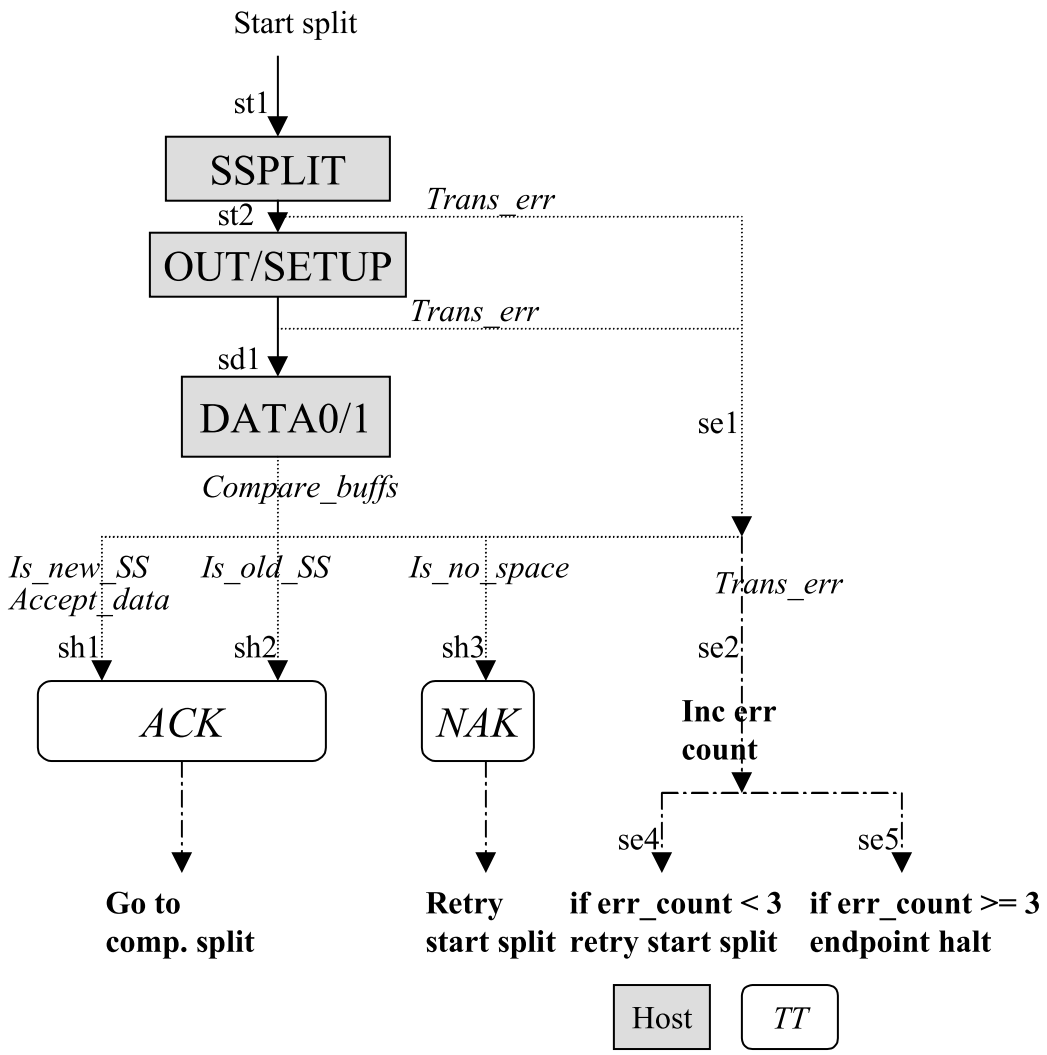



**Figure 11-48. Bulk/Control OUT Start-split Transaction Sequence**

The host must keep retrying the start-split for this endpoint until the err_count reaches three for this endpoint before continuing on to some other start-split for this endpoint. However, the host can issue other start-splits for other endpoints before it retries the start-split for this endpoint. The err_count is used to count how many errors have been experienced during attempts to issue a particular transaction for a particular endpoint.

If there is no space in the transaction buffers to hold the start-split, the high-speed handler 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. The host does not increase its err_count for a NAK handshake response. Once the host has received a NAK response to a start-split, it can skip other start-splits for this TT for bulk/control endpoints until it finishes a bulk/control complete-split.

If there is buffer space for the start-split, the high-speed handler 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 or ready, 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. That is, it increments an error count (err_count) and, if the count is less than three (transition “se4”), it will retry the transaction. If the err_count is greater or equal to three (transition “se5”), the host controller does endpoint halt processing and does not retry the transaction. If for some reason, a host memory or non-USB bus delay (e.g.,a system memory “hold off”) occurs that causes the transaction to not be completed normally, the err_count must not be incremented. Whenever a transaction completes normally, the err_count is reset to zero.

The high-speed handler in the TT has no immediate knowledge of what the host sees, so the “se2”, “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 11-49 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 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 bus, the 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.

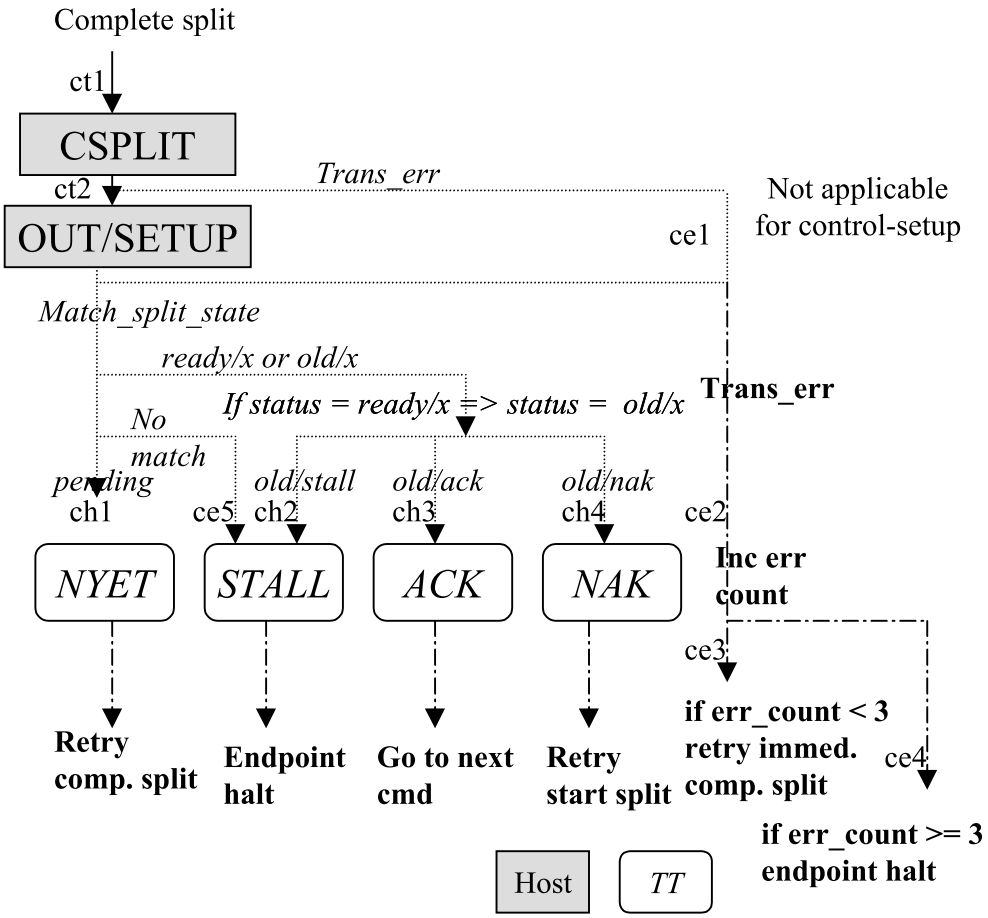



**Figure 11-49. Bulk/Control OUT Complete-split Transaction Sequence**

There is no timeout response status for a transaction because the full-/low-speed handler must perform a local retry of a full-/low-speed bulk or control transaction that experiences a transaction error. It locally implements a "three strikes and you're out" retry mechanism. This means that the full-/low-speed transaction will resolve to one of a NAK, STALL or ACK handshake results. If the transaction experiences a transaction error three times, the full-/low-speed handler will reflect this as a stall status result. The full-/low-speed handler must not do a local retry of the transaction in response to an ACK, NAK, or STALL handshake.

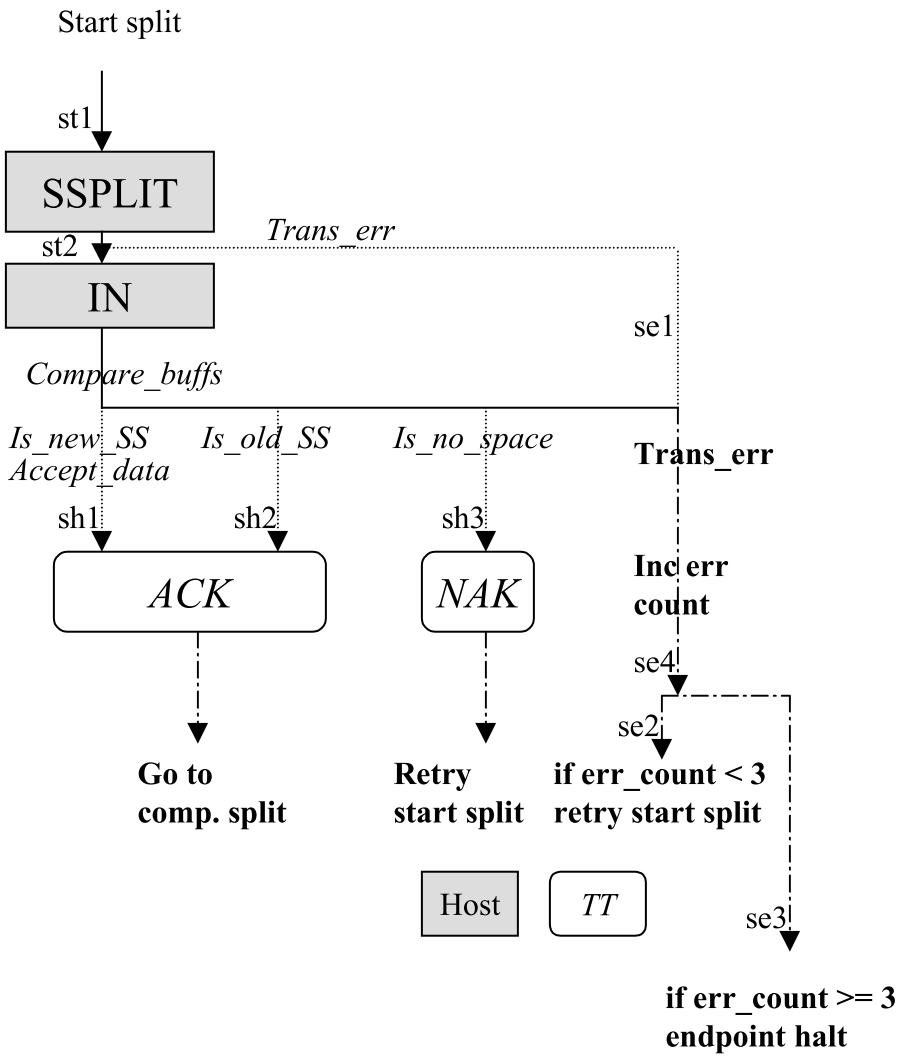



**Figure 11-50. Bulk/Control IN Start-split Transaction Sequence**

If the high-speed handler receives the complete-split token packet (and the token packet) while 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 receives a complete-split 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 saves 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 does not receive a valid handshake, it immediately retries the complete-split before going on to any other bulk/control transactions for this TT. The normal "three strikes" mechanism applies here also for the host; i.e., the err_count is incremented. If for some reason, a host memory or non-USB bus delay (e.g., a system memory "hold off") occurs that causes the transaction to not be completed normally, the err_count must not be incremented.

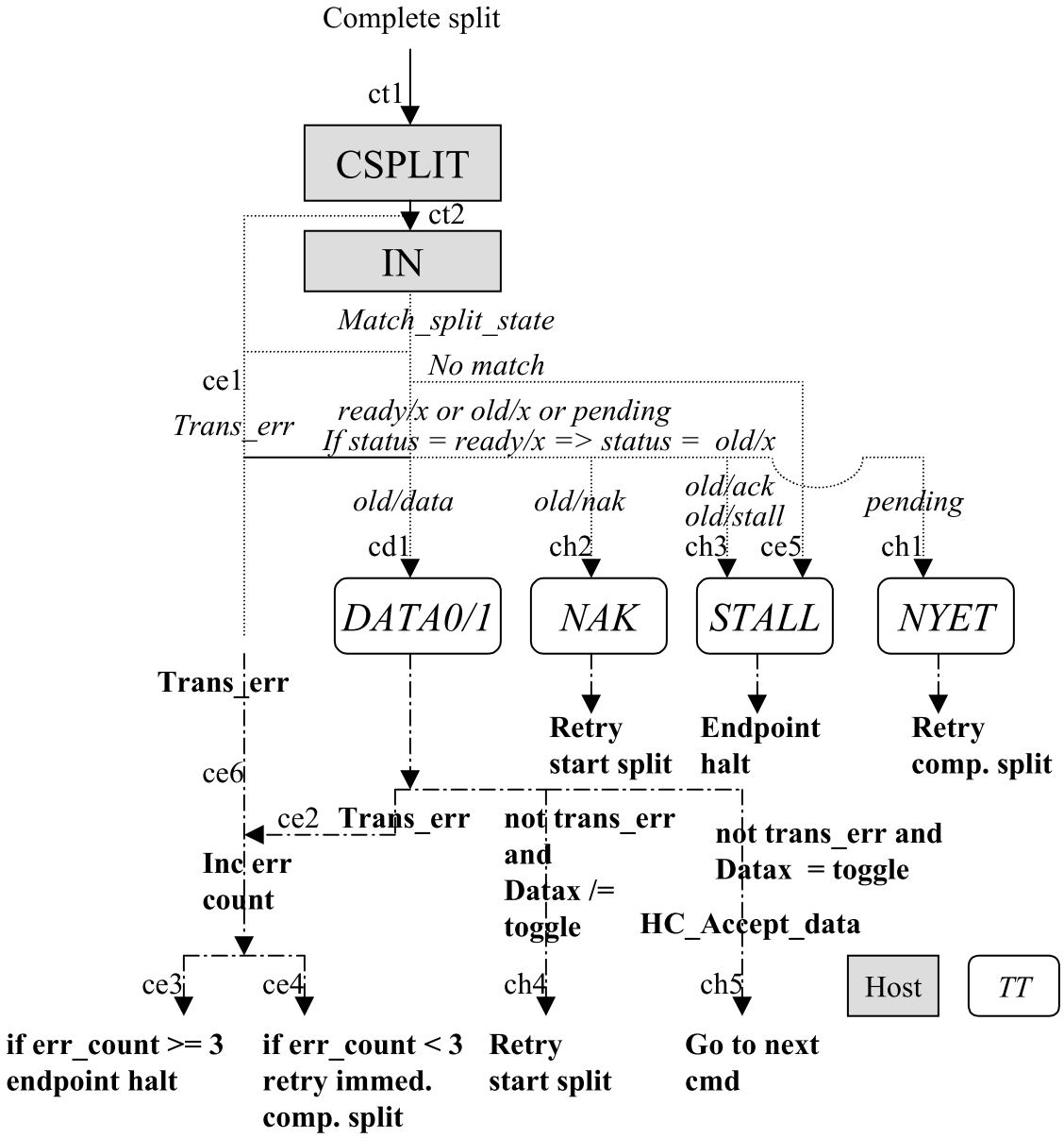



**Figure 11-51. Bulk/Control IN Complete-split Transaction Sequence**

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

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 the 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. The host must not increment the err_count in this case.

The host must keep retrying the current start-split until the err_count reaches three for this endpoint before proceeding to the next split transaction for this endpoint. However, the host can issue other start-splits for other endpoints before it retries the start-split for this endpoint.

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 11-50 and Figure 11-51 show the corresponding flow sequences for bulk/control IN split transactions.

## 11.17.2 Bulk/Control Split Transaction State Machines

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

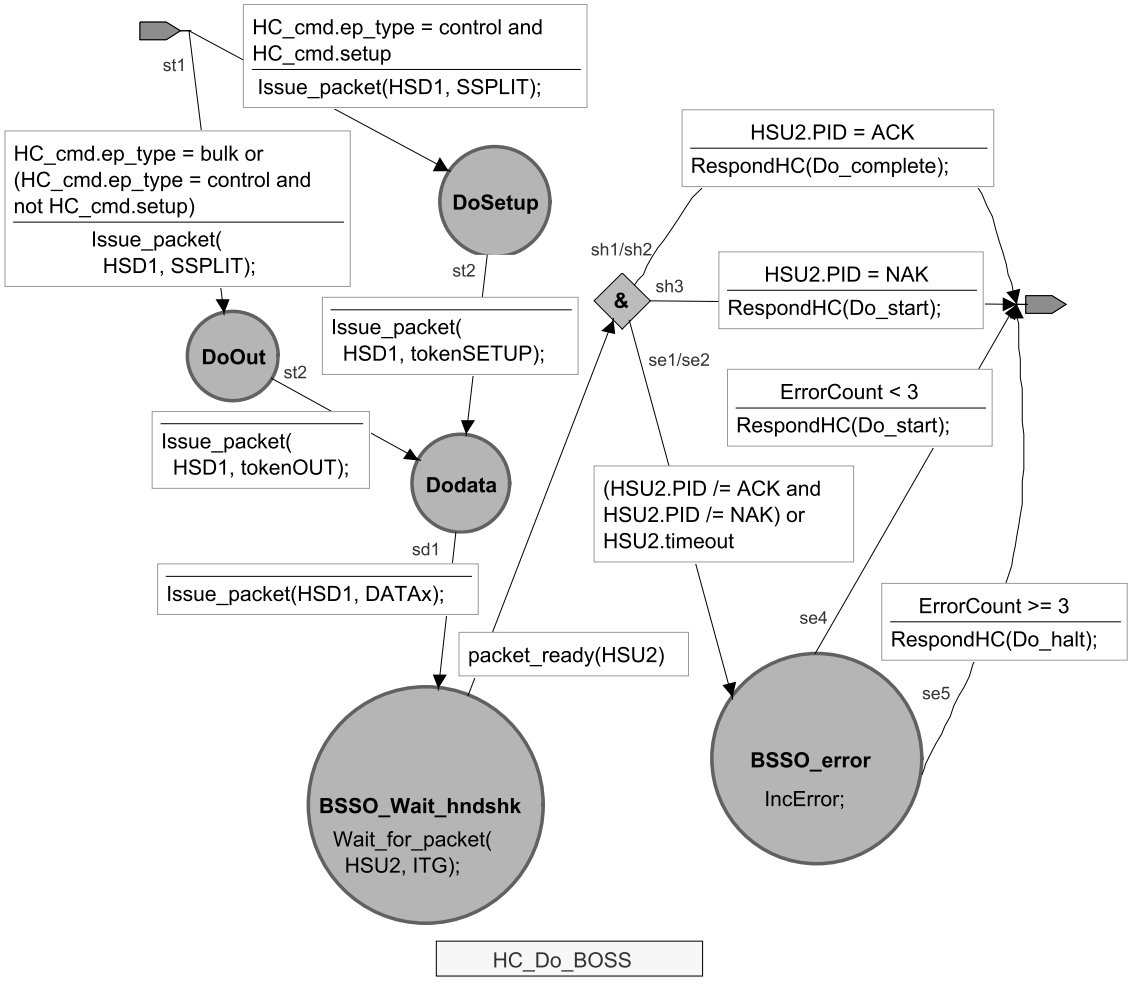



**Figure 11-52. Bulk/Control OUT Start-split Transaction Host State Machine**

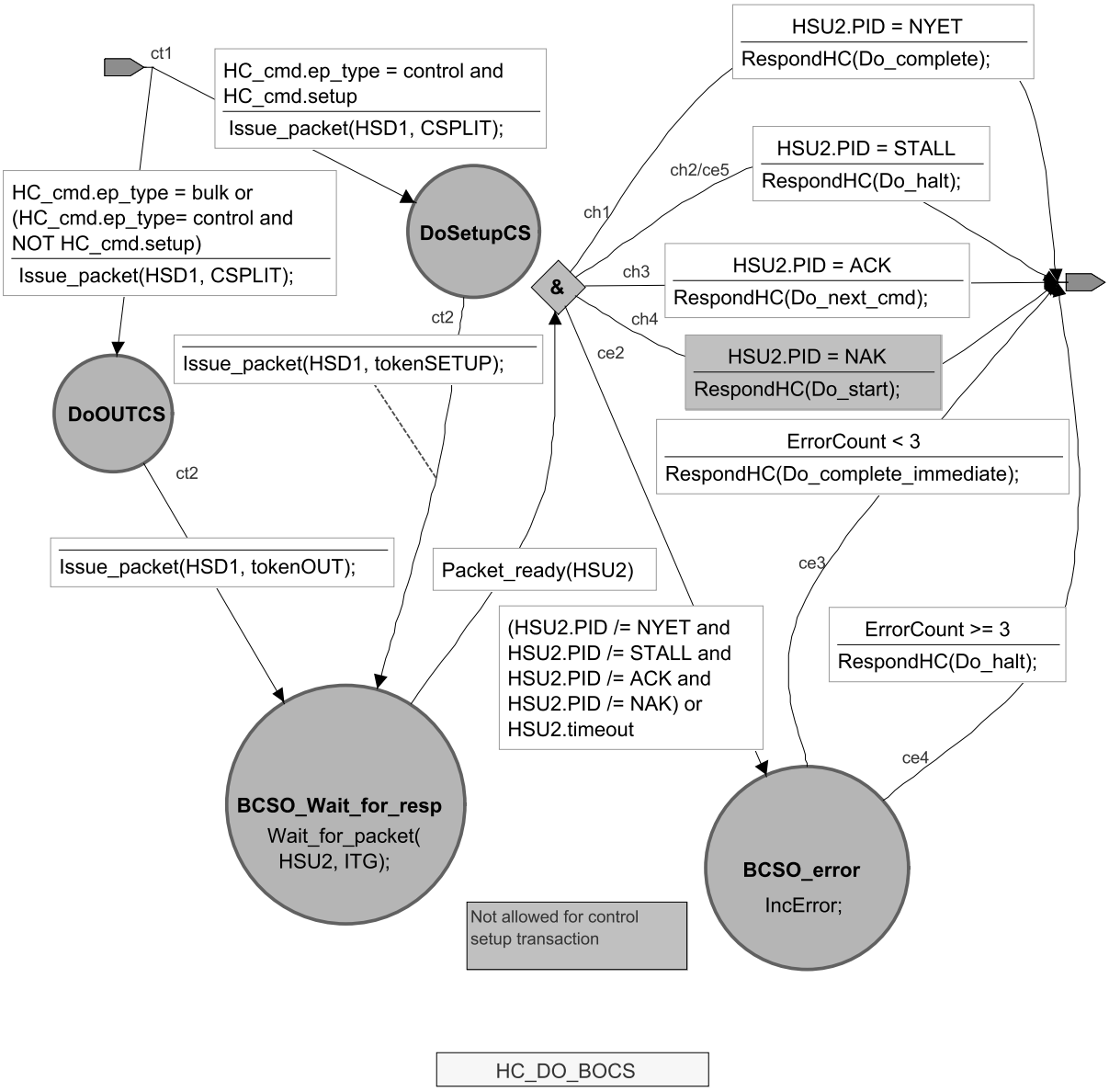



**Figure 11-53. Bulk/Control OUT Complete-split Transaction Host State Machine**

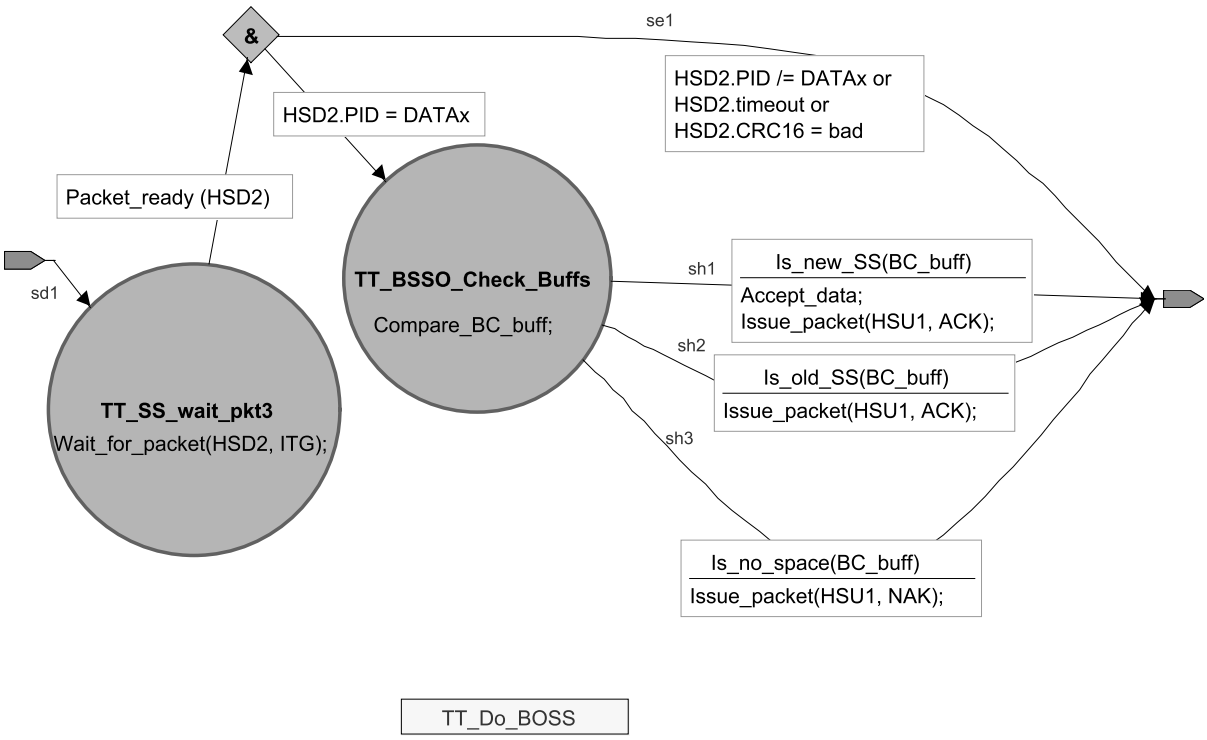



**Figure 11-54. Bulk/Control OUT Start-split Transaction TT State Machine**

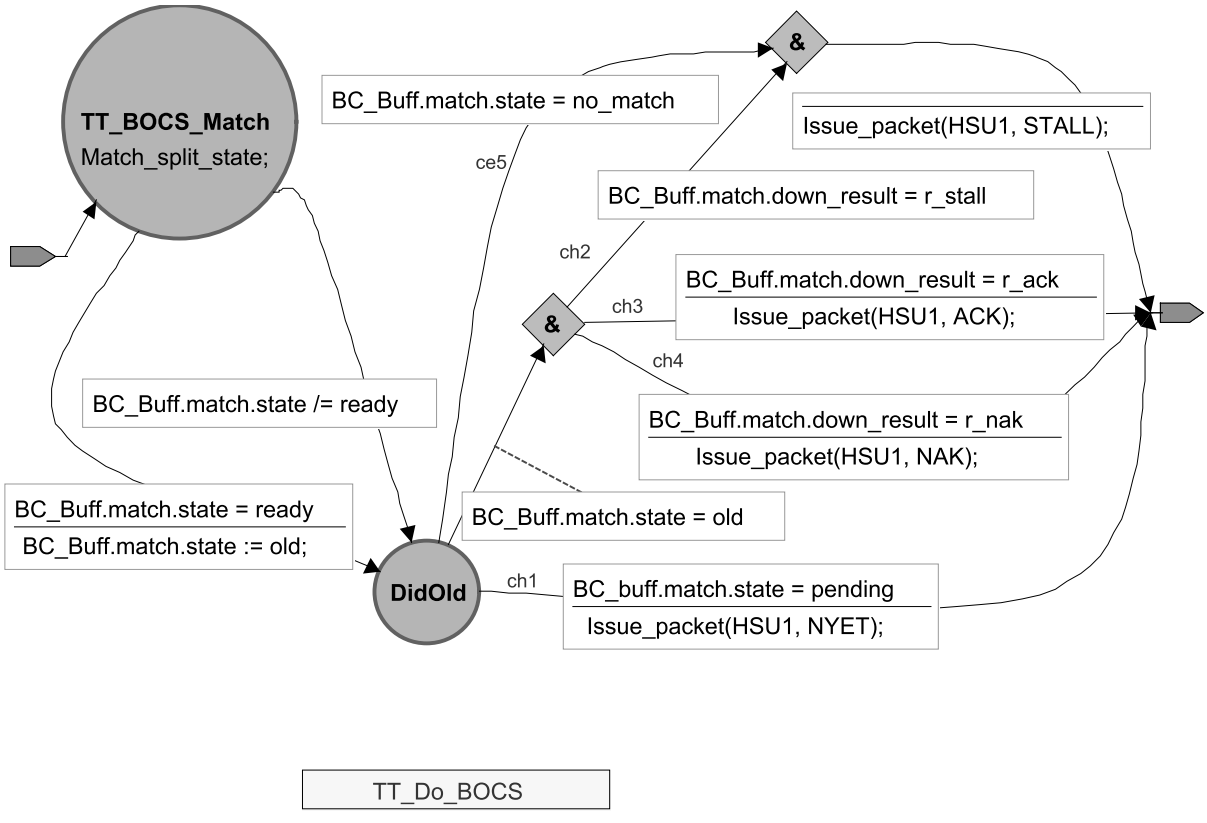



**Figure 11-55. Bulk/Control OUT Complete-split Transaction TT State Machine**

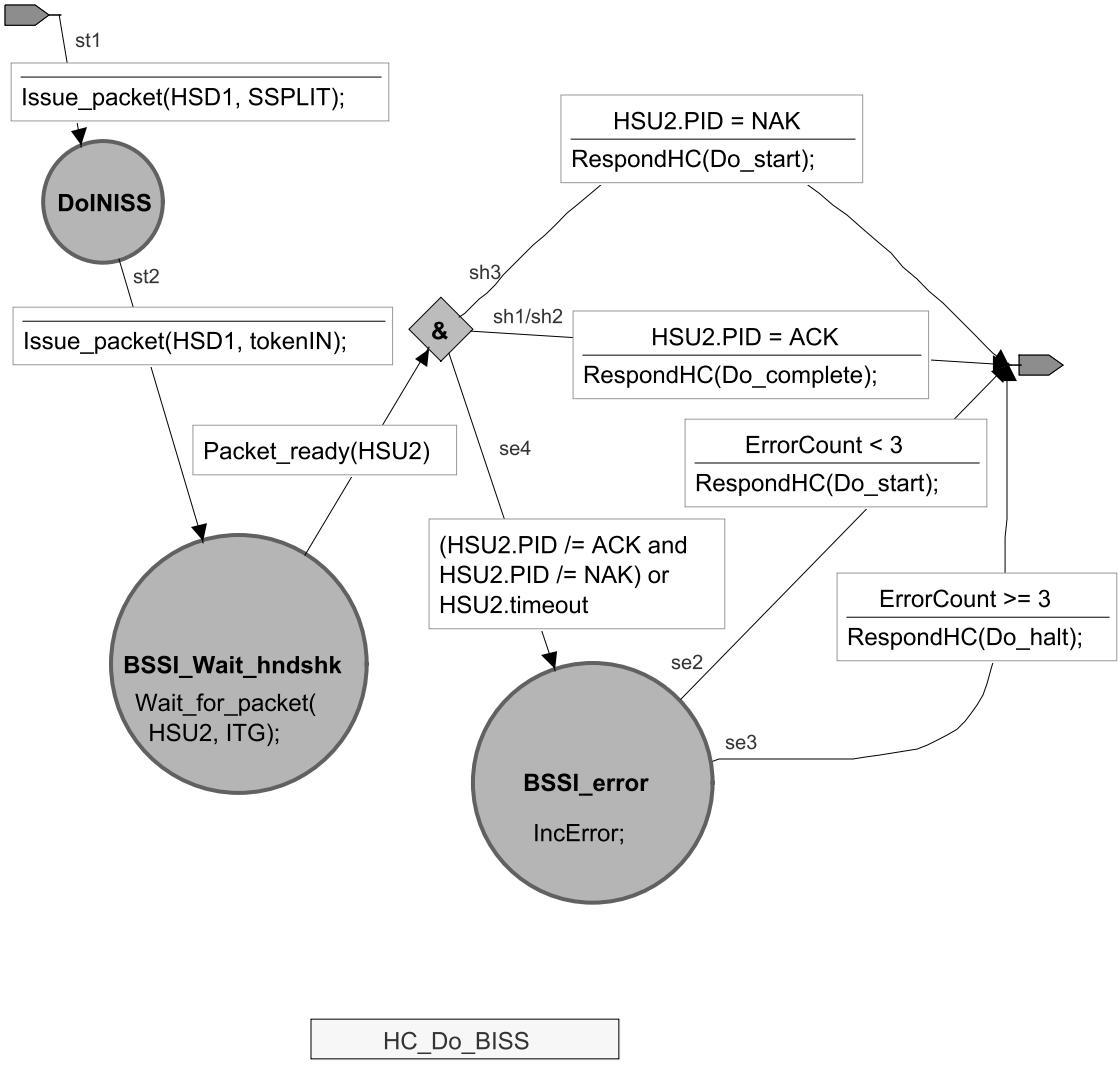



**Figure 11-56. Bulk/Control IN Start-split Transaction Host State Machine**

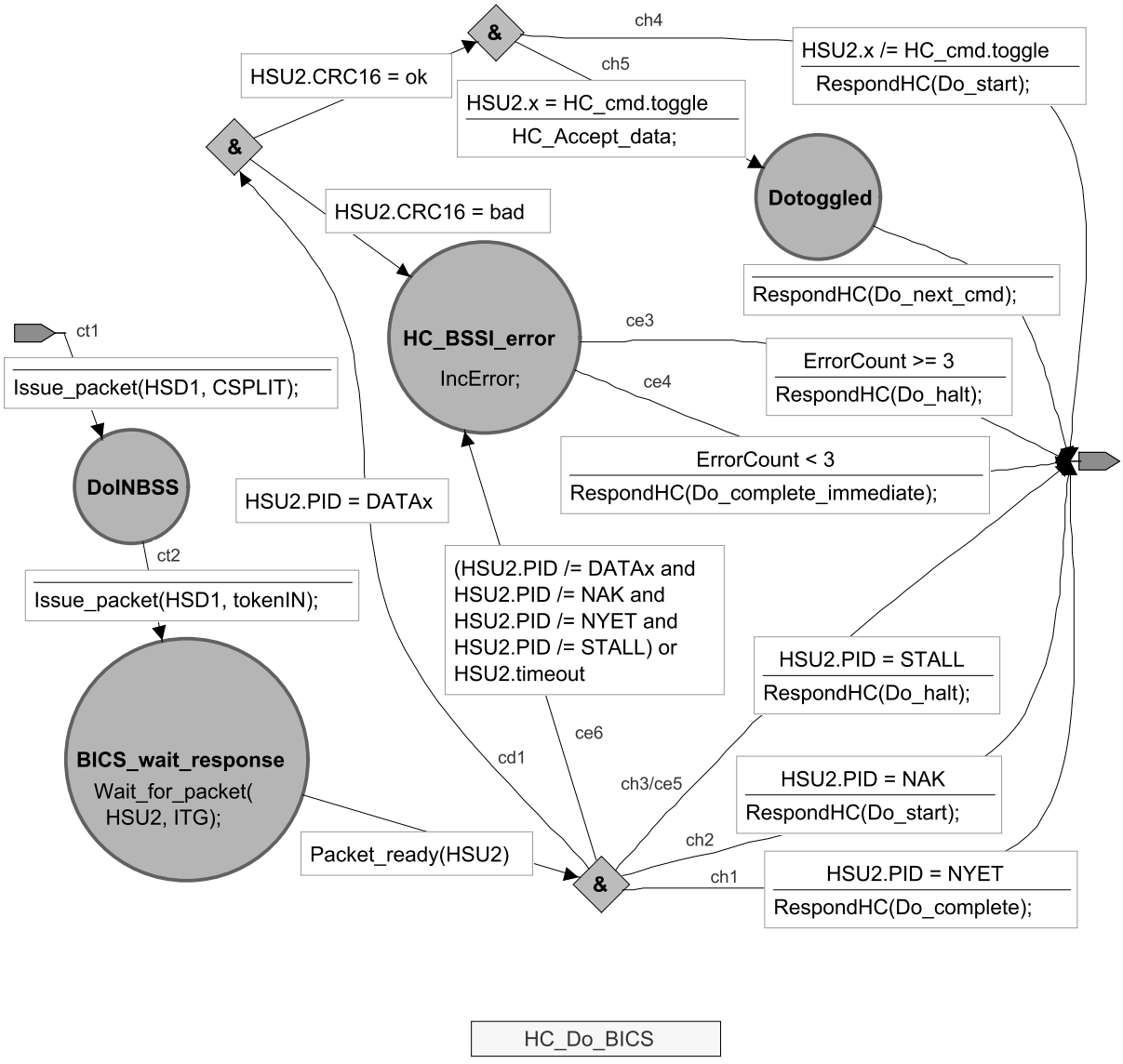



**Figure 11-57. Bulk/Control IN Complete-split Transaction Host State Machine**

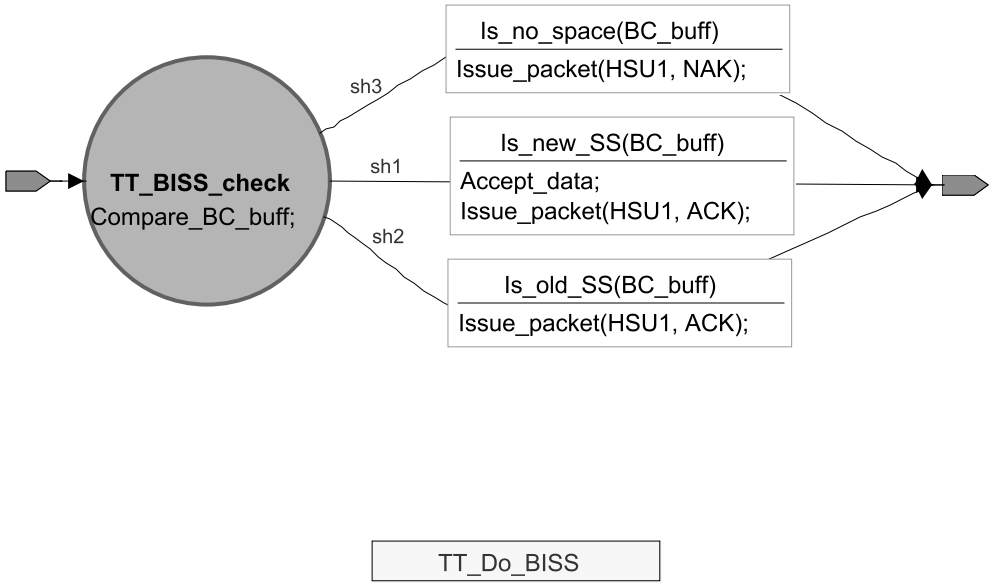



**Figure 11-58. Bulk/Control IN Start-split Transaction TT State Machine**

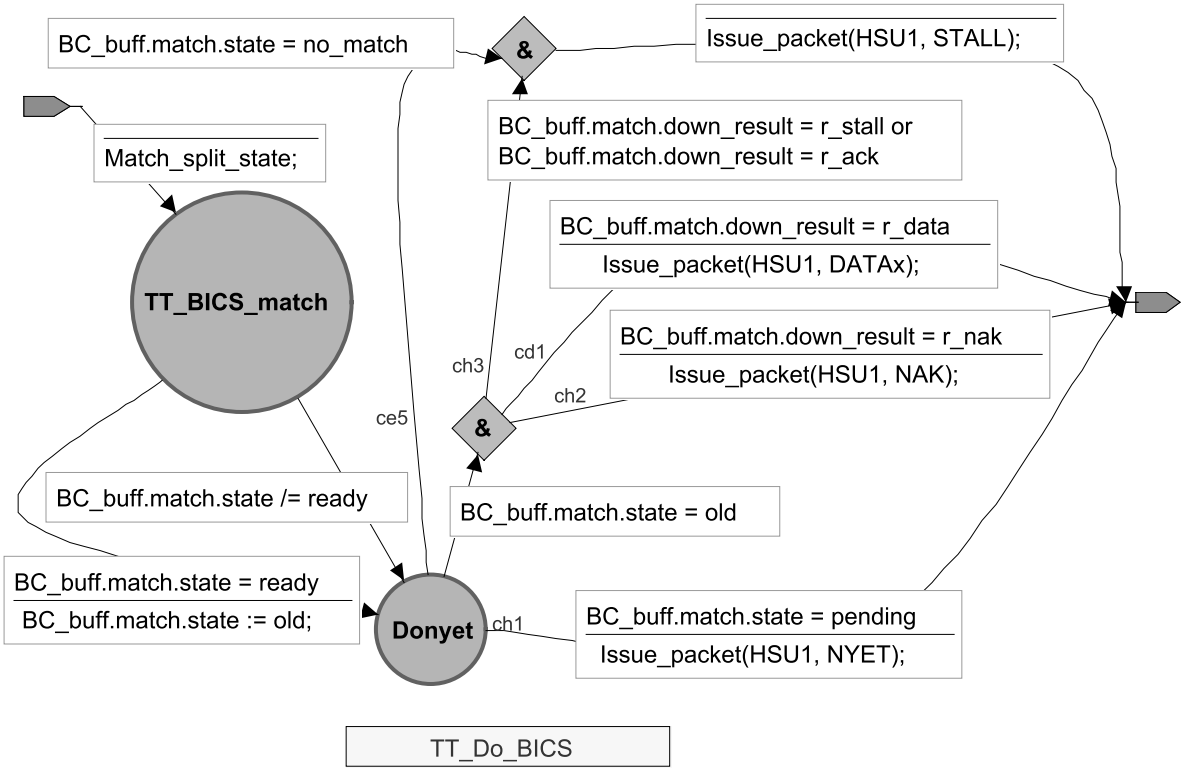



**Figure 11-59. Bulk/Control IN Complete-split Transaction TT State Machine**

## 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 issues 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. When the host sends the complete-split, 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 non-split transactions before doing this complete-split transaction retry.

If there is a bulk/control transaction in progress on the downstream facing bus when the EOF time occurs, the TT must adhere to the definition in Section 11.3 for its behavior on the downstream facing bus. This will cause an increase in the error count for this transaction. The normal retry rules will determine if the transaction will be retried or not on the downstream facing bus.

### 11.17.4 Bulk/Control Buffering Requirements

The TT must provide at least two transactions of non-periodic buffering to allow the TT 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 two 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 associated TT transaction buffer in a busy (ready/x) state. This buffer state will not 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.

Appendix A 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

There are requirements on the behavior of the host and the TT to ensure that the microframe pipeline correctly sequences full-/low-speed isochronous/interrupt transactions on downstream facing full-/low-speed buses. The host must determine the microframes in which a start-split and complete-split transaction must be issued on high-speed to correctly sequence a corresponding full-/low-speed transaction on the downstream facing bus. This is called "scheduling" the split transactions.

In the following descriptions, the 8 microframes within each full-speed (1 ms.) frame are referred to as microframe $Y_0$, $Y_1$, $Y_2$, …, $Y_7$. This notation means that the first microframe of each full-speed frame is labeled $Y_0$. The second microframe is labeled $Y_1$, etc. The last microframe of each full-speed frame is labeled $Y_7$. The labels repeat for each full-speed frame.

This section describes details of the microframe pipeline that affect both full-speed isochronous and full-/low-speed 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 Best Case Full-Speed Budget

A microframe of time allows at most 187.5 raw bytes of signaling on a full-speed bus. In order to estimate when full-/low-speed transactions appear on a downstream bus, the host must calculate a best case full-speed budget. This budget tracks in which microframes a full-/low-speed transaction appears. The best case full-speed budget assumes that 188 full-speed bytes occur in each microframe. Figure 11-60 shows how a 1 ms frame subdivided into microframes of budget time. This estimate assumes that no bit stuffing occurs to lengthen the time required to move transactions over the bus.

The maximum number of bytes in a 1 ms frame is calculated as:

1157 maximum_periodic_bytes_per_frame = 12 Mb/s * 1 ms / 8 bits_per_byte *

6 data_bits / 7 bit-stuffed_data_bits * 90% maximum_periodic_data_per_frame

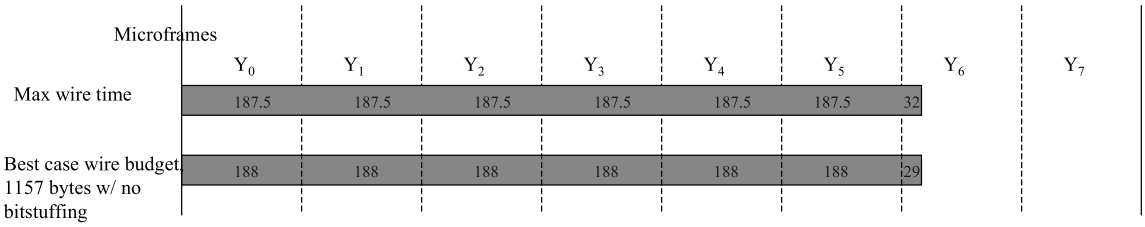



**Figure 11-60. Best Case Budgeted Full-speed Wire Time With No Bit Stuffing**

## 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 device is attached to the bus and configured, the host assigns 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.

The host always uses the maximum data payload size for a full-/low-speed endpoint in doing its budgeting. It does not attempt to schedule the actual data payloads that may be used in specific transactions to full-/low-speed endpoints. The host must include the maximum duration interpacket gap, bus turnaround times, and “TT think time”. The TT requires some time to proceed to the next full-/low-speed transaction. This time is called the “TT think time” and is specified in the hub descriptor field *wHubCharacteristics* bit 5 and 6.

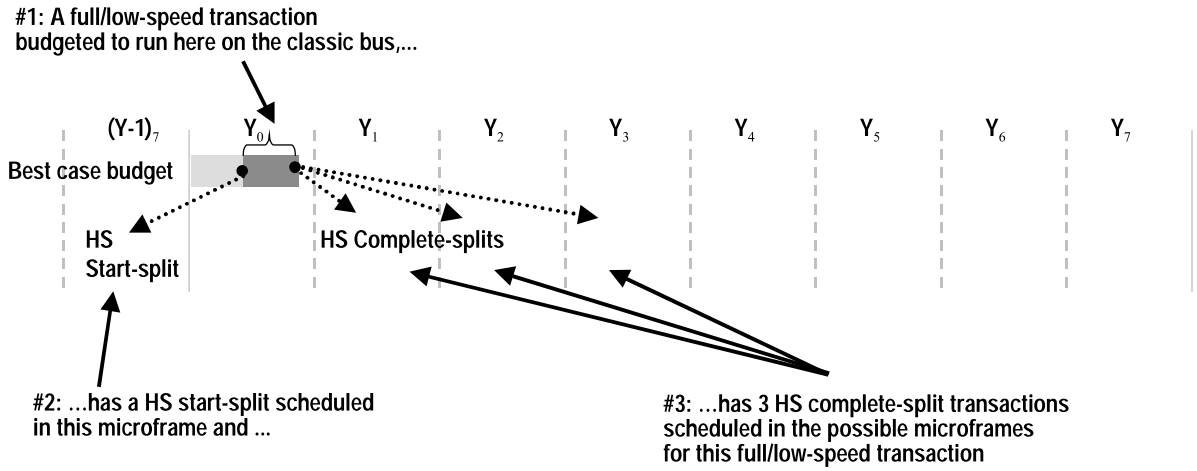



**Figure 11-61. Scheduling of TT Microframe Pipeline**

Figure 11-61 shows an example of a new endpoint that is assigned some portion of a full-/low-speed frame and where its start- and complete-splits are generally scheduled. The act of assigning some portion of the full-/low-speed frame to a particular transaction is called determining the budget for the transaction. More precise rules for scheduling and budgeting are presented later. The start-split for this example transaction is scheduled in microframe $Y\text{-}1_7$, the transaction is budgeted to run in microframe $Y_0$, and complete-splits are scheduled for microframes $Y_1$, $Y_2$, and $Y_3$. Section 11.18.4 describes the scheduling rules more completely.

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

## 11.18.3 Generation of Full-speed Frames

The TT must generate SOFs on the full-speed bus to establish the 1 ms frame clock within the defined jitter tolerances for full-speed devices. 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 zeroth microframe SOF. The zeroth high-speed microframe 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 frames as defined in other parts of this specification.

The TT must stop issuing full-speed SOFs after it detects 250 µs of high-speed idle. This is required to ensure that the full-/low-speed downstream facing bus enters suspend no more than 250 µs after the high-speed bus enters suspend.

The TT must generate a full-speed SOF on the downstream facing bus based on its frame timer. The generation of the full-speed SOF must occur within +/-3 full-speed bit time from the occurrence of the zeroth high-speed SOF. See Section 11.22.1 for more information about TT SOF generation.

## 11.18.4 Host Split Transaction Scheduling Requirements

Scheduling of split transactions is done by the host (typically in software) based on a best-case estimate of how the full-/low-speed transactions can be run on the downstream facing bus. This best-case estimate is called the best case budget. The host is free to issue the split transactions anytime within the scheduled microframe, but each split transaction must be issued sometime within the scheduled microframe. This description of the scheduling requirements applies to the split transactions for a single full-/low-speed transaction at a time.

1. The host must never schedule a start-split in microframe $Y_6$. Some error conditions may result in the host controller erroneously issuing a start-split in this microframe. The TT response to this start-split is undefined.

2. The host must compute the start-split schedule by determining the best case budget for the transaction and:

   a. For isochronous OUT full-speed transactions, for each microframe in which the transaction is budgeted, the host must schedule a 188 (or the remaining data size) data byte start-split transaction. The start-split transaction must be scheduled in the microframe before the data is budgeted to begin on the full-speed bus. The start-split transactions must use the beginning/middle/end/all split transaction token encodings corresponding to the piece of the full-speed data that is being sent on the high-speed bus. For example, if only a single start-split is required, an "all" encoding is used. If multiple start-splits are required, a "beginning" encoding is used for the first start-split and an "end" encoding is used for the final start-split. If there are more than two start-splits required, the additional start-splits that are not the first or last use a "middle" encoding. A zero length full-speed data payload must only be scheduled with an "all" start-split. A start-split transaction for a beginning, middle, or end start-split must always have a non-zero length data payload.
   Figure 11-62 shows an example of an isochronous OUT that would appear to have budgeted a zero length data payload in a start-split (end). This example instead must be scheduled with a start-split(all) transaction.

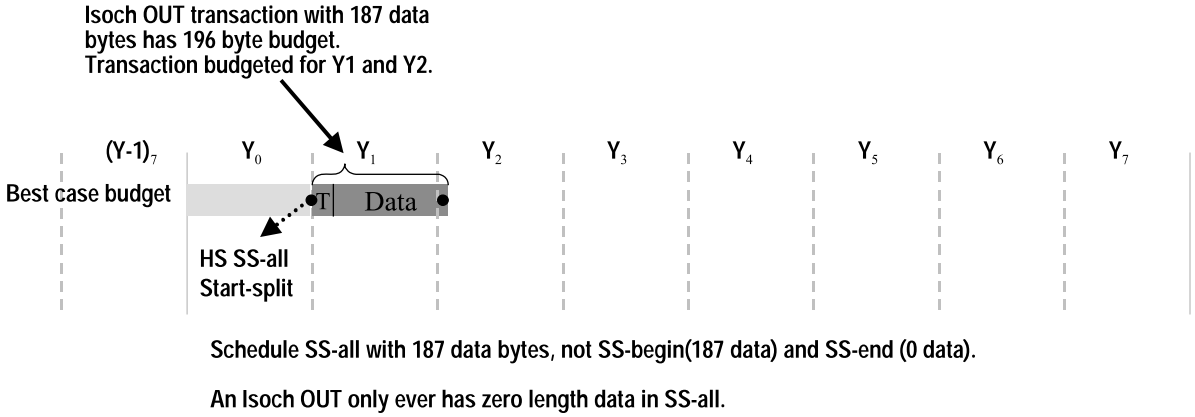



**Figure 11-62. Isochronous OUT Example That Avoids a Start-split-end With Zero Data**

   b. For isochronous IN and interrupt IN/OUT full-/low-speed transactions, a single start-split must be scheduled in the microframe before the transaction is budgeted to start on the full-/low-speed bus.

3. The host never schedules more than one complete-split in any microframe for the same full-/low-speed transaction.

   a. For isochronous OUT full-speed transactions, the host must never schedule a complete-split. The TT response to a complete-split for an isochronous OUT is undefined.

   b. For interrupt IN/OUT full-/low-speed transactions, the host must schedule a complete-split transaction in each of the two microframes following the first microframe in which the full-/low-speed transaction is budgeted. An additional complete-split must also be scheduled in the third following microframe unless the full-/low-speed transaction was budgeted to start in microframe $Y_6$. Figure 11-63 shows an example with only two complete-splits.

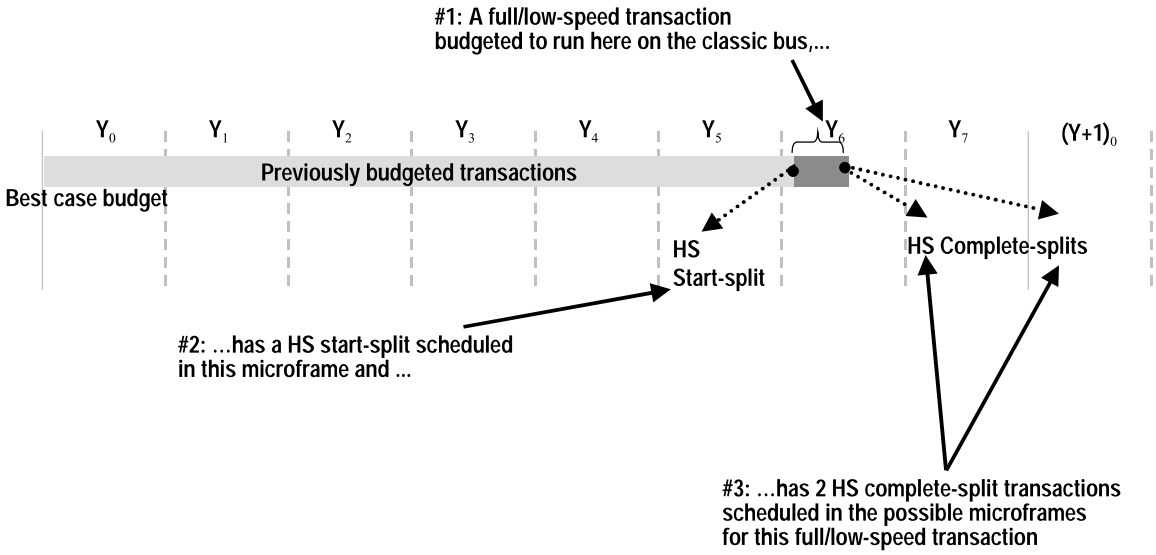



**Figure 11-63. End of Frame TT Pipeline Scheduling Example**

c. For isochronous IN full-speed transactions, for each microframe in which the full-speed transaction is budgeted, a complete-split must be scheduled for each following microframe. Also, determine the last microframe in which a complete-split is scheduled, call it L. If L is less than $Y_6$, schedule additional complete-splits in microframe L+1 and L+2.

   If L is equal to $Y_6$, schedule one complete-split in microframe $Y_7$. Also, schedule one complete-split in microframe $Y_0$ of the next frame, unless the full-speed transaction was budgeted to start in microframe $Y_0$.

   If L is equal to $Y_7$, schedule one complete-split in microframe $Y_0$ of the next frame, unless the full-speed transaction was budgeted to start in microframe $Y_0$. Figure 11-64 and Figure 11-65 show examples of the cases for L= $Y_6$ and L=$Y_7$.

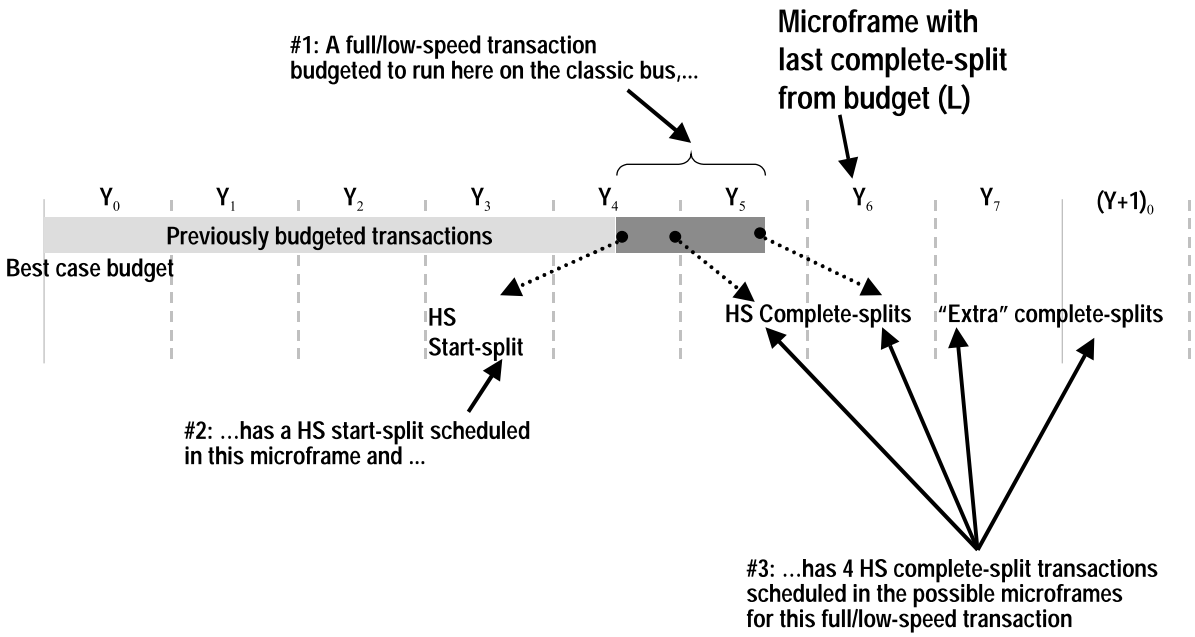



**Figure 11-64. Isochronous IN Complete-split Schedule Example at L=$Y_6$**

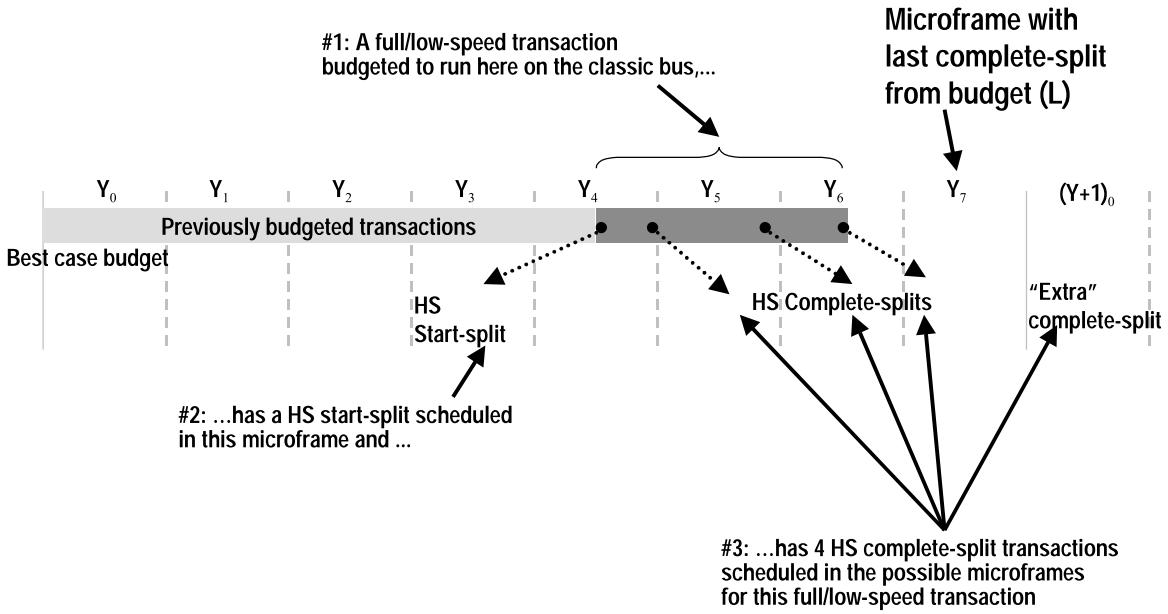



**Figure 11-65. Isochronous IN Complete-split Schedule Example at L=$Y_7$**

4. The host must never issue more than 16 start-splits in any high-speed microframe for any TT.
5. The host must only issue a split transaction in the microframe in which it was scheduled.
6. As precisely identified in the flow sequence and state machine figures, the host controller must immediately retry a complete-split after a high-speed transaction error ("trans_err").

The “pattern” of split transactions scheduled for a full-/low-speed transaction can be computed once when each endpoint is configured. Then the pattern does not change unless some change occurs to the collection of currently configured full-/low-speed endpoints attached via a TT.

Finally, for all periodic endpoints that have split transactions scheduled within a particular microframe, the host must issue complete-split transactions in the same relative order as the corresponding start-split transactions were issued.

### 11.18.5 TT Response Generation

The approach used for 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 to return 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 boundary to "carve up" the full-/low-speed data to be returned to the host. The TT does not do any computation on how much data to return at what time. In response to the "next" high-speed complete-split, the TT simply returns the endpoint data it has received from the full-/low-speed bus in a microframe.

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 for the data packet PID.

If the full-/low-speed handler completes the full-/low-speed isochronous/interrupt IN transaction during a microframe with a valid CRC16, it uses the DATA0/1 PID for the data packet of the complete-split transaction. This indicates that this is the last data of the full-/low-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 received.

If the full-/low-speed handler completes the full-/low-speed isochronous/interrupt IN transaction during a microframe with a bad CRC16, it uses the ERR response to the complete-split transaction and does not return the data received from the full-/low-speed device.

If the TT is still receiving data on the downstream facing bus at the microframe boundary, the TT will respond with either an MDATA PID or a NYET for the corresponding complete-split. If the TT has received more than two bytes of the data field of the full-/low-speed data packet, it will respond with an MDATA PID. Further, the data packet that will be returned in the complete-split must contain the data received from the full-/low-speed device minus the last two bytes. The last two bytes must not be included since they could be the CRC16 field, but the TT will not know this until the next microframe. The CRC16 field received from the full-/low-speed device is never returned in a complete-split data packet for isochronous/interrupt transactions. If less than three data bytes of the full-/low-speed data packet have been received at the end of a microframe, the TT must respond with a NYET to the corresponding high-speed complete-split. Both of these responses indicate to the host that more data is being received and another complete-split transaction is required.

When the host controller receives a DATA0/1 PID for interrupt or isochronous IN complete-splits (and ACK, NAK, STALL, ERR for interrupt IN/OUT complete-splits), it stops issuing any remaining complete-splits that might be scheduled for that endpoint for this full-/low-speed transaction.

If the TT has not started the full-/low-speed transaction when it receives a complete-split, the TT will not find an entry in the complete-split pipeline stage. When this happens, the protocol state machines show that the TT responds with a NYET (e.g., the “no match” case). This NYET response tells the host that there are no results available currently, but the host should continue with other scheduled split transactions for this endpoint in subsequent microframes.

In general, there will be two (or more) complete-split transactions scheduled for a periodic endpoint. However, for interrupt endpoints, the maximum size of the full-/low-speed transaction guarantees that it can never require more than two complete-split transactions. Two complete-split transactions are only required when the transaction spans a microframe boundary. In cases where the full-/low-speed transaction actually

starts and completes in the same microframe, only a single complete-split will return data; any other earlier complete-splits will have a NYET response.

For isochronous IN transactions, more complete-split transactions may be scheduled based on the length of the full-speed transaction. A full-speed isochronous IN transaction can be up to 1023 data bytes, which can require portions of up to 8 microframes of time on the downstream facing bus (with the worst alignment in the frame and worst case bit stuffing). Such a maximum sized full-speed transaction can require 8 complete-split transactions. If the device generates less data, the host will stop issuing complete-splits after the one that returns the final data from the device for a frame.

## 11.18.6 TT Periodic Transaction Handling Requirements

The TT has two methods it must use to react to timing related events that affect the microframe pipeline: current transaction abort and freeing pending start-splits. These methods must be used to manage the microframe pipeline.

The TT must also react (as described in Section 11.22.1) when its microframe or frame timer loses synchronization with the high-speed bus.

The TT must not issue too many full-/low-speed transactions in any microframe.

Each of these requirements are described below.

### 11.18.6.1 Abort of Current Transaction

When a current transaction is in progress on the downstream facing bus and it is no longer appropriate for the TT to continue the transaction, the transaction is "aborted."

The TT full-/low-speed handler must abort the current full-/low-speed transaction:

1. For all periodic transaction types, if the full-speed frame EOF time occurs
2. If the transaction is an interrupt transaction and the start-split for the transaction was received in some microframe (call it X) and the TT microframe timer indicates the X+4 microframe

Note that no additional abort handling is required for isochronous transactions besides the generic IN/OUT handling described below. Abort has different processing requirements with regards to the downstream facing bus for IN and OUT transactions. For any type of transaction, the TT must not generate a complete-split response for an aborted transaction; e.g., no entry is made in the complete-split pipeline stage for an aborted transaction.

1. At the time the TT decides to abort an IN transaction, the TT must not issue the handshake packet for the transaction if the handshake has not already been started on the downstream facing bus. The TT may choose to not issue the IN token packet, if possible. If the transaction is in the data phase (e.g., in the middle of the target device generated DATA packet), the TT simply awaits the completion of that packet and ignores any data received and must not respond with a full-/low-speed handshake. The TT must not make an entry in the complete-split pipeline stage. This processing will cause a NYET response to the corresponding complete-split on the high-speed bus.
2. At the time the TT decides to abort an OUT transaction, the TT may choose to not issue the TOKEN or DATA packets, if possible. If the TT is in the middle of the DATA packet, it must stop issuing data bytes as soon as possible and force a bit-stuffing error on the downstream facing bus. In any case, the TT must not make an entry in the complete-split pipeline stage. This processing will cause a NYET response to the corresponding complete-split on the high-speed bus.

### 11.18.6.2 Free of Pending Start-splits

A start-split can be buffered in the start-split pipeline stage that is no longer appropriate to cause a full-/low-speed transaction on the downstream facing bus. Such a start-split transaction must be "freed" from the

start-split pipeline stage. This means the start-split is simply ignored by the TT and the TT must respond to a corresponding complete-split with a NYET. For example, no entry is made in the complete-split pipeline stage for the freed start-split.

A start-split in the start-split pipeline must be freed:

1. If the full-speed frame EOF time occurs, except for start-splits received in $(Y\text{-}1)_7$
2. If the start-split transaction was received in some microframe (call it X) and the TT microframe timer indicates the X+4 microframe

If the TT receives a periodic start-split transaction in microframe $Y_6$, its behavior is undefined. This is a host scheduling error.

### 11.18.6.3 Maximum Full-/low-speed Transactions per Microframe

The TT must not start a full-/low-speed transaction unless it has space available in the complete-split pipeline stage to hold the results of the transaction. If there is not enough space, the TT must wait to issue the transaction until there is enough space. The maximum number of normally operating full-speed transactions that can ever be completed in a microframe is 16.

## 11.18.7 TT Transaction Tracking

Figure 11-66 shows the TT microframe pipeline of transactions. The 8 high-speed microframes that compose a full-/low-speed frame are labeled with $Y_0$ through $Y_7$ assuming the microframe timer 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 $Y_0$ (e.g., a start-split "B") can run on the full-/low-speed bus during microframe times $Y_1$ or $Y_2$ or $Y_3$. 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 transactions (if they are required) will run on the high-speed bus during microframes $Y_2$, $Y_3$, or $Y_4$.

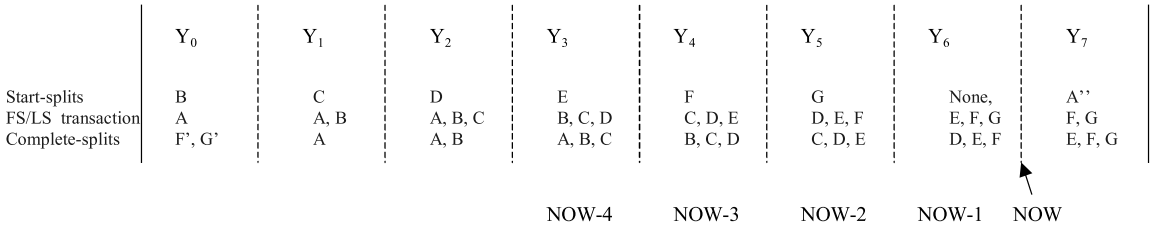



**Figure 11-66. Microframe Pipeline**

When the microframe timer indicates a new microframe, the high-speed handler must mark any start-splits in the start-split pipeline stage it received in the previous microframe 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 on the downstream bus too early.

At the beginning of each microframe (call it "NOW"), the high-speed handler must free (as defined in Section 11.18.6.2) any start-split transactions from the start-split pipeline stage that are still pending from microframe NOW-4 (or earlier) and ignore them. If the transaction is in progress on the downstream facing bus, the transaction must be aborted (with full-/low-speed methods as defined in Chapter 8). This is described in more detail in the previous sections. 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 periodic transaction pipeline running on time (e.g., transactions do not 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).

Finally, at the beginning of each microframe, the high-speed handler must change any complete-split transaction responses in the complete-split pipeline stage from microframe NOW-2 to the free state so that their space can be reused for responses in this microframe.

This algorithm is shown in pseudo code in Figure 11-67. 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
   Free start-split entry
End loop

-- Clean up complete-split pipeline in case no complete-splits were received
While complete-split transaction states from (microframe-2) loop
   Free complete-split response transaction entry
End loop

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

**Figure 11-67. Advance_Pipeline Pseudocode**

## 11.18.8 TT Complete-split Transaction State Searching

A host must issue complete-split transactions in a microframe for a set of full-/low-speed endpoints in the same relative order as the start-splits were issued in a microframe for this TT. However, errors on start- or complete-splits can cause the high-speed handler to receive a complete-split transaction that does not "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 does not 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 did not receive (due to complete-split errors on the high-speed bus).

The high-speed handler must 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 high-speed handler only needs to search the complete-split responses accumulated during the previous microframe. There only needs to be at most 1 microframe of complete-split response entries; the microframe of responses that have already been accumulated and are awaiting to be returned via high-speed complete-splits.

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".

The high-speed handler matches the complete-split transaction with the correct entry in the complete-split pipeline stage and advances the pipeline appropriately. There are five cases the TT must handle correctly:

1. If the high-speed complete-split token and first entry of the complete-split pipeline match, the high-speed handler responds with the indicated data/status. This case occurs the first time the TT receives a complete-split.

2. Same as above, but this is a retry of a complete-split that the TT has already received due to the host controller not receiving the (previous) response information.
3. If the complete-split transaction matches some other entry in the complete-split pipeline besides the first, the high-speed handler advances the complete-split pipeline (e.g., frees response information for previous complete-split entries) and responds with the information for the matching entry. This case can happen due to normal or missed previous complete-split transactions. An example abnormal case could be that the host controller was unsuccessful in issuing a complete-split transaction to the high-speed handler and has done endpoint halt processing for that endpoint. This means the next complete-split will not match the first entry of the complete-split pipeline stage.
4. 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). Note that a NYET response is possible in this case due to a transaction error on the start-split or a host (or TT) scheduling error.
5. The high-speed handler can timeout its first high-speed complete-split transaction while it is searching the complete-split pipeline stage for a matching entry. However, the high-speed handler must respond correctly to the subsequent complete-split transaction. If the high-speed handler did not 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 host controller must issue the complete-split transactions in the same relative order as the original corresponding start-split transactions.

## 11.19 Approximate TT Buffer Space Required

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

The TT microframe pipeline requires less than:

- 752 data bytes for the start-split stage
- 2x 188 data 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 and, 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, high-speed or full-/low-speed transactions for other endpoints may occur before or after these split transactions. 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 transaction error for full-/low-speed interrupt transactions.

### 11.20.1 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.

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 transaction errors (including timeout) of an interrupt transaction.

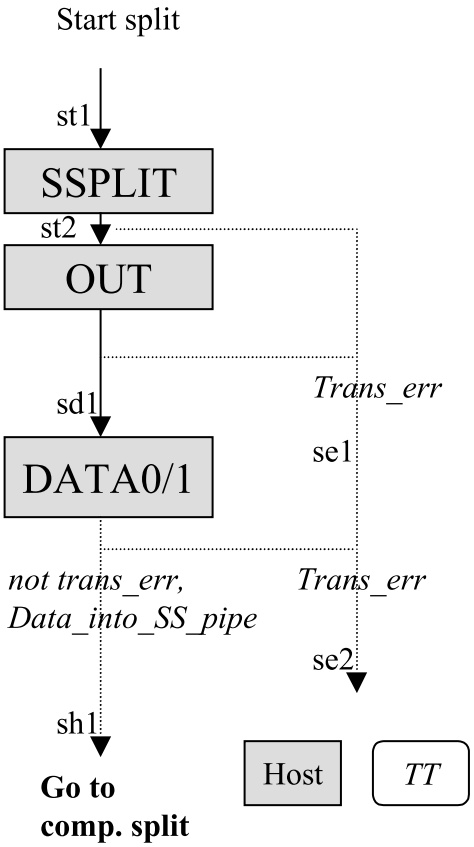



**Figure 11-68. Interrupt OUT Start-split Transaction Sequence**

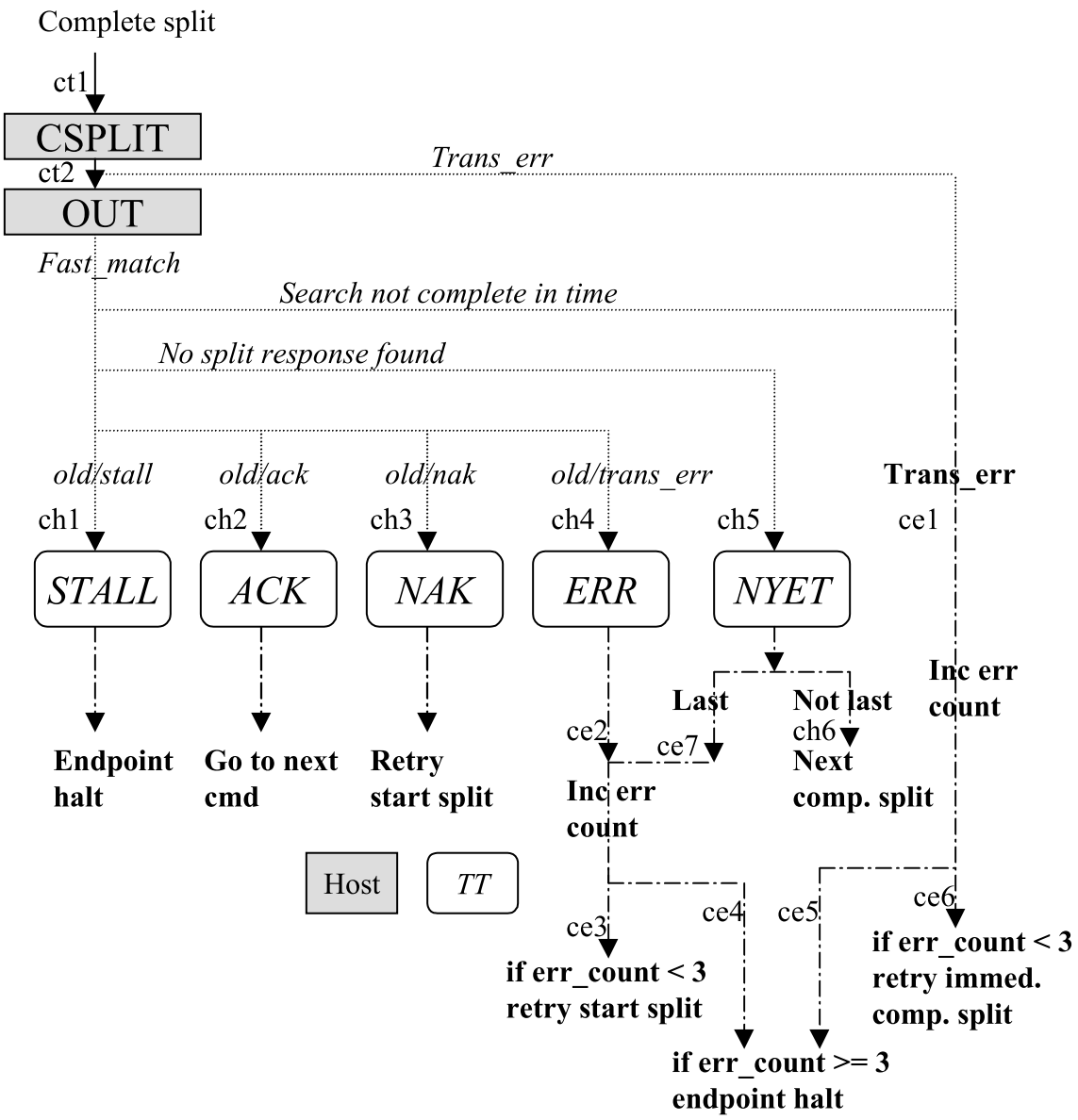



**Figure 11-69. Interrupt OUT Complete-split Transaction Sequence**

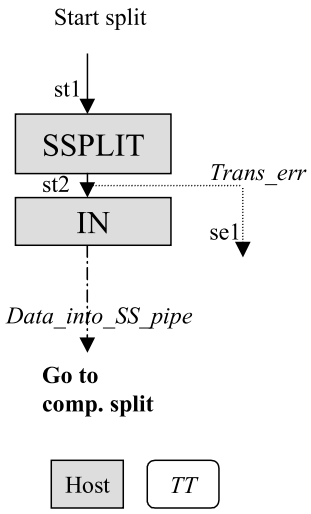



**Figure 11-70. Interrupt IN Start-split Transaction Sequence**

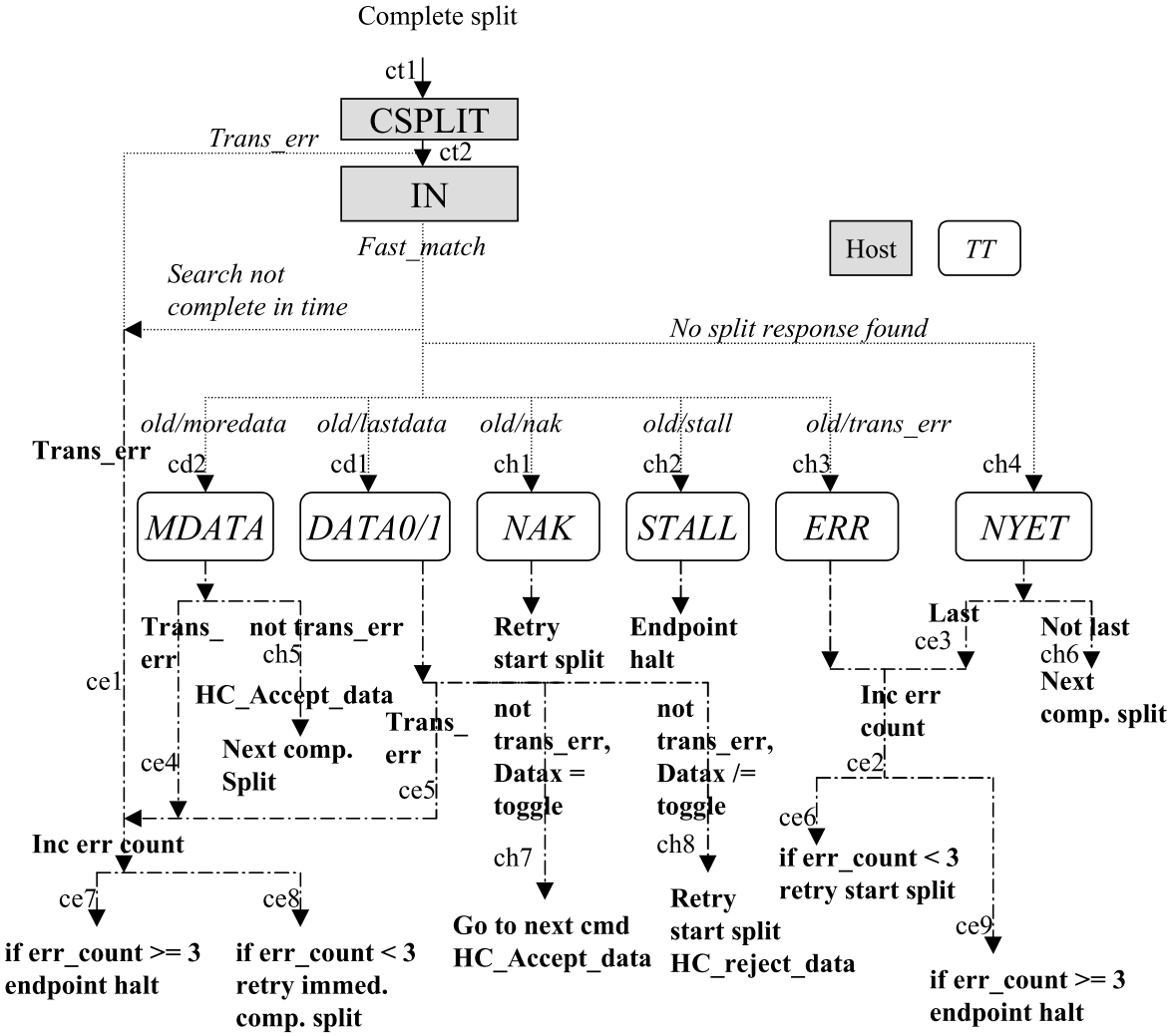



**Figure 11-71. Interrupt IN Complete-split Transaction Sequence**

## 11.20.2 Interrupt Split Transaction State Machines

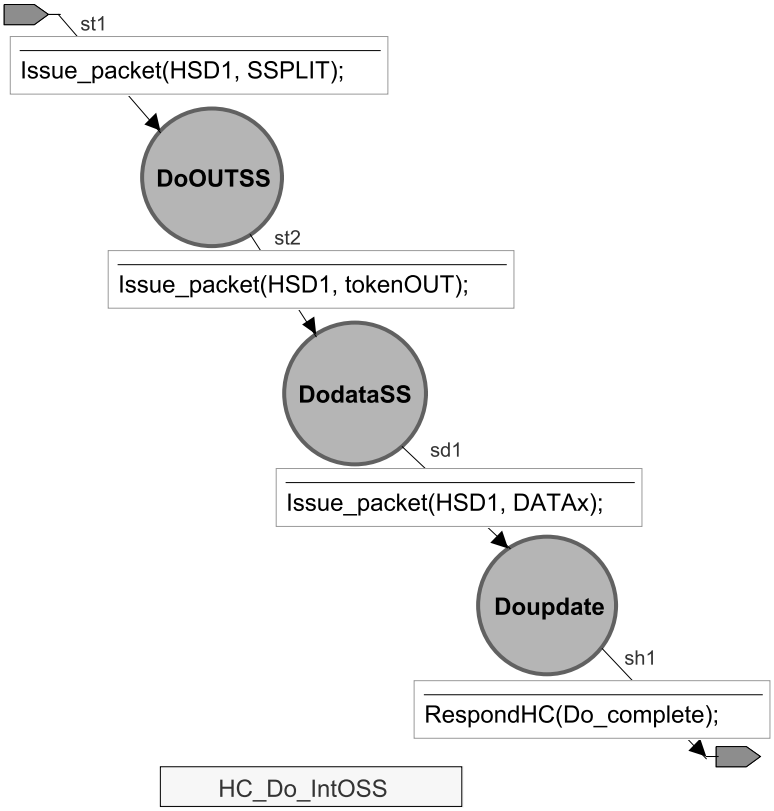



**Figure 11-72. Interrupt OUT Start-split Transaction Host State Machine**

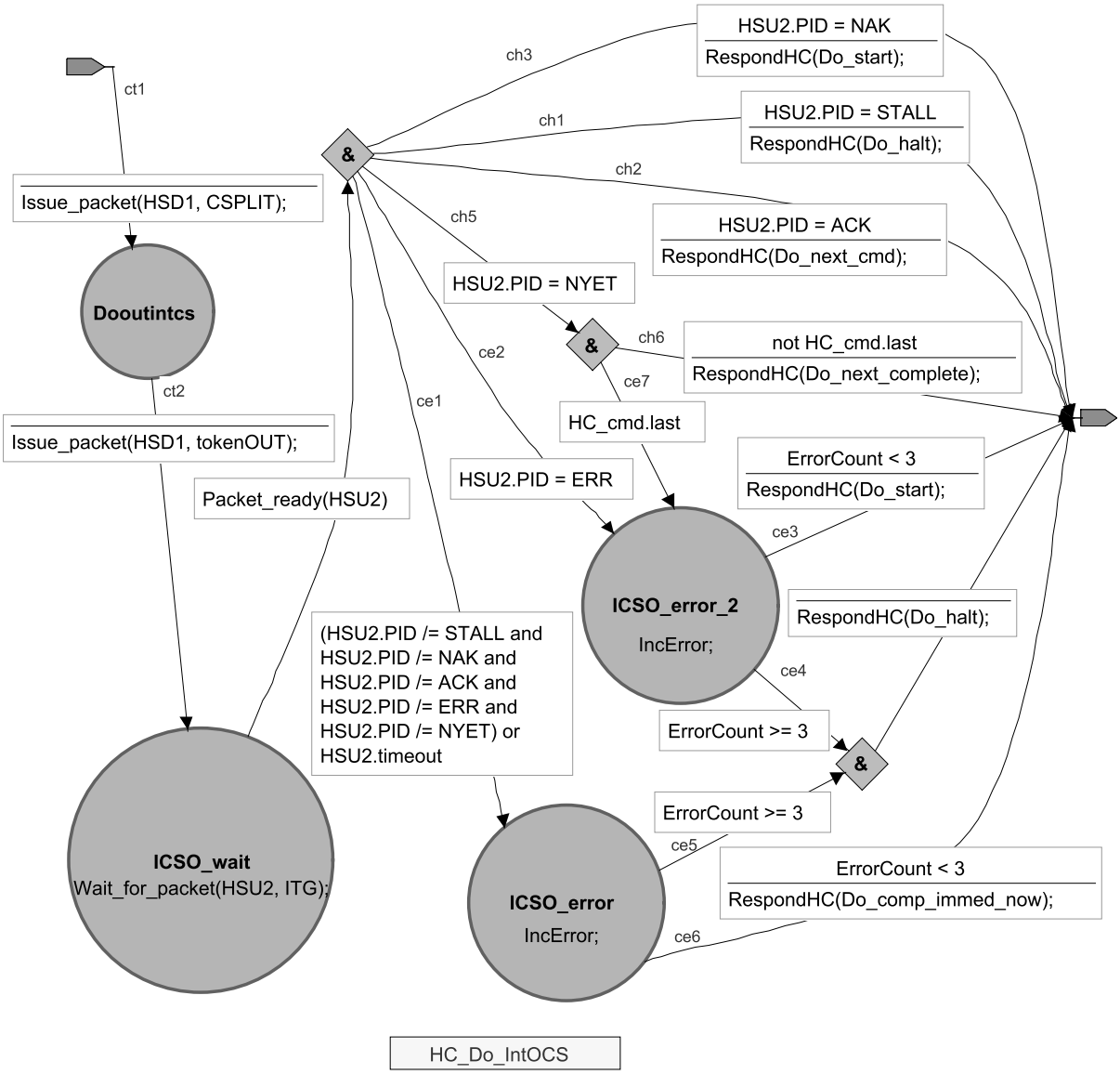



**Figure 11-73. Interrupt OUT Complete-split Transaction Host State Machine**

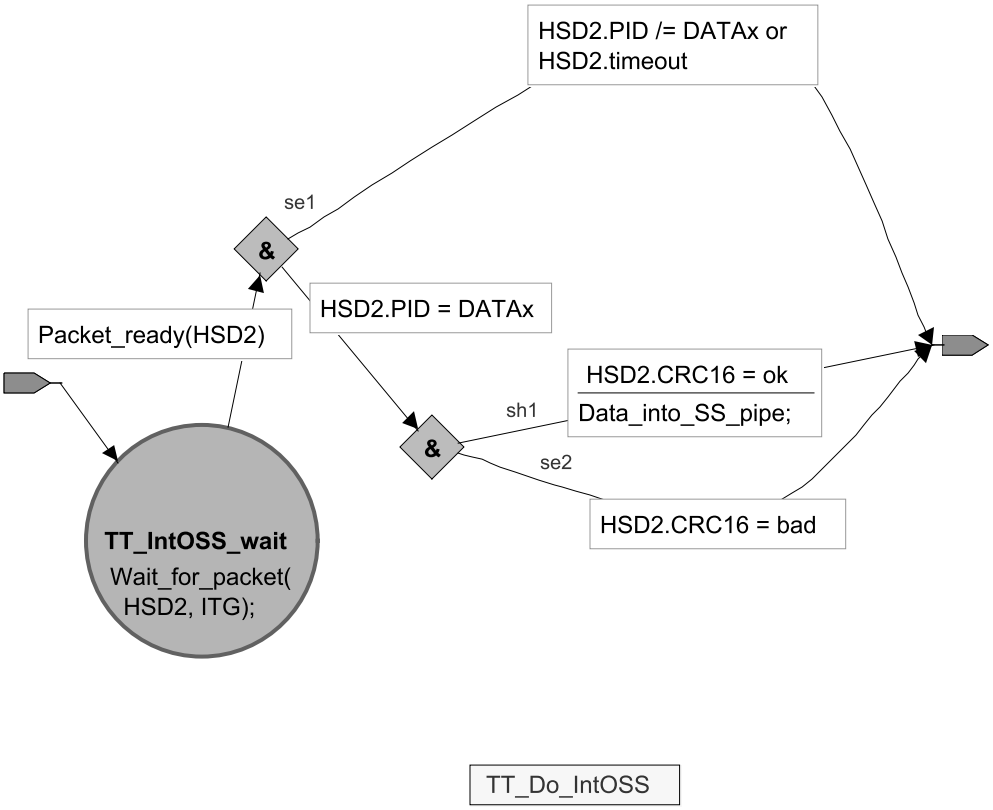



**Figure 11-74. Interrupt OUT Start-split Transaction TT State Machine**

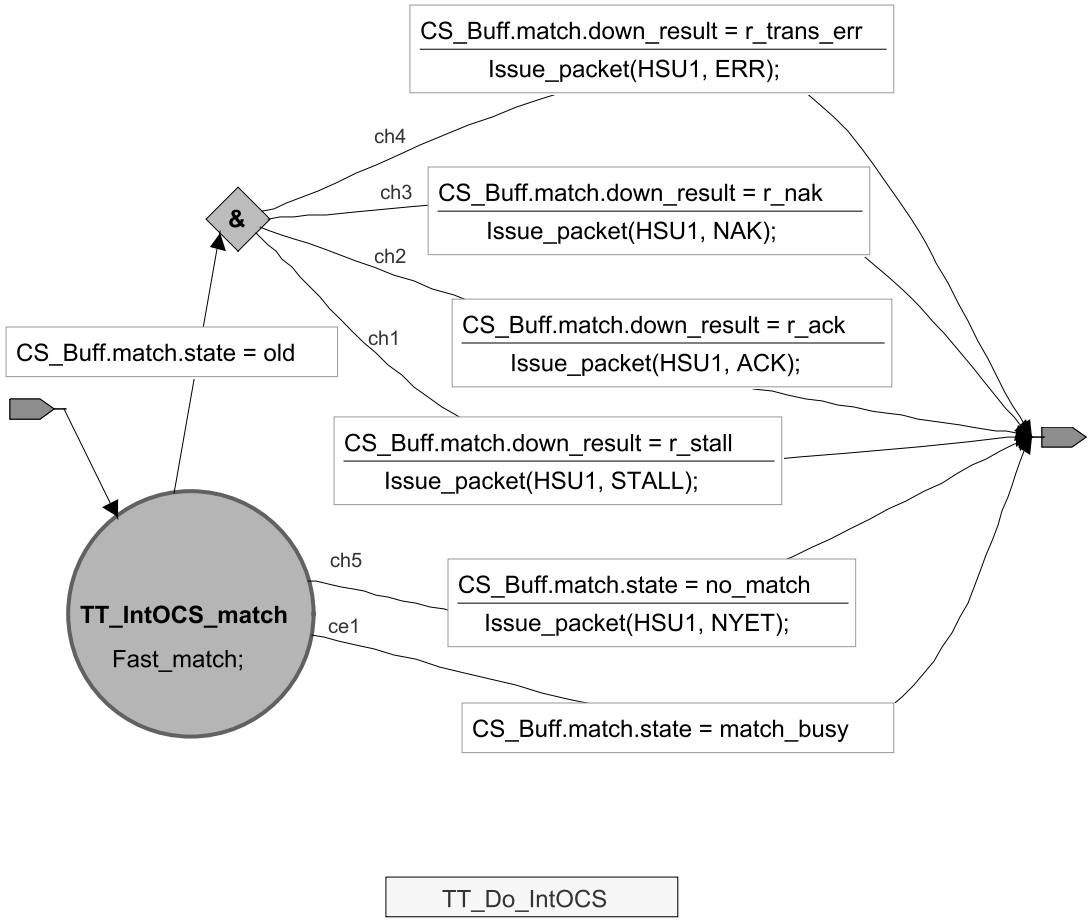



**Figure 11-75. Interrupt OUT Complete-split Transaction TT State Machine**

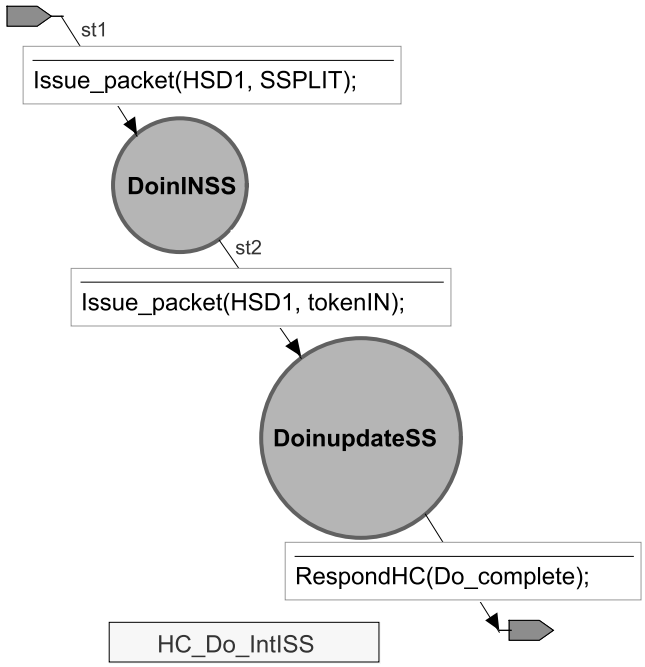



**Figure 11-76. Interrupt IN Start-split Transaction Host State Machine**

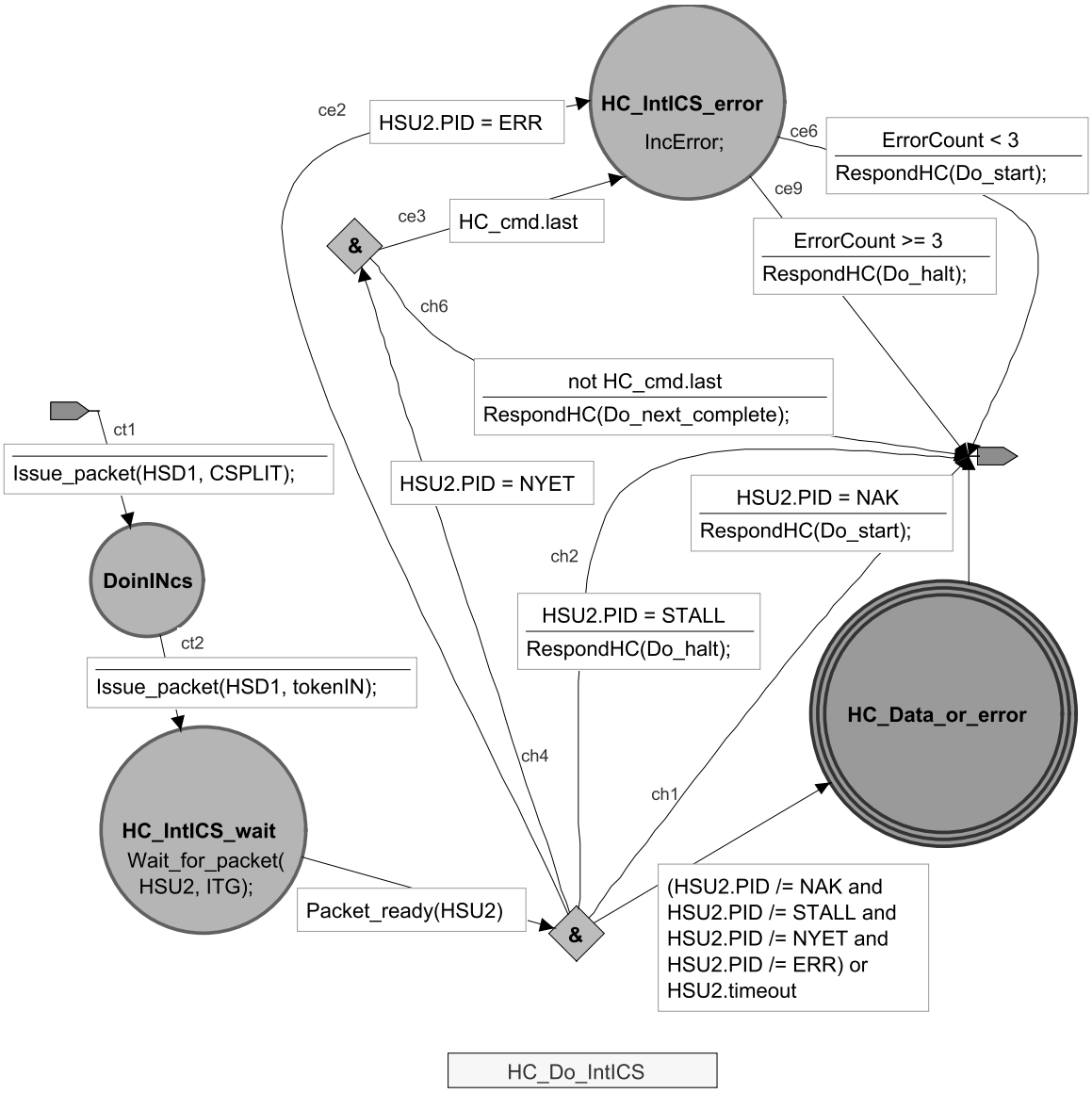



**Figure 11-77. Interrupt IN Complete-split Transaction Host State Machine**

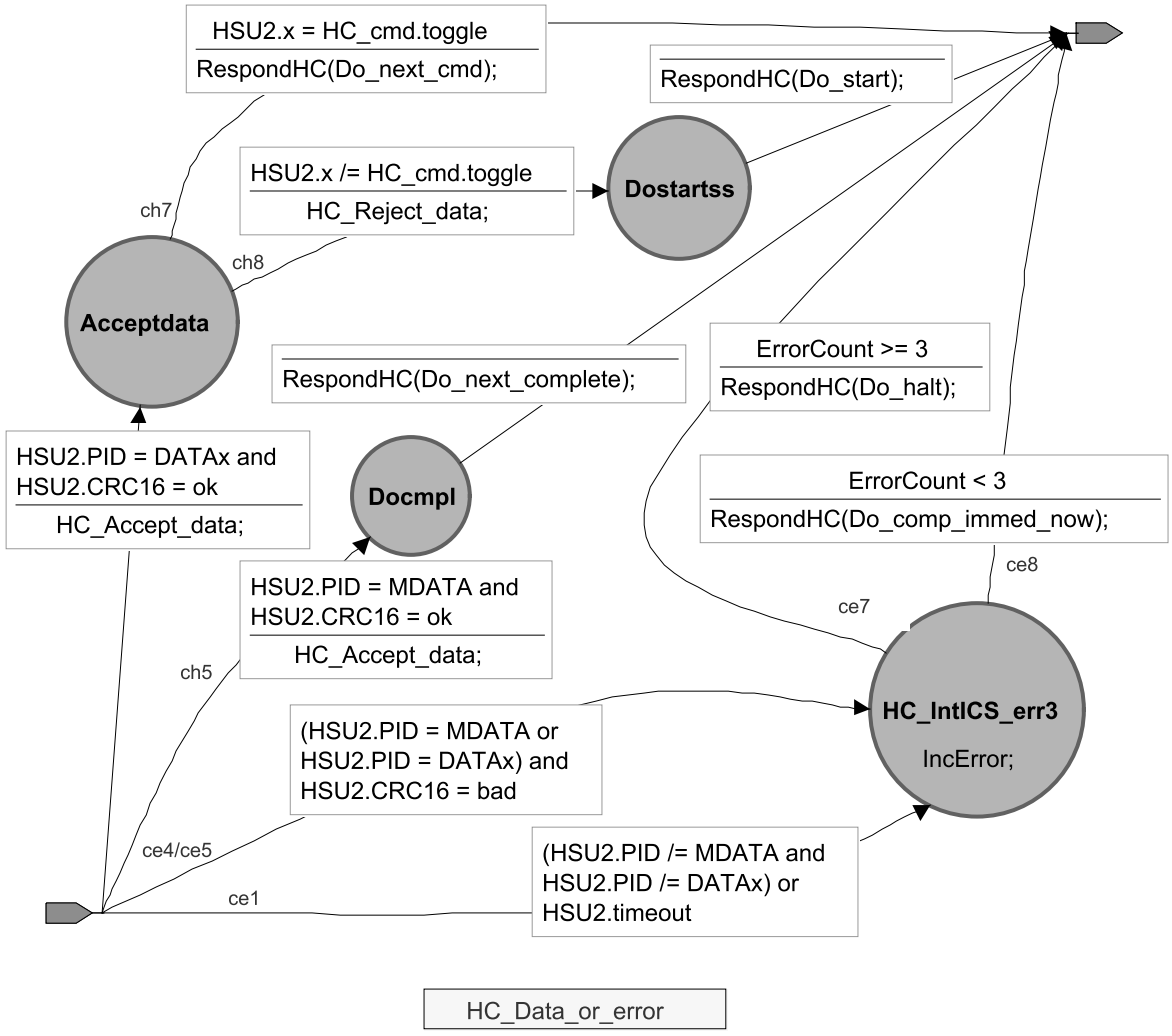



**Figure 11-78. HC_Data_or_Error State Machine**

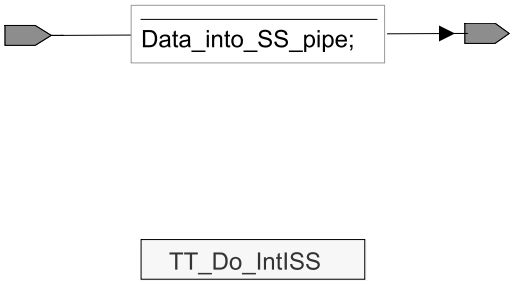



**Figure 11-79. Interrupt IN Start-split Transaction TT State Machine**

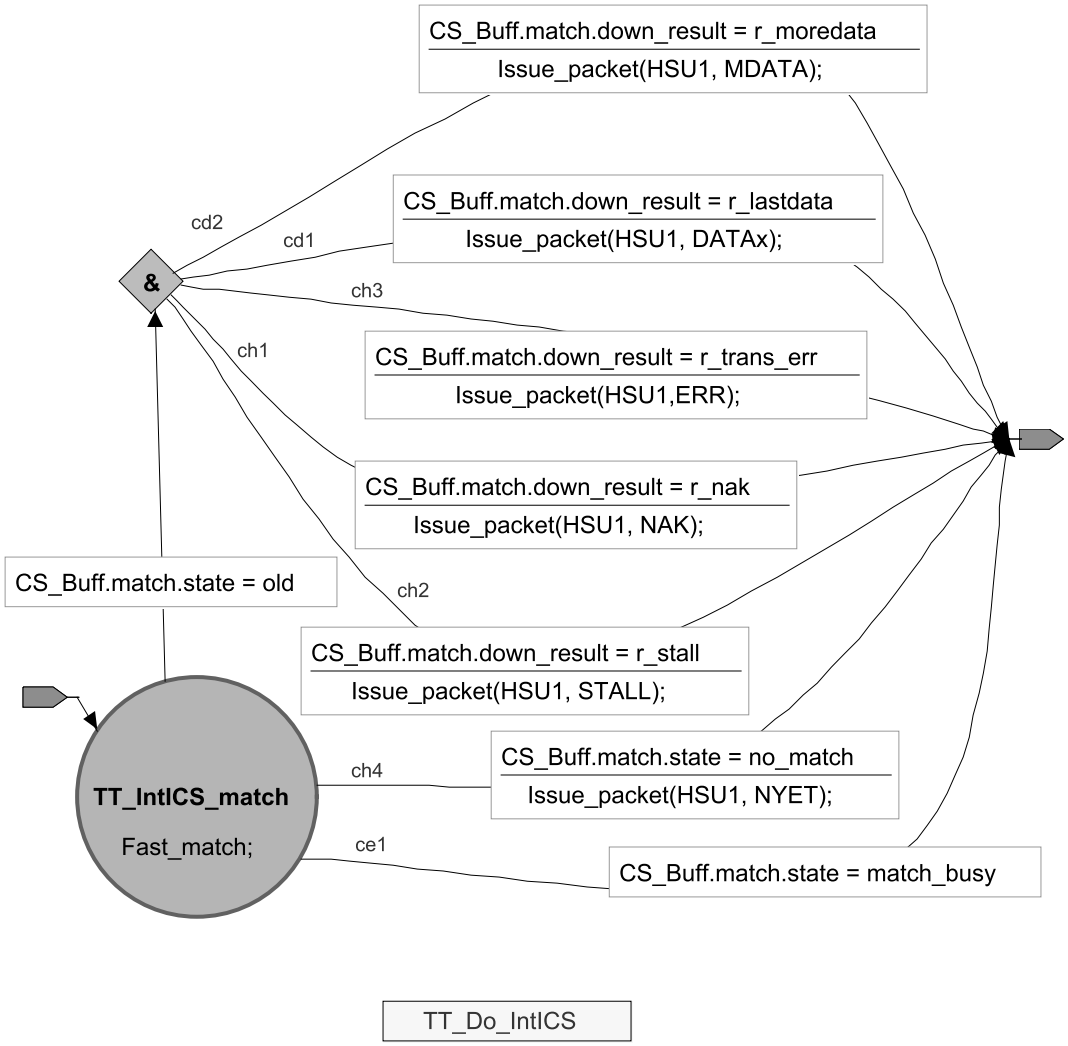



**Figure 11-80. Interrupt IN Complete-split Transaction TT State Machine**

## 11.20.3 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 the TT in the start-split pipeline stage. The start-splits are saved in the order they are received until the end of the microframe. At the end of the microframe, these transactions are available 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 a previous microframe since the high-speed schedule was built assuming best case bit stuffing and the full-/low-speed transactions could be taking longer on the full-/low-speed bus. As the full-/low-speed handler issues transactions on the downstream facing full-/low-speed bus, it saves the results in the periodic complete-split pipeline stage and then advances to the next transaction in the start-split pipeline.

In a following microframe (as described previously), the host controller issues a high-speed complete-split transaction and the TT responds appropriately.

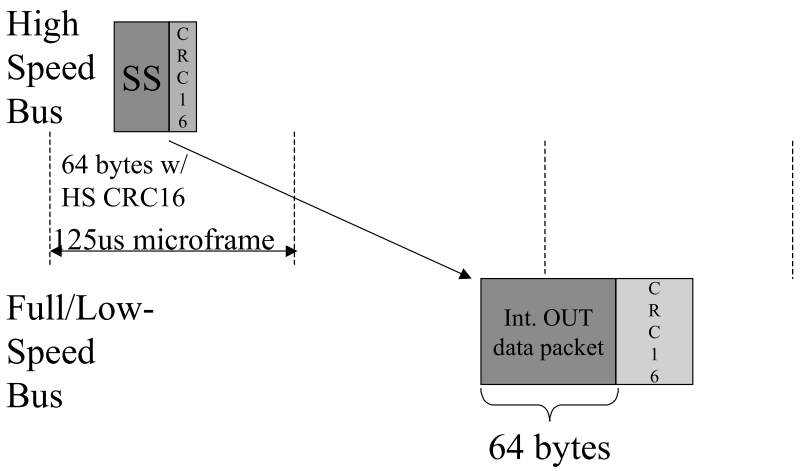



**Figure 11-81. Example of CRC16 Handling for Interrupt OUT**

The start-split transaction for an interrupt OUT transaction includes a normal CRC16 field for the high-speed data packet of the data phase of the start-split transaction. However, the data payload of the data packet contains only the data payload of the corresponding full-/low-speed data packet; i.e., there is only a single CRC16 in the data packet of 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 of the data packet. If the start-split has a CRC check failure, the full-speed transaction must not be started on the downstream bus. Figure 11-81 shows an example of the CRC16 handling for an interrupt OUT transaction and its start-split.

## 11.20.4 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 the next microframe (as described previously), these transactions are available to be issued by the full-/low-speed handler on the downstream full-/low-speed bus in the order they were saved in the start-split pipeline stage. The full-/low-speed handler issues each transaction on the downstream facing bus. The full-/low-speed handler responds to the full-/low-speed transaction with an appropriate handshake as described in Chapter 8. The full-/low-speed handler saves the results of the transaction (data, NAK, STALL, trans_err) in the complete-split pipeline stage.

During following microframes, the host controller issues high-speed complete-split transactions to retrieve the data/handshake from the high-speed handler. When the high-speed handler receives s complete-split transaction, the TT returns whatever data it has received during a microframe. If the full-/low-speed transaction was started and completed in a single microframe, the TT returns all the data for the transaction in the complete-split response occurring in the following microframe. If the full-/low-speed CRC check passes, the appropriate DATA0/1 PID for the data packet is used. If the full-/low-speed CRC check fails, an ERR handshake is used and there is no data packet as part of the complete-split transaction.

If the full-/low-speed transaction spanned a microframe, the TT requires two complete-splits (in two subsequent microframes) to return all the data for the full-/low-speed transaction. The data packet PID for the first complete-split must be an MDATA to tell the host controller that another complete-split is required for this endpoint. This MDATA response is made without performing a CRC check (since the CRC16 field has not yet been received on the full-/low-speed bus). The complete-split in the next microframe must use a DATA0/1 PID if the CRC check passes. If the CRC check fails, an ERR handshake response is made instead and there is no data packet as part of the complete-split transaction. Since full-speed interrupt transactions are limited to 64 data bytes or less (and low-speed interrupt transactions are limited to 8 data

bytes or less), no full-/low-speed interrupt transaction can span more than a single microframe boundary; i.e., no more than two microframes are ever required to complete the transaction.

The complete-split transaction for an interrupt IN transaction must not include the CRC16 field received from the full-/low-speed data packet (i.e., only a high-speed CRC16 field is used in split transactions). The TT must use a high-speed CRC16 on each complete-split data packet. If the full-speed handler detects a failed CRC check, it must use an ERR handshake response in the complete-split transaction to reflect that error to the high-speed host controller. The host controller must check the CRC16 on each returned complete-split data packet. A CRC failure (or ERR handshake) on any (partial) complete-split is reflected as a CRC failure on the total full-/low-speed transaction. This means that for a case where a full-/low-speed interrupt spans a microframe boundary, the host controller can accept the first complete-split without errors, then the second complete-split can indicate that the data from the first complete-split must be rejected as if it were never received by the host controller. Figure 11-82 shows an example of an interrupt IN and its CRC16 handling with corresponding complete-split responses.

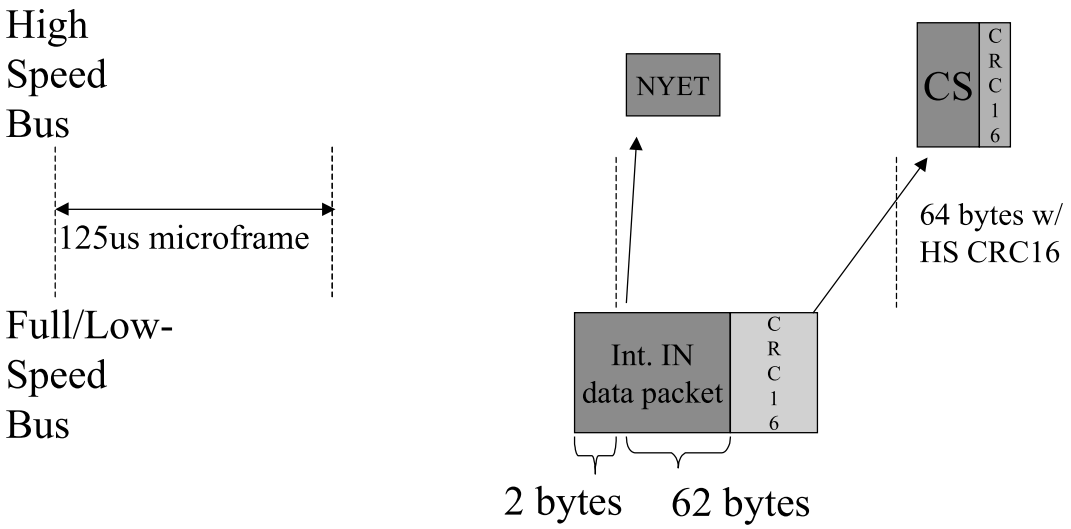



**Figure 11-82. Example of CRC16 Handling for Interrupt IN**

## 11.21 Isochronous Transaction Translation Overview

Isochronous split transactions are handled by the host by scheduling start- and complete-split transactions as described previously. Isochronous IN split transactions have more than two schedule entries:

- One entry for the start-split transaction in the microframe before the earliest the full-speed transaction can occur
- Other entries for the complete-splits in microframes after the data 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 multiple high-speed split transactions to move data to/from the TT. This allows any alignment of the data for each microframe.

Full-speed isochronous OUT transactions issued by a TT do not have corresponding 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 of the TT 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 a full-speed isochronous transfer type for a single endpoint. These figures must not be interpreted as showing any particular specific timing. In particular, high-speed or full-speed transactions for other endpoints may occur before or after these split transactions. 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 transaction errors (including timeout) of an isochronous transaction.

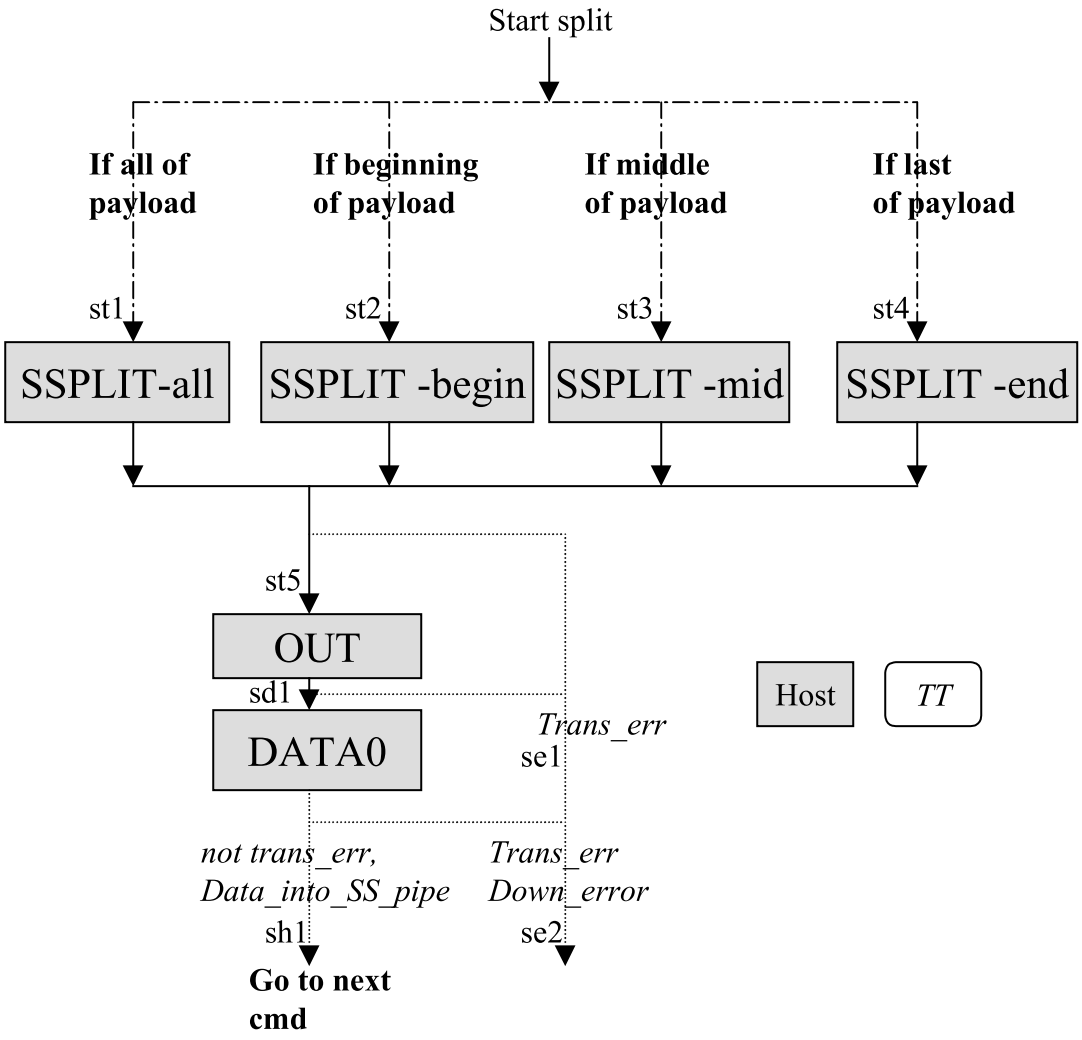



**Figure 11-83. Isochronous OUT Start-split Transaction Sequence**

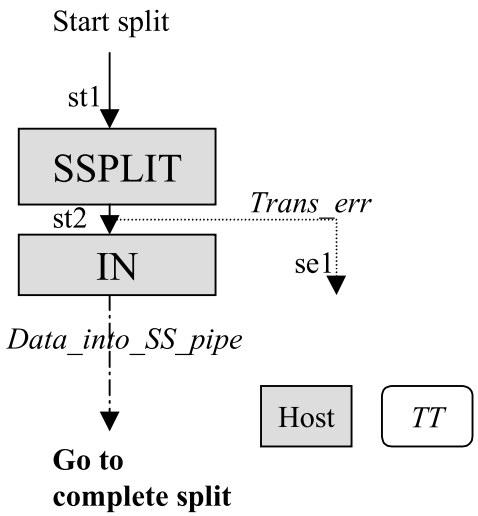



**Figure 11-84. Isochronous IN Start-split Transaction Sequence**

In Figure 11-85, the high-speed handler returns an ERR handshake for a “transaction error” of the full-speed transaction.

The high-speed handler returns an NYET handshake when it cannot 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 has 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 starting 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.

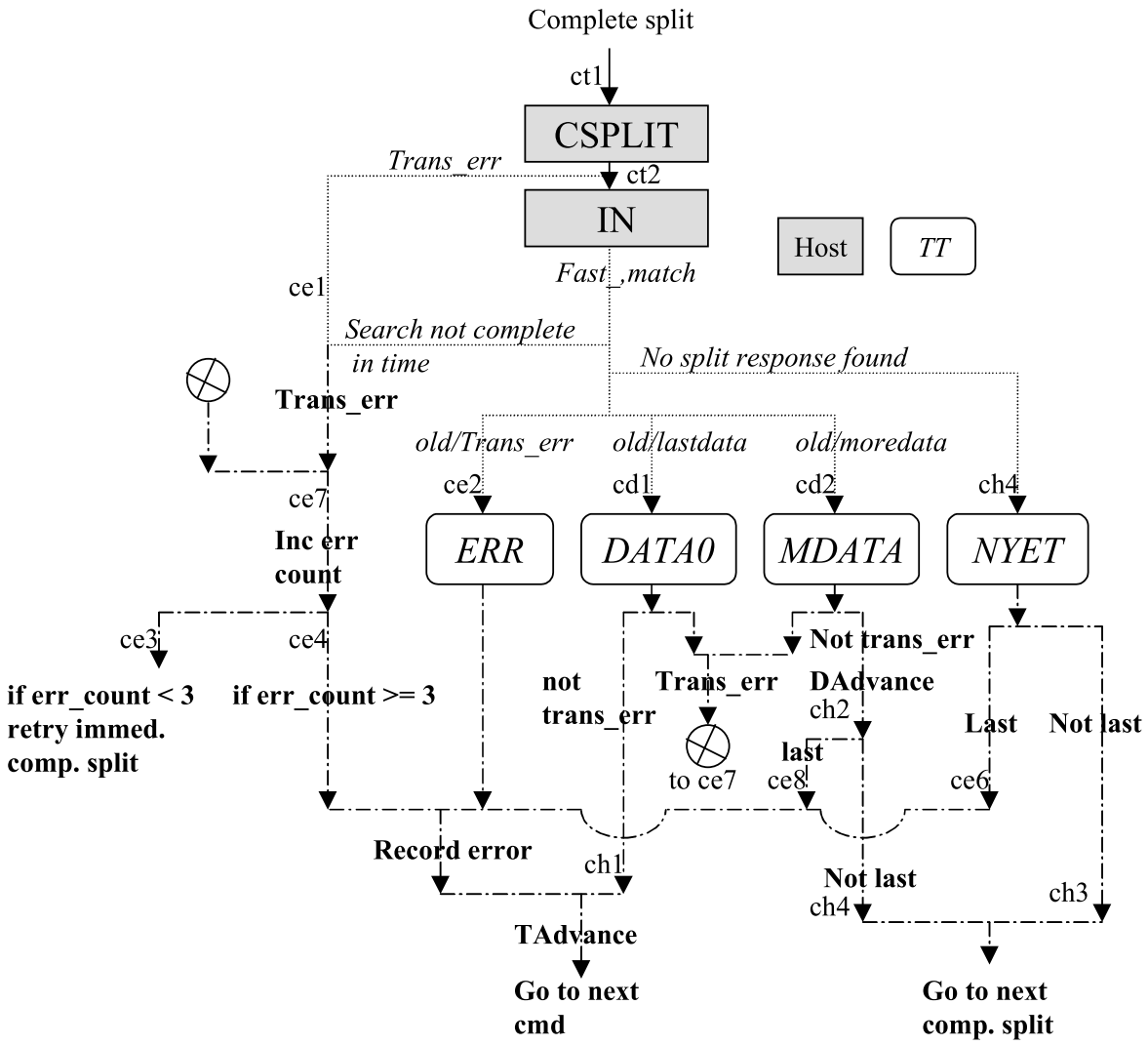



**Figure 11-85. Isochronous IN Complete-split Transaction Sequence**

## 11.21.2 Isochronous Split Transaction State Machines

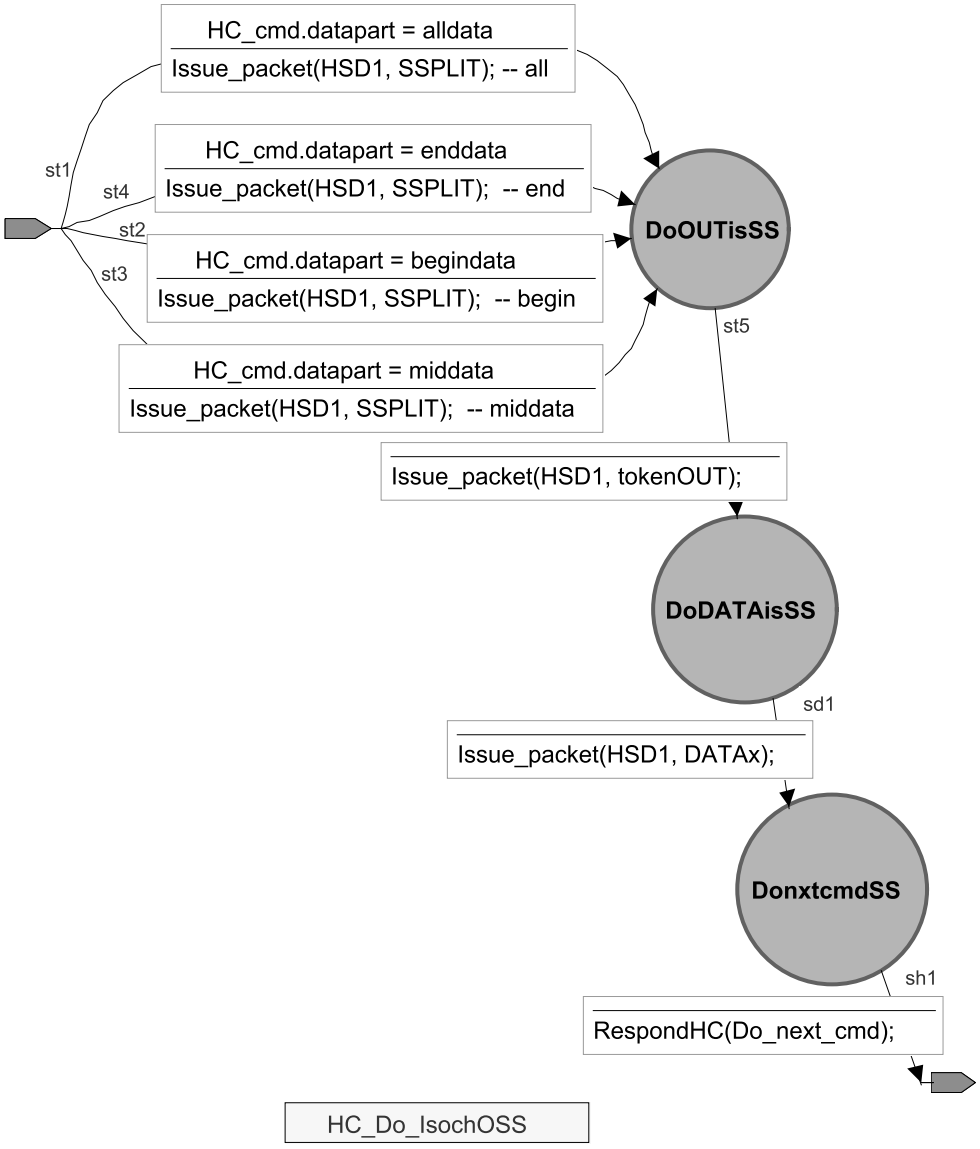



**Figure 11-86. Isochronous OUT Start-split Transaction Host State Machine**

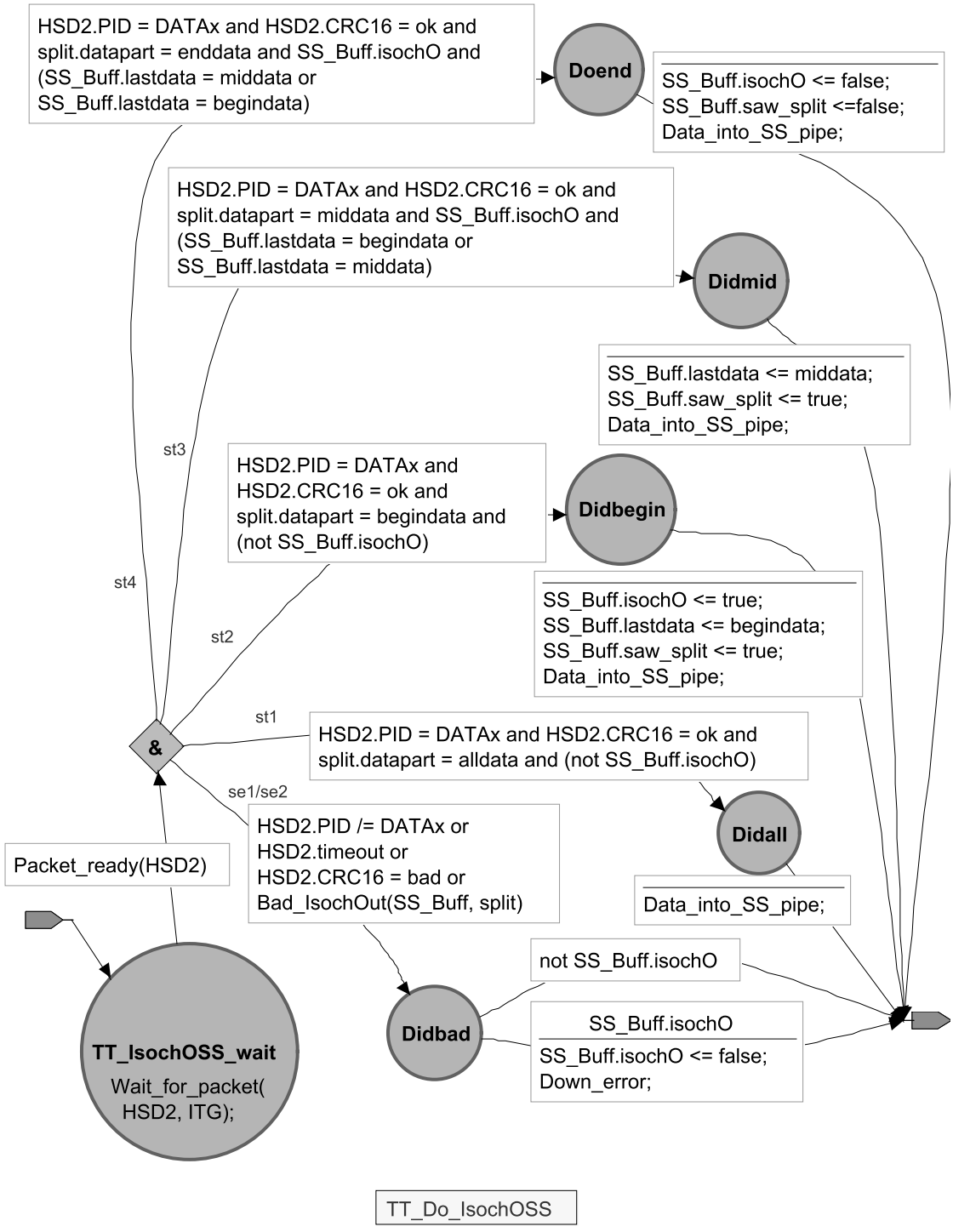



**Figure 11-87. Isochronous OUT Start-split Transaction TT State Machine**

There is a condition in Figure 11-87 on transition se1/se2 labeled “Bad_IsochOut”. This condition is true when none of the conditions on transitions st1 through st4 are true. The action labeled “Down_error” records an error to be indicated on the downstream facing full-speed bus for the transaction corresponding to this start-split.

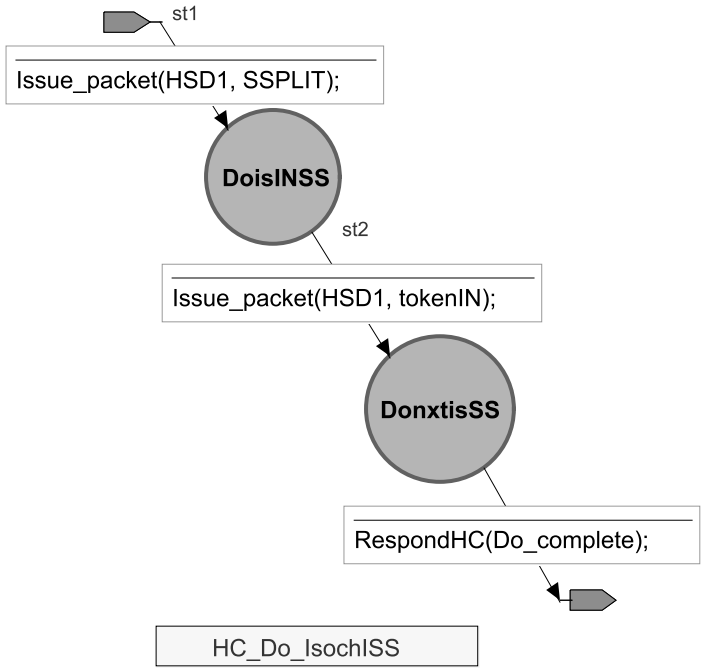



**Figure 11-88. Isochronous IN Start-split Transaction Host State Machine**

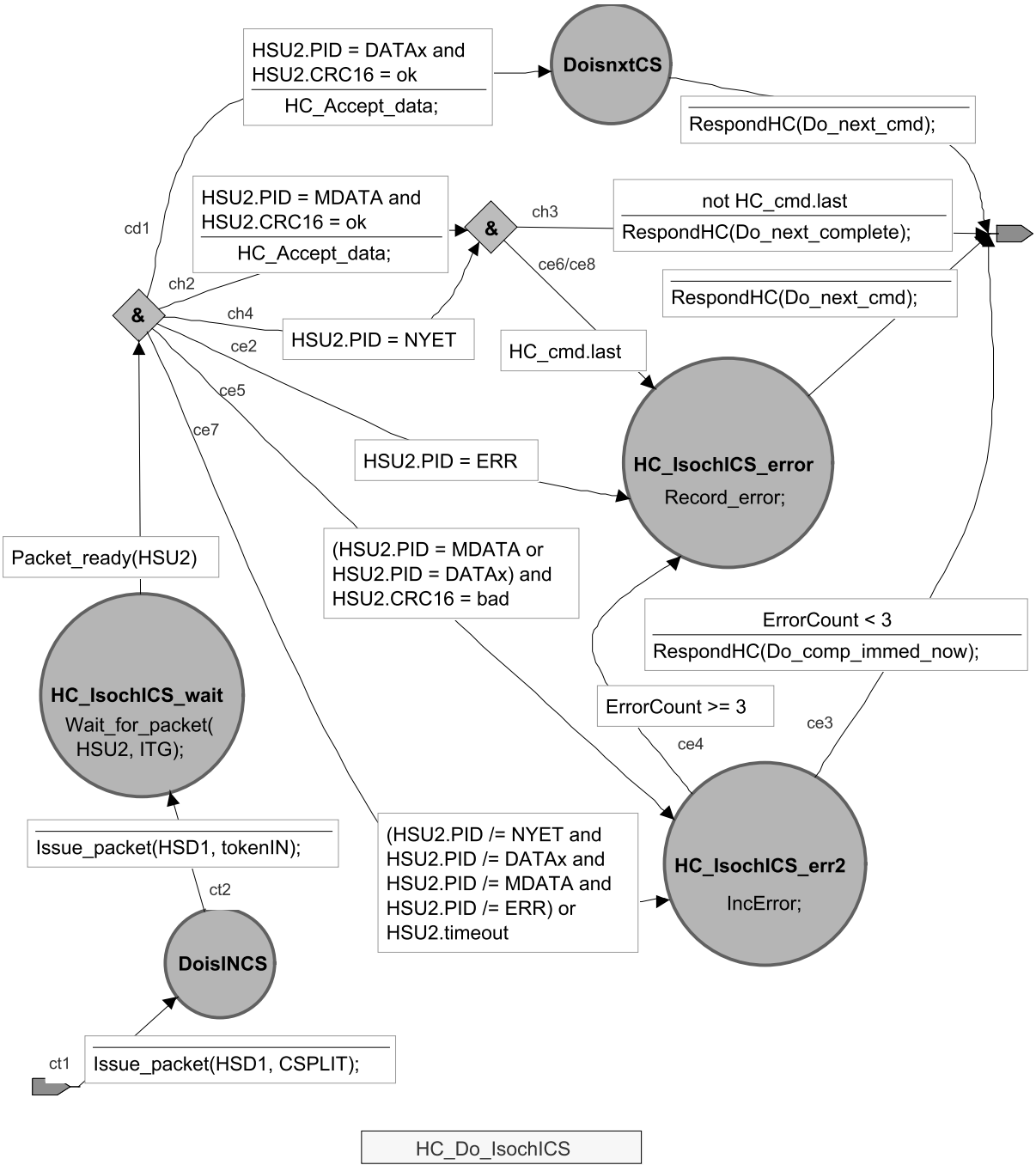



**Figure 11-89. Isochronous IN Complete-split Transaction Host State Machine**

In Figure 11-89, the transition “ce8” occurs when the high-speed handler responds with an MDATA to indicate there is more data for the full-speed transaction, but the host controller knows that this is the last scheduled complete-split for this endpoint for this frame. If a DATA0 response from the high-speed handler is not received before the last scheduled complete-split, the host controller records an error and proceeds to the next transaction for this endpoint (in the next frame).

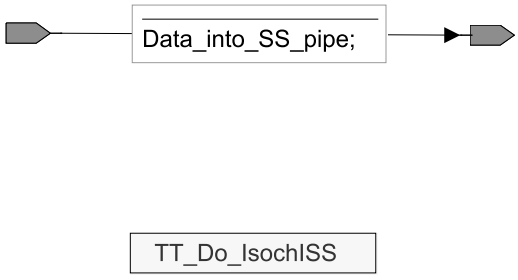



**Figure 11-90. Isochronous IN Start-split Transaction TT State Machine**

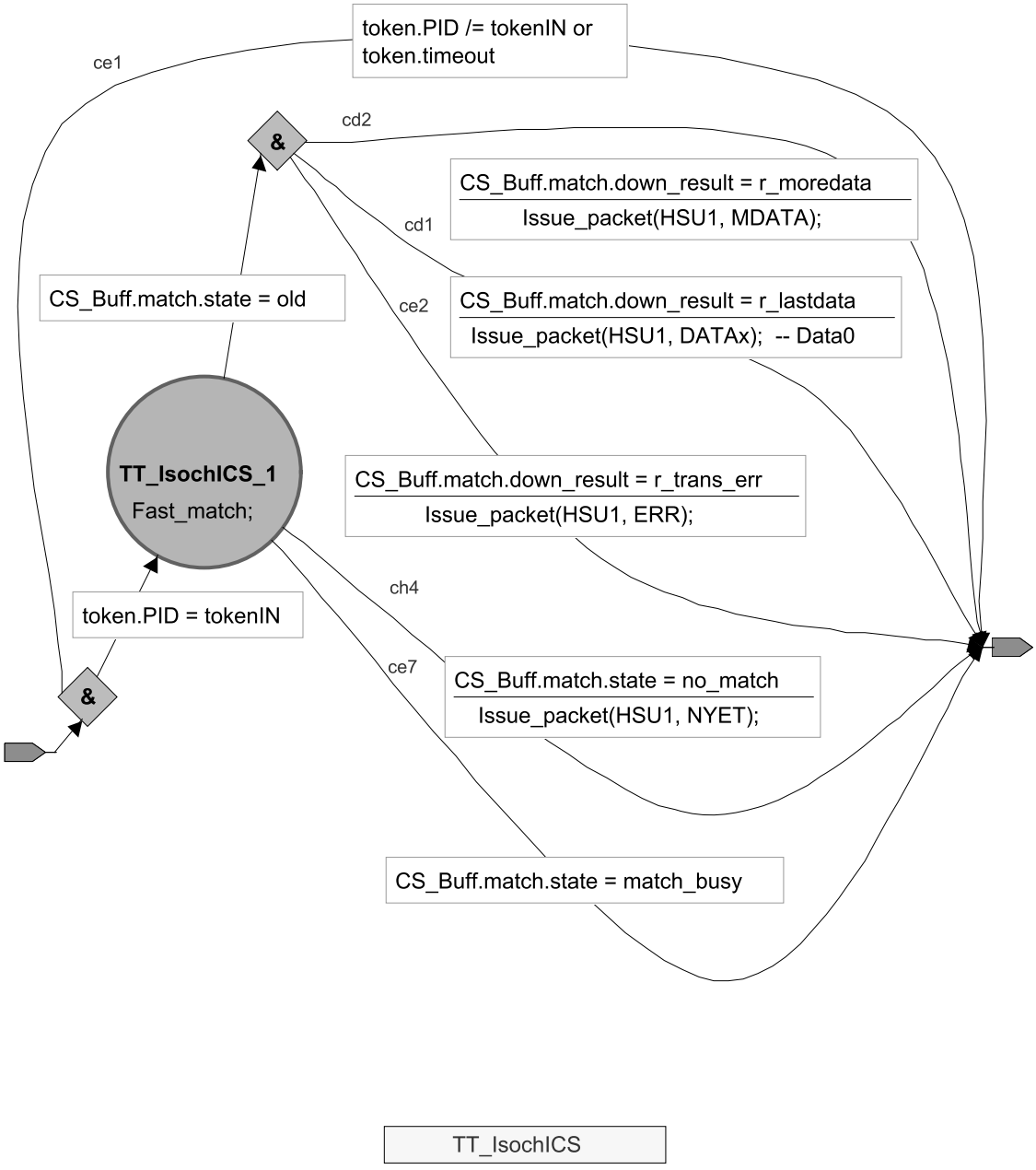



**Figure 11-91. Isochronous IN Complete-split Transaction TT State Machine**

### 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 "SSPLIT-begin" start-split transaction token packet, the high-speed handler must track start-split transactions that are received for this endpoint. The high-speed handler must track that a start-split transaction is received each and every microframe until an "SSPLIT-end" split transaction token packet is received for this endpoint. 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 forces a bitstuff error on the full-speed transaction. Any subsequent "SPLIT-middle" or "SPLIT-end" start-splits for the same endpoint must be ignored until the next non "SPLIT-middle" and non "SPLIT-end" is received (for any endpoint supported by this TT).

The start-split transaction for an isochronous OUT transaction must not include the CRC16 field for the full-speed data packet. For a full-speed transaction, the host would compute the CRC16 of the data packet for the full data packet (e.g., a 1023 byte data packet uses a single CRC16 field that is computed once by the host controller). For a split transaction, any isochronous OUT full-speed transaction is subdivided into multiple start-splits, each with a data payload of 188 bytes or less. For each of these start-splits, the host computes a high-speed CRC16 field for each start-split data packet. The TT high-speed handler must check each high-speed CRC16 value on each start-split. The TT full-speed handler must locally generate the CRC16 value for the complete full-speed data packet. Figure 11-92 shows an example of a full-speed isochronous OUT packet and the high-speed start-splits with their CRC16 fields.

If there is a CRC check failure on the high-speed start-split, the high-speed handler must indicate to the full-speed handler that there was an error in the start-split for the full-speed transaction. If the transaction has been indicated as having a CRC failure (or if there is a missed start-split), 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.

Additional high-speed start-split transactions for the same endpoint must be ignored after a CRC check fails, until the high-speed handler receives either an "SSPLIT-end" start-split transaction token packet for that endpoint or a start-split for a different endpoint.

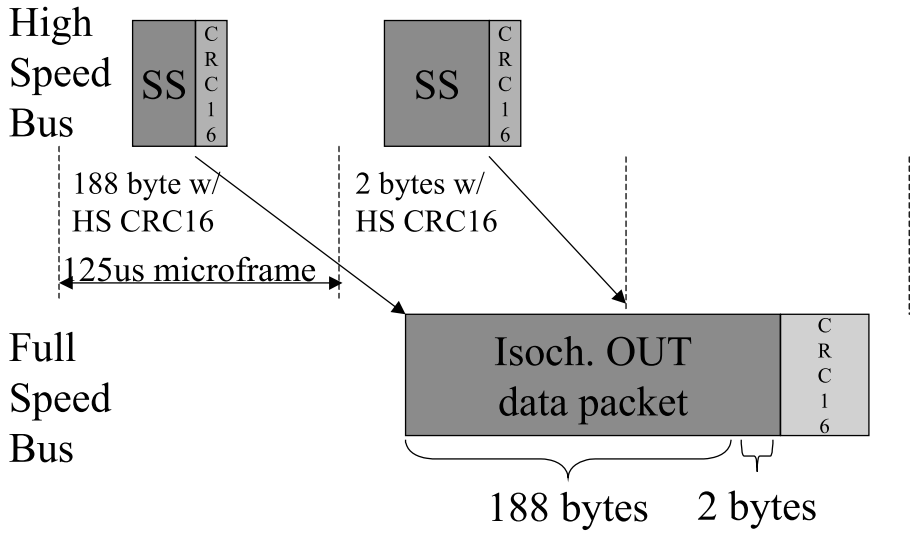



**Figure 11-92. Example of CRC16 Isochronous OUT Data Packet Handling**

### 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 high-speed CRC16 values for complete-split transactions. If the full-speed handler detects a failed CRC check at the end of the data packet (e.g., after potentially several complete-split transactions on high-speed), the handler must use an ERR handshake response to reflect that error to the high-speed host controller. The host controller must check the CRC16 on each returned high-speed 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. Figure 11-93 shows an example of the relationships of the full-speed data packet and the high-speed complete-splits and their CRC16 fields.

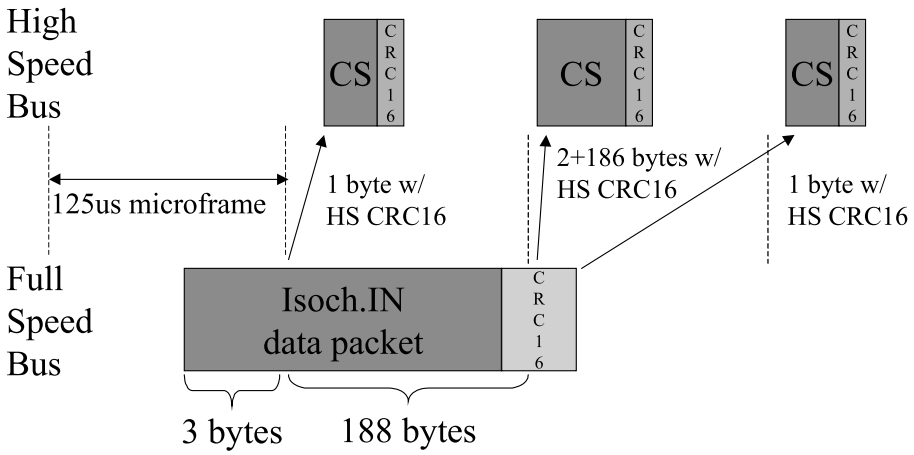



**Figure 11-93. Example of CRC16 Isochronous IN Data Packet Handling**

## 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 EOF2 of the downstream facing bus, it must disable the downstream facing port that is currently transmitting.
- If the TT is transmitting a packet near EOF1 of the downstream facing bus, it must force an abnormal termination sequence as defined in Section 11.3.3 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 EOF1 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 allowed for the transfer type in its determination. Periodic transactions do not need to be included in this test since the microframe pipeline is maintained separately.

### 11.22.1 Loss of TT Synchronization With HS SOFs

The hub has a timer it uses for (micro)frame maintenance. It has a 1 ms frame timer when operating at full-/low-speed for enforcing EOF with downstream connected devices. It has a 125 µs microframe timer when operating at high-speed for enforcing EOF with high-speed devices. It also uses the 125 µs microframe timer to create a 1 ms frame timer for enforcing EOF with downstream full-/low-speed devices when operating at high-speed. The hub (micro)frame timer must always stay synchronized with host generated SOFs to keep the bus operating correctly

In normal hub repeater (full- or high-speed) operation (e.g., not involving a TT), the (micro)frame timer loses synchronization whenever it has missed SOFs for three consecutive microframes. While timer synchronization is lost, the hub does not establish upstream connectivity. Downstream connectivity is established normally, even when timer synchronization is lost. When the timer is synchronized, the hub allows upstream connectivity to be established when required. The hub is responsible for ensuring that there is no signaling being repeated/transmitted upstream from a device after the EOF2 point in any (micro)frame. The hub must not establish upstream connectivity if it has lost (micro)frame timer synchronization since it no longer knows accurately where the EOF2 point is.

### 11.22.2 TT Frame and Microframe Timer Synchronization Requirements

When the hub is operating at high-speed and has full-/low-speed devices connected on its downstream facing ports (e.g., a TT is active), the hub has additional responsibilities beyond enforcement of the (high-speed) EOF2 point on its upstream facing port in every microframe. The TT must also generate full-speed SOFs downstream and ensure that the TT operates correctly (in bridging high-speed and full-/low-speed operation).

A high-speed operating hub synchronizes its microframe timer to 125 µs SOFs. However, in order to generate full-speed downstream SOFs, it must also have a 1 ms frame timer. It generates this 1 ms frame timer by recognizing zeroth microframe SOFs, e.g., a high-speed SOF when the frame number value changes compared to SOF of the immediately previous microframe.

In order to create the 1 ms frame timer, the hub must successfully receive a zeroth microframe SOF after its microframe timer is synchronized. In order to recognize a zeroth microframe SOF, the hub must successfully receive SOFs for two consecutive microframes where the frame number increments by 1 (mod 2^11). When the hub has done this, it knows that the second SOF is a zeroth microframe SOF and thereby establishes a 1 ms frame timer starting time. Note that a hub can synchronize both timers with as few as two SOFs if the SOFs are for microframe 7 and microframe 0, i.e., if the second SOF is a zeroth microframe SOF.

Once the hub has synchronized its 1 ms frame timer, it can keep that timer synchronized as long as it keeps its 125 µs microframe timer synchronized (since it knows that every 8 microframes from the zeroth microframe SOF is a 1 ms frame). In particular, the hub can keep its frame timer synchronized even if it misses zeroth microframe SOFs (as long as the microframe timer stays synchronized).

So in summary, the hub can synchronize its 125 µs microframe timer after receiving SOFs of two consecutive microframes. It synchronizes its 1 ms frame timer when it receives a zeroth microframe SOF (and the microframe timer is synchronized). The 125 µs microframe timer loses synchronization after three SOFs for consecutive microframes have been missed. This also causes the 1 ms frame timer to lose synchronization at the same time.

The TT must only generate full-speed SOFs downstream when its 1 ms frame timer is synchronized.

Correct internal operation of the TT is dependent on both timers. The TT must accurately know when microframes occur to enforce its microframe pipeline abort/free rules. It knows this based on a synchronized microframe timer (for generally incrementing the microframe number) and a synchronized frame timer (to know when the zeroth microframe occurs).

Since loss of microframe timer synchronization immediately causes loss of frame timer synchronization, the TT stops normal operation once the microframe timer loses synchronization. In an error free environment, microframe timer synchronization can be restored after receiving the two SOFs for the next two consecutive microframes (e.g., synchronization is restored at least 250 µs after synchronization loss). As long as SOFs are not missed, frame timer synchronization will be restored in less than 1 ms after microframe synchronization. Note that frame timer synchronization can be restored in a high-speed operating case in much less time (0.250-1.250 ms) than the 2-3 ms required in full-speed operation. Once the frame timer is synchronized, SOFs can be issued on downstream facing full-speed ports for the beginning of the next frame.

Once the hub detects loss of microframe timer synchronization, its TT(s):

1. Must respond to periodic complete-splits with any responses buffered in the periodic pipeline (only good for at most 1 microframe of complete-splits).
2. Must abort any buffered periodic start-split transactions in the periodic pipeline.
3. Must ignore any high-speed periodic start-splits.
4. Must stop issuing full-speed SOFs on downstream facing full-speed ports (and low-speed keep-alives on low-speed ports).
5. Must not start issuing subsequent periodic full-/low-speed transactions on downstream facing full-/low-speed ports.
6. Must respond to high-speed start-split bulk/control transactions.
7. Buffered bulk/control results must respond to high-speed complete-split transactions.
8. Pending bulk/control transactions must not be issued to full-/low-speed downstream facing ports. The TT buffers used to hold bulk/control transactions must be preserved until the microframe timer is re-synchronized. (Or until a Clear_TT_Buffer request is received for the transaction).

Note that in any case a TT must not issue transactions of any speed on downstream facing ports when its upstream facing port is suspended.

A TT only restores normal operation on downstream facing full-/low-speed ports after both microframe and frame timers are synchronized. Figure Figure 11-94 summarizes the relationship between high-speed SOFs and the TT frame and microframe timer synchronization requirements on start-splits.

For suspend sequencing of a hub, a hub will first lose microframe/frame timer synchronization at the same time. This will cause its TT(s) to stop issuing SOFs (which should be the only transactions keeping the downstream facing full-/low-speed ports out of suspend). Then the hub (along with any downstream devices) will enter suspend.

Upon a resume, the hub will first restore its microframe timer synchronization (after high-speed transactions continue). Then in less than 1 ms (assuming no errors), the frame timer will be synchronized and the TT can start normal operation (including SOFs/keep-alives on downstream facing full-/low-speed ports).

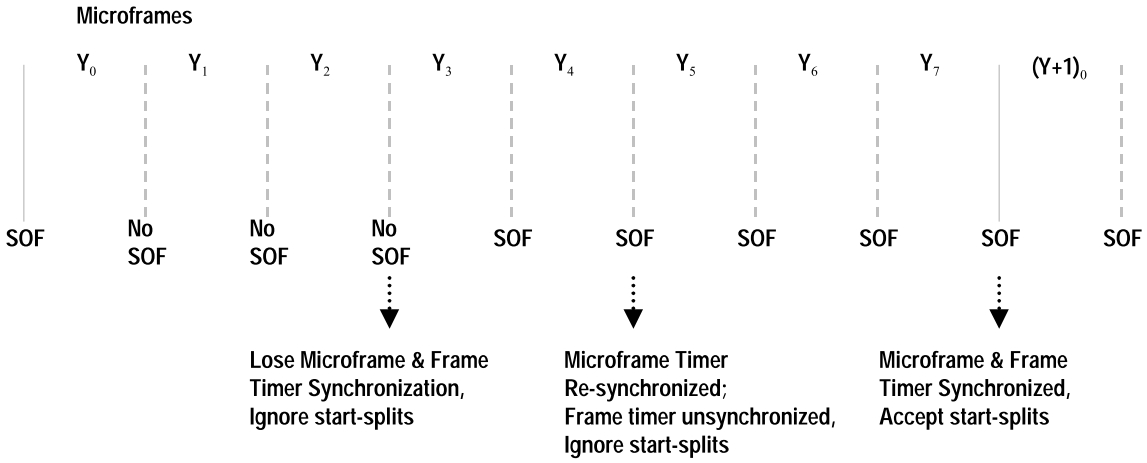



**Figure 11-94. Example Frame/Microframe Synchronization Events**

## 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 for Hub Class

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

A hub returns different descriptors based on whether it is operating at high-speed or full-/low-speed. A hub can report three different sets of the descriptors: one descriptor set for full-/low-speed operation and two sets for high-speed operation.

A hub operating at full-/low-speed has a device descriptor with a bDeviceProtocol field set to zero(0) and an interface descriptor with a bInterfaceProtocol field set to zero(0). The rest of the descriptors are the same for all speeds.

A hub operating at high-speed can have one of two TT organizations: single TT or multiple TT. All hubs must support the single TT organization. A multiple TT hub has an additional interface descriptor (with a corresponding endpoint descriptor). The first set of descriptors shown below must be provided by all hubs. A hub that has a single TT must set the bDeviceProtocol field of the device descriptor to one(1) and the interface descriptor bInterfaceProtocol field set to 0.

A multiple TT hub must set the bDeviceProtocol field of the device descriptor to two (2). The first interface descriptor has the bInterfaceProtocol field set to one(1). Such a hub also has a second interface descriptor where the bInterfaceProtocol is set to two(2). When the hub is configured with an interface protocol of one(1), it will operate as a single TT organized hub. When the hub is configured with an interface protocol of two(2), it will operate as a multiple TT organized hub. The TT organization must not be changed while the hub has full-/low-speed transactions in progress.

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.

**Full-/Low-speed Operating Hub**

Device Descriptor (full-speed information):

| | |
|---|---|
| *bLength* | 12H |
| *bDescriptorType* | 1 |
| *bcdUSB* | 0200H |
| *bDeviceClass* | HUB_CLASSCODE (09H) |
| *bDeviceSubClass* | 0 |
| *bDeviceProtocol* | 0 |
| *bMaxPacketSize0* | 64 |
| *bNumConfigurations* | 1 |

Device_Qualifier Descriptor (high-speed information):

| | |
|---|---|
| *bLength* | 0AH |
| *bDescriptorType* | 6 |
| *bcdUSB* | 200H |
| *bDeviceClass* | HUB_CLASSCODE (09H) |
| *bDeviceSubClass* | 0 |
| *bDeviceProtocol* | 1 (for single TT) or 2 (for multiple TT) |
| *bMaxPacketSize0* | 64 |
| *bNumConfigurations* | 1 |

Configuration Descriptor (full-speed information):

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 2 |
| *wTotalLength* | N |
| *bNumInterfaces* | 1 |
| *bConfigurationValue* | X |
| *iConfiguration* | Y |
| *bmAttributes* | Z |
| *bMaxPower* | The maximum amount of bus power the hub will consume in full-/low-speed configuration |

Interface Descriptor:

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 4 |
| *bInterfaceNumber* | 0 |
| *bAlternateSetting* | 0 |
| *bNumEndpoints* | 1 |
| *bInterfaceClass* | HUB_CLASSCODE (09H) |
| *bInterfaceSubClass* | 0 |
| *bInterfaceProtocol* | 0 |
| *iInterface* | i |

Endpoint Descriptor (for Status Change Endpoint):

| | |
|---|---|
| *bLength* | 07H |
| *bDescriptorType* | 5 |
| *bEndpointAddress* | Implementation-dependent;<br>Bit 7: Direction = In(1) |
| *bmAttributes* | Transfer Type = Interrupt (00000011B) |
| *wMaxPacketSize* | Implementation-dependent |
| *bInterval* | FFH (Maximum allowable interval) |

Other_Speed_Configuration Descriptor (High-speed information):

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 7 |
| *wTotalLength* | N |
| *bNumInterfaces* | 1 (for single TT) or 2 (for multiple TT) |
| *bConfigurationValue* | X |
| *iConfiguration* | Y |
| *bmAttributes* | Z |
| *bMaxPower* | The maximum amount of bus power the hub will consume in high-speed configuration |

Interface Descriptor:

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 4 |
| *bInterfaceNumber* | 0 |
| *bAlternateSetting* | 0 |
| *bNumEndpoints* | 1 |
| *bInterfaceClass* | HUB_CLASSCODE (09H) |
| *bInterfaceSubClass* | 0 |
| *bInterfaceProtocol* | 0 (for single TT)<br>1 (for multiple TT) |
| *iInterface* | i |

Endpoint Descriptor (for Status Change Endpoint):

| | |
|---|---|
| *bLength* | 07H |
| *bDescriptorType* | 5 |
| *bEndpointAddress* | Implementation-dependent;<br>Bit 7: Direction = In(1) |
| *bmAttributes* | Transfer Type = Interrupt<br>(00000011B ) |
| *wMaxPacketSize* | Implementation-dependent |
| *bInterval* | FFH (Maximum allowable interval) |

Interface Descriptor (present if multiple TT hub):

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 4 |
| *bInterfaceNumber* | 0 |
| *bAlternateSetting* | 0 |
| *bNumEndpoints* | 1 |
| *bInterfaceClass* | HUB_CLASSCODE (09H) |
| *bInterfaceSubClass* | 0 |
| *bInterfaceProtocol* | 2 |
| *iInterface* | i |

Endpoint Descriptor (present if multiple TT hub):

| | |
|---|---|
| *bLength* | 07H |
| *bDescriptorType* | 5 |
| *bEndpointAddress* | Implementation-dependent; Bit 7: Direction = In(1) |
| *bmAttributes* | Transfer Type = Interrupt (00000011B ) |
| *wMaxPacketSize* | Implementation-dependent |
| *bInterval* | FFH (Maximum allowable interval) |

**High-speed Operating Hub with Single TT**

Device Descriptor (High-speed information):

| | |
|---|---|
| *bLength* | 12H |
| *bDescriptorType* | 1 |
| *bcdUSB* | 200H |
| *bDeviceClass* | HUB_CLASSCODE (09H) |
| *bDeviceSubClass* | 0 |
| *bDeviceProtocol* | 1 |
| *bMaxPacketSize0* | 64 |
| *bNumConfigurations* | 1 |

Device_Qualifier Descriptor (full-speed information):

| | |
|---|---|
| *bLength* | 0AH |
| *bDescriptorType* | 6 |
| *bcdUSB* | 200H |
| *bDeviceClass* | HUB_CLASSCODE (09H) |
| *bDeviceSubClass* | 0 |
| *bDeviceProtocol* | 0 |
| *bMaxPacketSize0* | 64 |
| *bNumConfigurations* | 1 |

Configuration Descriptor (high-speed information):

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 2 |
| *wTotalLength* | N |
| *bNumInterfaces* | 1 |
| *bConfigurationValue* | X |
| *iConfiguration* | Y |
| *bmAttributes* | Z |
| *bMaxPower* | The maximum amount of bus power the hub will consume in this configuration |

Interface Descriptor:

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 4 |
| *bInterfaceNumber* | 0 |
| *bAlternateSetting* | 0 |
| *bNumEndpoints* | 1 |
| *bInterfaceClass* | HUB_CLASSCODE (09H) |
| *bInterfaceSubClass* | 0 |
| *bInterfaceProtocol* | 0 (single TT) |
| *iInterface* | i |

Endpoint Descriptor (for Status Change Endpoint):

| | |
|---|---|
| *bLength* | 07H |
| *bDescriptorType* | 5 |
| *bEndpointAddress* | Implementation-dependent;<br>Bit 7: Direction = In(1) |
| *bmAttributes* | Transfer Type = Interrupt (00000011B) |
| *wMaxPacketSize* | Implementation-dependent |
| *bInterval* | FFH (Maximum allowable interval) |

Other_Speed_Configuration Descriptor (full-speed information):

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 7 |
| *wTotalLength* | N |
| *bNumInterfaces* | 1 |
| *bConfigurationValue* | X |
| *iConfiguration* | Y |
| *bmAttributes* | Z |
| *bMaxPower* | The maximum amount of bus power the hub will consume in high-speed configuration |

Interface Descriptor:

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 4 |
| *bInterfaceNumber* | 0 |
| *bAlternateSetting* | 0 |
| *bNumEndpoints* | 1 |
| *bInterfaceClass* | HUB_CLASSCODE (09H) |
| *bInterfaceSubClass* | 0 |
| *bInterfaceProtocol* | 0 |
| *iInterface* | i |

Endpoint Descriptor (for Status Change Endpoint):

| | |
|---|---|
| *bLength* | 07H |
| *bDescriptorType* | 5 |
| *bEndpointAddress* | Implementation-dependent;<br>Bit 7: Direction = In(1) |
| *bmAttributes* | Transfer Type = Interrupt (00000011B ) |
| *wMaxPacketSize* | Implementation-dependent |
| *bInterval* | FFH (Maximum allowable interval) |

### High-speed Operating Hub with Multiple TTs

Device Descriptor (High-speed information):

| | |
|---|---|
| *bLength* | 12H |
| *bDescriptorType* | 1 |
| *bcdUSB* | 200H |
| *bDeviceClass* | HUB_CLASSCODE (09H) |
| *bDeviceSubClass* | 0 |
| *bDeviceProtocol* | 2 (multiple TTs) |
| *bMaxPacketSize0* | 64 |
| *bNumConfigurations* | 1 |

Device_Qualifier Descriptor (full-speed information):

| | |
|---|---|
| *bLength* | 0AH |
| *bDescriptorType* | 6 |
| *bcdUSB* | 200H |
| *bDeviceClass* | HUB_CLASSCODE (09H) |
| *bDeviceSubClass* | 0 |
| *bDeviceProtocol* | 0 |
| *bMaxPacketSize0* | 64 |
| *bNumConfigurations* | 1 |

Configuration Descriptor (high-speed information):

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 2 |
| *wTotalLength* | N |
| *bNumInterfaces* | 1 |
| *bConfigurationValue* | X |
| *iConfiguration* | Y |
| *bmAttributes* | Z |
| *bMaxPower* | The maximum amount of bus power the hub will consume in this configuration |

Interface Descriptor:

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 4 |
| *bInterfaceNumber* | 0 |
| *bAlternateSetting* | 0 |
| *bNumEndpoints* | 1 |
| *bInterfaceClass* | HUB_CLASSCODE (09H) |
| *bInterfaceSubClass* | 0 |
| *bInterfaceProtocol* | 1 (single TT) |
| *iInterface* | i |

Endpoint Descriptor (for Status Change Endpoint):

| | |
|---|---|
| *bLength* | 07H |
| *bDescriptorType* | 5 |
| *bEndpointAddress* | Implementation-dependent;<br>Bit 7: Direction = In(1) |
| *bmAttributes* | Transfer Type = Interrupt (00000011B) |
| *wMaxPacketSize* | Implementation-dependent |
| *bInterval* | FFH (Maximum allowable interval) |

Interface Descriptor:

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 4 |
| *bInterfaceNumber* | 0 |
| *bAlternateSetting* | 1 |
| *bNumEndpoints* | 1 |
| *bInterfaceClass* | HUB_CLASSCODE (09H) |
| *bInterfaceSubClass* | 0 |
| *bInterfaceProtocol* | 2 (multiple TTs) |
| *iInterface* | i |

Endpoint Descriptor:

| | |
|---|---|
| *bLength* | 07H |
| *bDescriptorType* | 5 |
| *bEndpointAddress* | Implementation-dependent;<br>Bit 7: Direction = In(1) |
| *bmAttributes* | Transfer Type = Interrupt<br>(00000011B ) |
| *wMaxPacketSize* | Implementation-dependent |
| *bInterval* | FFH (Maximum allowable interval) |

Other_Speed_Configuration Descriptor (full-speed information):

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 7 |
| *wTotalLength* | N |
| *bNumInterfaces* | 1 |
| *bConfigurationValue* | X |
| *iConfiguration* | Y |
| *bmAttributes* | Z |
| *bMaxPower* | The maximum amount of bus power the hub will consume in high-speed configuration |

Interface Descriptor:

| | |
|---|---|
| *bLength* | 09H |
| *bDescriptorType* | 4 |
| *bInterfaceNumber* | 0 |
| *bAlternateSetting* | 0 |
| *bNumEndpoints* | 1 |
| *bInterfaceClass* | HUB_CLASSCODE (09H) |
| *bInterfaceSubClass* | 0 |
| *bInterfaceProtocol* | 0 |
| *iInterface* | i |

Endpoint Descriptor (for Status Change Endpoint):

| | |
|---|---|
| *bLength* | 07H |
| *bDescriptorType* | 5 |
| *bEndpointAddress* | Implementation-dependent;<br>Bit 7: Direction = In(1) |
| *bmAttributes* | Transfer Type = Interrupt (00000011B) |
| *wMaxPacketSize* | Implementation-dependent |
| *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 11-13 outlines the various fields contained by the hub descriptor.

**Table 11-13. 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 facing ports that this hub supports |
| 3 | *wHubCharacteristics* | 2 | D1...D0: Logical Power Switching Mode<br>00: Ganged power switching (all ports' power at once)<br>01: Individual port power switching<br>1X: Reserved. Used only on 1.0 compliant hubs that implement no power switching<br><br>D2: Identifies a Compound Device<br>0: Hub is not part of a compound device.<br>1: Hub is part of a compound device.<br><br>D4...D3: Over-current Protection Mode<br>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.<br>01: Individual Port Over-current Protection. The hub reports over-current on a per-port basis. Each port has an over-current status.<br>1X: No Over-current Protection. This option is allowed only for bus-powered hubs that do not implement over-current protection. |

| Offset | Field | Size | Description |
|---|---|---|---|
| | | | D6...D5: TT Think TIme<br>00: TT requires at most 8 FS bit times of inter transaction gap on a full-/low-speed downstream bus.<br>01: TT requires at most 16 FS bit times.<br>10: TT requires at most 24 FS bit times.<br>11: TT requires at most 32 FS bit times.<br><br>D7: Port Indicators Supported<br>0: Port Indicators are not supported on its downstream facing ports and the PORT_INDICATOR request has no effect.<br>1: Port Indicators are supported on its downstream facing ports and the PORT_INDICATOR request controls the indicators. See Section 11.5.3.<br><br>D15...D8: Reserved |
| 5 | *bPwrOn2PwrGood* | 1 | Time (in 2 ms 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. |
| 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.<br><br>Bit value definition:<br>0B - Device is removable.<br>1B - Device is non-removable<br><br>This is a bitmap corresponding to the individual ports on the hub:<br>Bit 0: Reserved for future use.<br>Bit 1: Port 1<br>Bit 2: Port 2<br>....<br>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 5 ms for all requests.

Table 11-14 outlines the various standard device requests.

**Table 11-14. 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 11-15. Table 11-16 defines the hub class request codes. All requests in the table below except SetHubDescriptor() are mandatory.

**Table 11-15. 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 | Selector, Port | Zero | None |
| ClearTTBuffer | 00100011B | CLEAR_TT_BUFFER | Dev_Addr, EP_Num | TT_port | Zero | None |
| 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 Status |
| GetPortStatus | 10100011B | GET_ STATUS | Zero | Port | Four | Port Status and Change Status |
| ResetTT | 00100011B | RESET_TT | Zero | Port | Zero | None |
| 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 | Selector, Port | Zero | None |
| GetTTState | 10100011B | GET_TT_STATE | TT_Flags | Port | TT State Length | TT State |
| StopTT | 00100011B | STOP_TT | Zero | Port | Zero | None |

**Table 11-16. Hub Class Request Codes**

| bRequest | Value |
|---|---|
| GET_ STATUS | 0 |
| CLEAR_ FEATURE | 1 |
| RESERVED (used in previous specifications for GET_STATE) | 2 |
| SET_ FEATURE | 3 |
| *Reserved for future use* | *4-5* |
| GET_DESCRIPTOR | 6 |
| SET_DESCRIPTOR | 7 |
| CLEAR_TT_BUFFER | 8 |
| RESET_TT | 9 |
| GET_TT_STATE | 10 |
| STOP_TT | 11 |

Table 11-17 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 11-17. 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 |

**Table 11-17. Hub Class Feature Selectors (continued)**

| | Recipient | Value |
|---|---|---|
| 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 |
| PORT_INDICATOR | Port | 22 |

### 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 11-17 for the feature selector definitions that apply to the hub as a recipient. If the feature selector is associated with a status change, clearing that status change 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 11-17 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 | Selector | Port | Zero | None |

The port number must be a valid port number for that hub, greater than zero. The port field is located in bits 7..0 of the *wIndex* field.

Clearing a feature disables that feature or starts a process associated with the feature; refer to Table 11-17 for the feature selector definitions. If the feature selector is associated with a status change, clearing that status change acknowledges the change. This request format is used to clear the following features:

- PORT_ENABLE
- PORT_SUSPEND
- PORT_POWER
- PORT_INDICATOR
- 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 be placed 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.

The selector field identifies the port indicator selector when clearing a port indicator. The selector field is in bits 15..8 of the *wIndex* field.

It is a Request Error if *wValue* is not a feature selector listed in Table 11-17, if *wIndex* specifies a port that does not 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 functional 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. This request is only defined for non-periodic endpoints; e.g., if it is issued for a periodic endpoint, the response is undefined. 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_port | Zero | None |

If the hub supports a TT per port, then wIndex must specify the port 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 one.

The device_address, endpoint_number, and endpoint_type of the full-/low-speed endpoint (as specified in the corresponding split transaction) that may have a busy TT buffer must be specified in the wValue field. The specific bit fields used are shown in Table 11-18.

It is a Request Error if *wIndex* specifies a port that does not exist, or if *wLength* is not as specified above. It is not an error for this request to try to clear a buffer for a transaction that is not buffered by the TT ( should treat as a functional no-operation).

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

**Table 11-18. wValue Field for Clear_TT_Buffer**

| Bits | Field |
|---|---|
| 3..0 | Endpoint Number |
| 10..4 | Device Address |
| 12..11 | Endpoint Type |
| 14..13 | Reserved, must be zero |
| 15 | Direction, 1 = IN, 0 = OUT |

### 11.24.2.4 Get Bus State

Previous versions of USB defined a GetBusState request. This request is no longer defined.

### 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 Status |

The first word of data contains *wHubStatus* (refer to Table 11-19). The second word of data contains *wHubChange* (refer to Table 11-20).

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 11-19. Hub Status Field, *wHubStatus***

| Bit | Description |
|---|---|
| 0 | **Local Power Source:** This is the source of the local power supply.<br><br>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.<br>0 = Local power supply good<br>1 = Local power supply lost (inactive) |
| 1 | **Over-current:**<br><br>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 placed 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.<br>0 = No over-current condition currently exists.<br>1 = A hub over-current condition exists. |
| 2-15 | **Reserved**<br>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 11-20. 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*.<br><br>This field is initialized to zero when the hub receives a bus reset.<br>0 = No change has occurred to Local Power Status.<br>1 = Local Power Status has changed. |
| 1 | **Over-Current Change**: (C_HUB_OVER_CURRENT) This field indicates if a change has occurred in the Over-Current field in *wHubStatus*.<br><br>This field is initialized to zero when the hub receives a bus reset.<br>0 = No change has occurred to the Over-Current Status.<br>1 = Over-Current Status has changed. |
| 2-15 | **Reserved**<br>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 Status |

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 11-21). The second word of data contains *wPortChange* (refer to Table 11-20).

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 11-21. 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.<br>0 = No device is present.<br>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.<br>0 = Port is disabled.<br>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 downstream. This field may be reset by a request or by resume signaling from the device attached to the port.<br>0 = Not suspended.<br>1 = Suspended or resuming. |
| 3 | **Over-current:** (PORT_OVER_CURRENT)<br><br>If the hub reports over-current conditions on a per-port basis, this field will indicate that the current drain on the port exceeds the specified maximum. For more details, see Section 7.2.1.2.1.<br>0 = All no over-current condition exists on this port.<br>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.<br>0 = Reset signaling not asserted.<br>1 = Reset signaling asserted. |
| 5-7 | **Reserved**<br>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.<br>0 = This port is in the Powered-off state.<br>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.<br>0 = Full-speed or High-speed device attached to this port (determined by bit 10).<br>1 = Low-speed device attached to this port. |
| 10 | **High-speed Device Attached:** (PORT_HIGH_SPEED) This is relevant only if a device is attached.<br>0 = Full-speed device attached to this port.<br>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.<br>0 = This port is not in the Port Test Mode.<br>1 = This port is in Port Test Mode. |
| 12 | **Port Indicator Control**: (PORT_INDICATOR) This field is set to reflect software control of the port indicator. For more details see Sections 11.5.3, 11.24.2.7.1.10, and 11.24.2.13.<br>0 = Port indicator displays default colors.<br>1 = Port indicator displays software controlled color. |
| 13-15 | **Reserved**<br>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 7.1.7.3), 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, 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 7.1.20 for details on each test mode. See Section 11.24.2.13 for more information about using SetPortFeature to control 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, Disconnected, or Suspended states. If the port is not in one of these states, the hub must respond with a request error.

This field may be cleared as a result of resetting the hub.

#### 11.24.2.7.1.10 PORT_INDICATOR

When the Port Indicator port status is set to a (1B), the port indicator selector is non-zero. The specific indicator mode is specified in the SetPortFeature(PORT_INDICATOR) request by the indicator selector. The GetPortStatus(PORT_INDICATOR) provides no standard mechanism to report a specific indicator mode; therefore, system software must track which indicator mode was used. Refer to Sections 11.5.3 and 11.24.2.13 for details on each indicator mode.

This field may only be set as a result of a SetPortFeature(PORT_INDICATOR) request.

This field may be cleared as a result of a SetPortFeature(PORT_INDICATOR) request with Indicator Selector = Default or a ClearPortFeature(PORT_INDICATOR) request.

This feature must be set when host software detects an error on a port that requires user intervention. This feature must be utilized by system software if it determines that any of the following conditions are true:

- A high power device is plugged into a low power port.
- A defective device is plugged into a port (Babble conditions, excessive errors, etc.).
- An overcurrent condition occurs which causes software or hardware to set the indicator.

The PORT_OVER_CURRENT status bit will set the default port indicator color to amber. Setting the PORT_POWER feature, sets the indicator to off.

This feature is also used when host software determines that a port requires user attention. Many error conditions can be resolved if the user moves a device from one port to another that has the proper capabilities.

A typical scenario is when a user plugs a high power device in to a bus-powered hub. If there is an available high power port, then the user can be directed to move the device from the low-power port to the high power port.

1. Host software would cycle the PORT_INDICATOR feature of the low-power port to blink the indicator and display a message to the user to unplug the device from the port with the blinking indicator.
2. Using the C_PORT_CONNECTION status change feature, host software can determine when the user physically removed the device from the low-power port.
3. Host software would next issue a ClearPortFeature(PORT_INDICATOR) to the low-power port (restoring the default color), begin cycling the PORT_INDICATOR of the high-power port, and display a message telling the user to plug the device into the port with the blinking indicator.
4. Using the C_PORT_CONNECTION status change feature host software can determine when the user physically inserted the device onto the high power port.

Host software must cycle the PORT_INDICATOR feature to blink the current color at approximately 0.5 Hz rate with a 30-50% duty cycle.

#### 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 11-22 describes the various bits in the *wPortChange* field.

**Table 11-22. 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 11.24.2.7.2.1.<br>0 = No change has occurred to Current Connect status.<br>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 state. This field is set only when the entire resume process has completed. That is, the hub has ceased signaling resume on this port.<br>0 = No change.<br>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).<br>0 = No change has occurred to Over-Current Indicator.<br>1 = Over-Current Indicator has changed.<br><br>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.<br>0 = No change.<br>1 = Reset complete. |
| 5-15 | **Reserved**<br>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 7.1.7.3). 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 Get_TT_State

This request returns the internal state of the transaction translator in a vendor specific format. A TT receiving this request must have first been stopped via the Stop_TT request. This request is provided for debugging purposes.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 10100011B | GET_TT_STATE | TT_Flags | TT_Port | TT State Length | TT State |

The TT_Flags bits 7..0 are reserved for future USB definition and must be set to zero. The TT_Flags bits 15..8 are for vendor specific usage.

The TT state returned in the data stage of the control transfer for this request is shown in Table 11-23.

**Table 11-23. Format of Returned TT State**

| Offset | Field | Size (bytes) | Comments |
|---|---|---|---|
| 0 | TT_Return_Flags | 4 | Bits 15..0 are reserved for future USB definition and must be set to zero. Bits 31..16 are for vendor specific usage. |
| 4 | TT_specific_state | Implementation dependent | |

If the hub supports multiple TTs, then *wIndex* must specify the port number of the TT that will return TT_state. If the hub provides only a single TT, then Port must be set to one.

The state of the TT after processing this request is undefined.

It is a Request Error, if *wIndex* specifies a port that does not exist. If *wLength* is larger than the actual length of this request, then only the actual length is returned. If *wLength* is less than the actual length of this request, then only the first *wLength* bytes of this request are returned; this is not considered an error even if *wLength* is zero.

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

### 11.24.2.9 Reset_TT

This request returns the transaction translator in a hub to a known state.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 00100011B | RESET_TT | Zero | TT_Port | Zero | None |

Under some circumstances, a Transaction Translator (TT) in a hub may be in an unknown state such that it is no longer functioning correctly. The Reset_TT request allows the TT to be returned to the state it is in immediately after the hub is configured. Reset_TT only resets the TT internal data structures (buffers) and pipeline and its related state machines. After the reset is completed, the TT can resume its normal operation. Reset of the TT is de-coupled from the other parts of the hub (including downstream facing ports of the hub, the hub repeater, the hub controller, etc). Other parts of the hub are not reset and can continue their normal operation. The downstream facing ports are not reset, so that when the TT resumes its normal operation, the corresponding attached devices continue to work; i.e., a new enumeration process is not required. The working of downstream FS/LS devices are disrupted only during the reset time of the TT to which they belong.

If the hub supports multiple TTs, then *wIndex* must specify the port number of the TT that is to be reset. If the hub provides only a single TT, then Port must be set to one. For a single TT Hub, the Hub can ignore the Port number.

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

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

### 11.24.2.10 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.

### 11.24.2.11 Stop_TT

This request stops the normal execution of the transaction translator so that the internal TT state can be retrieved via Get_TT_State. This request is provided for debugging purposes.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 00100011B | STOP_TT | Zero | TT_Port | Zero | None |

The only standardized method to restart a TT after a Stop_TT request is via the Reset_TT request.

If the hub supports multiple TTs, then *wIndex* must specify the port number of the TT that is being stopped. If the hub provides only a single TT, then Port must be set to one. For a single TT Hub, the Hub can ignore the Port number.

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

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

### 11.24.2.12 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 11-17 for the feature selector definitions that apply to the hub as recipient. Status changes may not be acknowledged using this request.

It is a Request Error if *wValue* is not a feature selector listed in Table 11-17 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.

### 11.24.2.13 Set Port Feature

This request sets a value reported in the port status.

| bmRequestType | bRequest | wValue | wIndex | | wLength | Data |
|---|---|---|---|---|---|---|
| 00100011B | SET_ FEATURE | Feature Selector | 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 (bits 7..0) 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 11-17 for the feature selector definitions that apply to a port as a recipient. Status change may not be acknowledged using this request. Features that can be set with this request are:

- PORT_RESET
- PORT_SUSPEND
- PORT_POWER
- PORT_TEST
- PORT_INDICATOR
- 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 status change and immediately enables the port. Also see Section 11.24.2.7.1 for further details.

When the feature selector is PORT_TEST, the most significant byte (bits 15..8) of the *wIndex* field is the selector identifying the specific test mode. Table 11-24 lists the test selector definitions. Refer to Section 7.1.20 for definitions of each test mode. Test mode of a downstream facing port can only be used in a well defined sequence of hub states. This sequence is defined as follows:

1) All enabled downstream facing ports of the hub containing the port to be tested must be (selectively) suspended via the SetPortFeature(PORT_SUSPEND) request. Each downstream facing port of the hub must be in the disabled, disconnected, or suspended state (see Figure 11-9).
2) A SetPortFeature(PORT_TEST) request must be issued to the downstream facing port to be tested. Only a single downstream facing port can be in test_mode at a time. The transition to test mode must be complete no later than 3 ms after the completion of the status stage of the request.
3) The downstream facing port under test can now be tested.
4) During test_mode, a port disconnect or resume status change on one of the suspended ports (not including the port under test) must cause a status change (C_PORT_CONNECTION or C_PORT_SUSPEND) report (See Section 11.12.3 and 11.24.2.7.2) from the hub. Note: Other

status changes may or may not be supported in a hub with a downstream facing port in test mode. The reporting of these status changes can allow a test application to restore normal operation of a root hub without requiring a non-USB keyboard or mouse for user input. For example. a USB device attached to the root hub can be disconnected to notify the test application to restore normal root hub operation.

5) During test_mode, the state of the hub downstream facing ports must not be changed by the host (i.e., hub class requests other than the Get_Port_Status() request must not be issued by the host). Note: The hub must also be reset before a SetPortFeature(PORT_TEST) can be used to place the port into another test mode.

6) After the test is completed, the hub (with the port under test) must be reset by the host or user. This must be accomplished by manipulating the port of the parent hub to which the hub under test is attached. This manipulation can consist of one of the following:

   a) Issuing a SetPortFeature(PORT_RESET) to port of the parent hub to which the hub under test is attached.

   b) Issuing a ClearPortFeature(PORT_POWER) and SetPortFeature(PORT_POWER) to cycle power of a parent hub that supports per port power control.

   c) Disconnecting and re-connecting the hub under test from its parent hub port.

   d) For a root hub under test, a reset of the Host Controller may be required as there is no parent hub of the root hub.

7) Behavior of the hub under test and its downstream facing ports is undefined if these requirements are not met.

**Table 11-24. Test Mode Selector Codes**

| Value | Test Mode Description |
|---|---|
| 0H | Reserved |
| 1H | Test_J |
| 2H | Test_K |
| 3H | Test_SE0_NAK |
| 4H | Test_Packet |
| 5H | Test_Force_Enable |
| 06H-3FH | Reserved for Standard Test selections |
| 40H-BFH | Reserved |
| C0H-FFH | Reserved for Vendor-Unique test selections |

When the feature selector is PORT_INDICATOR, the most significant byte of the *wIndex* field is the selector identifying the specific indicator mode. Table 11-25 lists the indicator selector definitions. Refer to Sections 11.5.3 and 11.24.2.7.1.10 for indicator details. The hub will respond with a request error if the request contains an invalid indicator selector.

**Table 11-25. Port Indicator Selector Codes**

| Value | Port Indicator Color | Port Indicator Mode |
|---|---|---|
| 0 | Color set automatically, as defined in Table 11-6 | Automatic |
| 1 | Amber | Manual |
| 2 | Green | |
| 3 | Off | |
| 4-FFH | Reserved | Reserved |

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 11-17, if *wIndex* specifies a port that does not exist, or if *wLength* is not as specified above.

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

# Appendix A
# Transaction Examples

This appendix contains transaction examples for different split transaction cases. The cases are for bulk/control OUT and SETUP, bulk/control IN, interrupt OUT, interrupt IN, isochronous OUT, and isochronous IN.

## A.1 Bulk/Control OUT and SETUP Transaction Examples

Legend:

(S): Start Split

(C): Complete Split

Summary of cases for bulk/control OUT and SETUP transaction

- Normal cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash | Figure A-1 | |
| HS SSPLIT smash | | Figure A-2 |
| HS SSPLIT 3 strikes smash | | Figure A-3 |
| HS OUT/SETUP(S) smash | | Figure A-2 |
| HS OUT/SETUP(S) 3 strikes smash | | Figure A-3 |
| HS DATA0/1 smash | Figure A-2 | |
| HS DATA0/1 3 strikes smash | Figure A-3 | |
| HS ACK(S) smash | Figure A-4<br>Figure A-5 | |
| HS ACK(S) 3 strikes smash | Figure A-6 | |
| HS CSPLIT smash | Figure A-7 | |
| HS CSPLIT 3 strikes smash | Figure A-8 | |
| HS OUT/SETUP(C) smash | | Figure A-7 |
| HS OUT/SETUP(C) 3 strikes smash | | Figure A-8 |

| | | |
|---|---|---|
| HS ACK(C) smash | Figure A-9 | |
| HS ACK(C) 3 strikes smash | Figure A-10 | |
| FS/LS OUT/SETUP smash | | Figure A-11 |
| FS/LS OUT/SETUP 3 strikes smash | | Figure A-12 |
| FS/LS DATA0/1 smash | Figure A-11 | |
| FS/LS DATA0/1 3 strikes smash | Figure A-12 | |
| FS/LS ACK smash | Figure A-13 | |
| FS/LS ACK 3 strikes smash | Figure A-14 | |

- No buffer(on hub) avallable cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash(HS NAK(S)) | Figure A-15 | |
| HS NAK(S) smash | Figure A-16 | |
| HS NAK(S) 3 strikes smash | Figure A-17 | |

- CS(Complete-split transaction) earlier cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash(HS NYET) | Figure A-18 | |
| HS NYET smash | Figure A-19<br>Figure A-20 | |
| HS NYET 3 strikes smash | Figure A-21 | |

- Device busy cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash(HS NAK(C)) | Figure A-22 | |
| HS NAK(C) smash | | Figure A-9 |

| | | |
|---|---|---|
| HS NAK(C) 3 strikes smash | | Figure A-10 |
| FS/LS NAK smash | | Figure A-13 |
| FS/LS NAK 3 strikes smash | | Figure A-14 |

- Device stall cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash | Figure A-23 | |
| HS STALL(C) smash | | Figure A-9 |
| HS STALL(C) 3 strikes smash | | Figure A-10 |
| FS/LS STALL smash | | Figure A-13 |
| FS/LS STALL 3 strikes smash | | Figure A-14 |

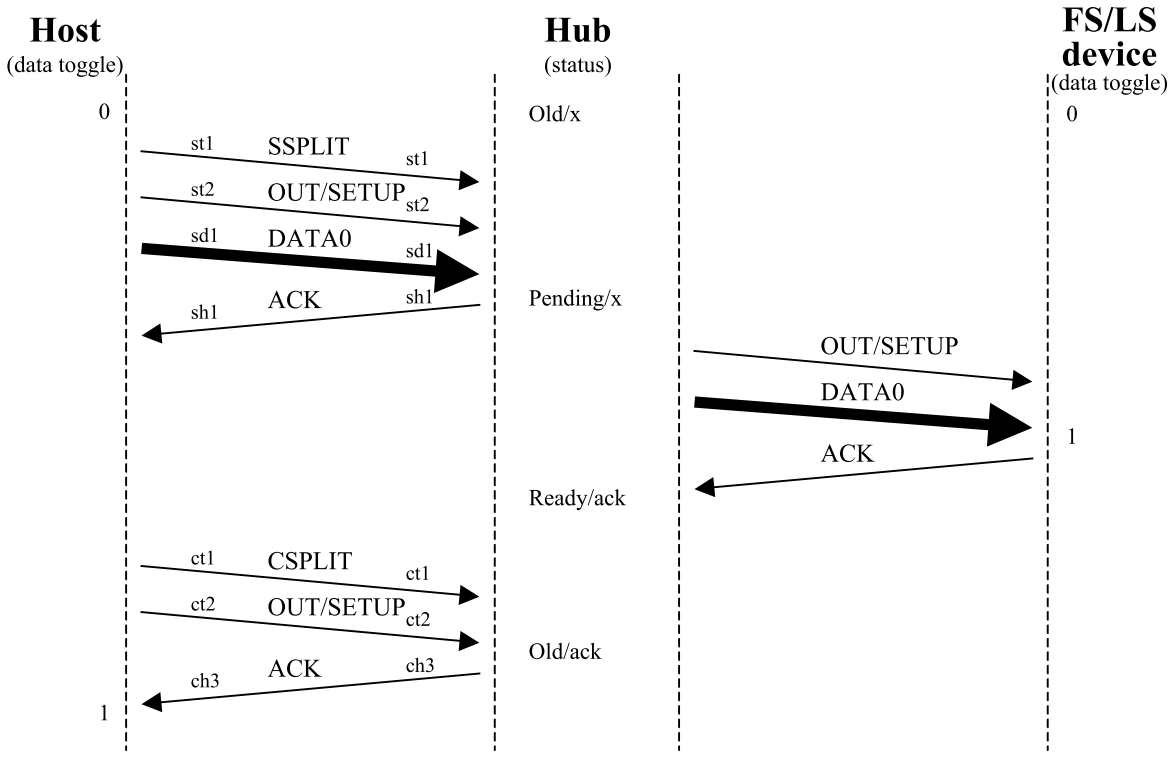



**Figure A-1. Normal No Smash**

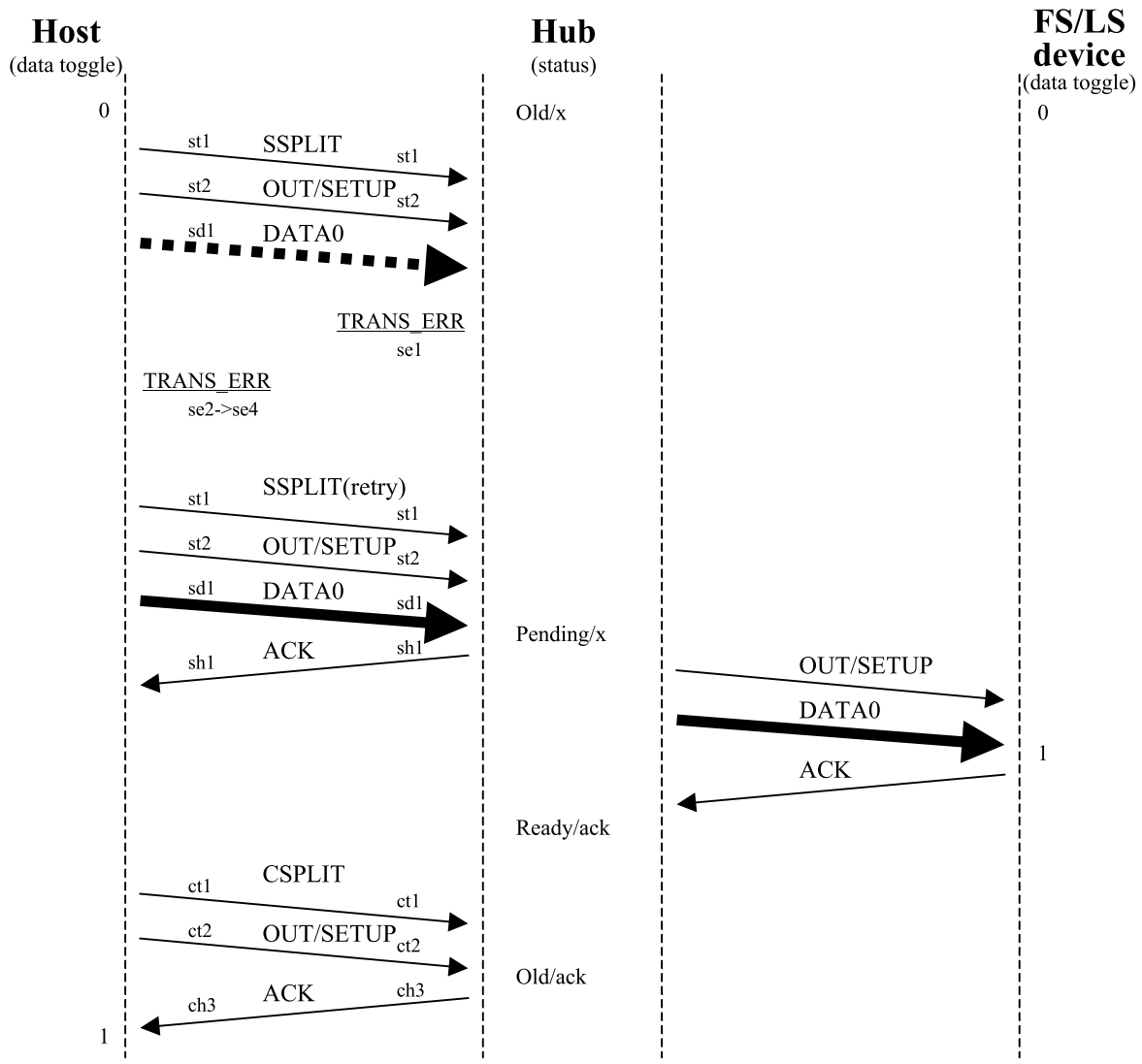



**Figure A-2. Normal HS DATA0/1 Smash**

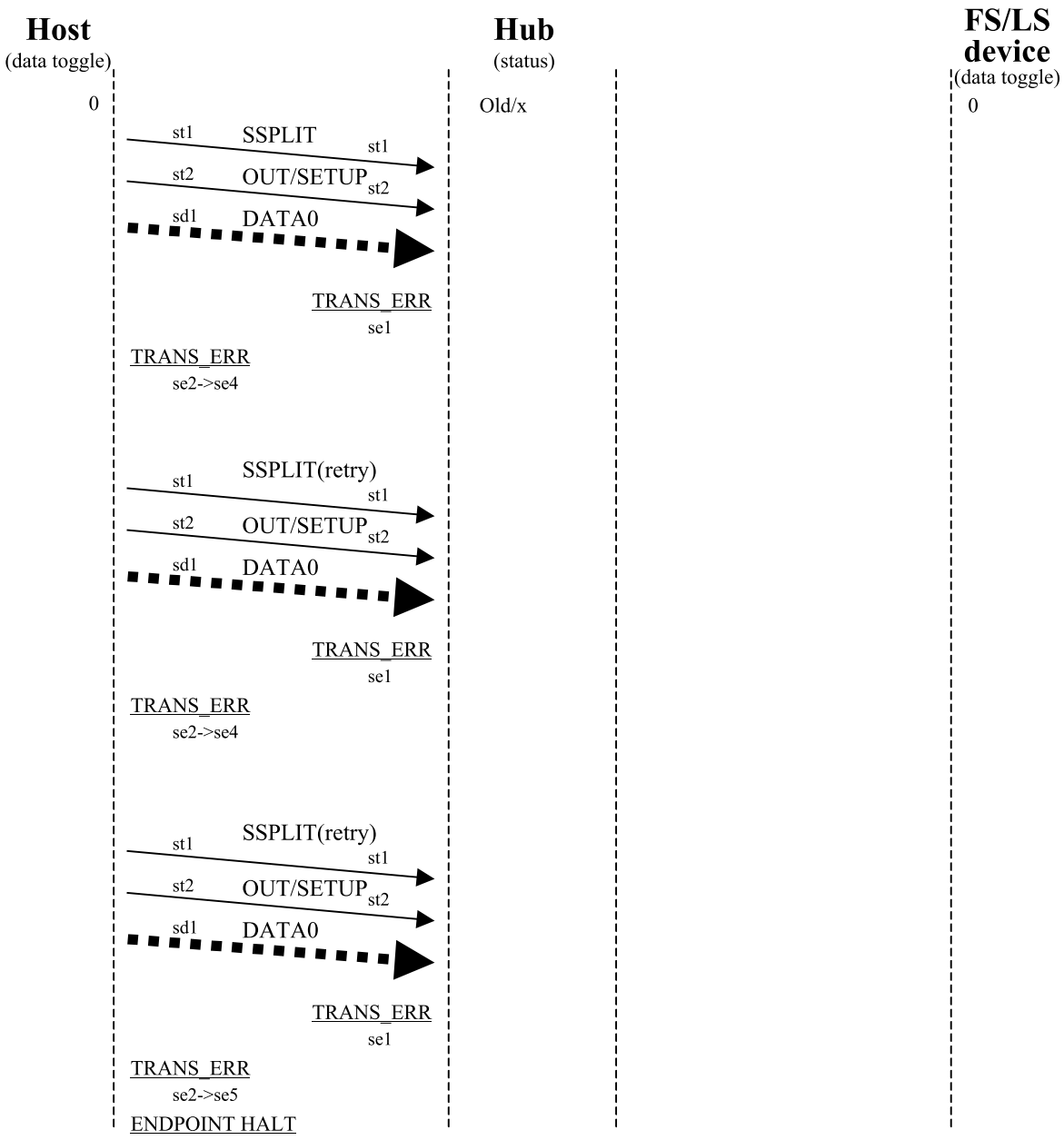



**Figure A-3. Normal HS DATA0/1 3 Strikes Smash**

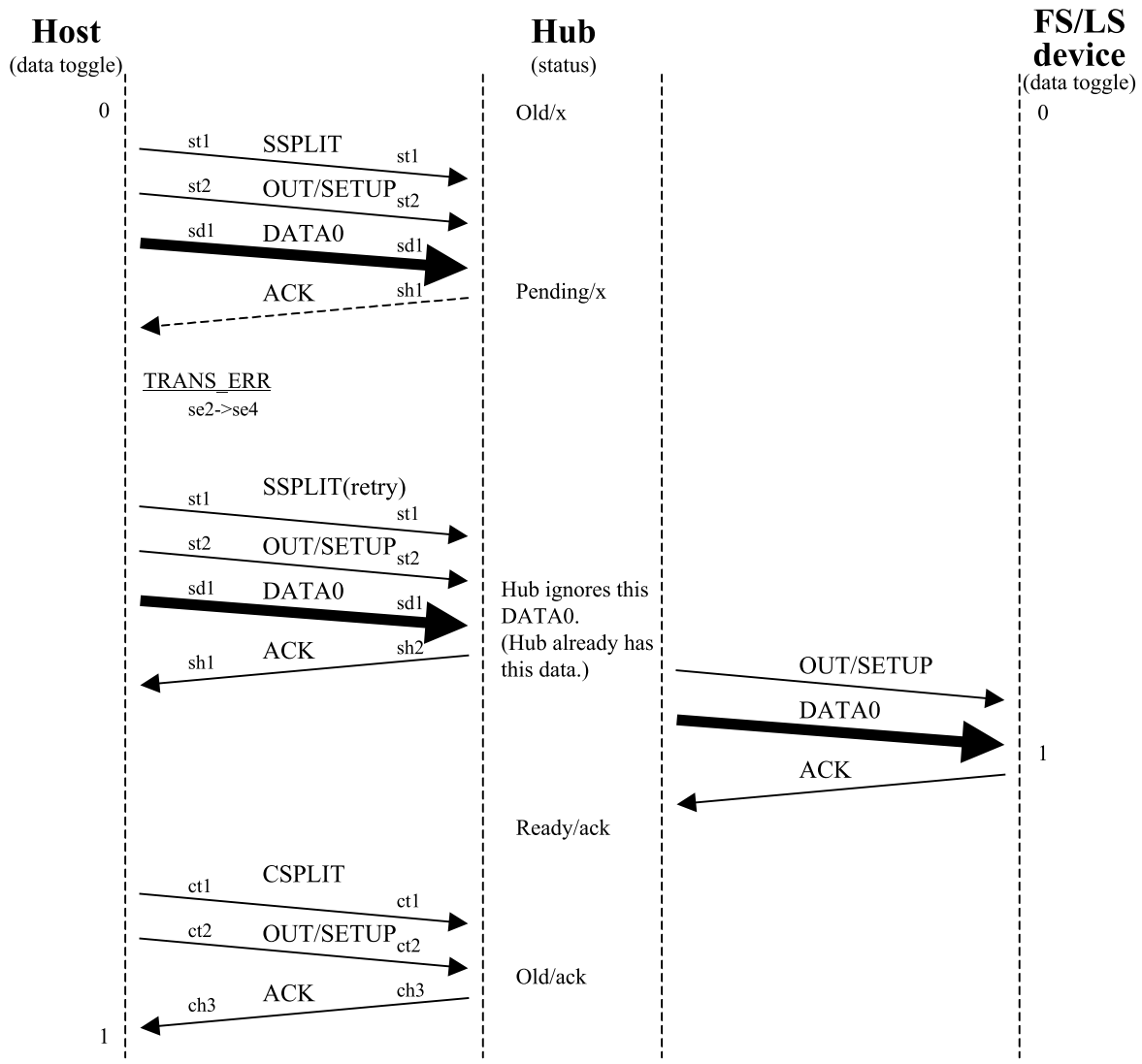



**Figure A-4. Normal HS ACK(S) Smash(case 1)**

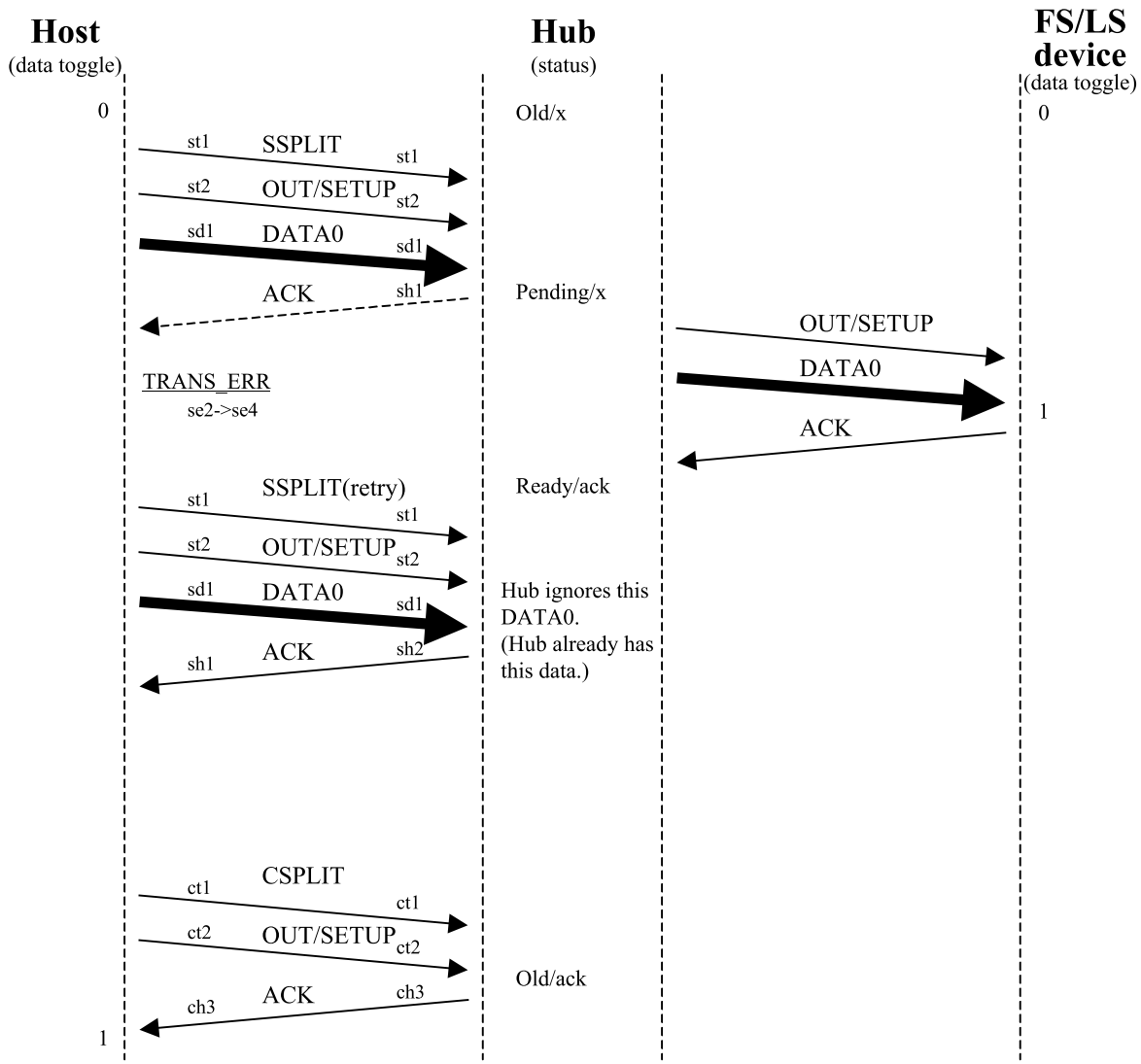



**Figure A-5. Normal HS ACK(S) Smash(case 2)**

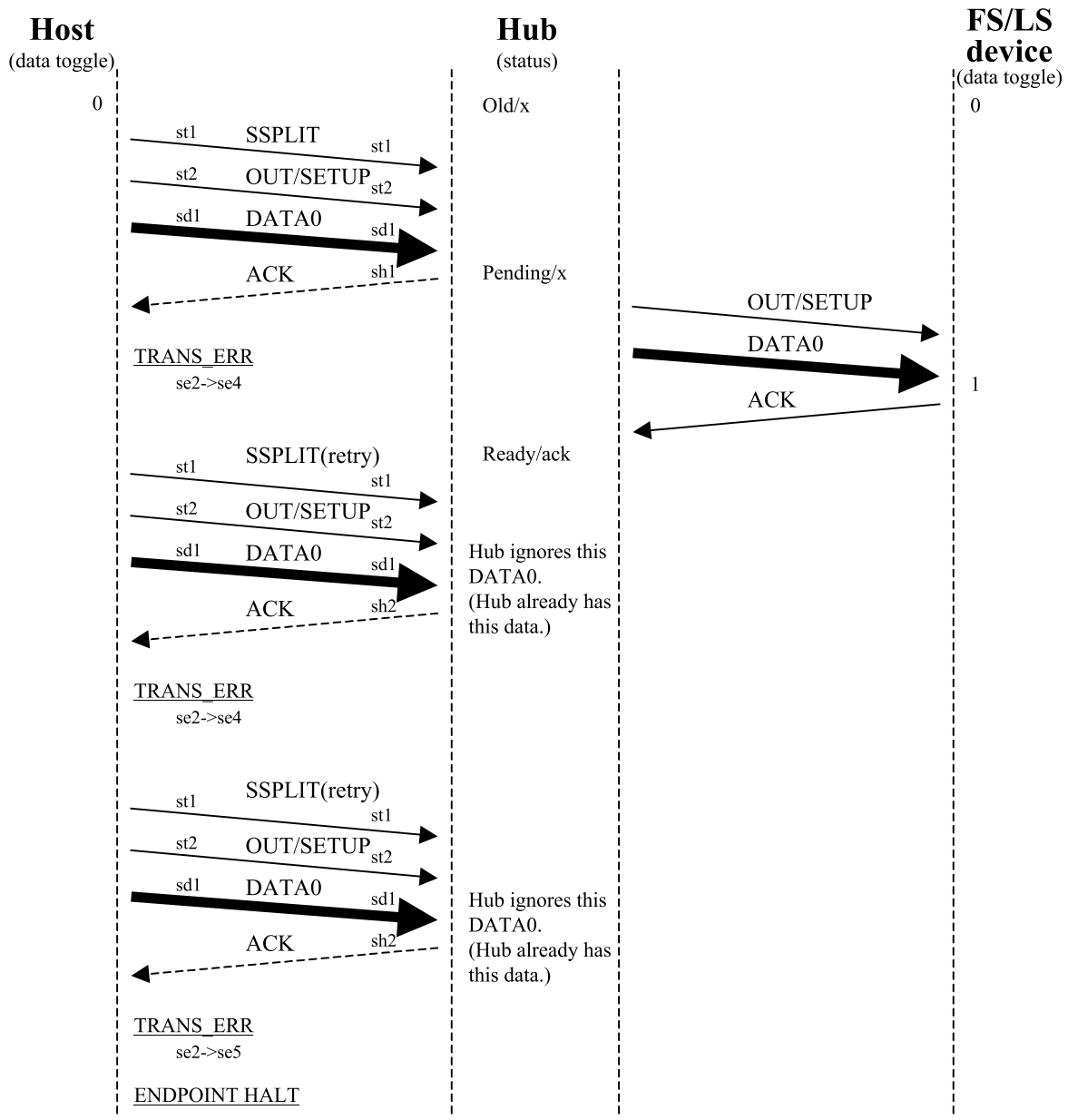



**Figure A-6. Normal HS ACK(S) 3 Strikes Smash**

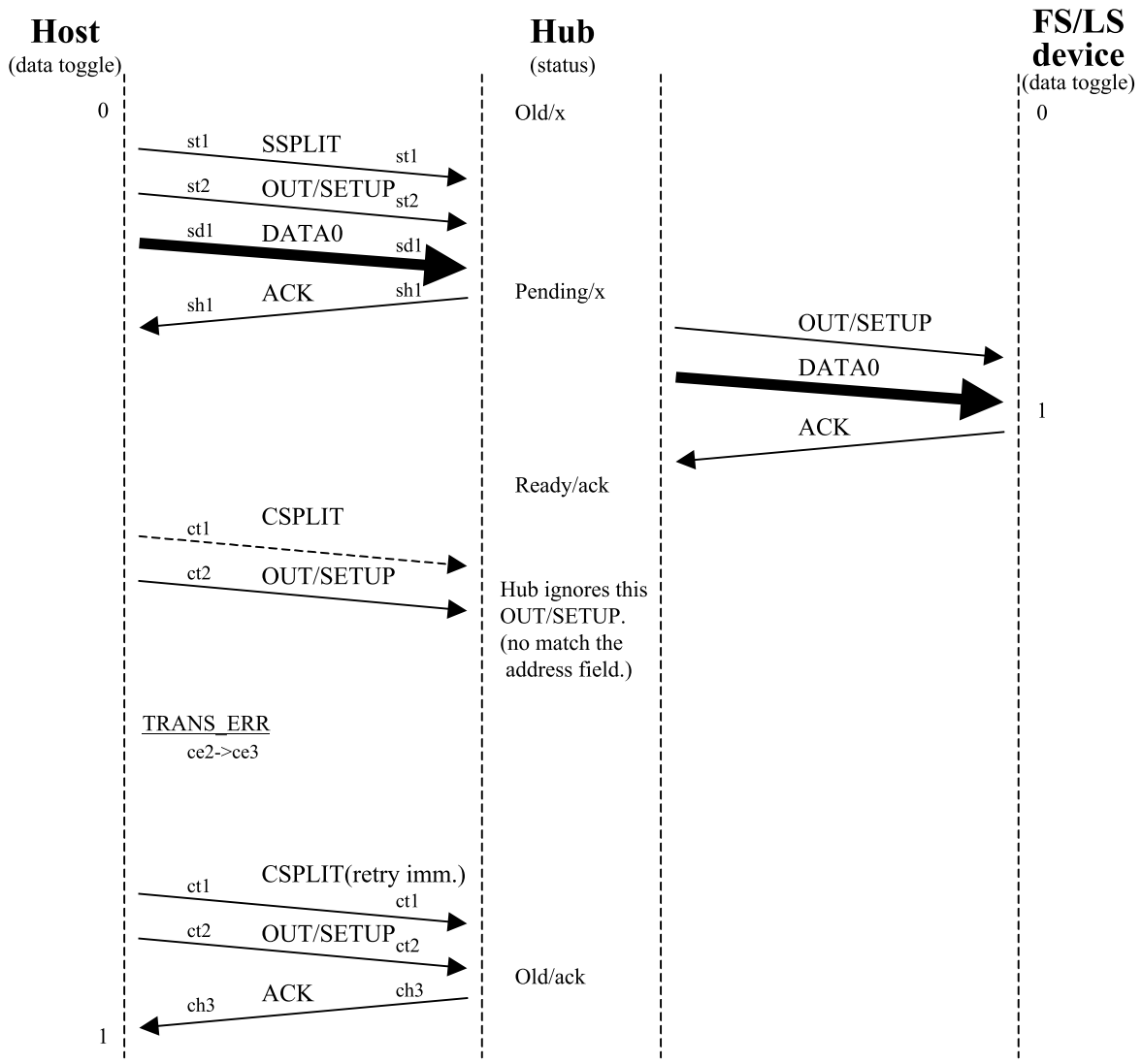



**Figure A-7. Normal HS CSPLIT Smash**

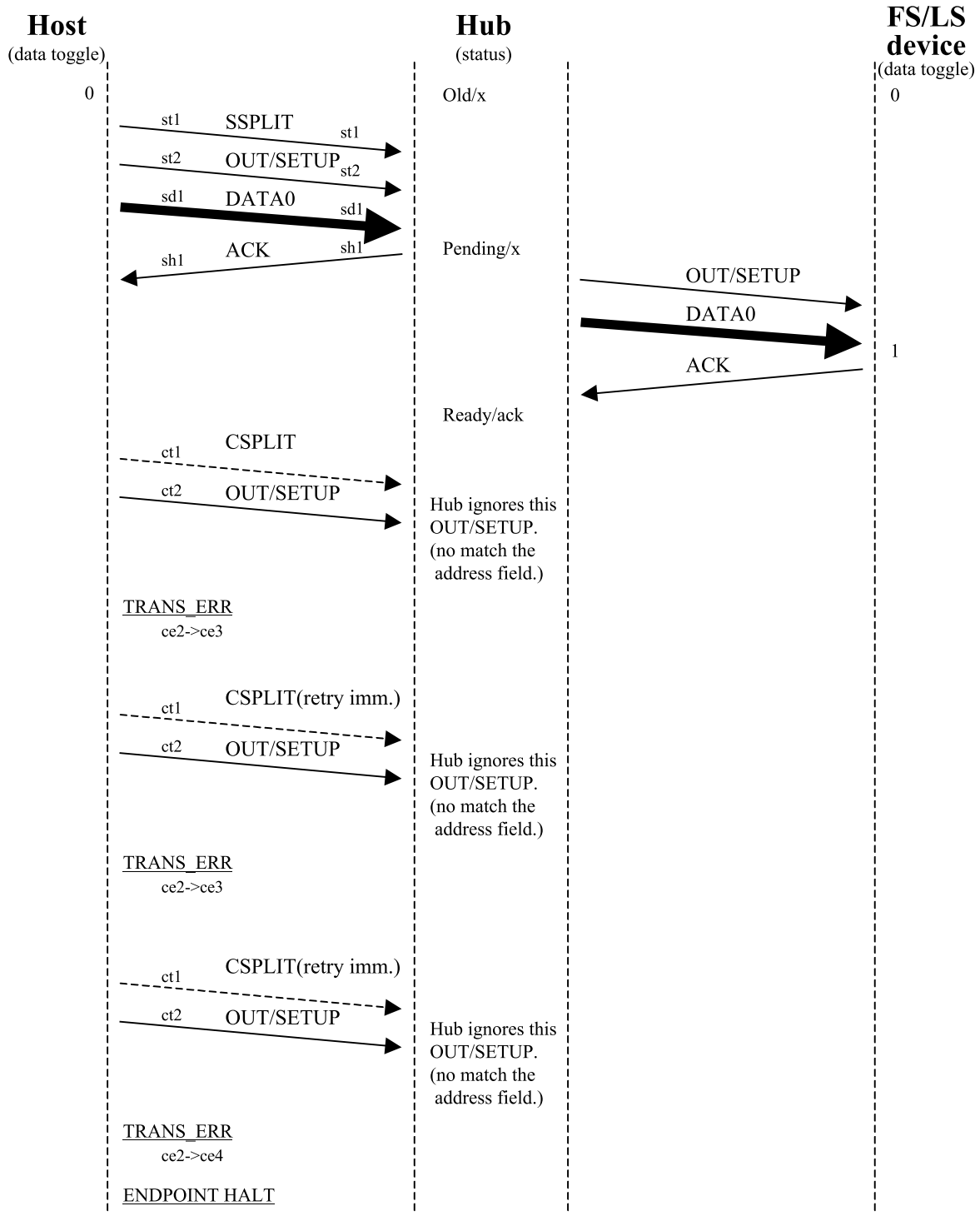



**Figure A-8. Normal HS CSPLIT 3 Strikes Smash**

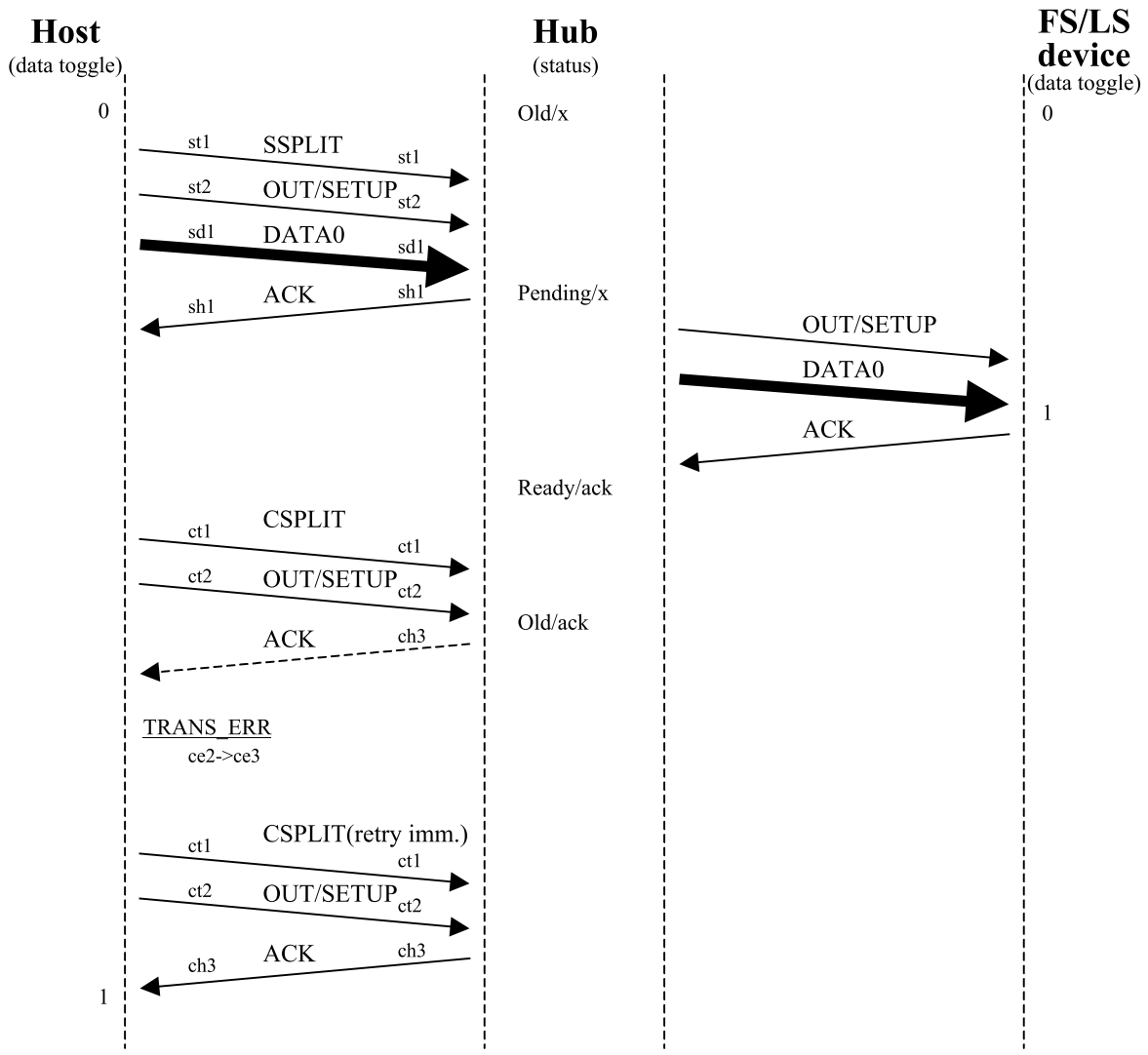



**Figure A-9. Normal HS ACK(C) Smash**

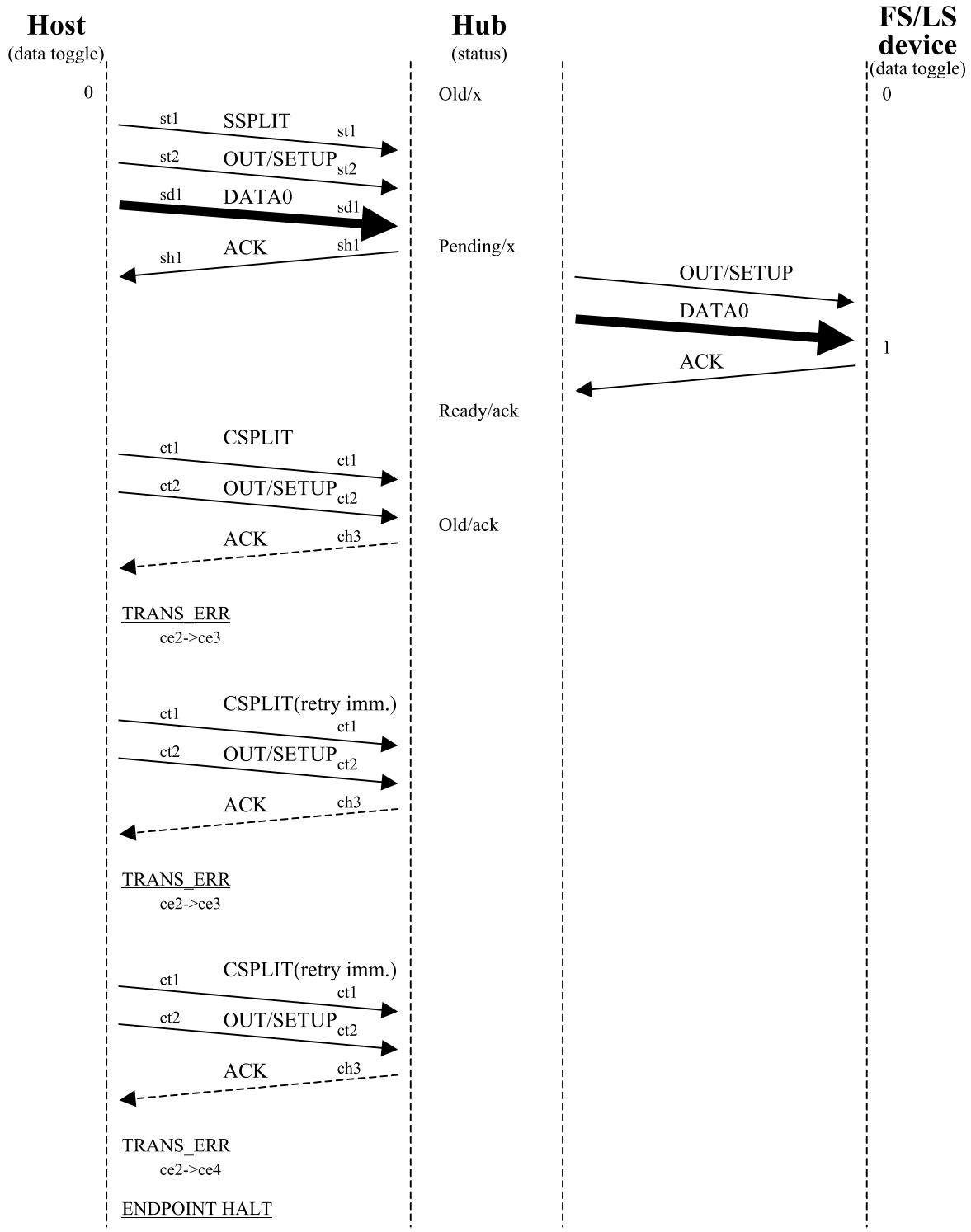



**Figure A-10. Normal S ACK(C) 3 Strikes Smash**

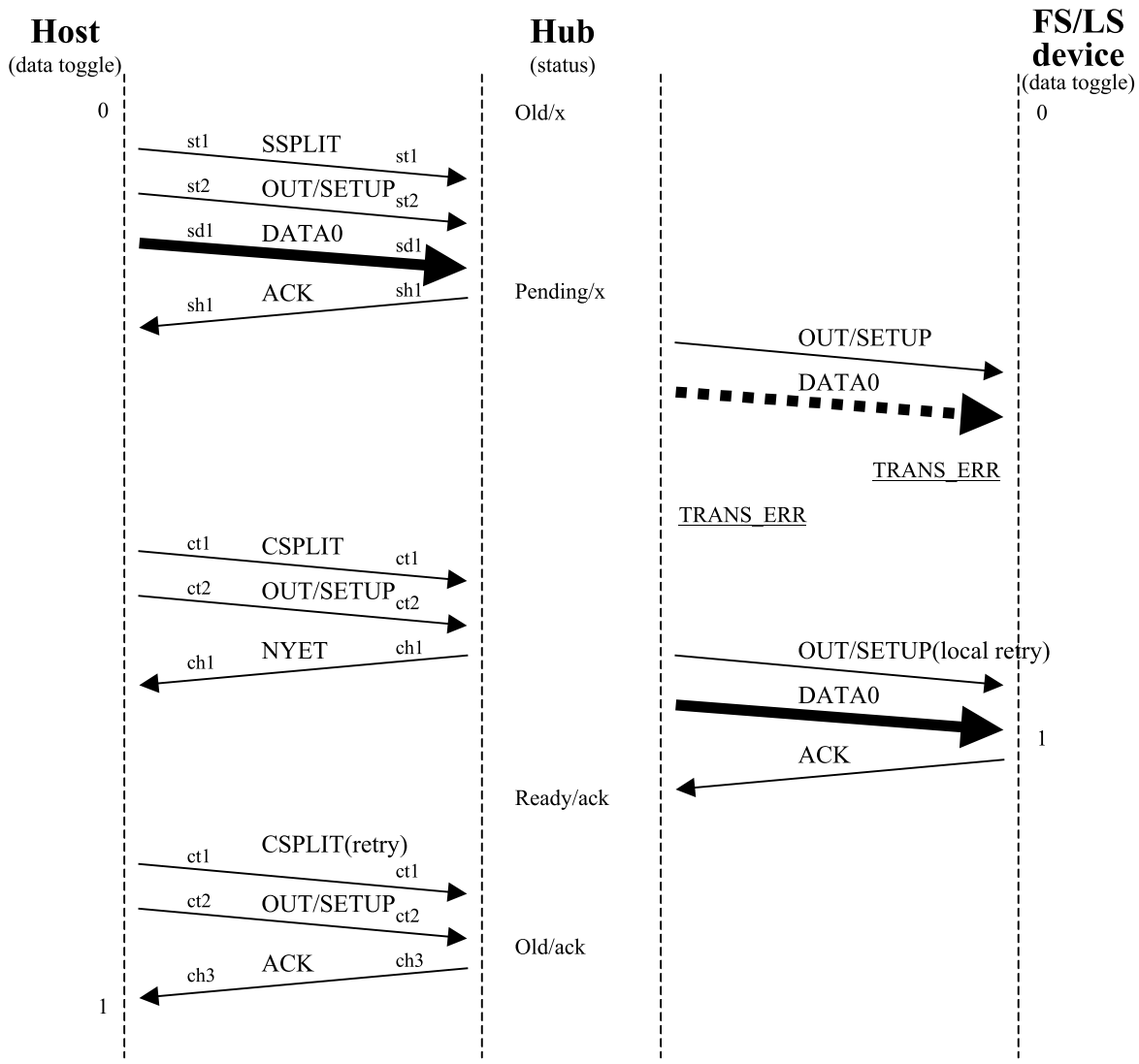



**Figure A-11. Normal FS/LS DATA0/1 Smash**

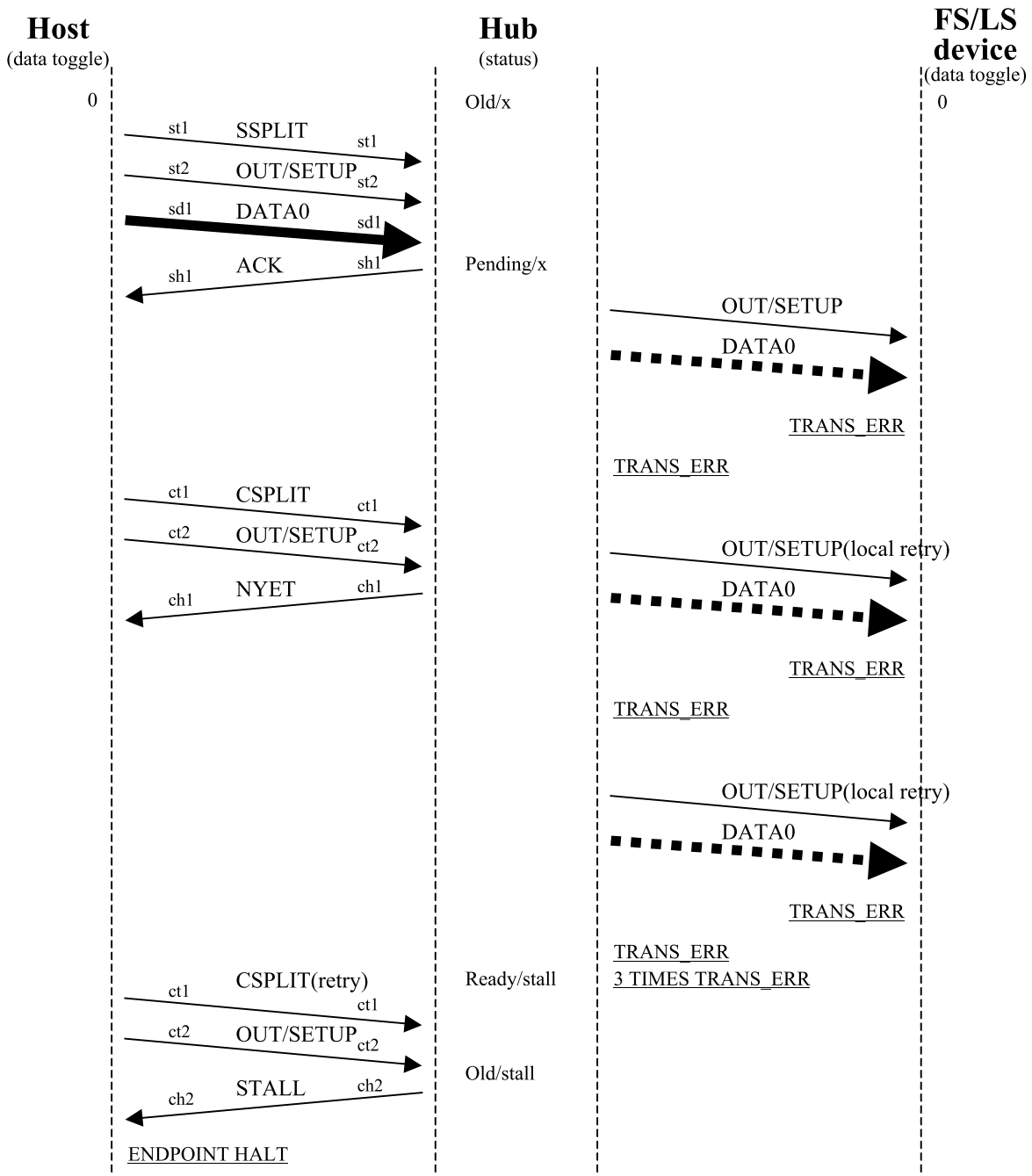



**Figure A-12. Normal FS/LS DATA0/1 3 Strikes Smash**

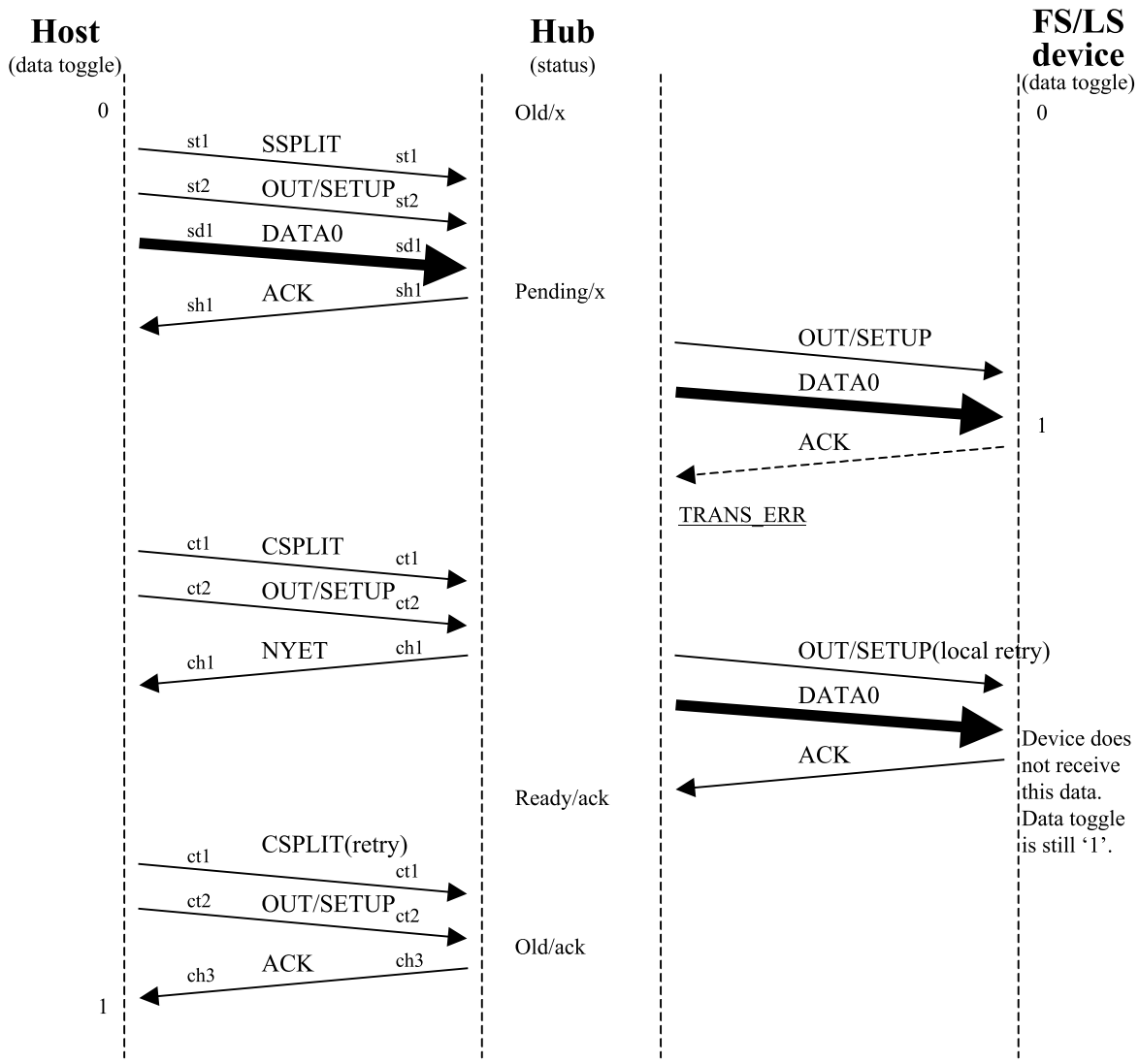



**Figure A-13. Normal FS/LS ACK Smash**

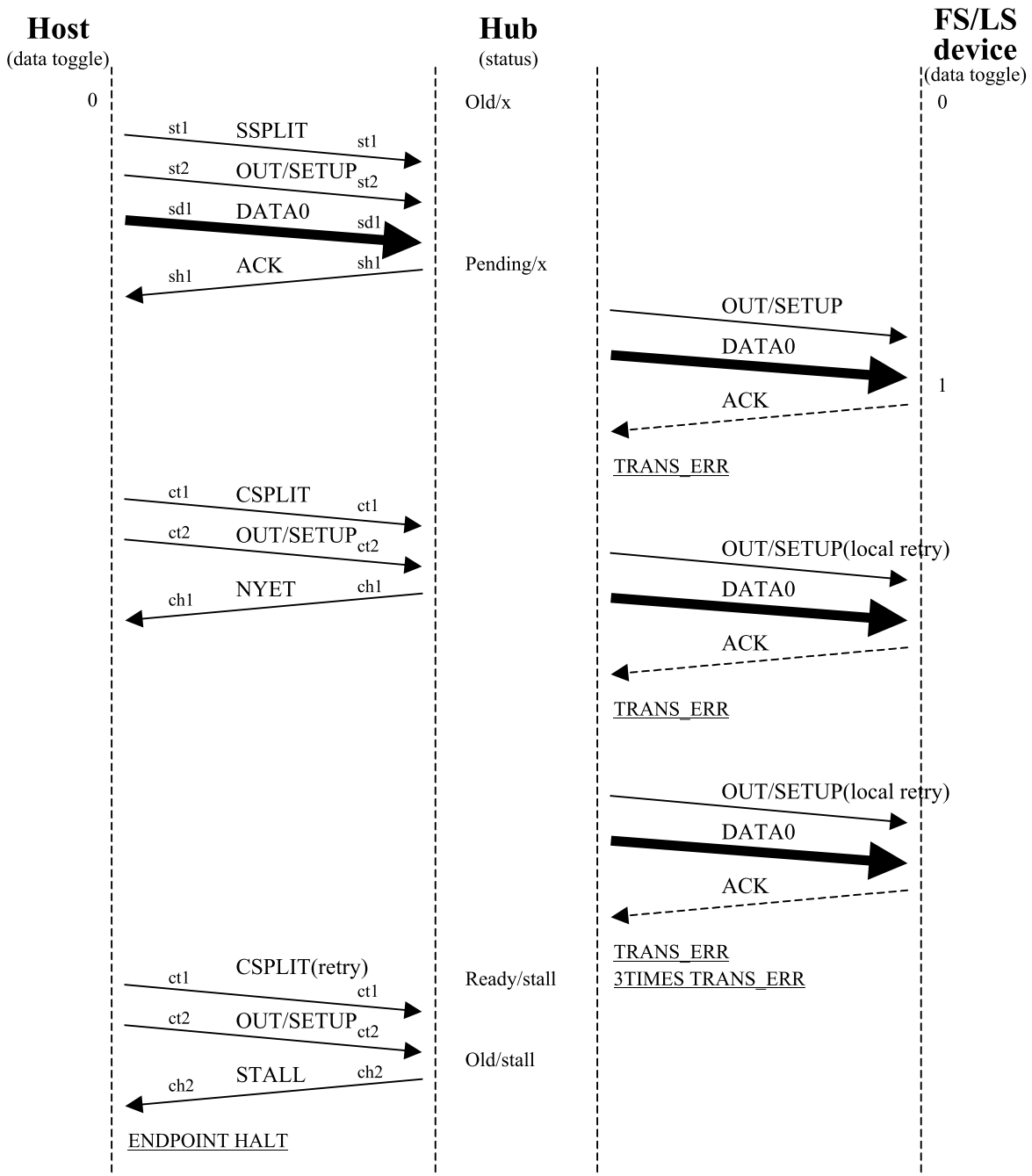



**Figure A-14. Normal FS/LS ACK 3 Strikes Smash**

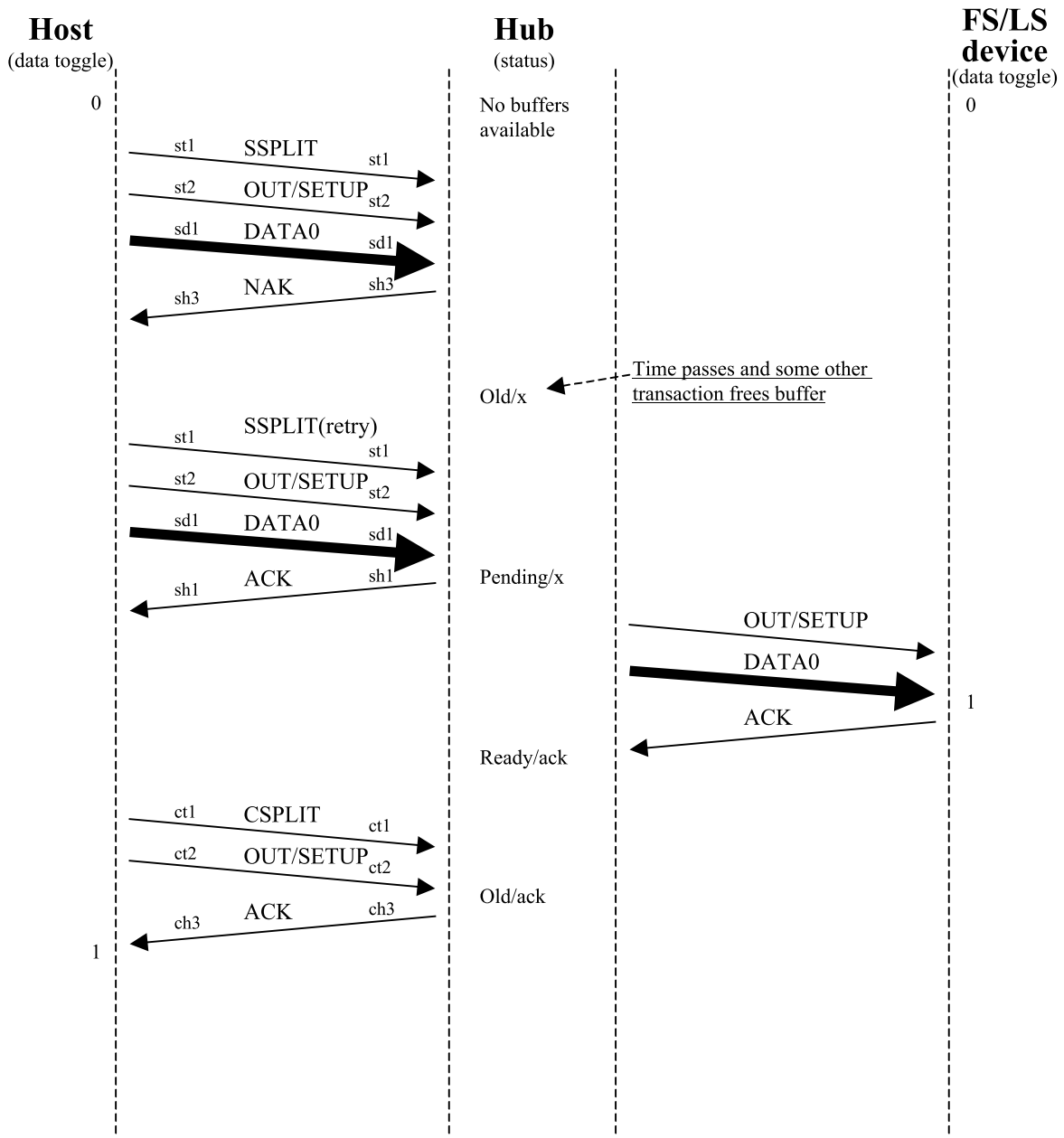



**Figure A-15. No buffer Available No Smash (HS NAK(S))**

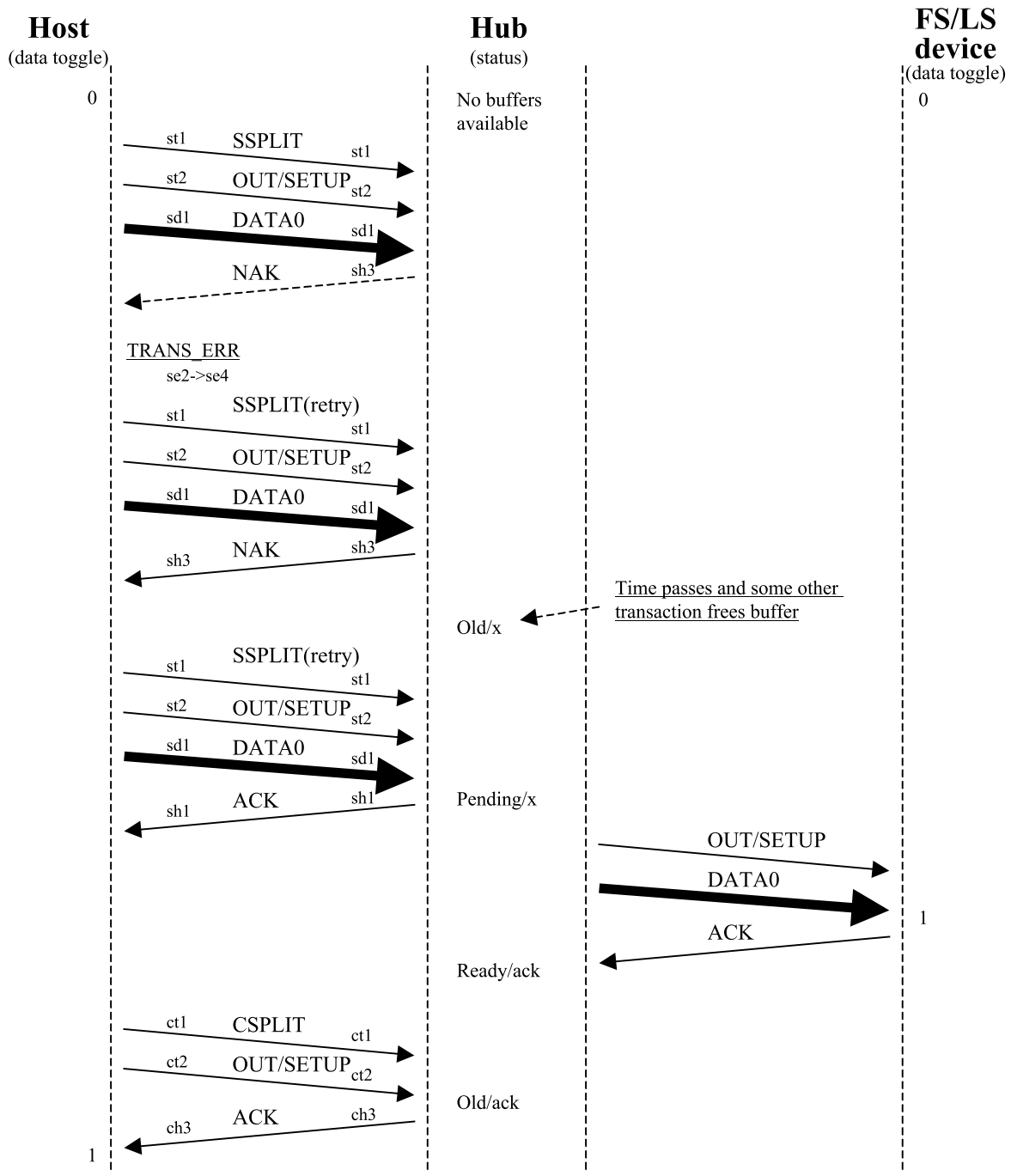



**Figure A-16. No Buffer Available HS NAK(S) Smash**

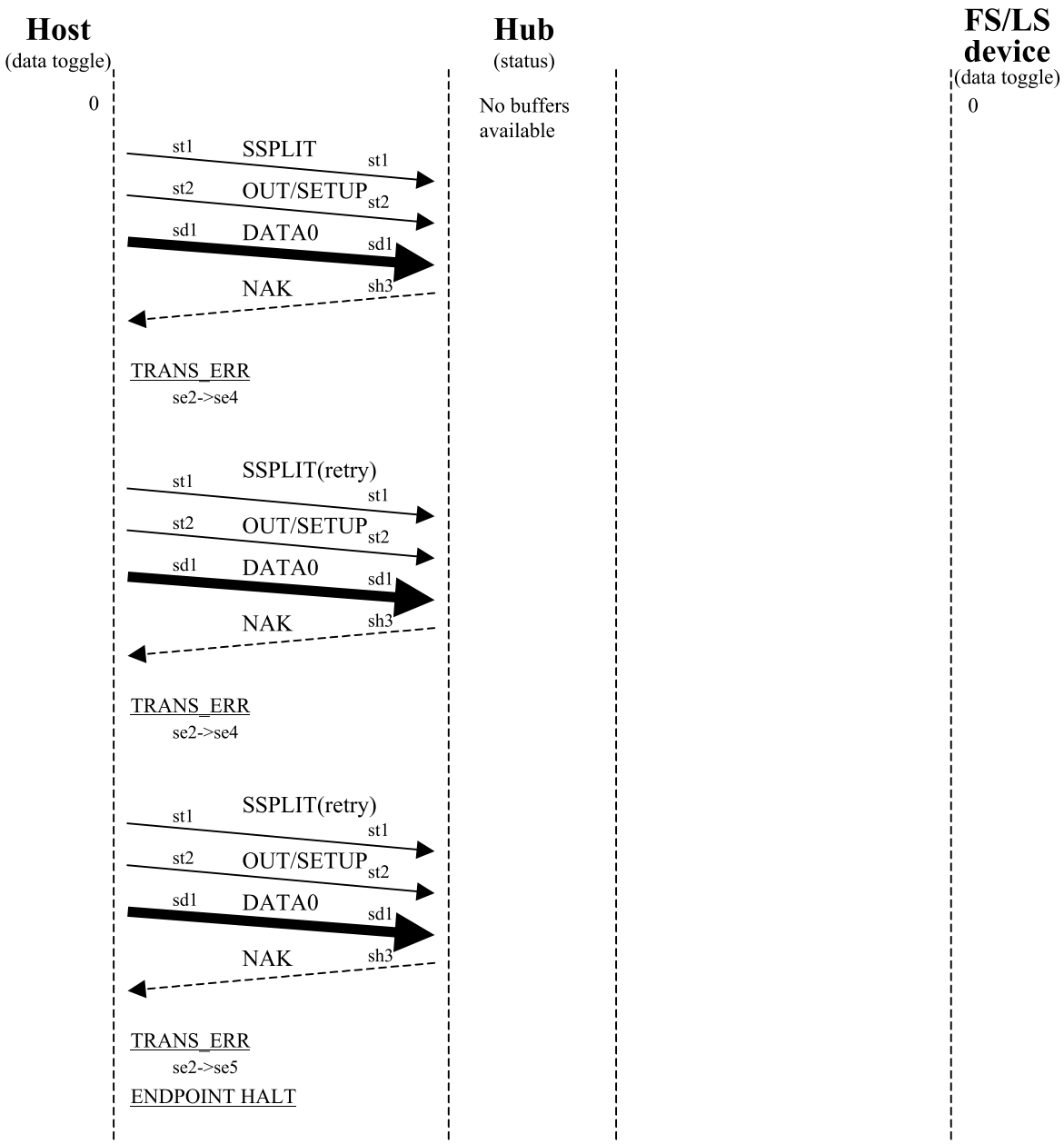



**Figure A-17. No Buffer Available HS NAK(S) 3 Strikes Smash**

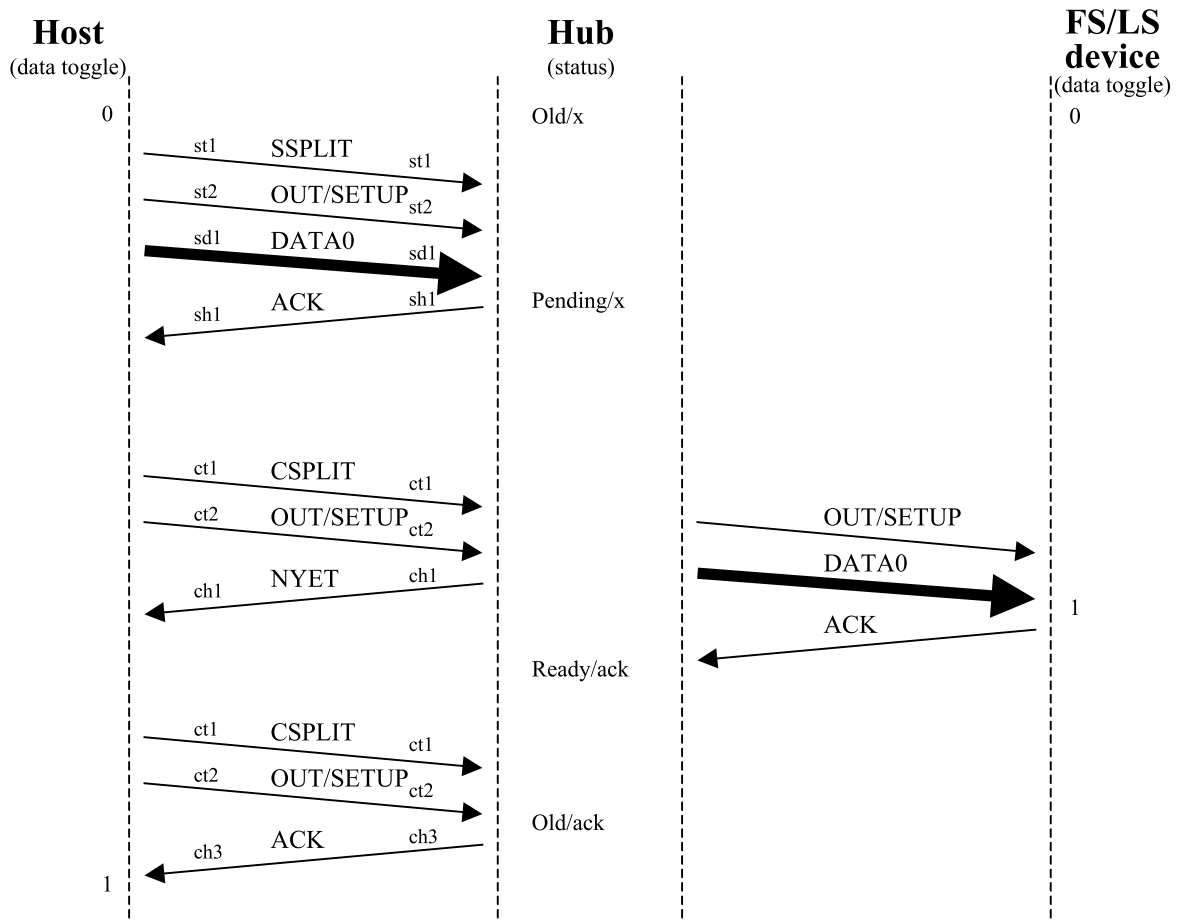



**Figure A-18. CS Earlier No Smash (HS NYET)**

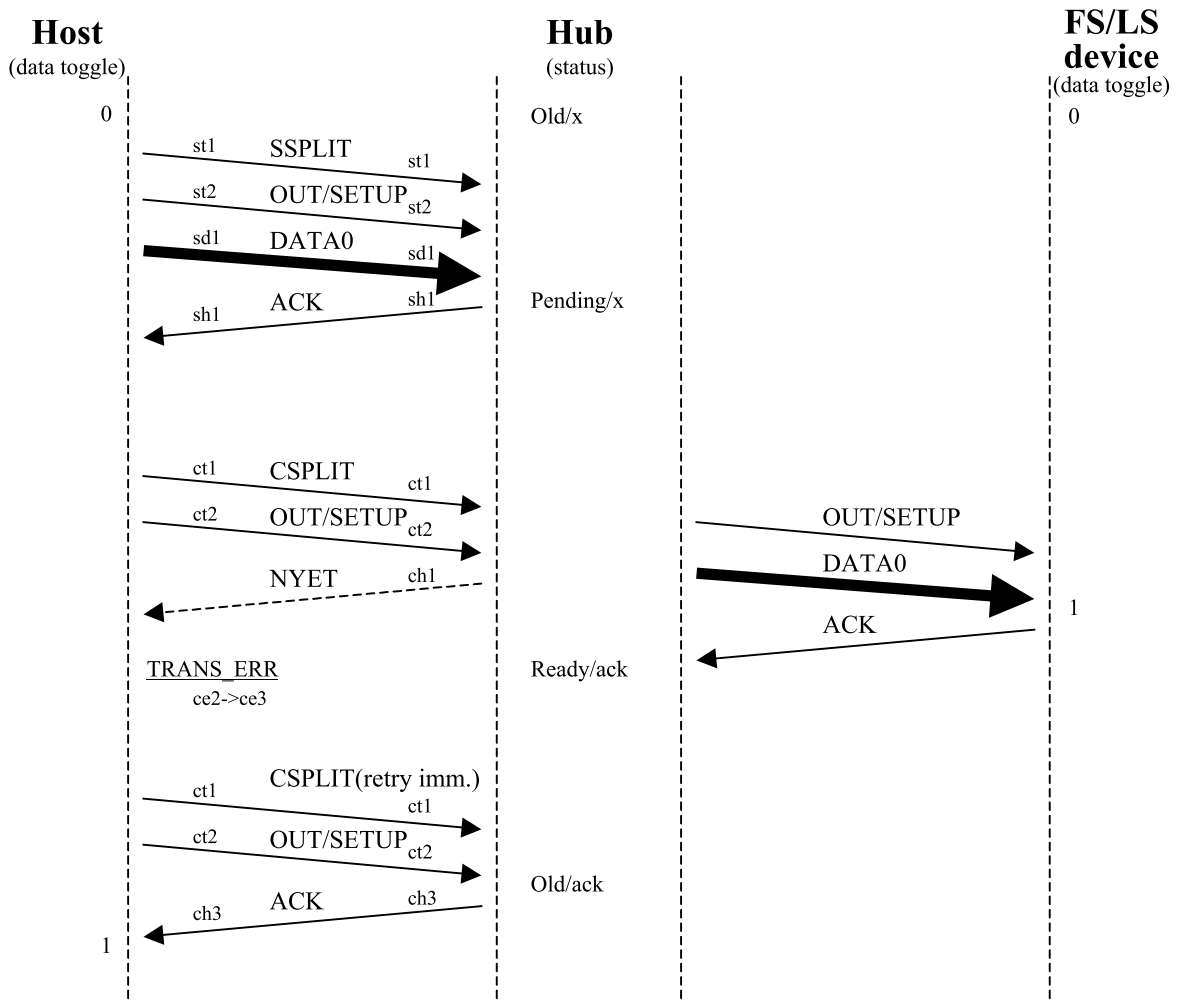



**Figure A-19. CS Earlier HS NYET Smash(case 1)**

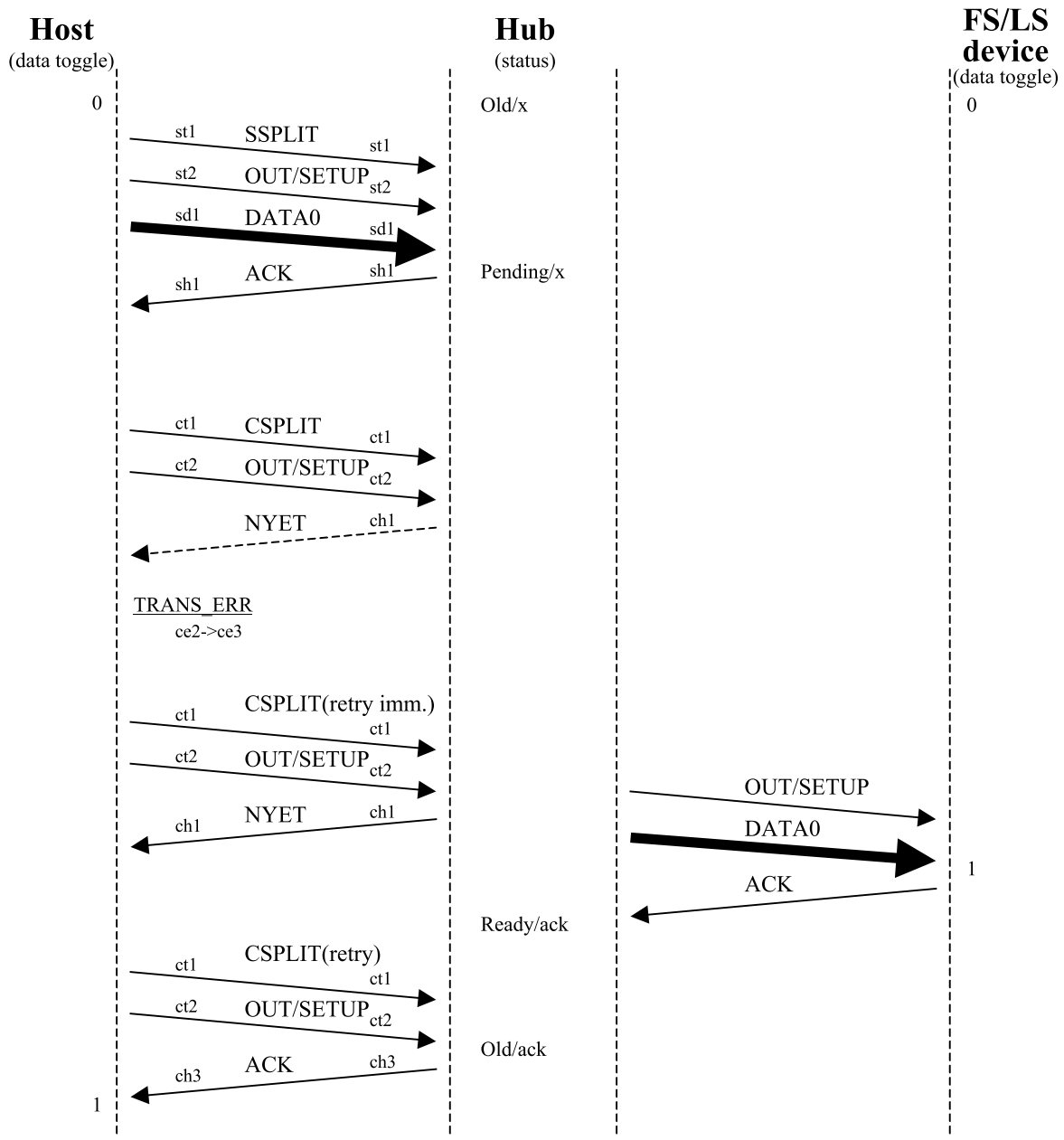



**Figure A-20. CS Earlier HS NYET Smash(case 2)**

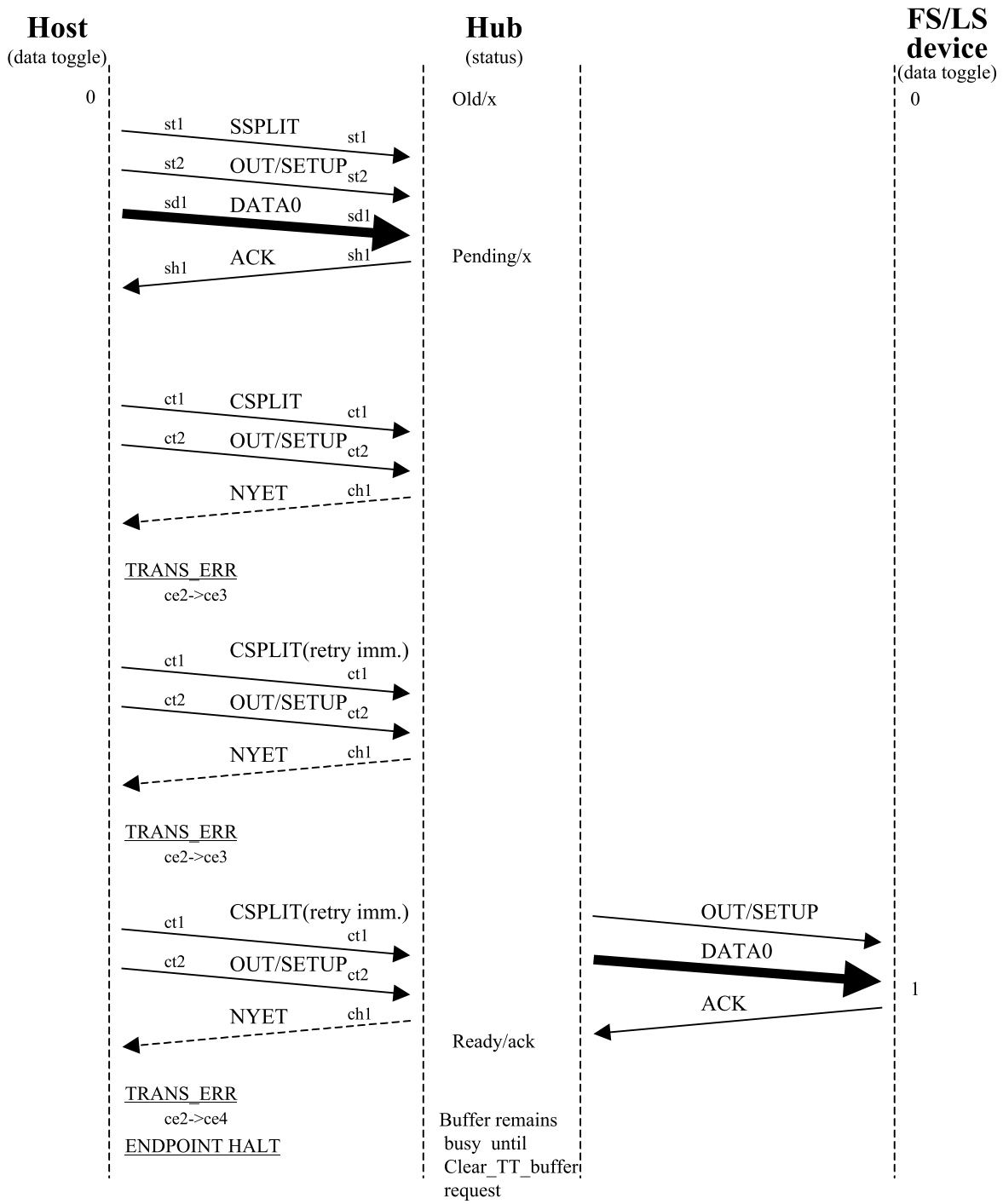



**Figure A-21. CS Earlier HS NYET 3 Strikes Smash**

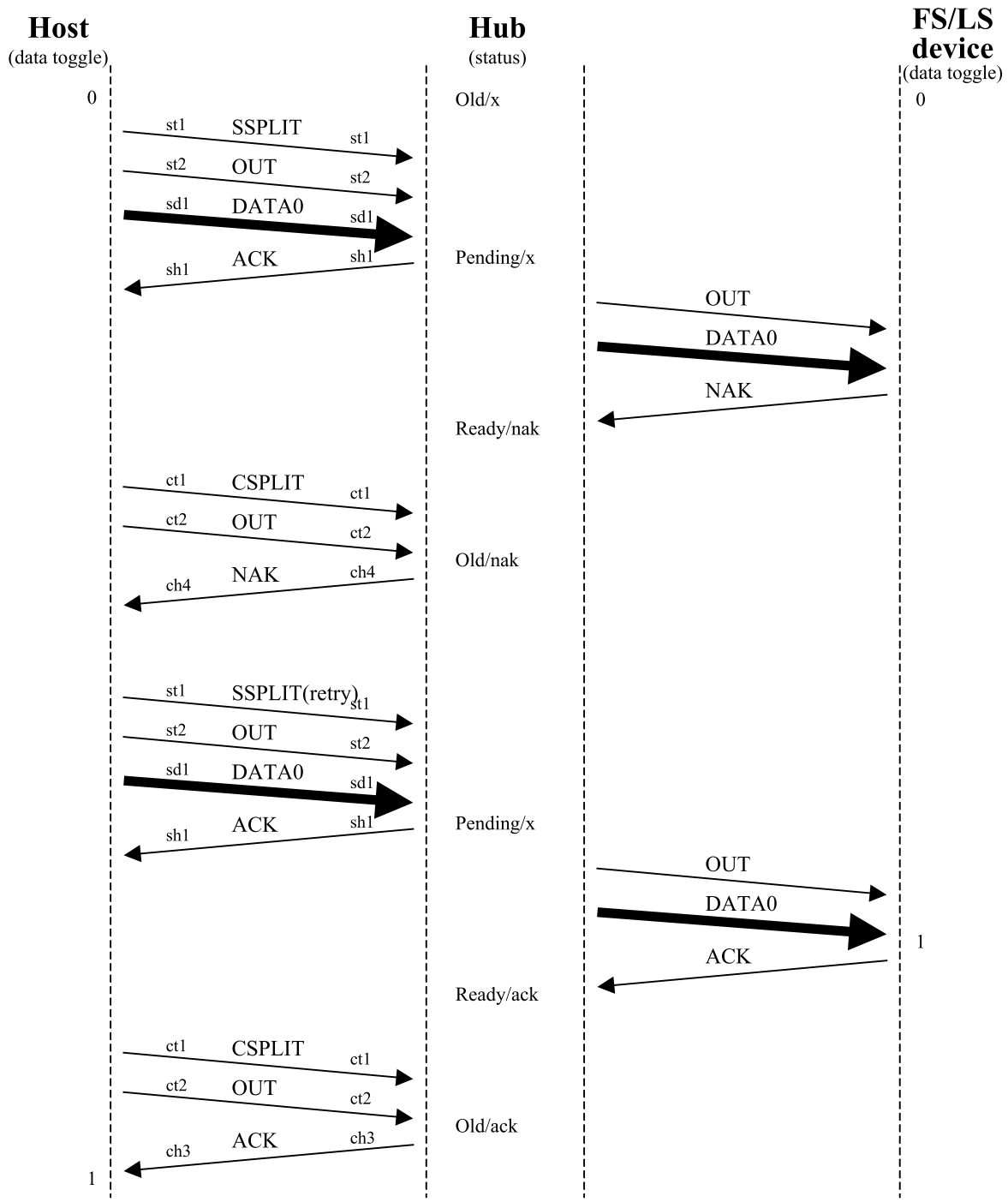



**Figure A-22. Device Busy No Smash(FS/LS NAK)**

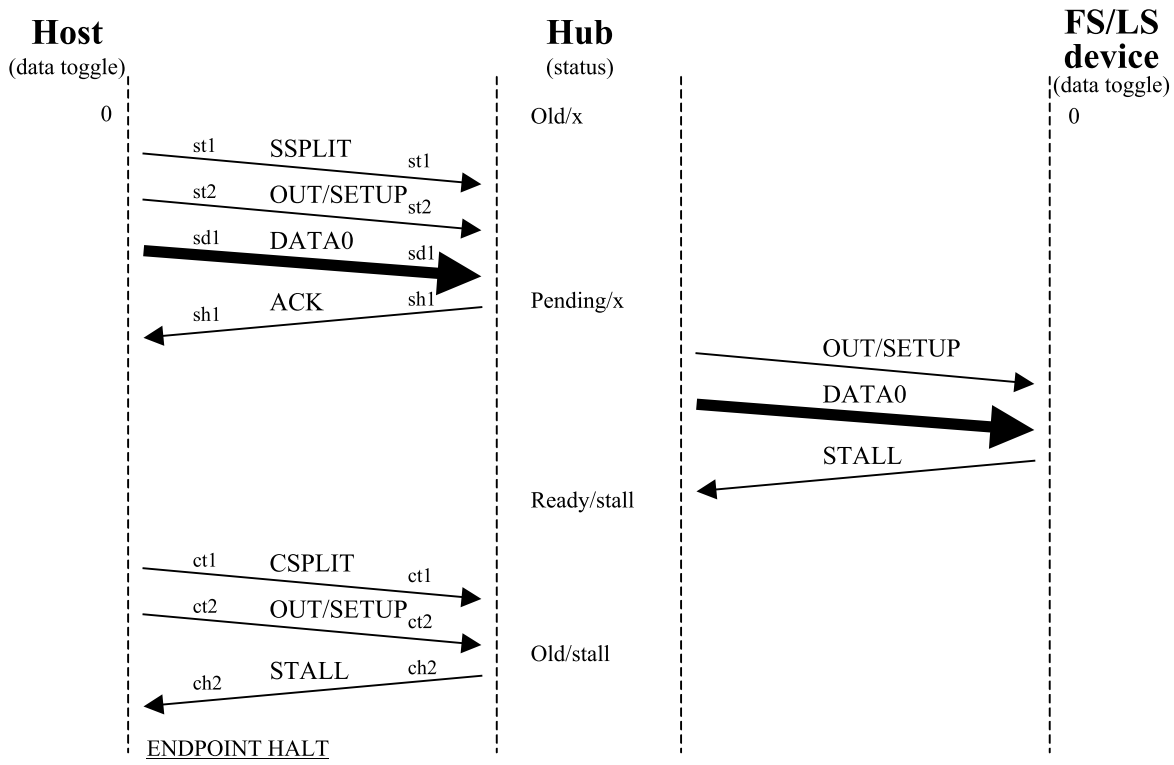



**Figure A-23. Device Stall No Smash(FS/LS STALL)**

## A.2 Bulk/Control IN Transaction Examples

Legend:

(S): Start Split

(C): Complete Split

Summary of cases for bulk/control IN transaction

- Normal cases

| Case | Reference figure | Similar figure |
|---|---|---|
| No smash | Figure A-24 | |
| HS SSPLIT smash | Figure A-25 | |
| HS SSPLIT 3 strikes smash | Figure A-26 | |
| HS IN(S) smash | | Figure A-25 |
| HS IN(S) 3 strikes smash | | Figure A-26 |
| HS ACK(S) smash | Figure A-27<br>Figure A-28 | |
| HS ACK(S) 3 strikes smash | Figure A-29 | |
| HS CSPLIT smash | Figure A-30 | |
| HS CSPLIT 3 strikes smash | Figure A-31 | |
| HS IN(C) smash | | Figure A-30 |
| HS IN(C) 3 strikes smash | | Figure A-31 |
| HS DATA0/1 smash | Figure A-32 | |
| HS DATA0/1 3 strikes smash | Figure A-33 | |
| FS/LS IN smash | Figure A-34 | |
| FS/LS IN 3 strikes smash | Figure A-35 | |
| FS/LS DATA0/1 smash | Figure A-36 | |
| FS/LS DATA0/1 3 strikes smash | Figure A-37 | |

| | | |
|---|---|---|
| FS/LS ACK smash | Figure A-38 | |
| FS/LS ACK 3 strikes smash | No figure | |

- No buffer(on hub) avallable cases

| Case | Reference figure | Similar figure |
|---|---|---|
| No smash(HS NAK(S)) | Figure A-39 | |
| HS NAK(S) smash | Figure A-40 | |
| HS NAK(S) 3 strikes smash | Figure A-41 | |

- CS(Complete-split transaction) earlier cases

| Case | Reference figure | Similar figure |
|---|---|---|
| No smash(HS NYET) | Figure A-42 | |
| HS NYET smash | Figure A-43<br>Figure A-44 | |
| HS NYET 3 strikes smash | No figure | |

- Device busy cases

| Case | Reference figure | Similar figure |
|---|---|---|
| No smash(HS NAK(C)) | Figure A-45 | |
| HS NAK(C) smash | | Figure A-32 |
| HS NAK(C) 3 strikes smash | | Figure A-33 |
| FS/LS NAK smash | | Figure A-36 |
| FS/LS NAK 3 strikes smash | | Figure A-37 |

- Device stall cases

| Case | Reference figure | Similar figure |
|---|---|---|
| No smash | Figure A-46 | |
| HS STALL(C) smash | | Figure A-32 |
| HS STALL(C) 3 strikes smash | | Figure A-33 |
| FS/LS STALL smash | | Figure A-36 |
| FS/LS STALL 3 strikes smash | | Figure A-37 |

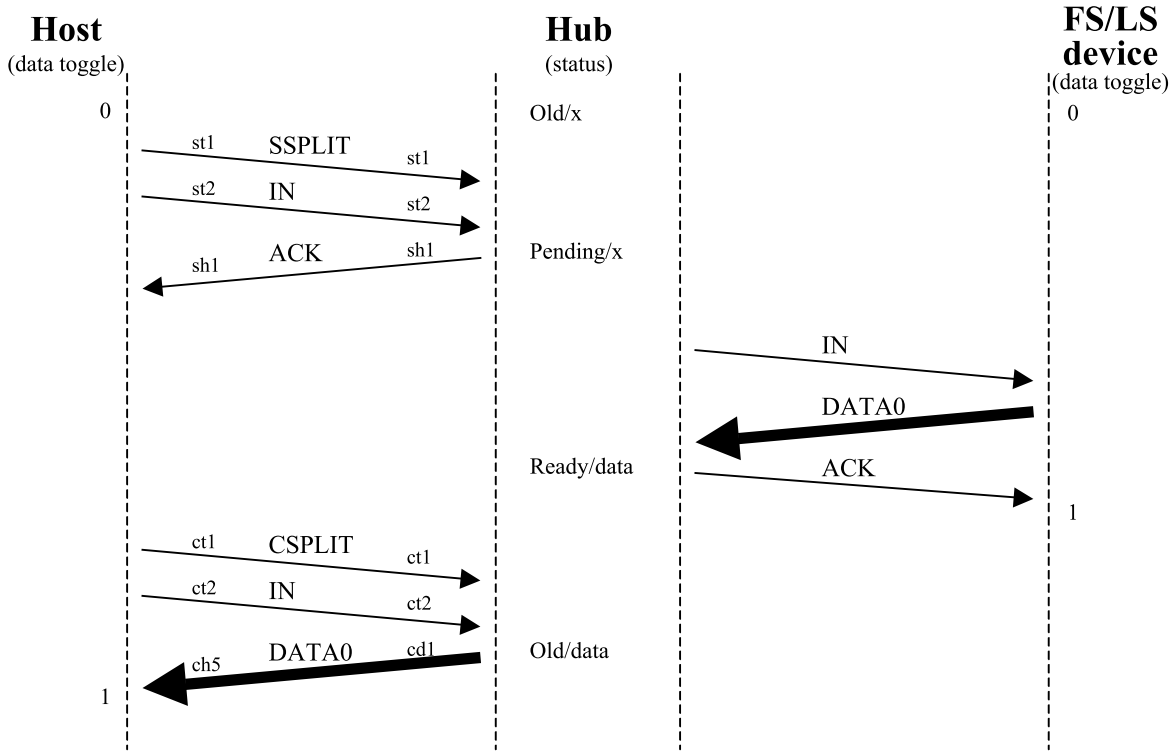



**Figure A-24. Normal No Smash**

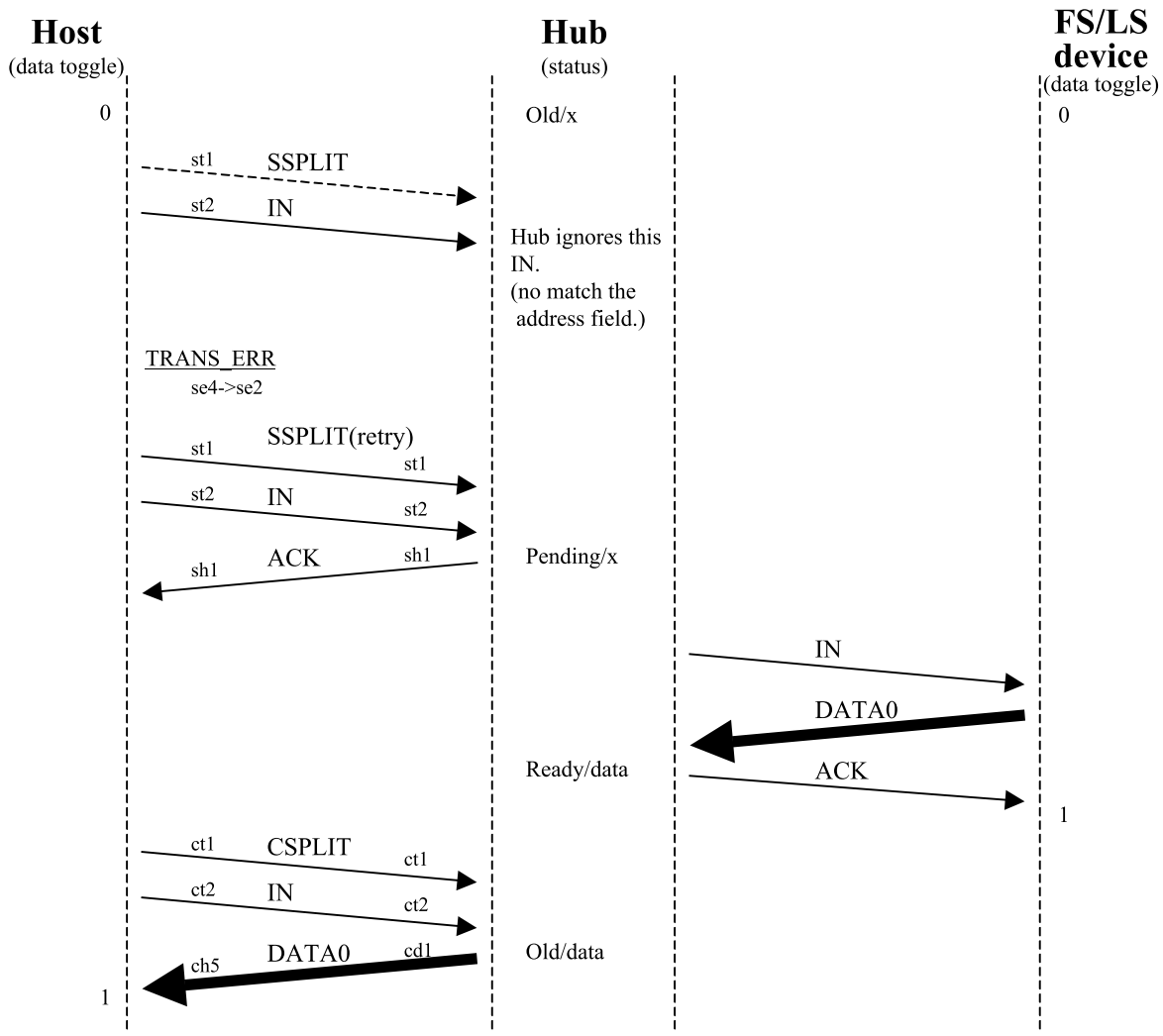



**Figure A-25. Normal HS SSPLIT Smash**

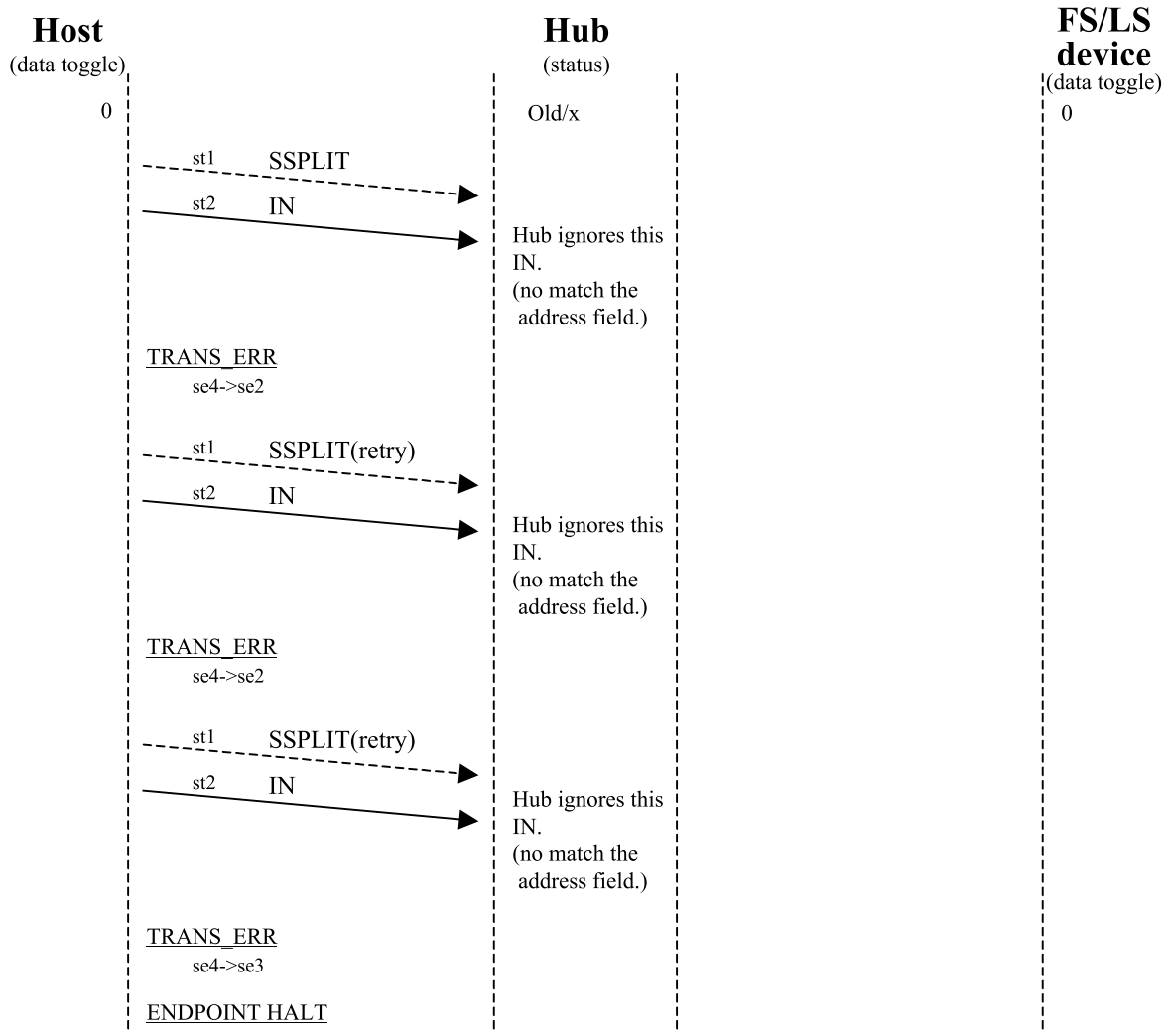



**Figure A-26. Normal SSPLIT 3 Strikes Smash**

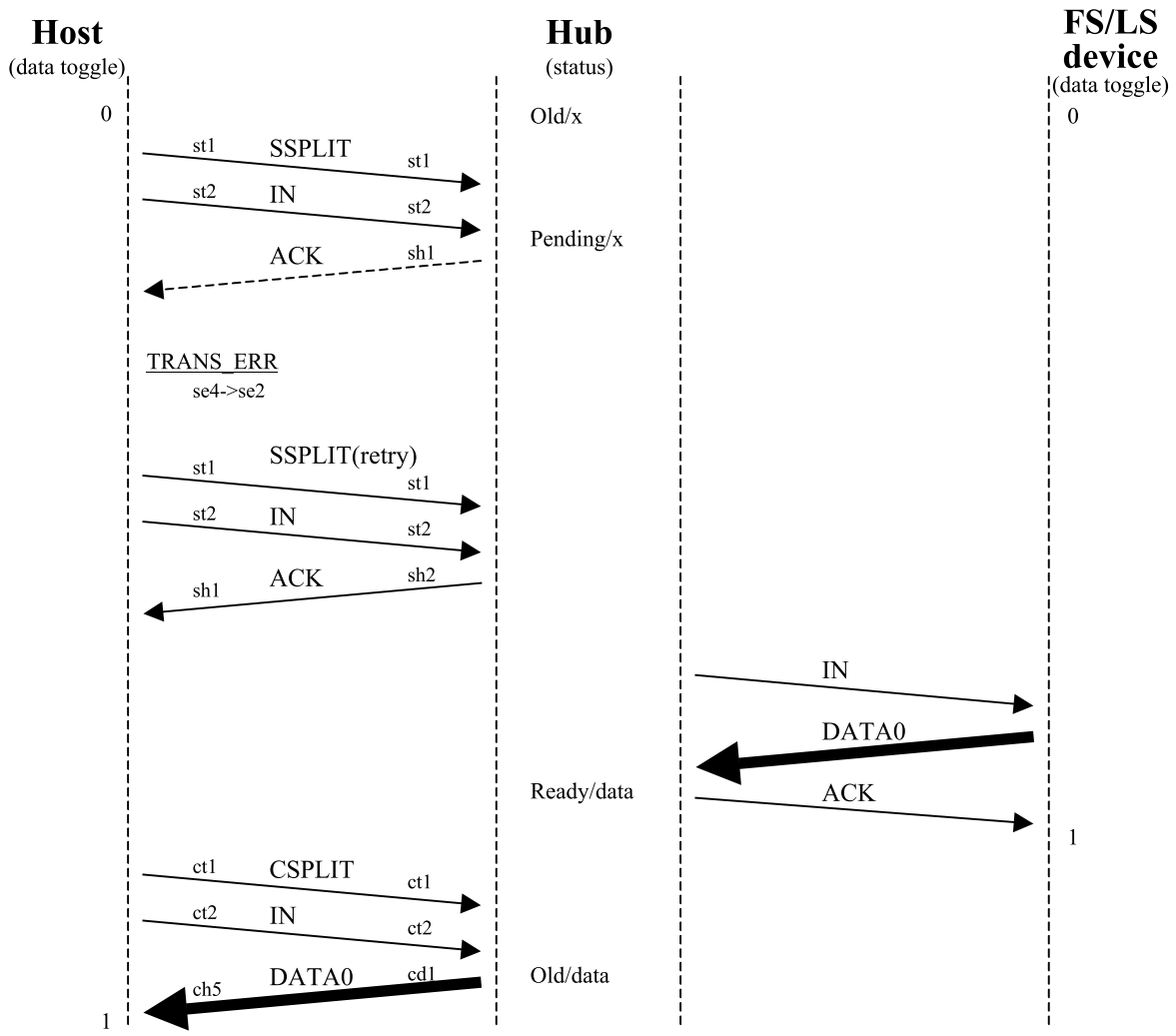



**Figure A-27. Normal HS ACK(S) Smash(case 1)**

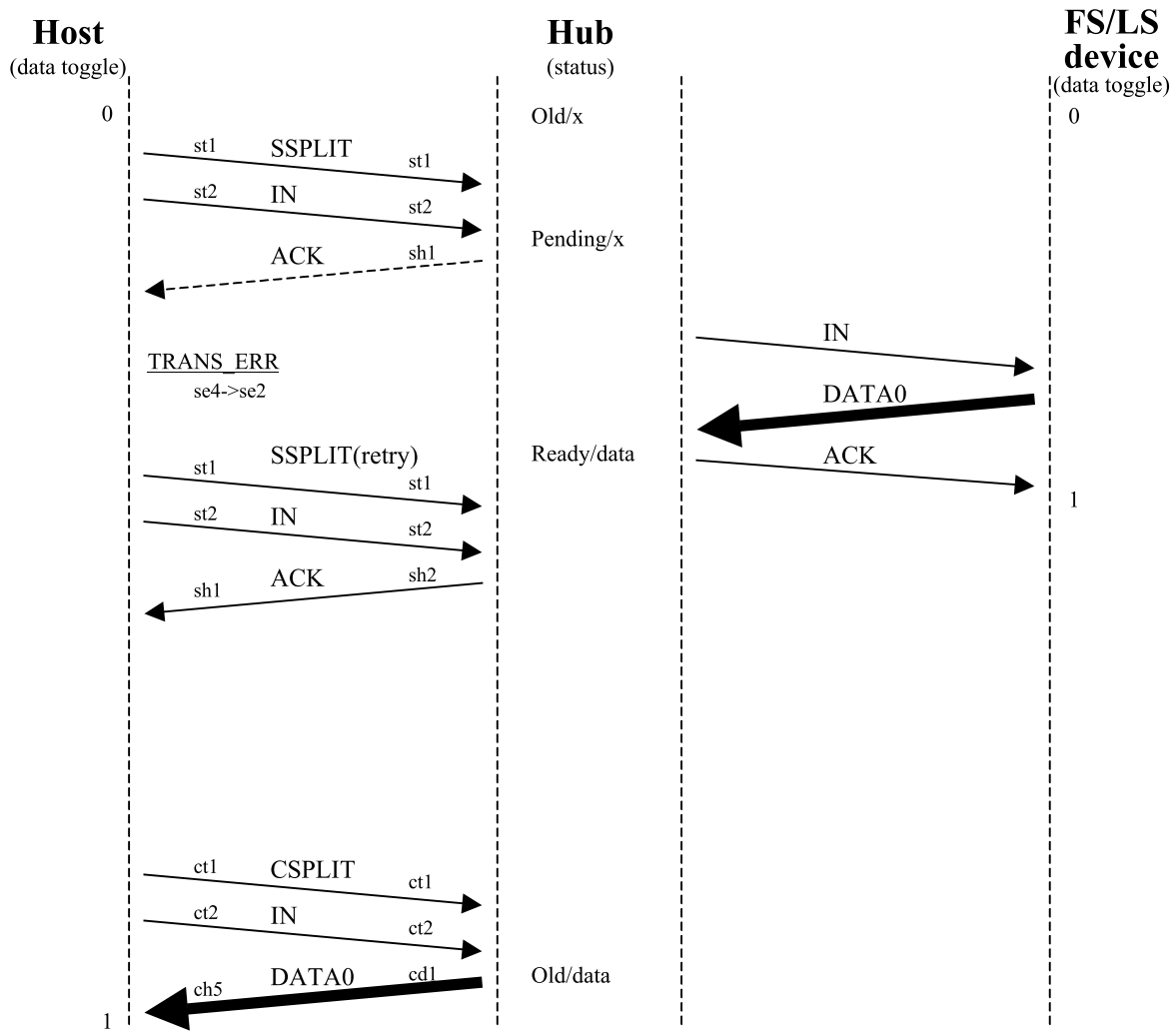



**Figure A-28. Normal HS ACK(S) Smash(case 2)**

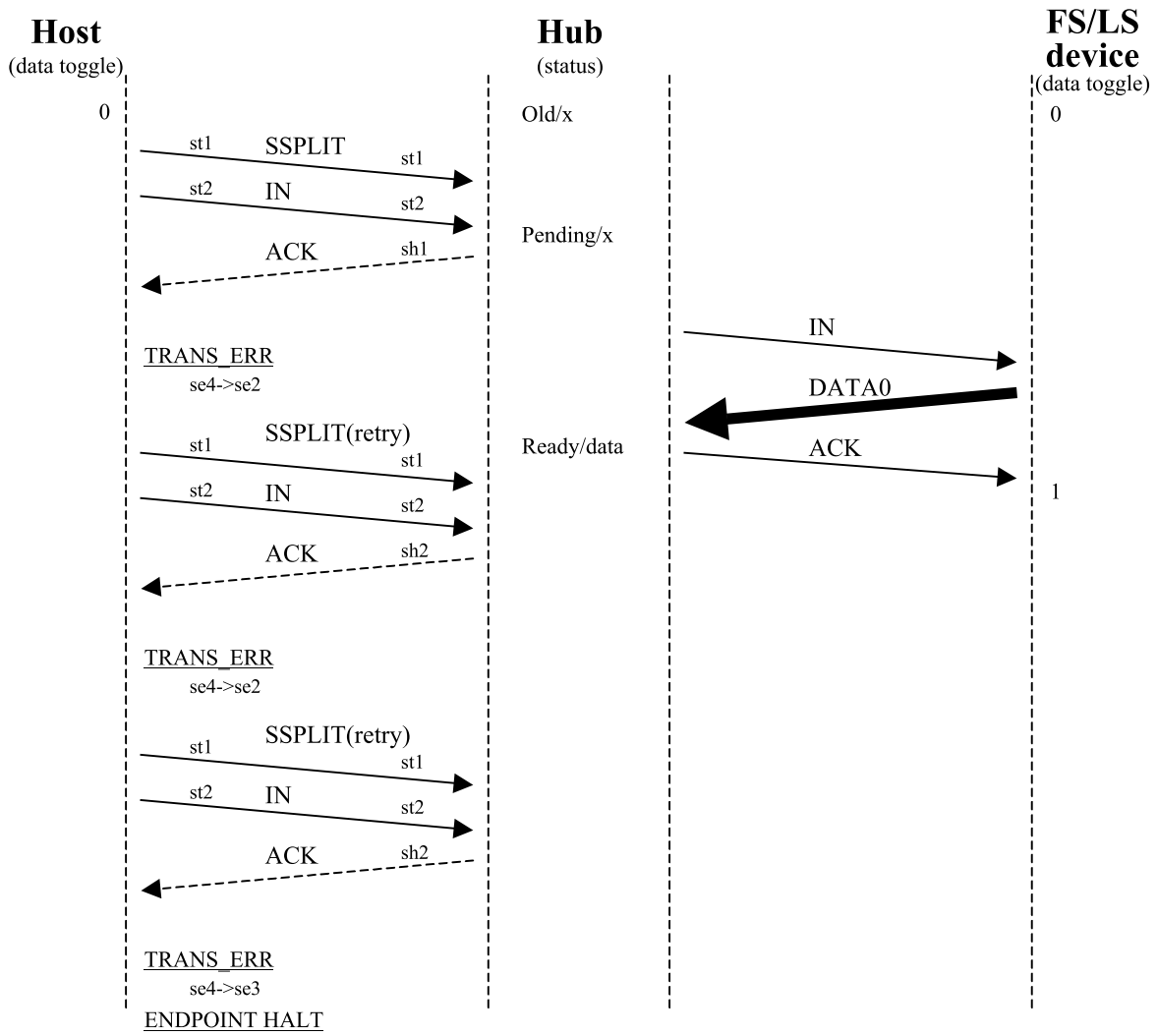



**Figure A-29. Normal HS ACK(S) 3 Strikes Smash**

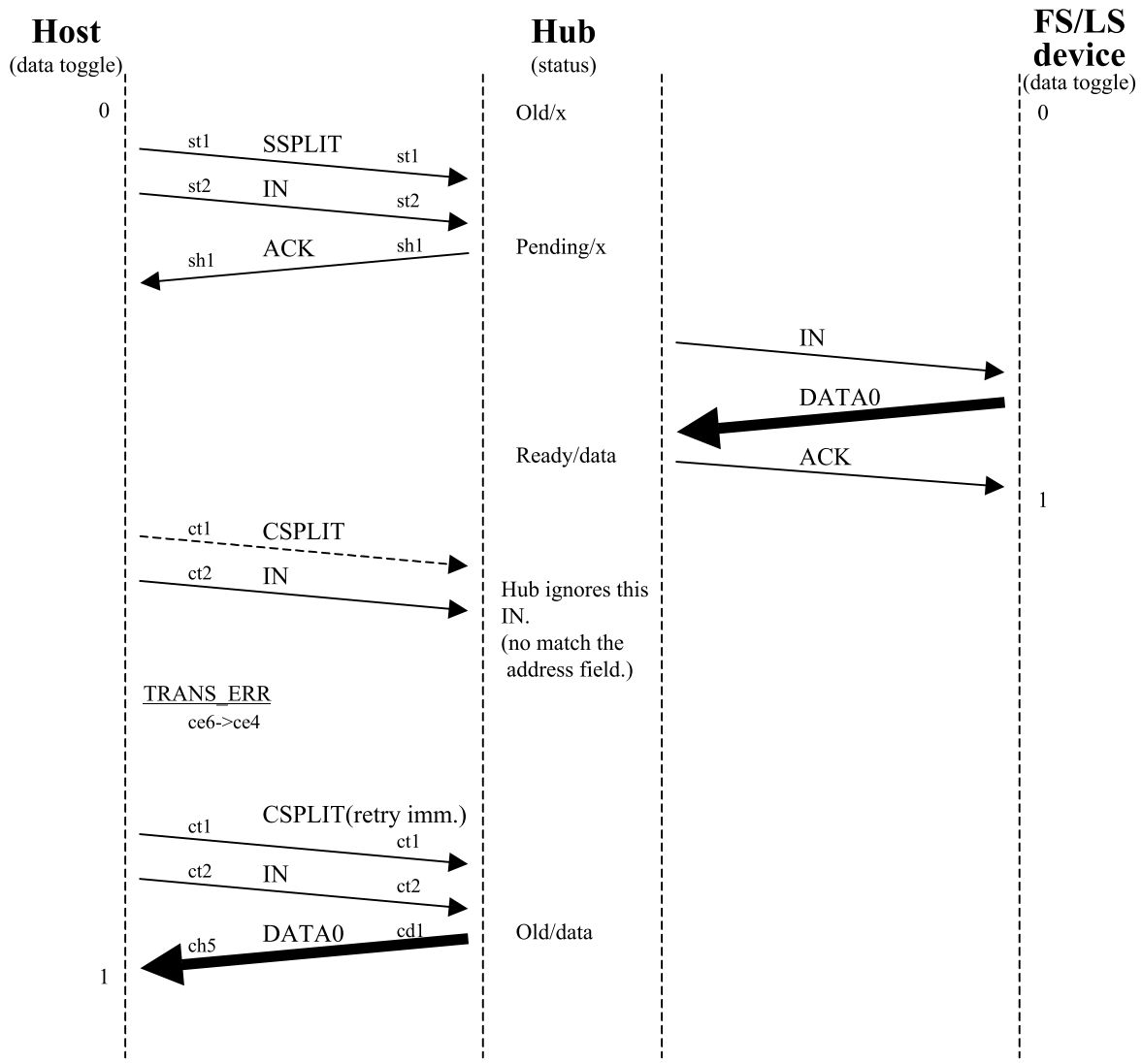



**Figure A-30. Normal HS CSPLIT Smash**

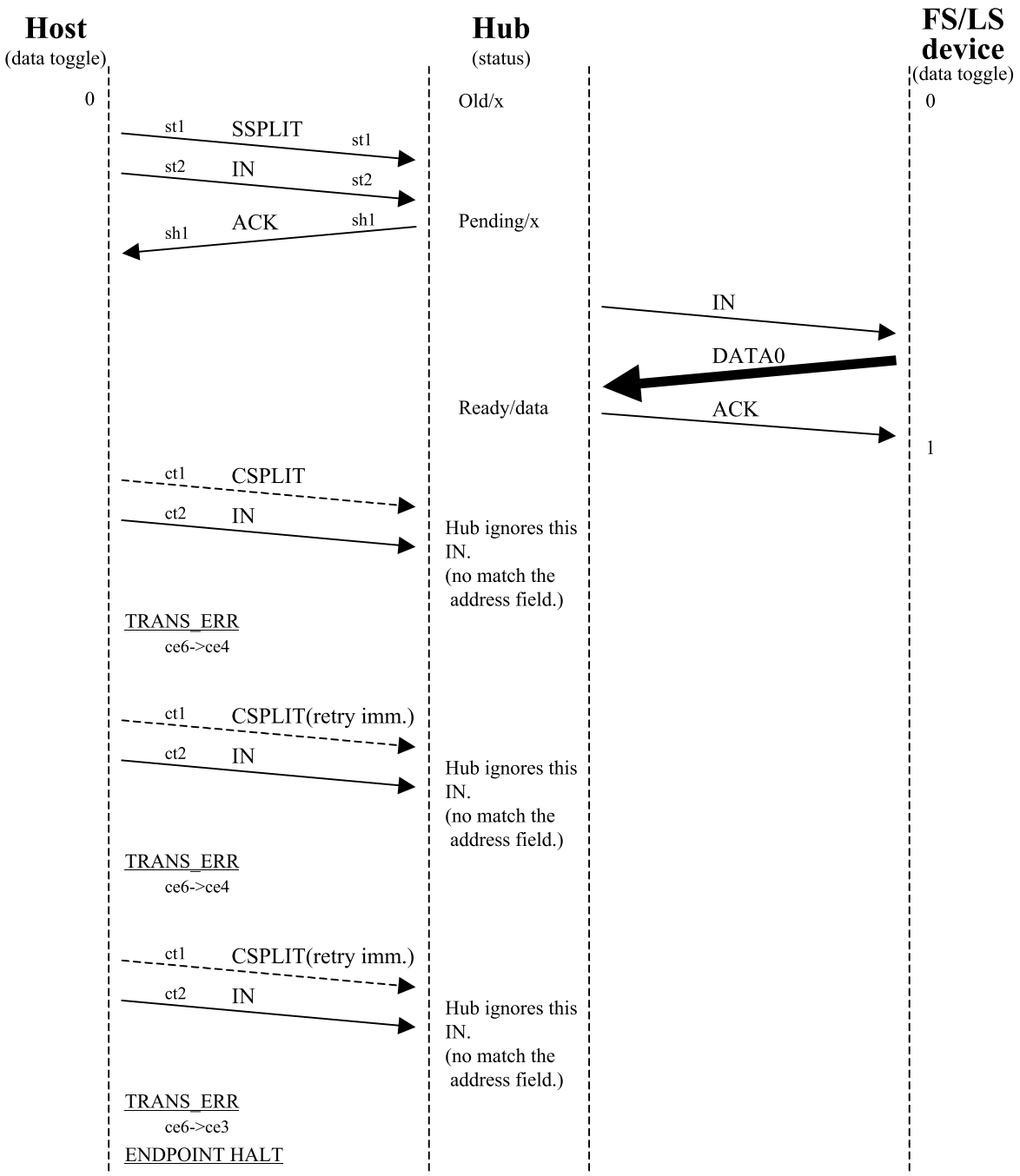



**Figure A-31. Normal HS CSPLIT 3 Strikes Smash**

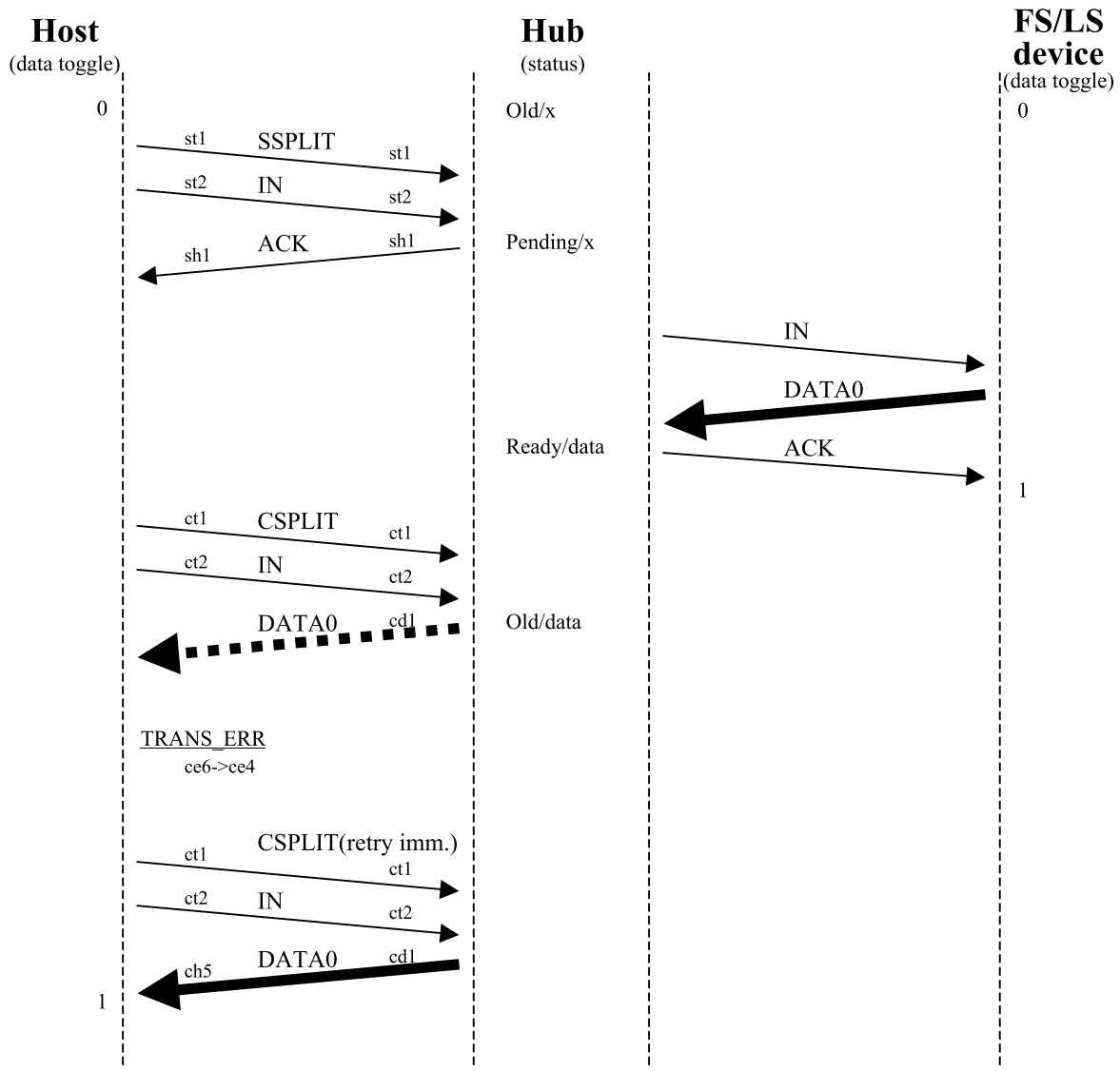



**Figure A-32. Normal HS DATA0/1 Smash**

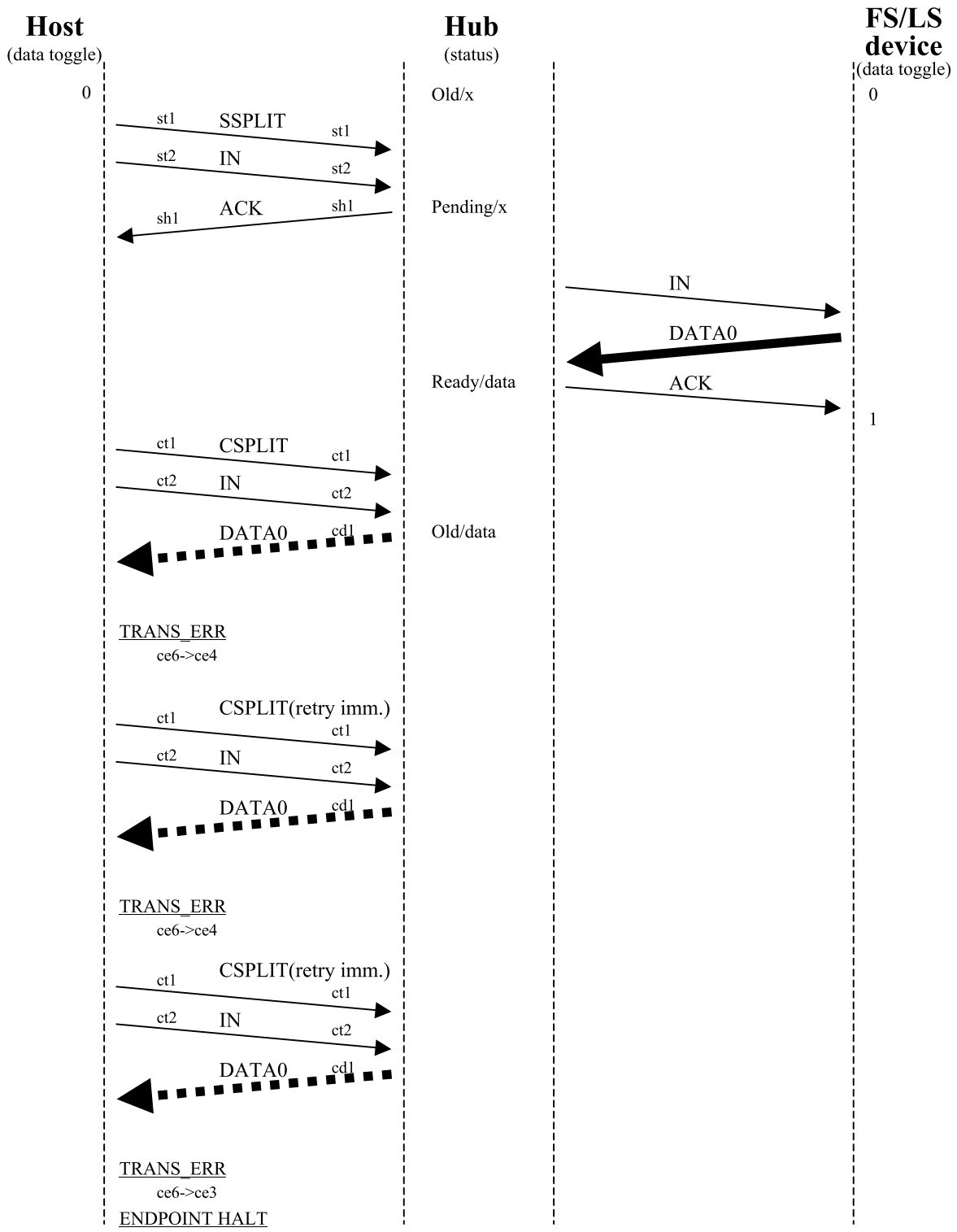



**Figure A-33. Normal HS DATA0/1 3 Strikes Smash**

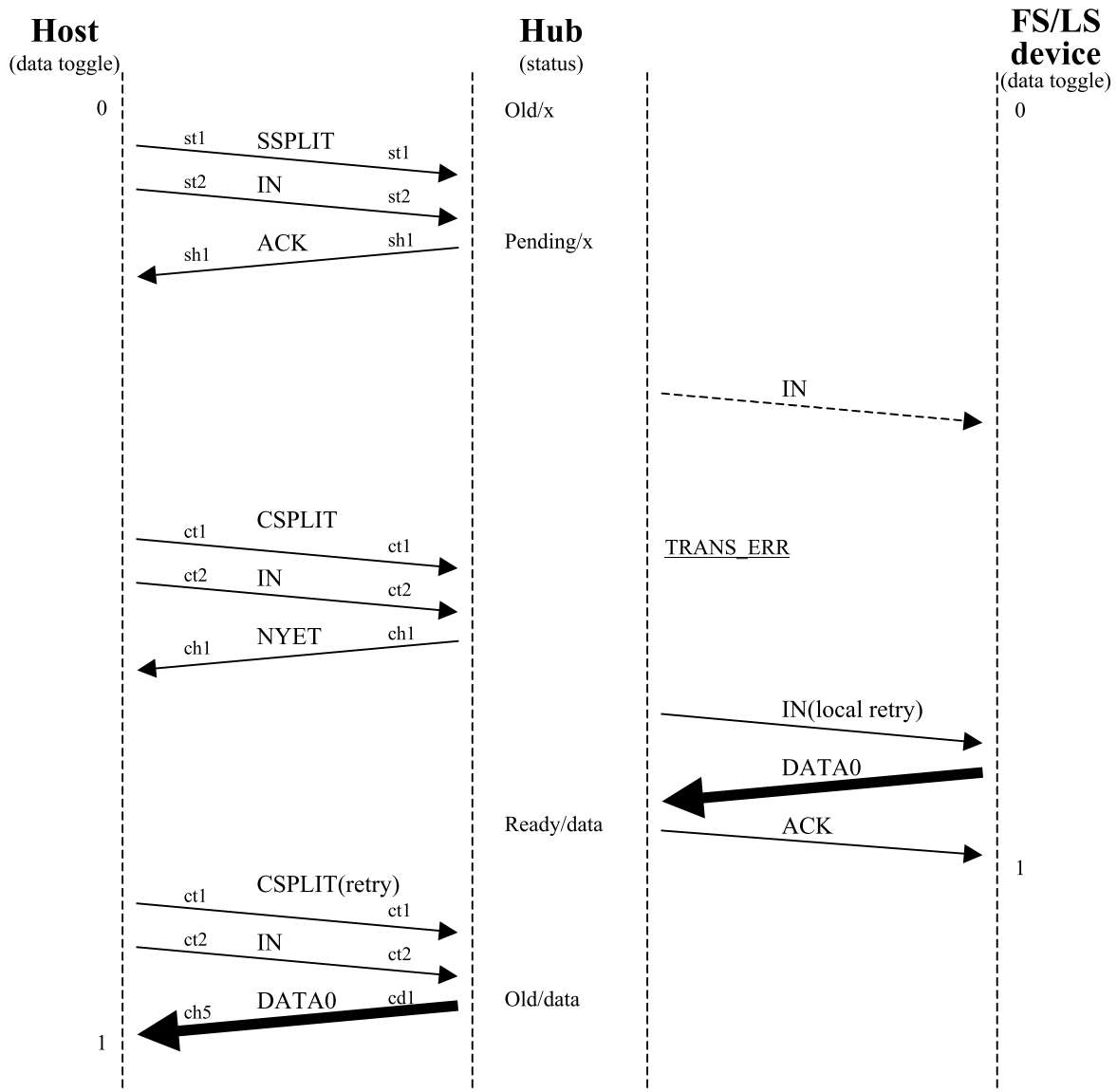



**Figure A-34. Normal FS/LS IN Smash**

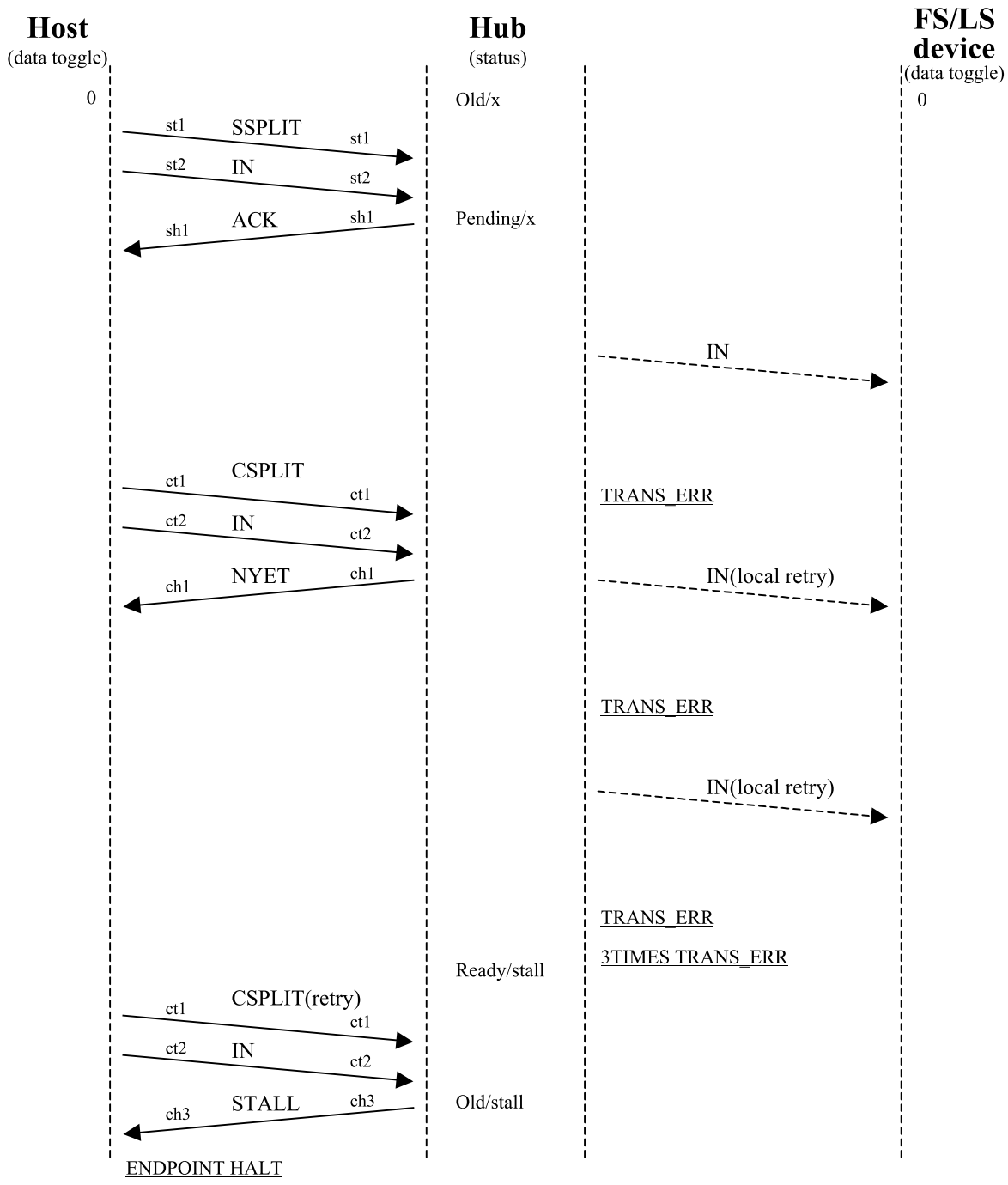



**Figure A-35. Normal FS/LS IN 3 Strikes Smash**

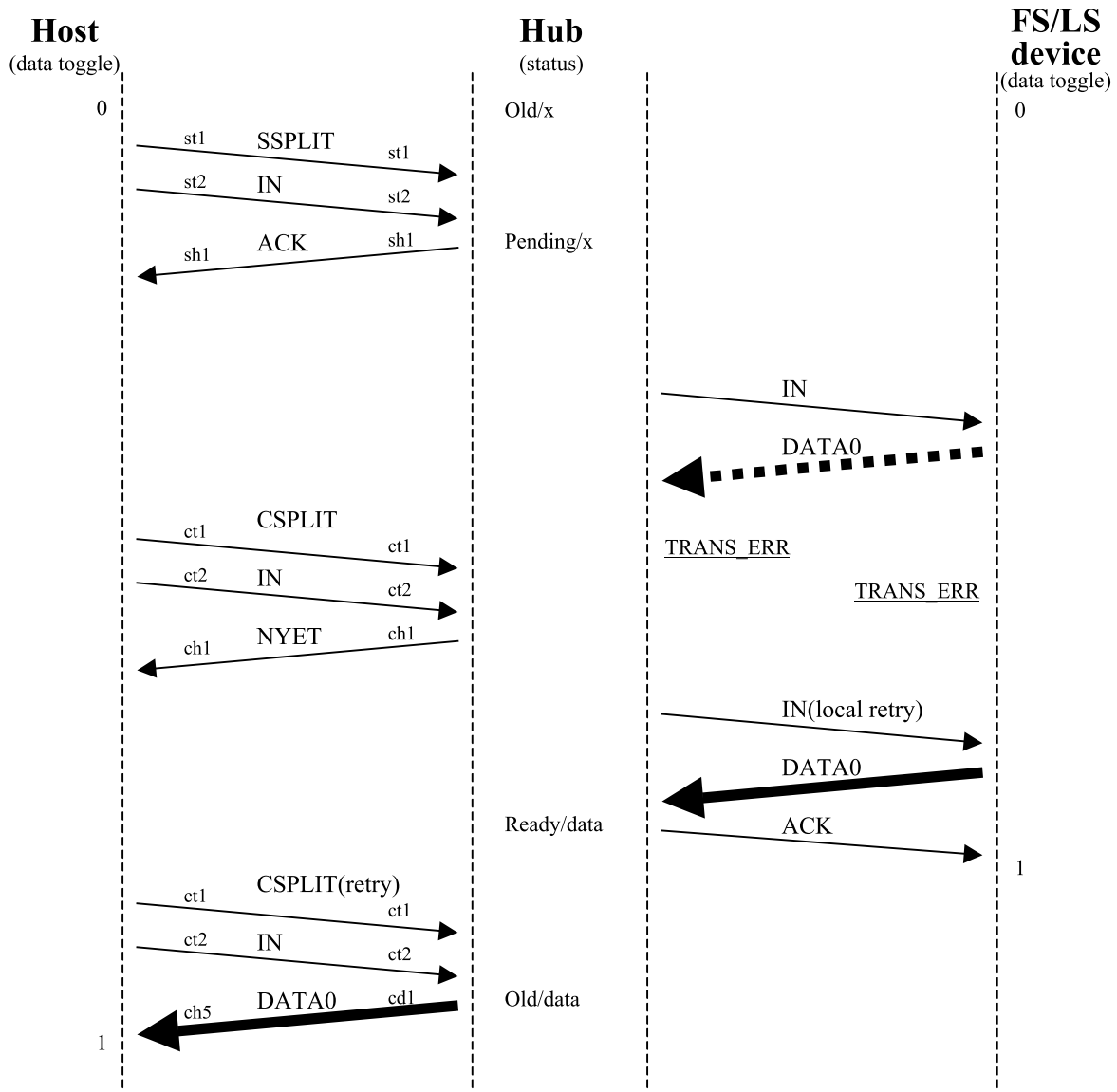



**Figure A-36. Normal FS/LS DATA0/1 Smash**

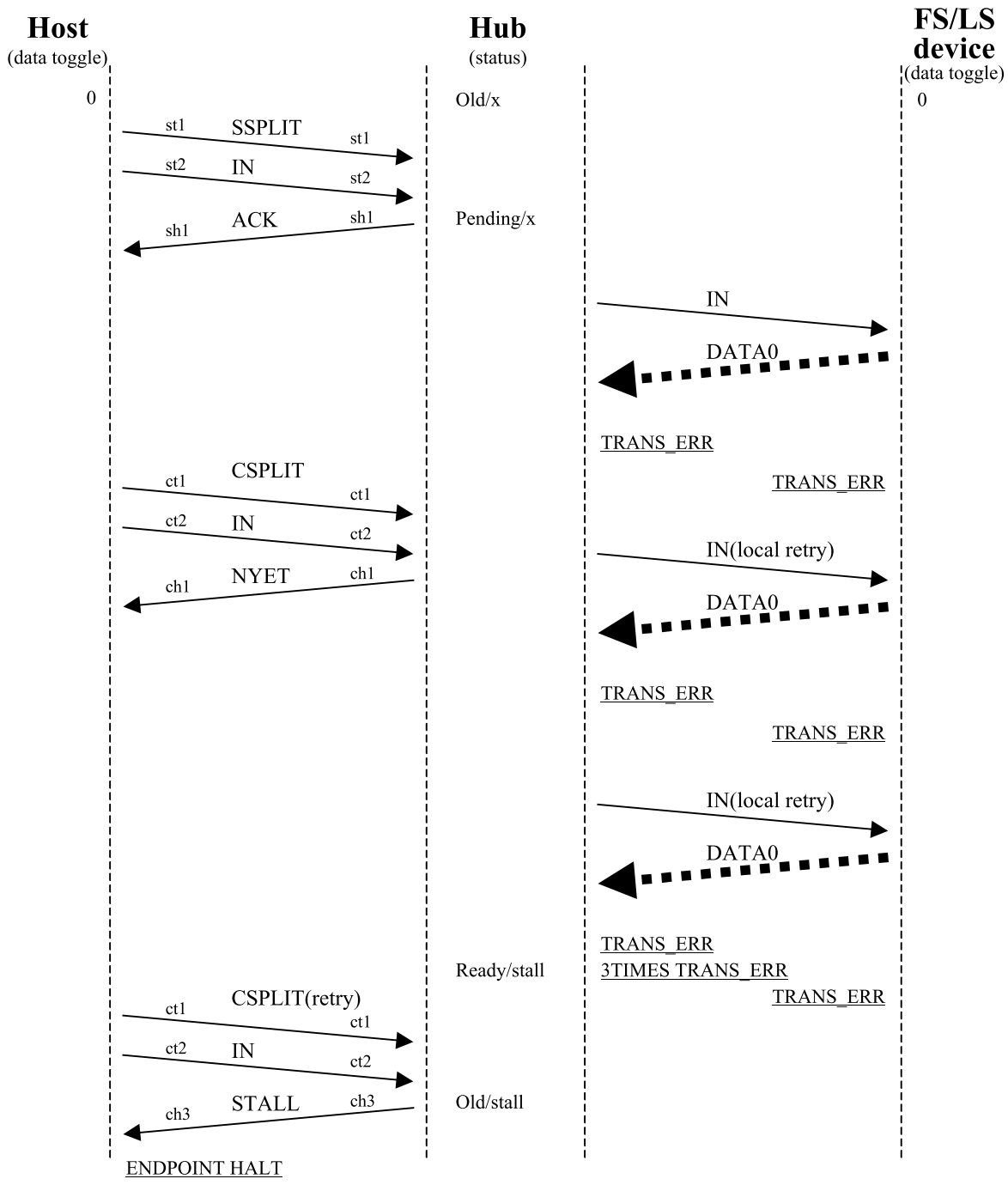



**Figure A-37. Normal FS/LS DATA0/1 3 Strikes Smash**

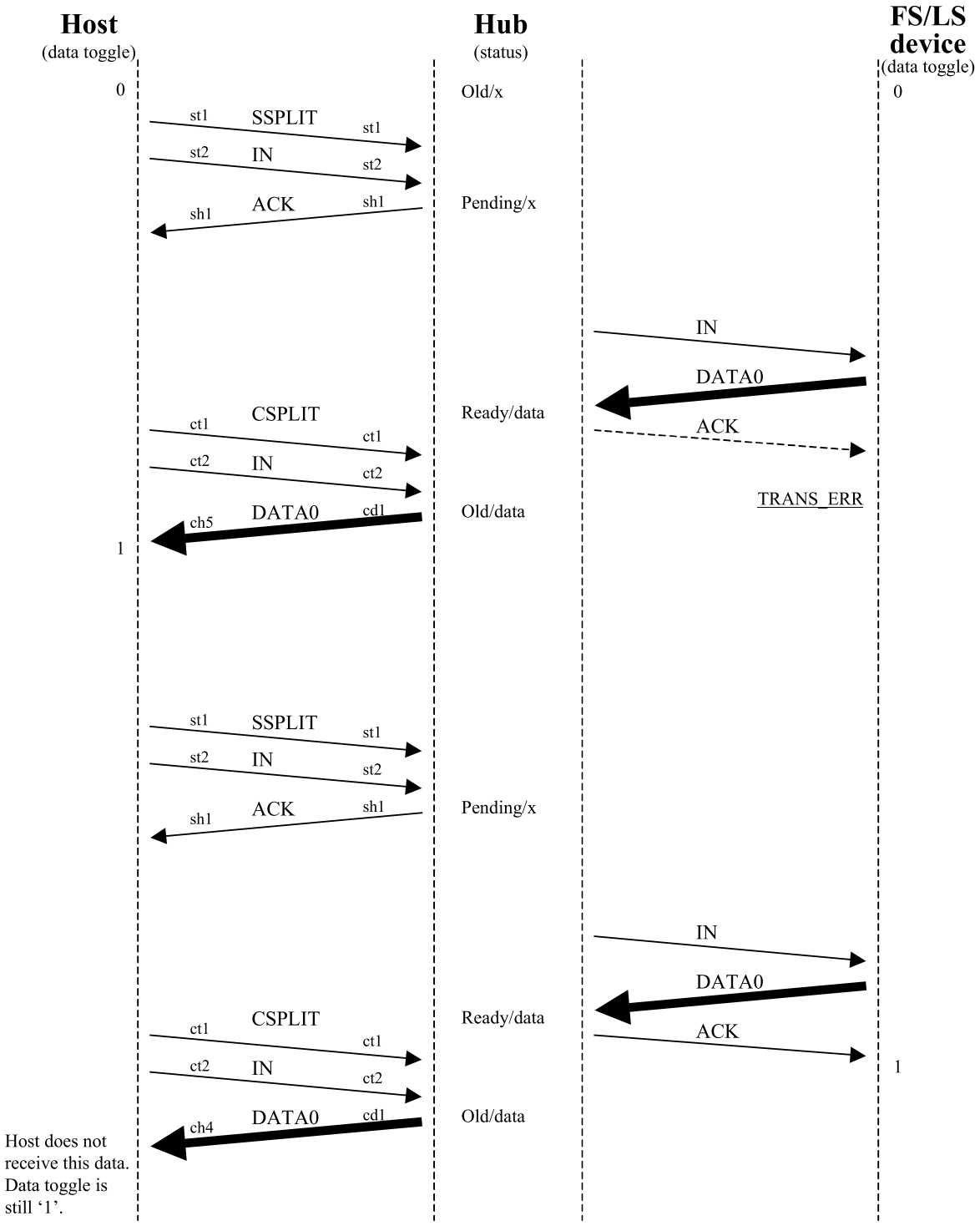



**Figure A-38. Normal FS/LS ACK Smash**

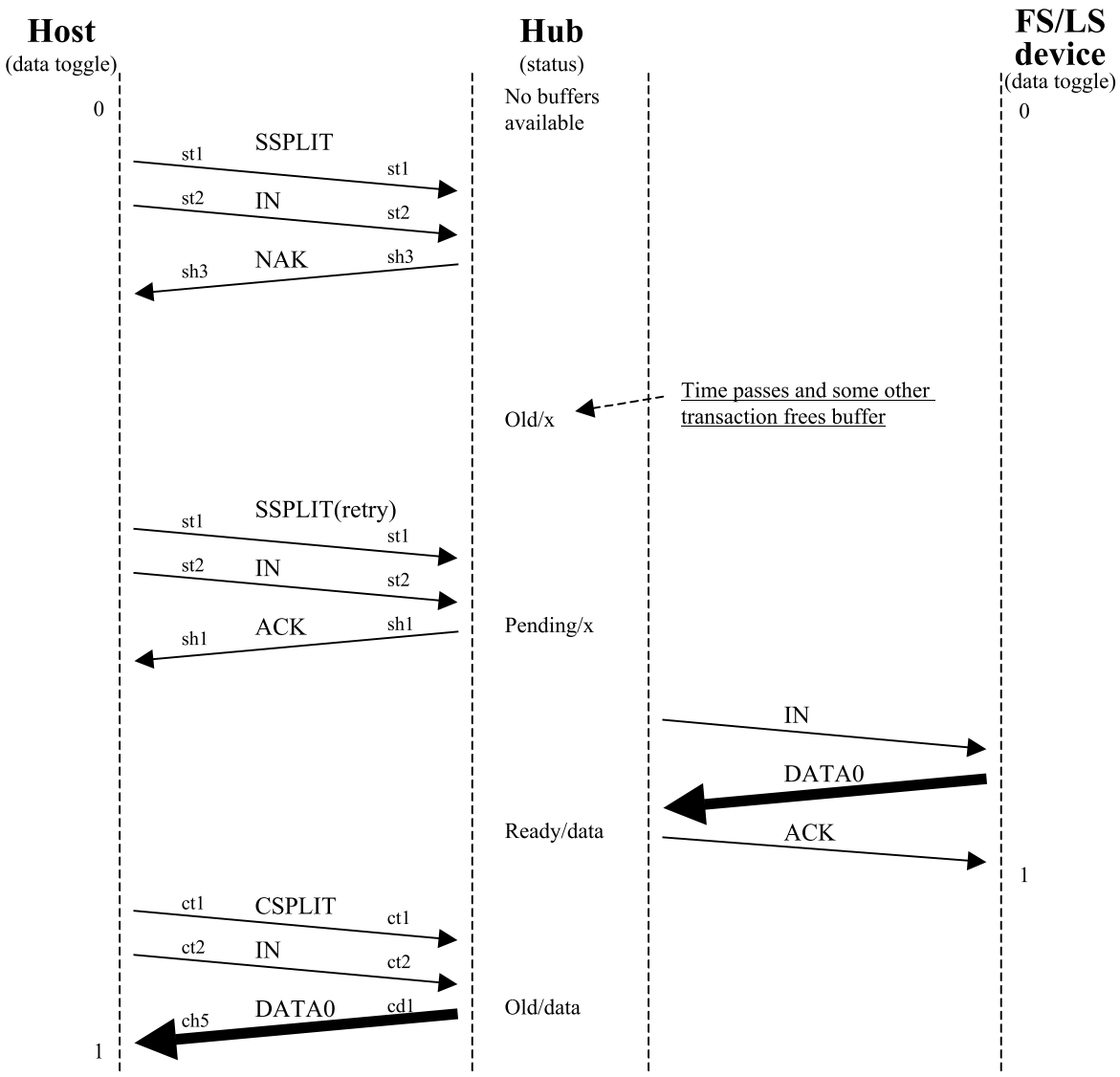



**Figure A-39. No Buffer Available No Smash(HS NAK(S))**

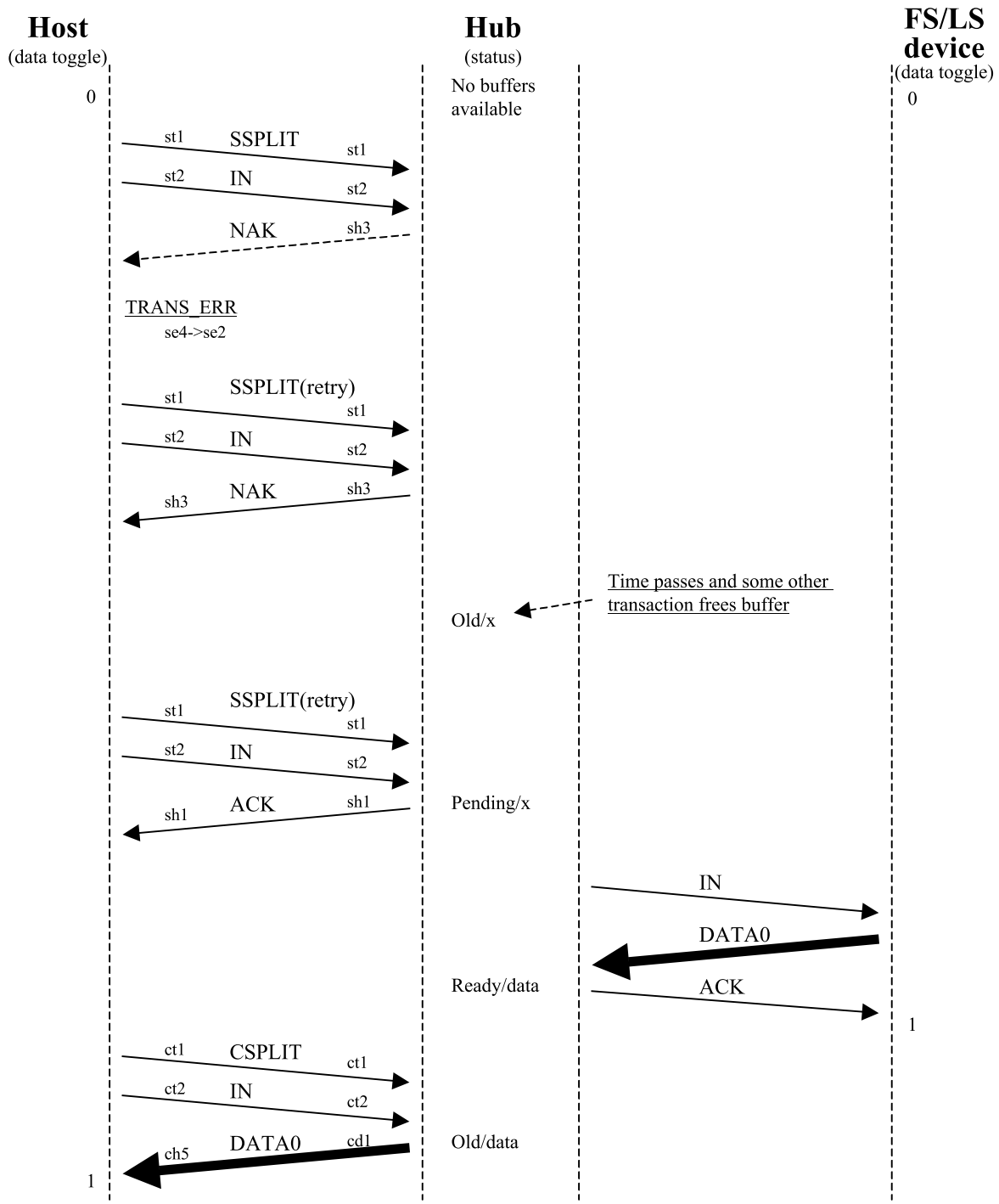



**Figure A-40. No Buffer Available HS NAK(S) Smash**

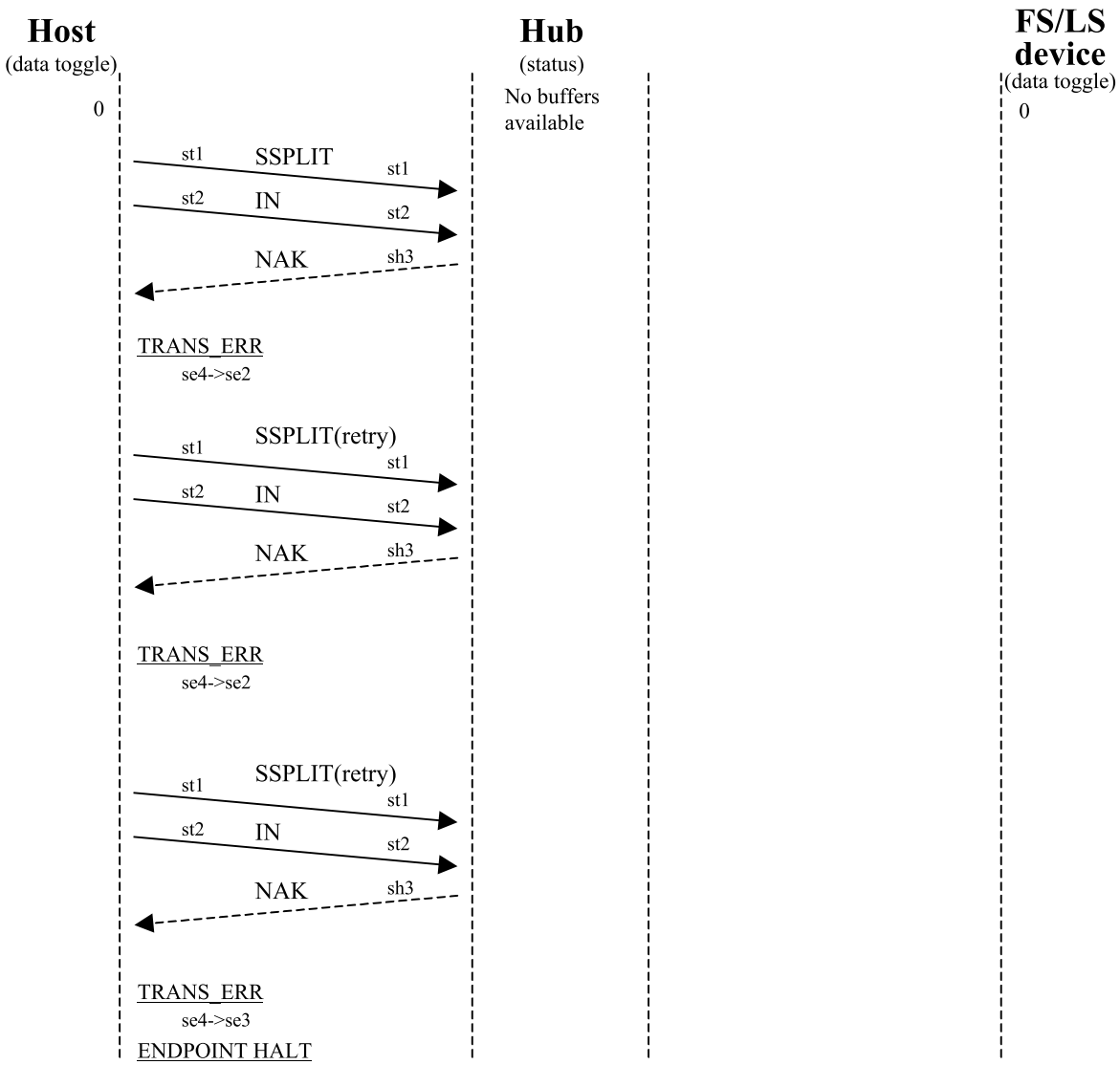



**Figure A-41. No Buffer Available HS NAK(S) 3 Strikes Smash**

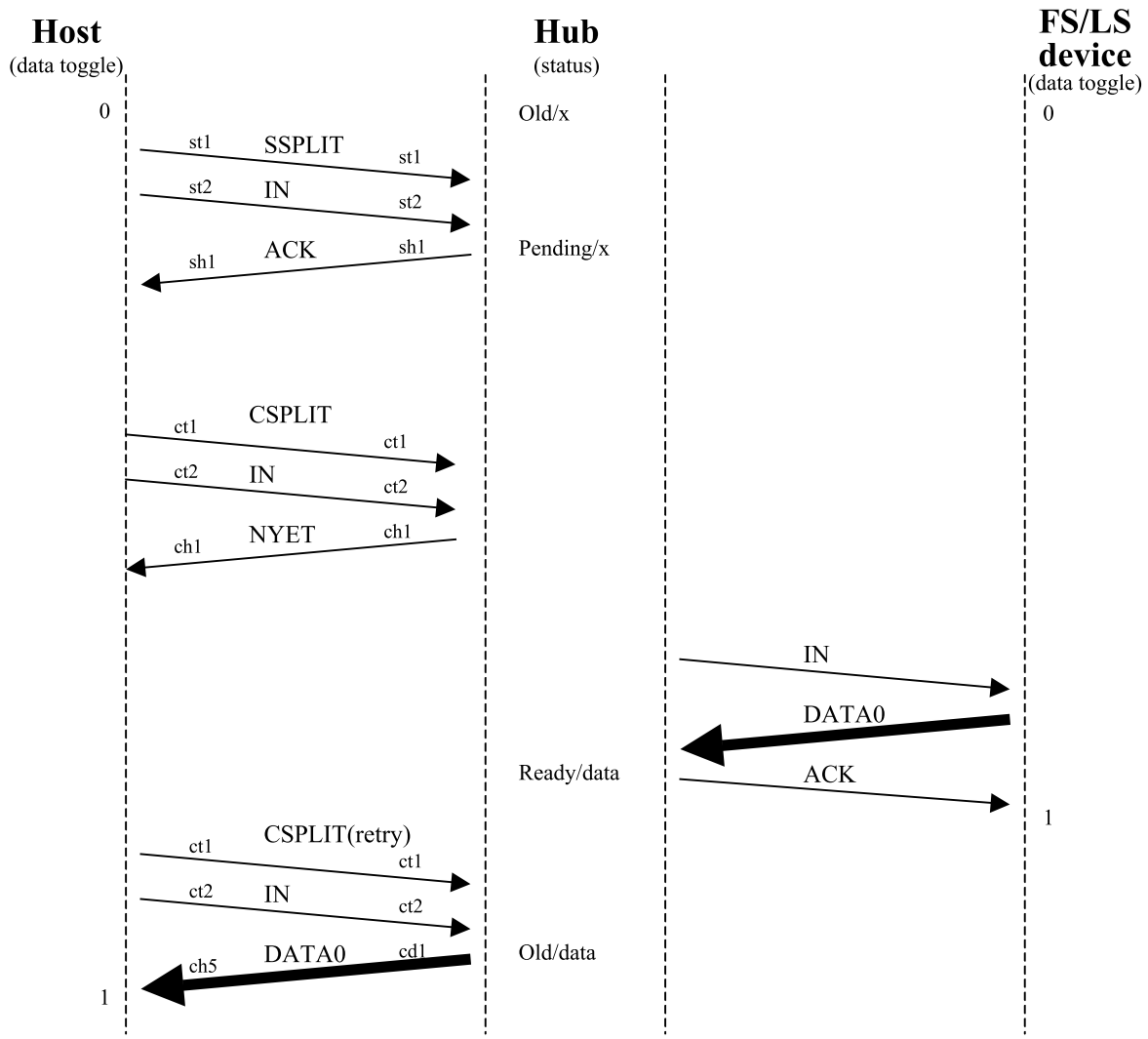



**Figure A-42. CS Earlier No Smash(HS NYET)**

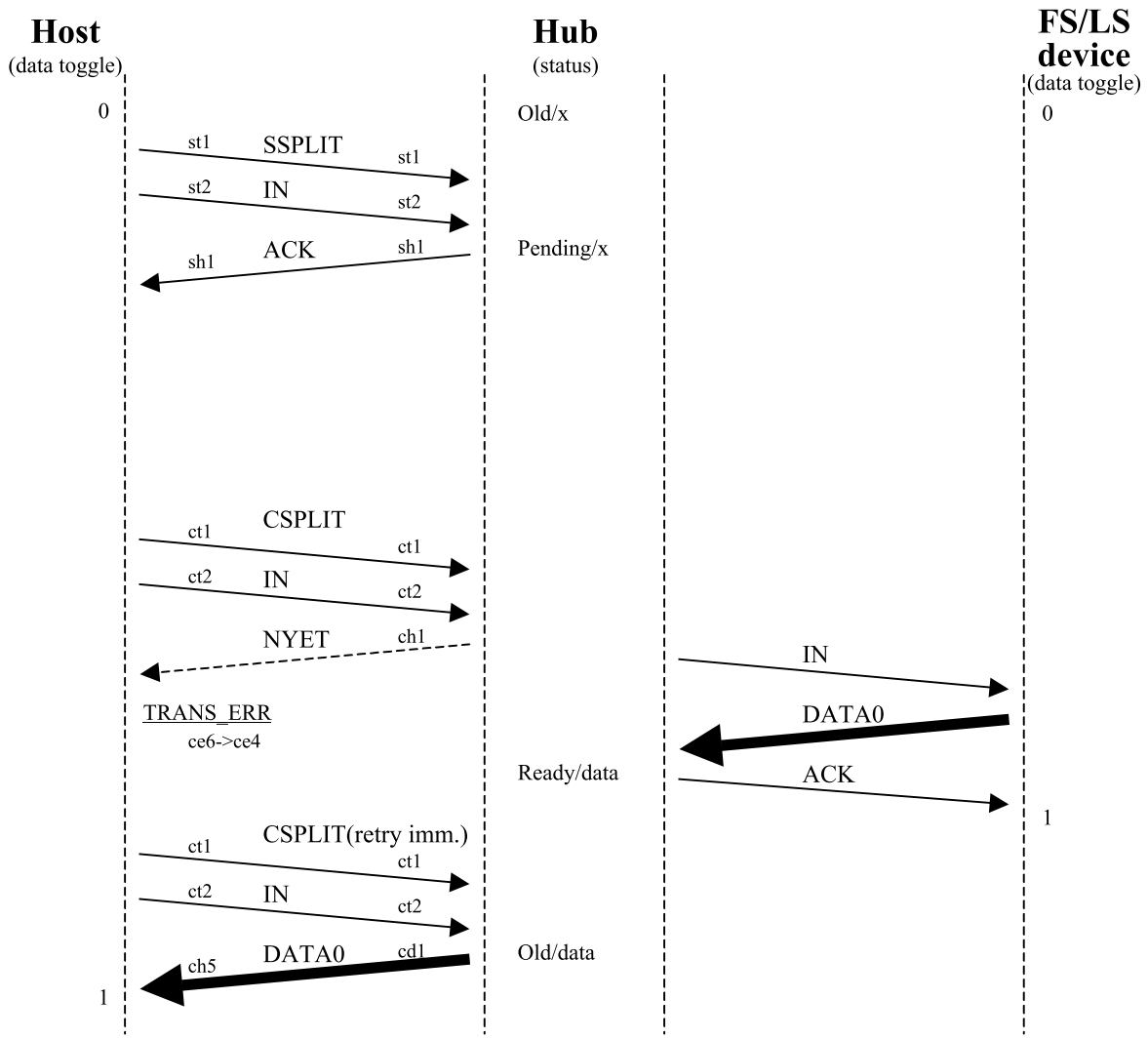



**Figure A-43. CS Earlier HS NYET Smash(case 1)**

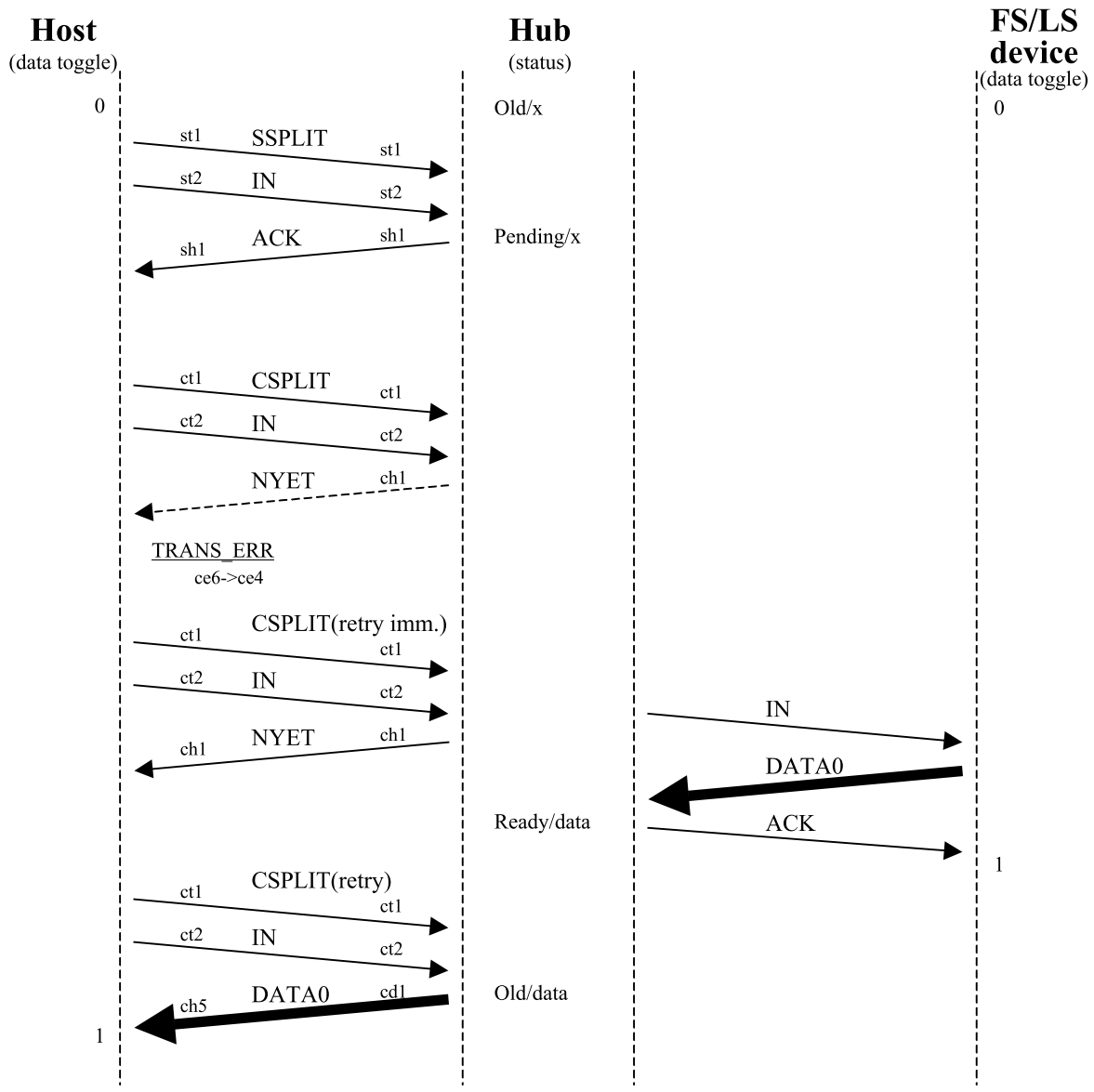



**Figure A-44. CS Earlier HS NYET Smash(case 2)**

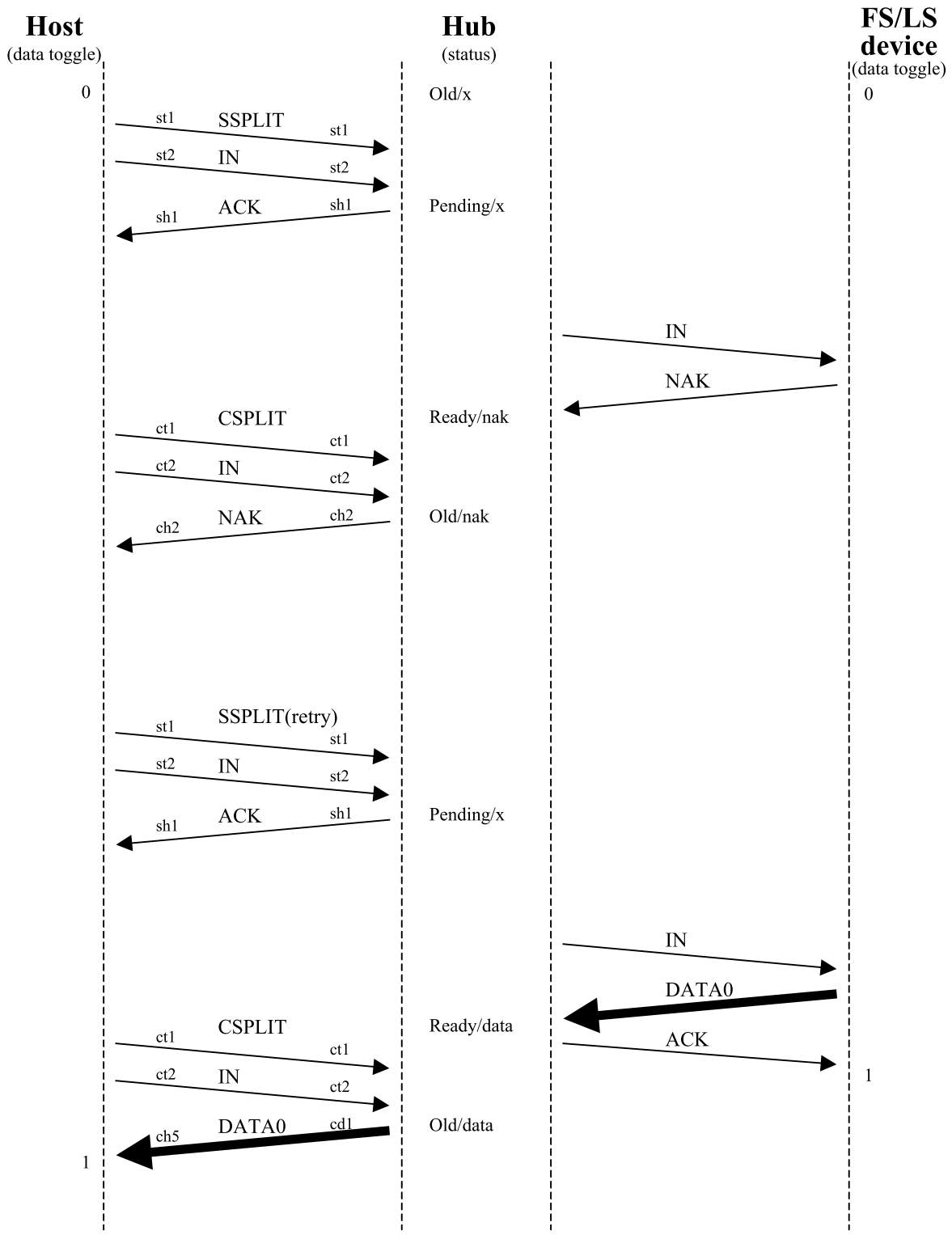



**Figure A-45. Device Busy No Smash(FS/LS NAK)**

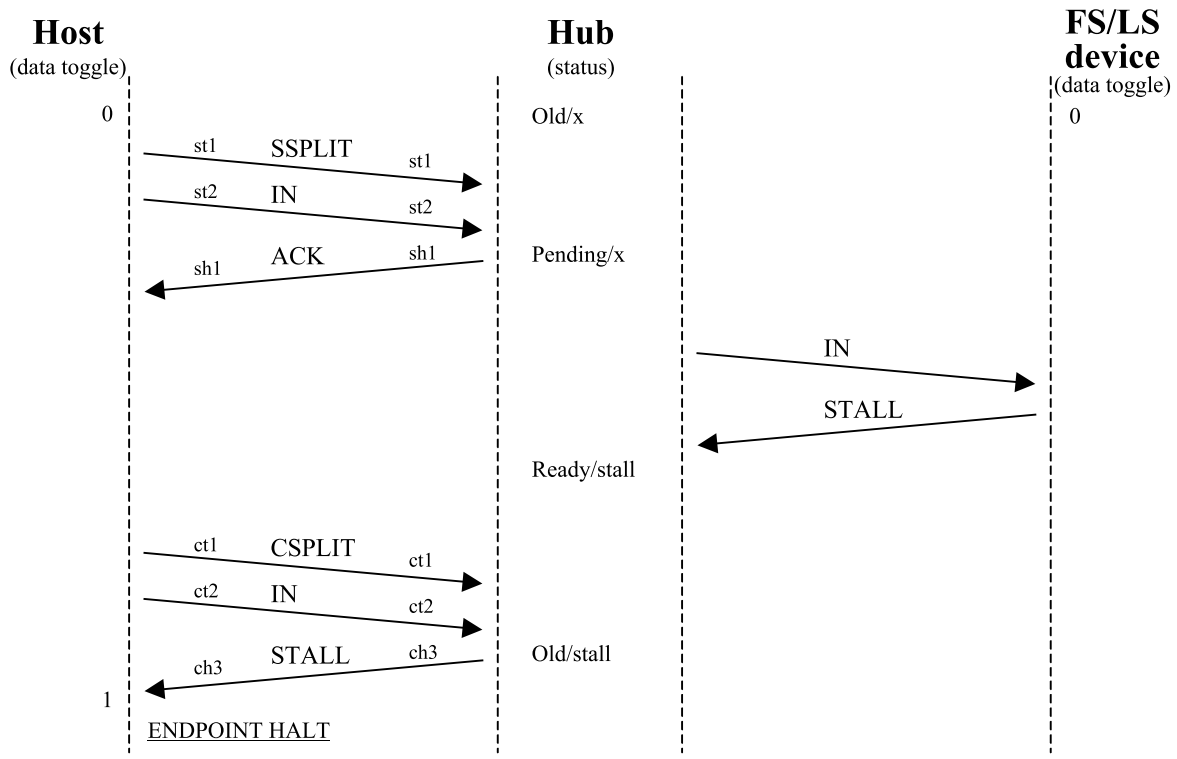



**Figure A-46. Device Stall No Smash(FS/LS STALL)**

## A.3 Interrupt OUT Transaction Examples

Legend:

(S): Start Split

(C): Complete Split

Summary of cases for Interrupt OUT transaction

- Normal cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash<br><br>(FS/LS handshake packet is done by M+1) | Figure A-47 | |
| HS SSPLIT smash | | Figure A-48 |
| HS SSPLIT 3 strikes smash | No figure | |
| HS OUT(S) smash | | Figure A-48 |
| HS OUT(S) 3 strikes smash | No figure | |
| HS DATA0/1 smash | Figure A-48 | |
| HS DATA0/1 3 strikes smash | No figure | |
| HS CSPLIT smash | Figure A-49 | |
| HS CSPLIT 3 strikes smash | Figure A-50 | |
| HS OUT(C) smash | | Figure A-49 |
| HS OUT(C) 3 strikes smash | | Figure A-50 |
| HS ACK(C) smash | Figure A-51 | |
| HS ACK(C) 3 strikes smash | Figure A-52 | |
| FS/LS OUT smash | | Figure A-53 |
| FS/LS OUT 3 strikes smash | No figure | |
| FS/LS DATA0/1 smash | Figure A-53 | |
| FS/LS DATA0/1 3 strikes smash | No figure | |
| FS/LS ACK smash | Figure A-54 | |

| | | |
|---|---|---|
| FS/LS ACK 3 strikes smash | No figure | |

- Searcing

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash | Figure A-55 | |

- CS(Complete-split transaction) earlier cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash (HS NYETand FS/LS handshake packet is done by M+2) | Figure A-56 | |
| No smash(HS NYET and FS/LS handshake packet is done by M+3) | Figure A-57 | |
| HS NYET smash | Figure A-58 | |
| HS NYET 3 strikes smash | Figure A-59 | |

- Abort and Free cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash and abort (HS NYETand FS/LS transaction is continued at end of M+3) | Figure A-60 | |
| No smash and free(HS NYETand FS/LS transaction is not started at end of M+3) | Figure A-61 | |

- FS/LS transaction error cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| HS ERR smash | | Figure A-51 |
| HS ERR 3 strikes smash | | Figure A-52 |

- Device busy cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash(HS NAK(C)) | Figure A-62 | |
| HS NAK(C) smash | | Figure A-51 |
| HS NAK(C) 3 strikes smash | | Figure A-52 |
| FS/LS NAK smash | | Figure A-53 |
| FS/LS NAK 3 strikes smash | No figure | |

- Device stall cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash | Figure A-63 | |
| HS STALL(C) smash | | Figure A-51 |
| HS STALL(C) 3 strikes smash | | Figure A-52 |
| FS/LS STALL smash | | Figure A-53 |
| FS/LS STALL 3 strikes smash | No figure | |

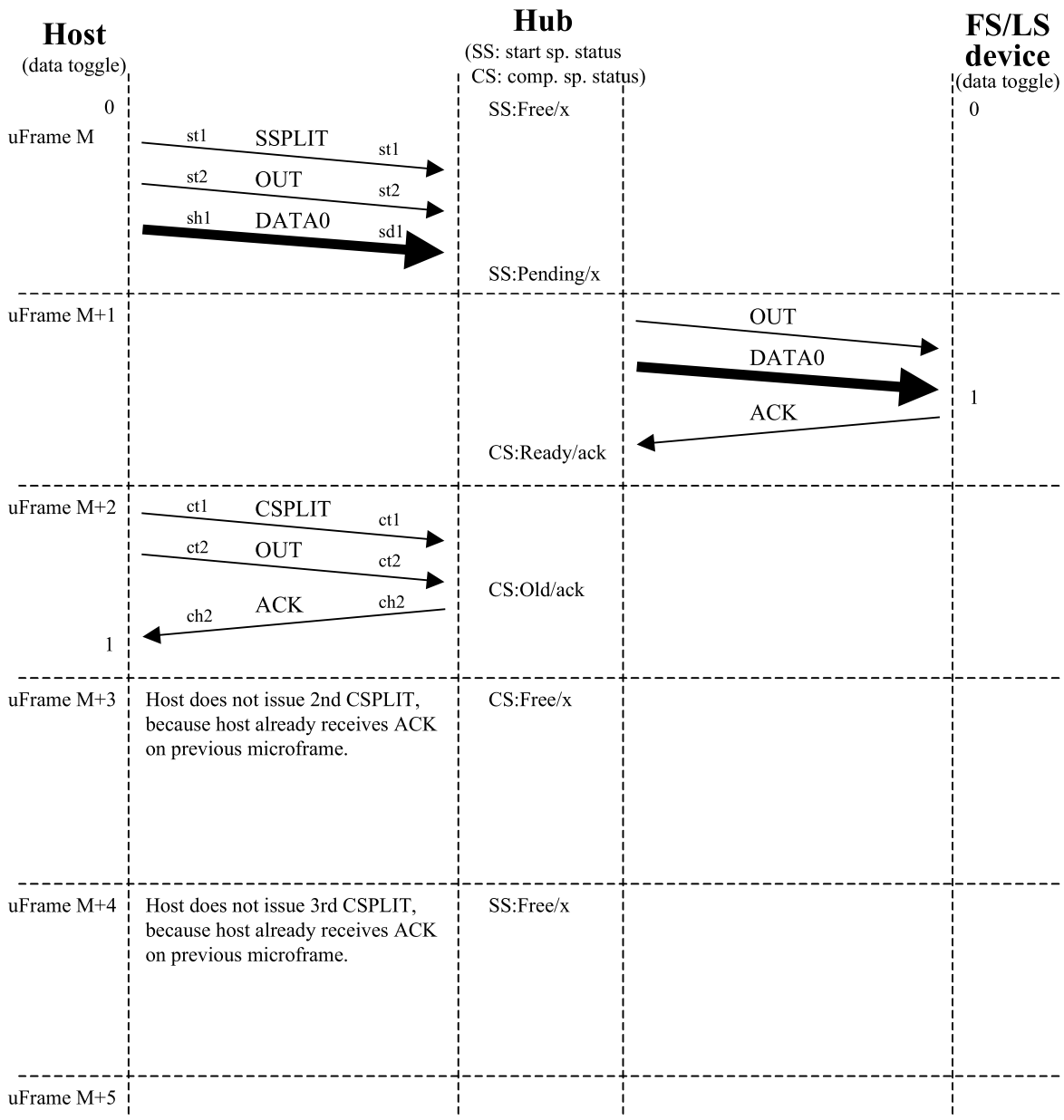



**Figure A-47. Normal No Smash(FS/LS Handshake Packet is Done by M+1)**

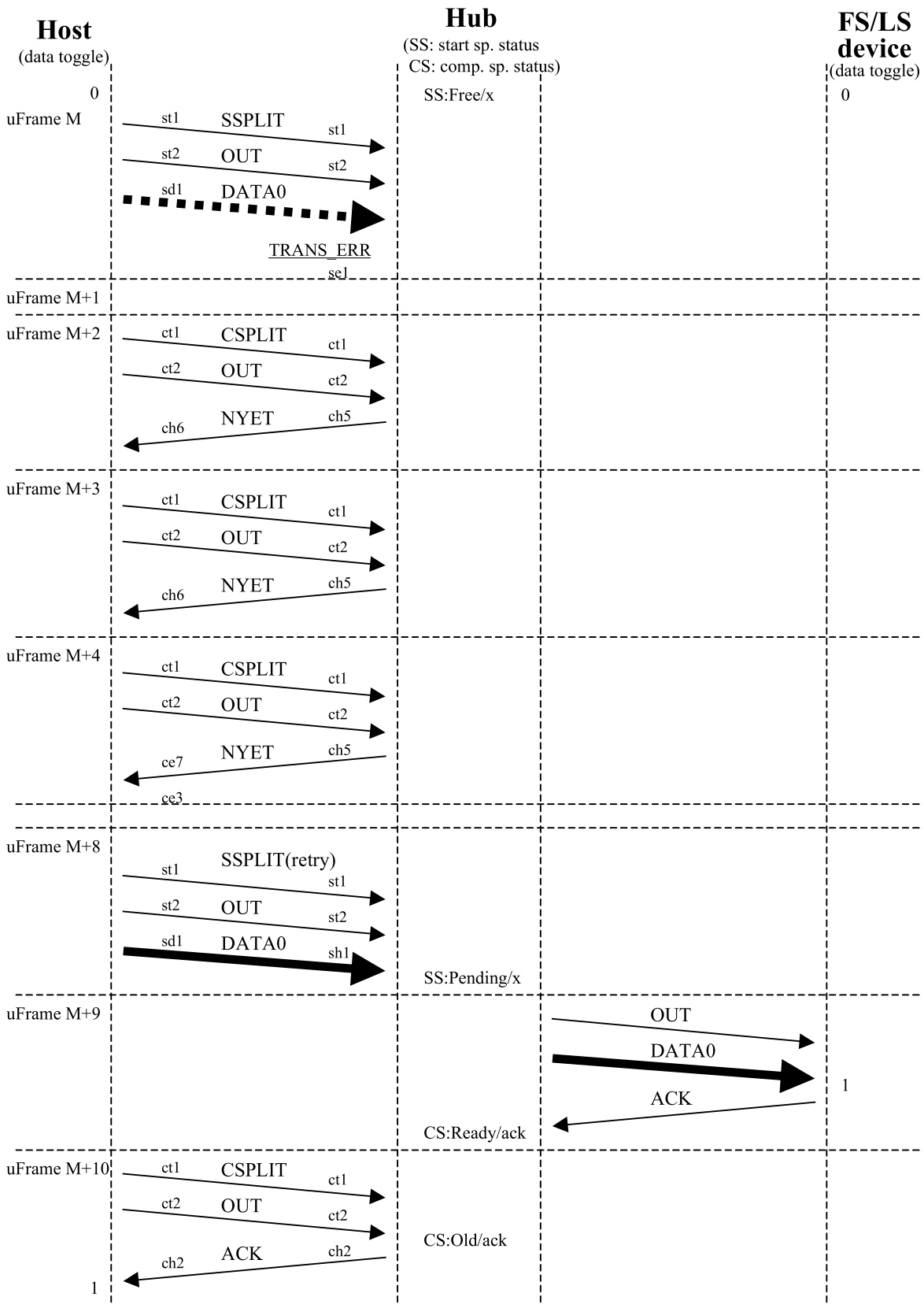



**Figure A-48. Normal HS DATA0/1 Smash**

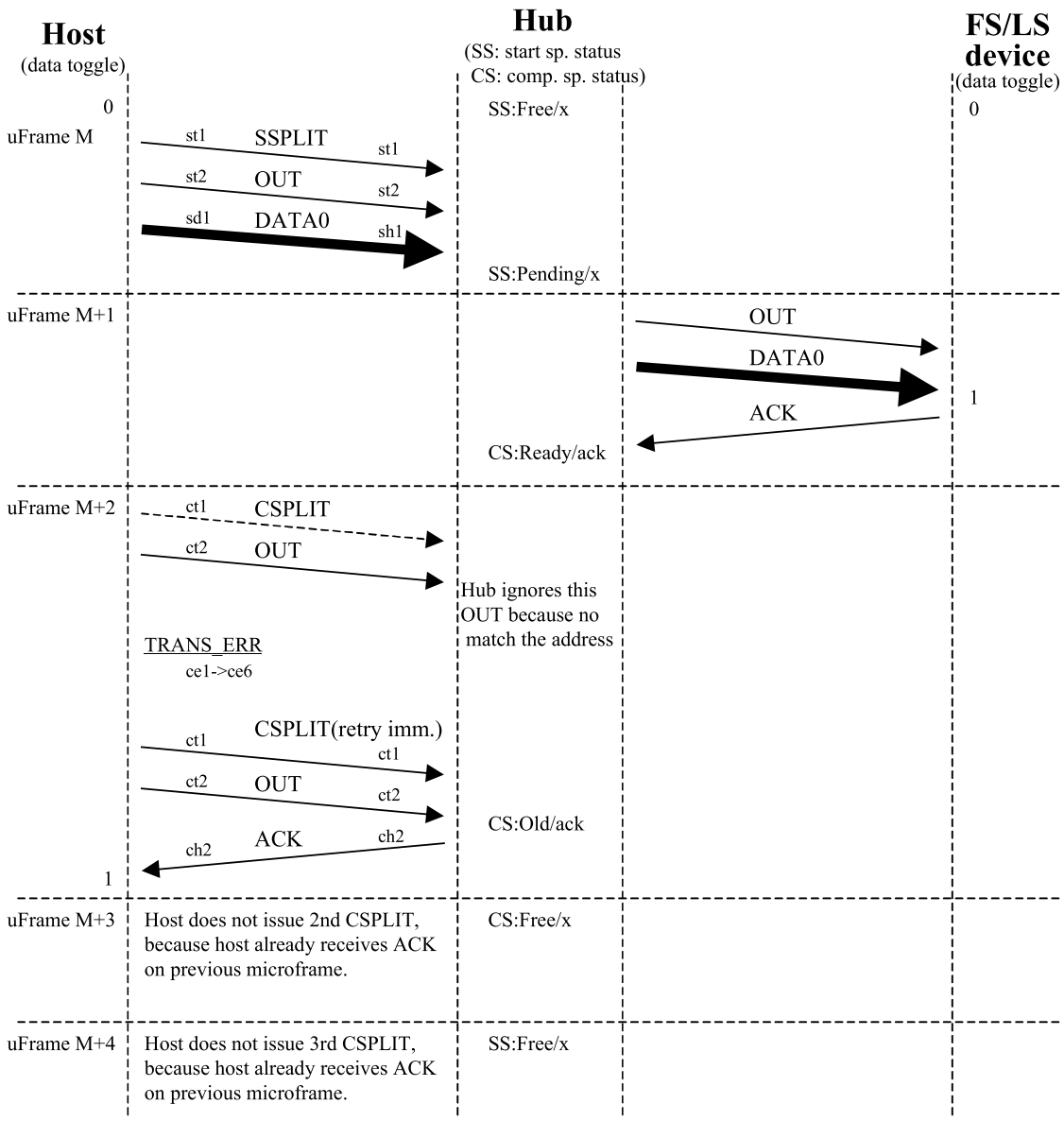



**Figure A-49. Normal HS CSPLIT Smash**

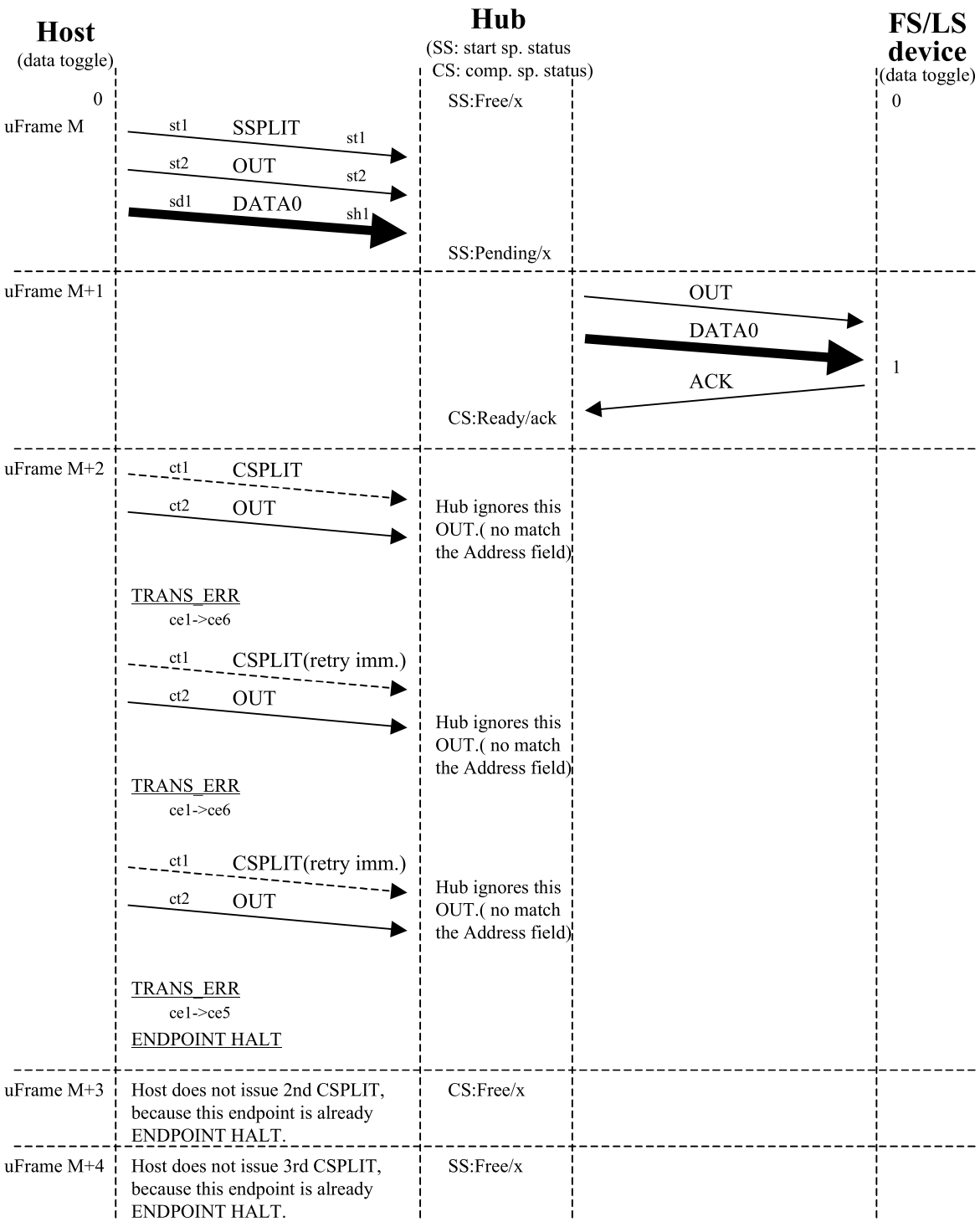



**Figure A-50. Normal HS CSPLIT 3 Strikes Smash**

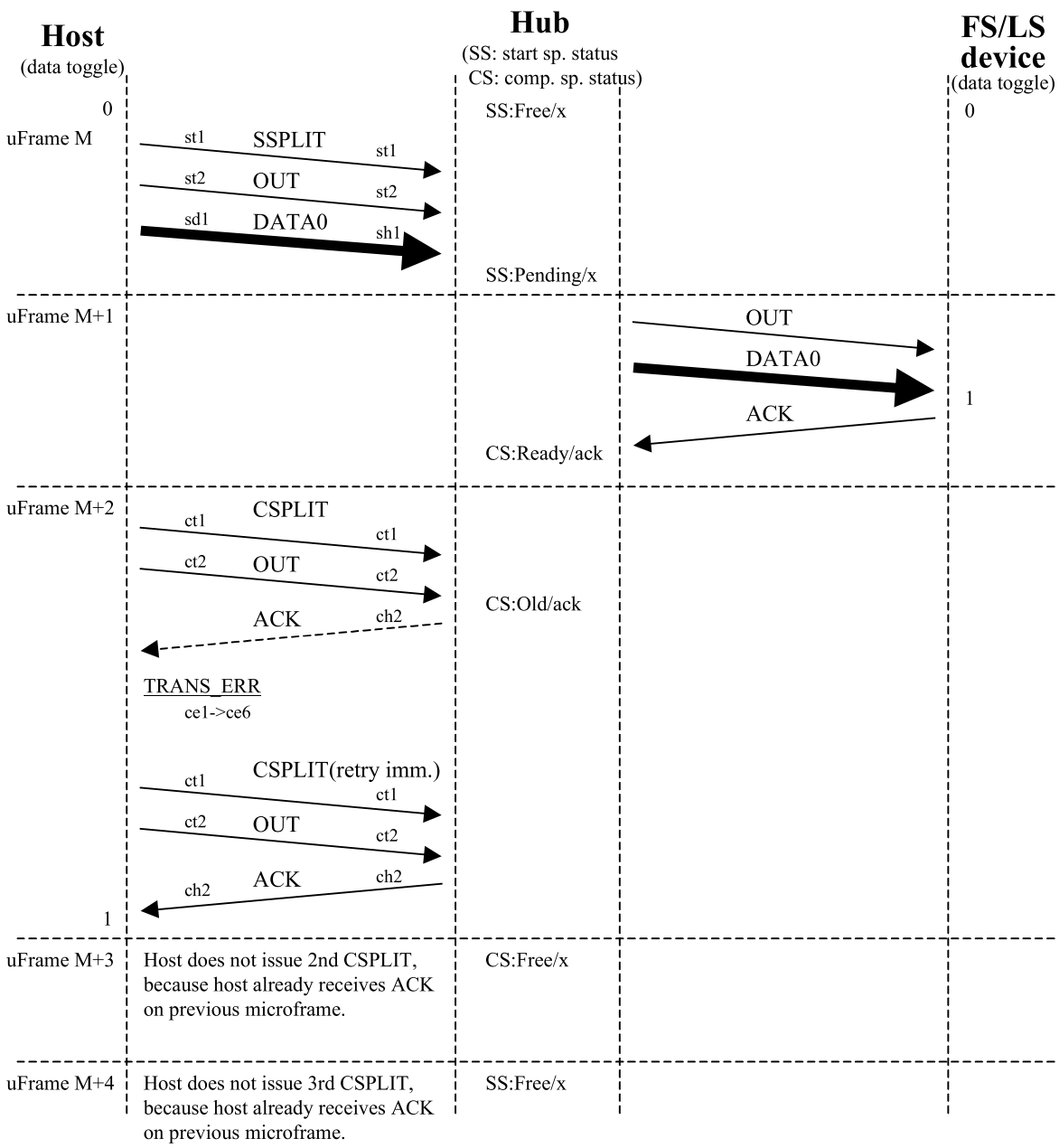



**Figure A-51. Normal HS ACK(C) Smash**

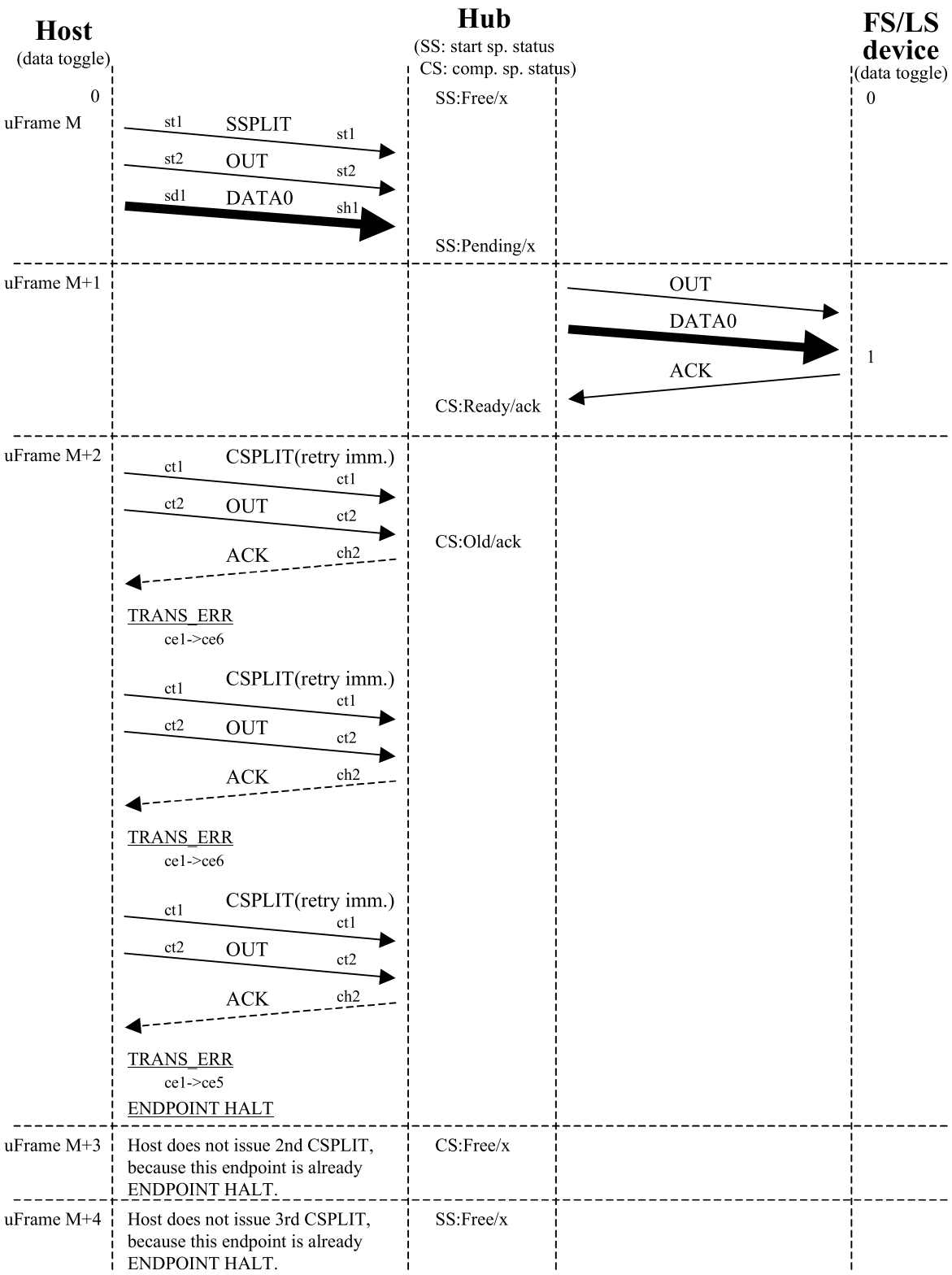



**Figure A-52. Normal HS ACK(C) 3 Strikes Smash**

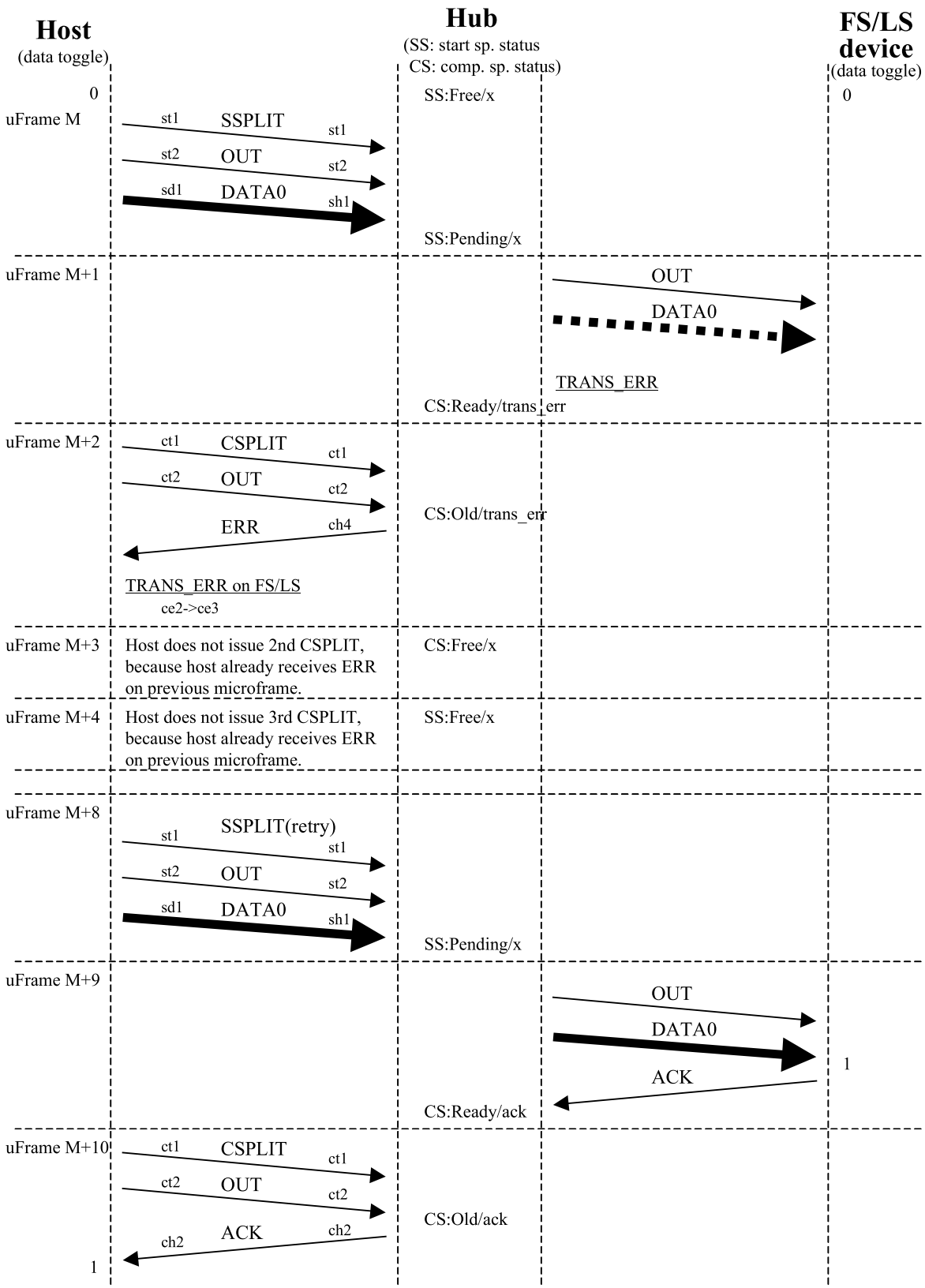



**Figure A-53. Normal FS/LS DATA0/1 Smash**

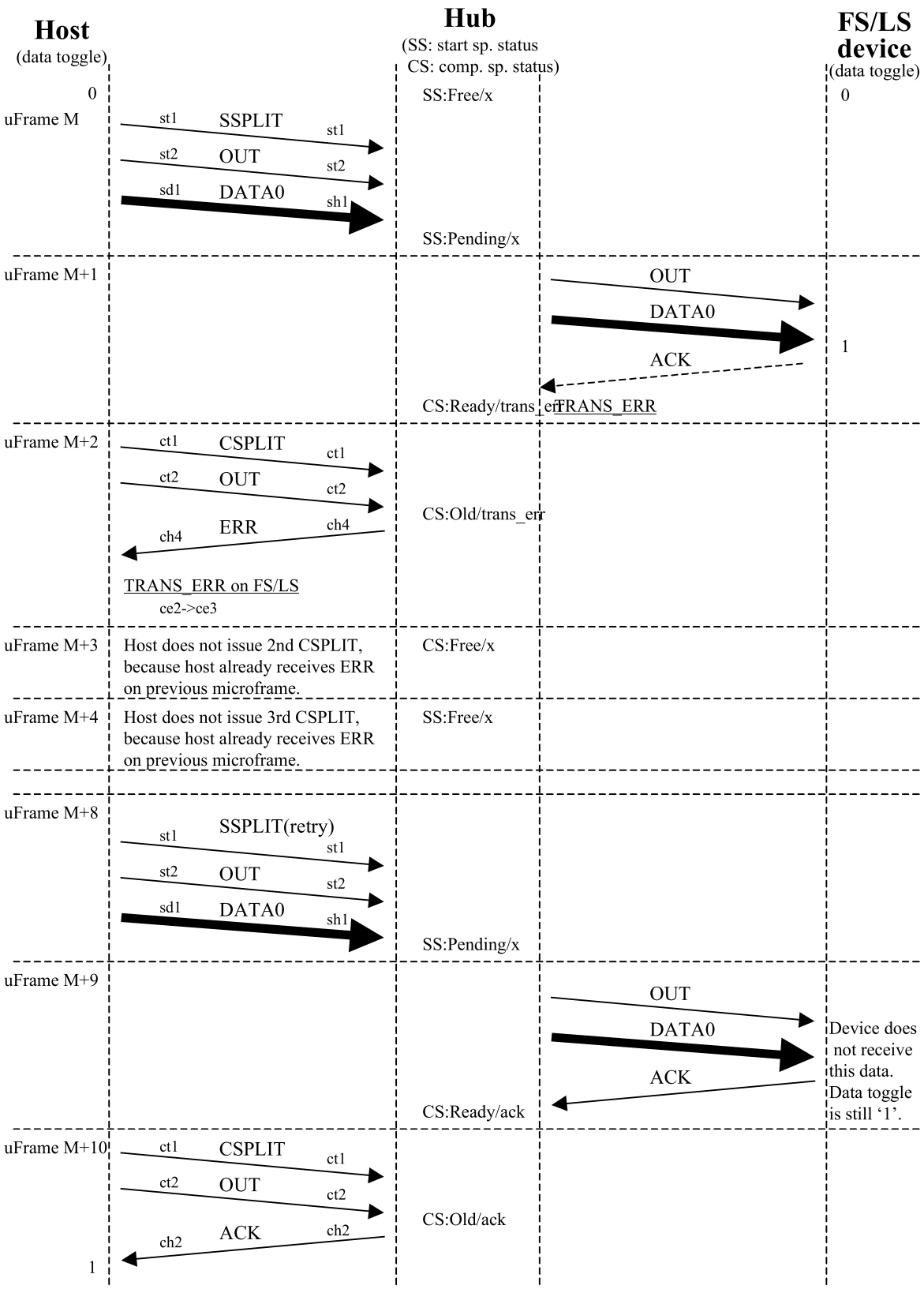



**Figure A-54. Normal FS/LS ACK Smash**

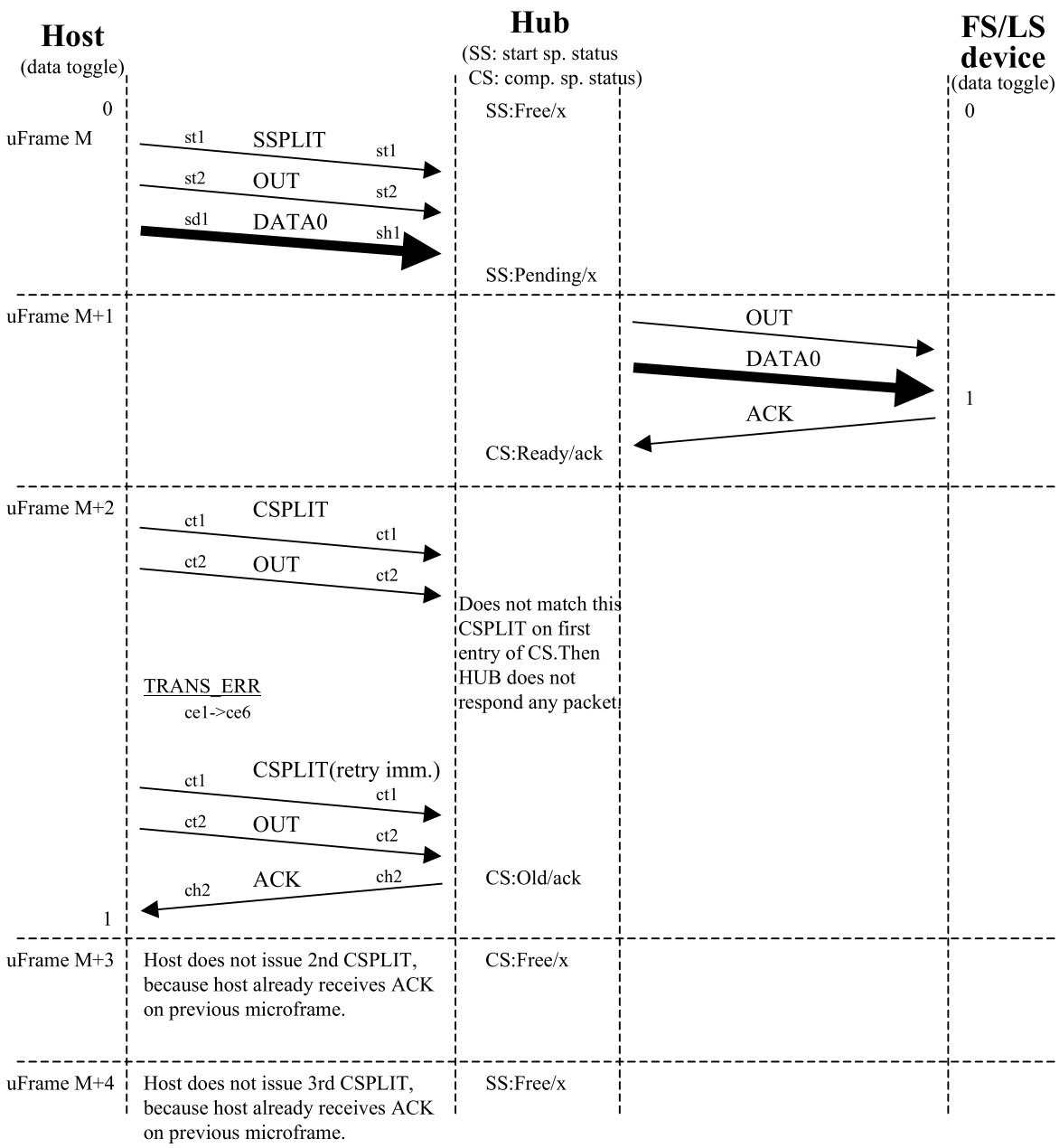



**Figure A-55. Searching No Smash**

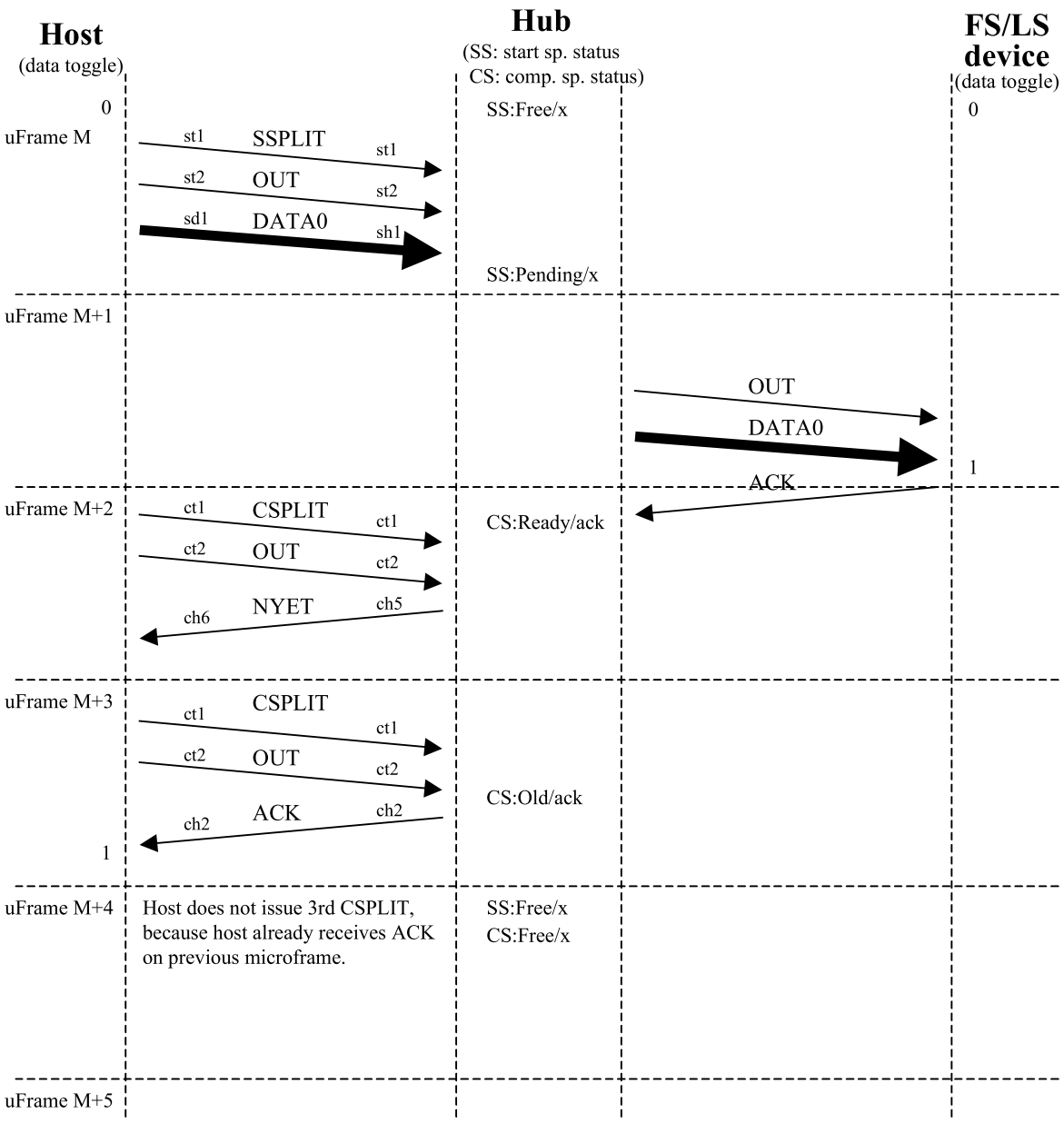



**Figure A-56. CS Earlier No Smash(HS NYET and FS/LS Handshake Packet is Done by M+2)**

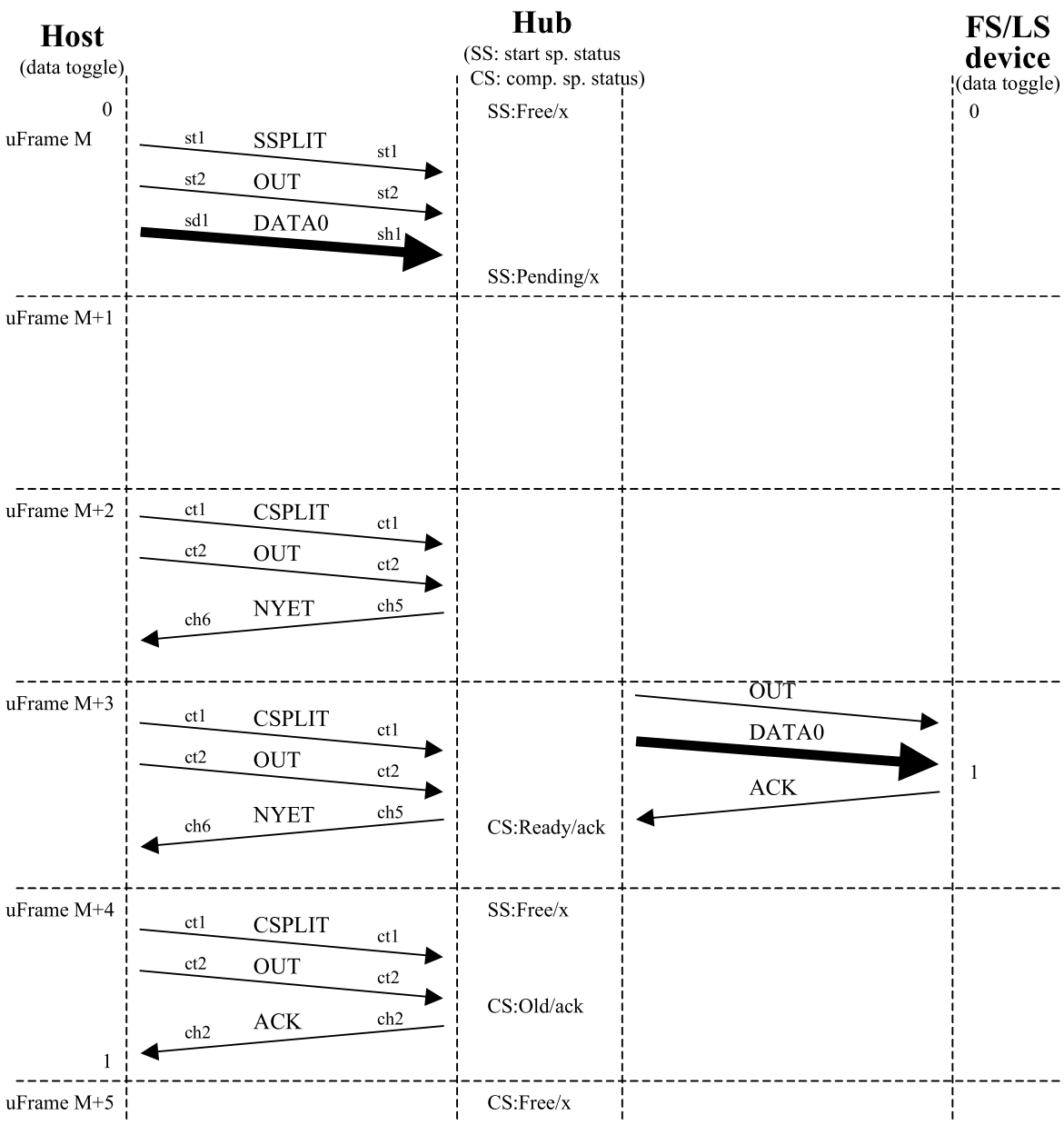



**Figure A-57. CS Earlier No Smash(HS NYET and FS/LS Handshake Packet is Done by M+3)**

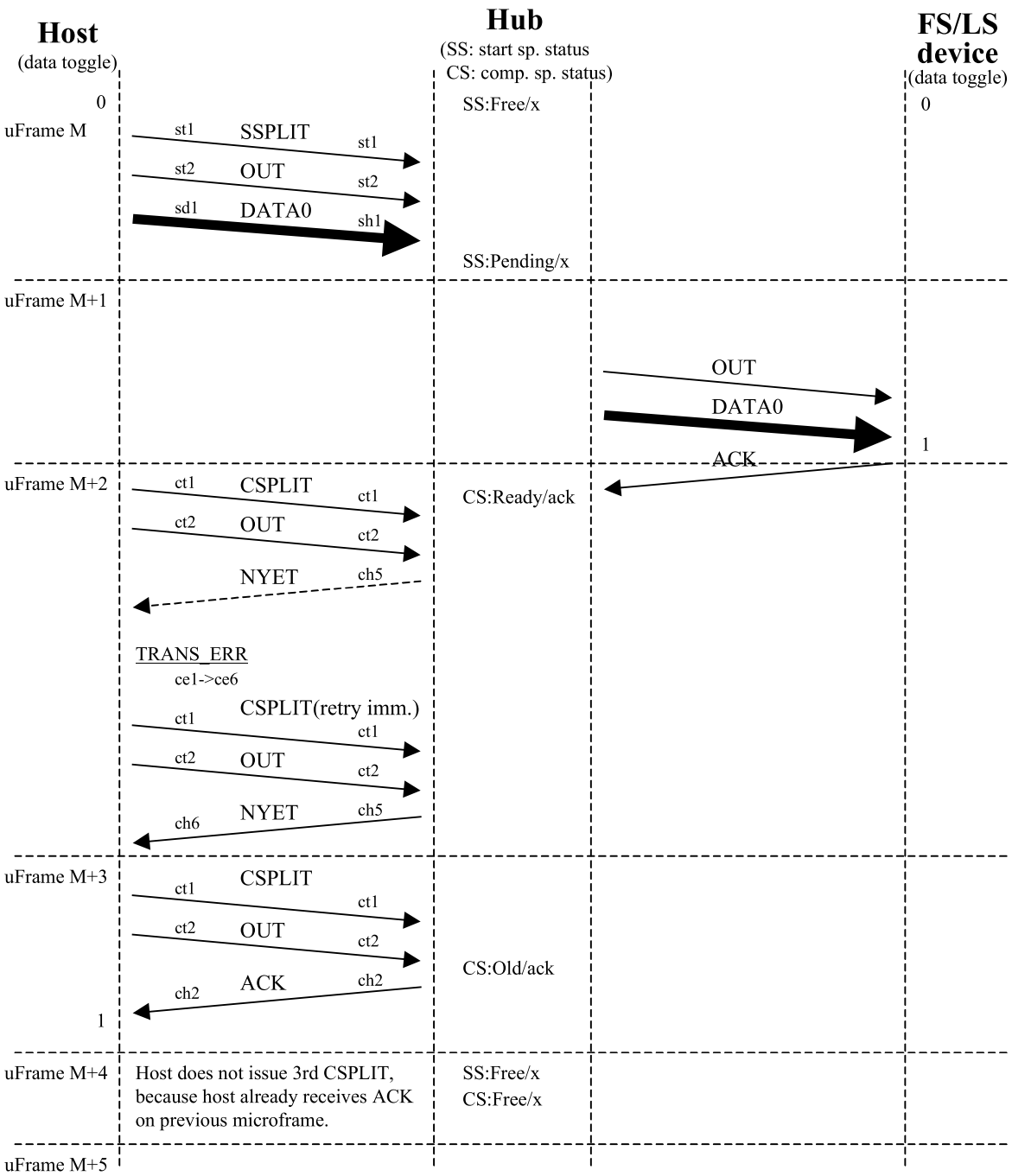



**Figure A-58. CS Earlier HS NYET Smash**

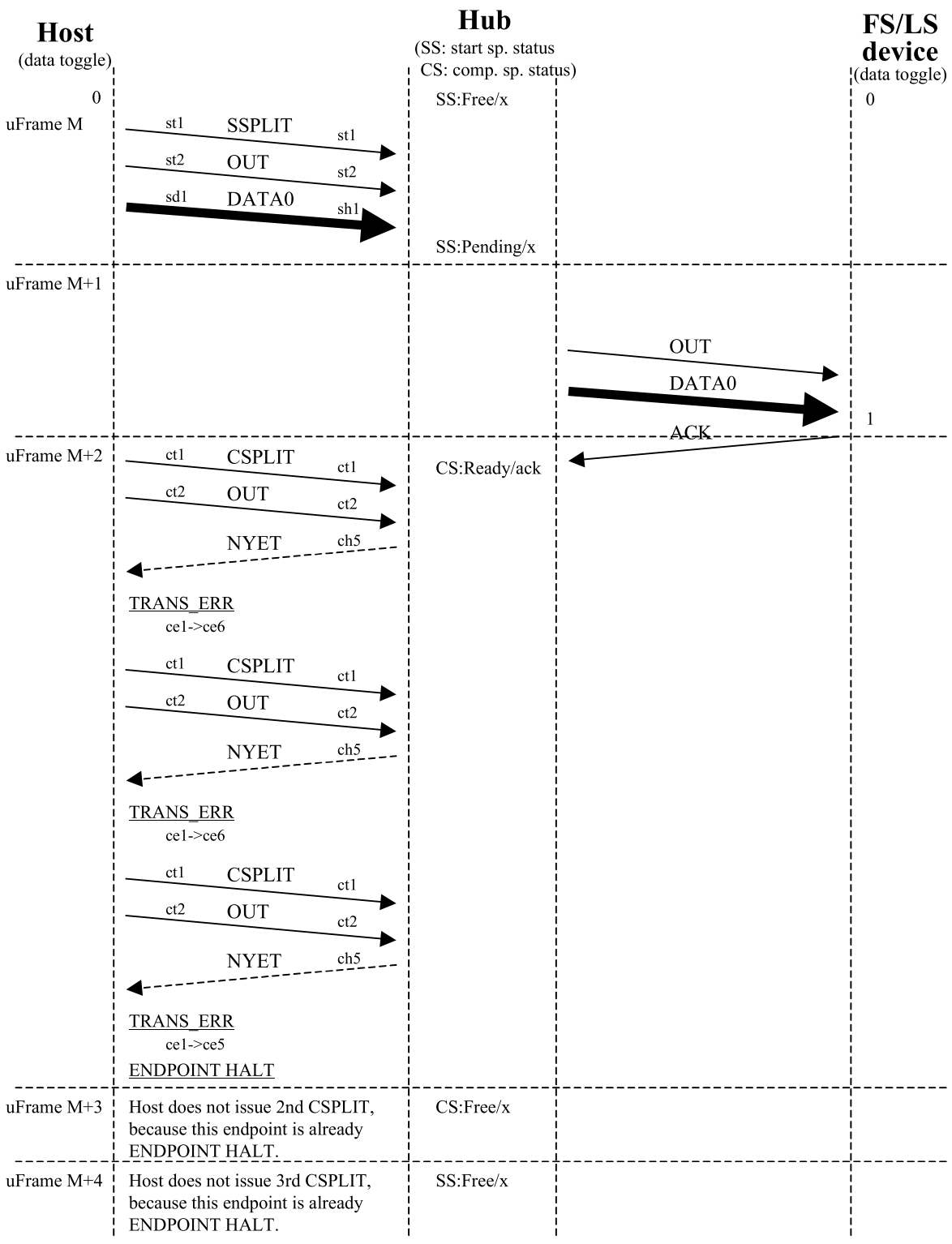



**Figure A-59. CS Earlier HS NYET 3 Strikes Smash**

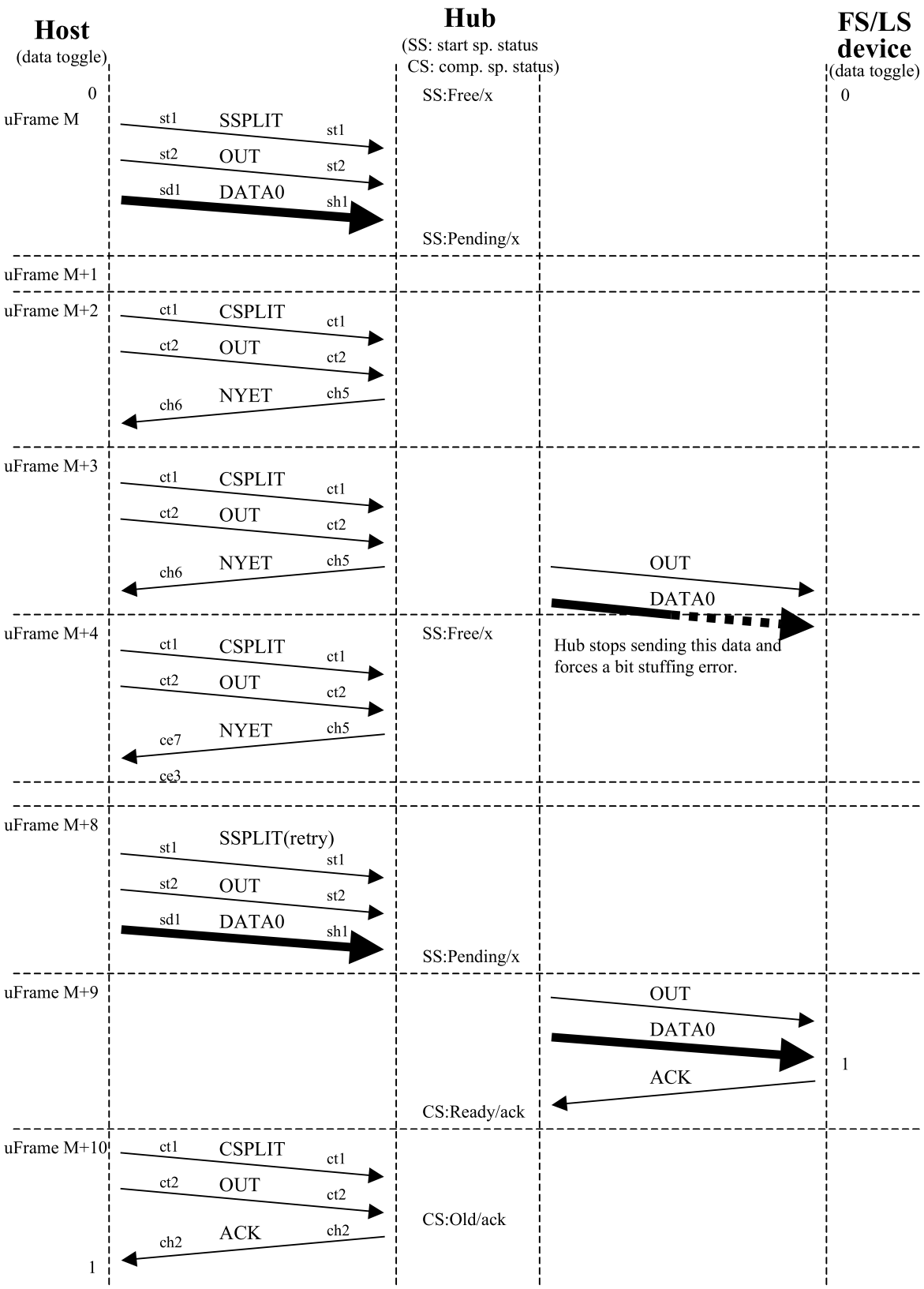



**Figure A-60. Abort and Free Abort(FS/LS Transaction is Continued at End of M+3)**

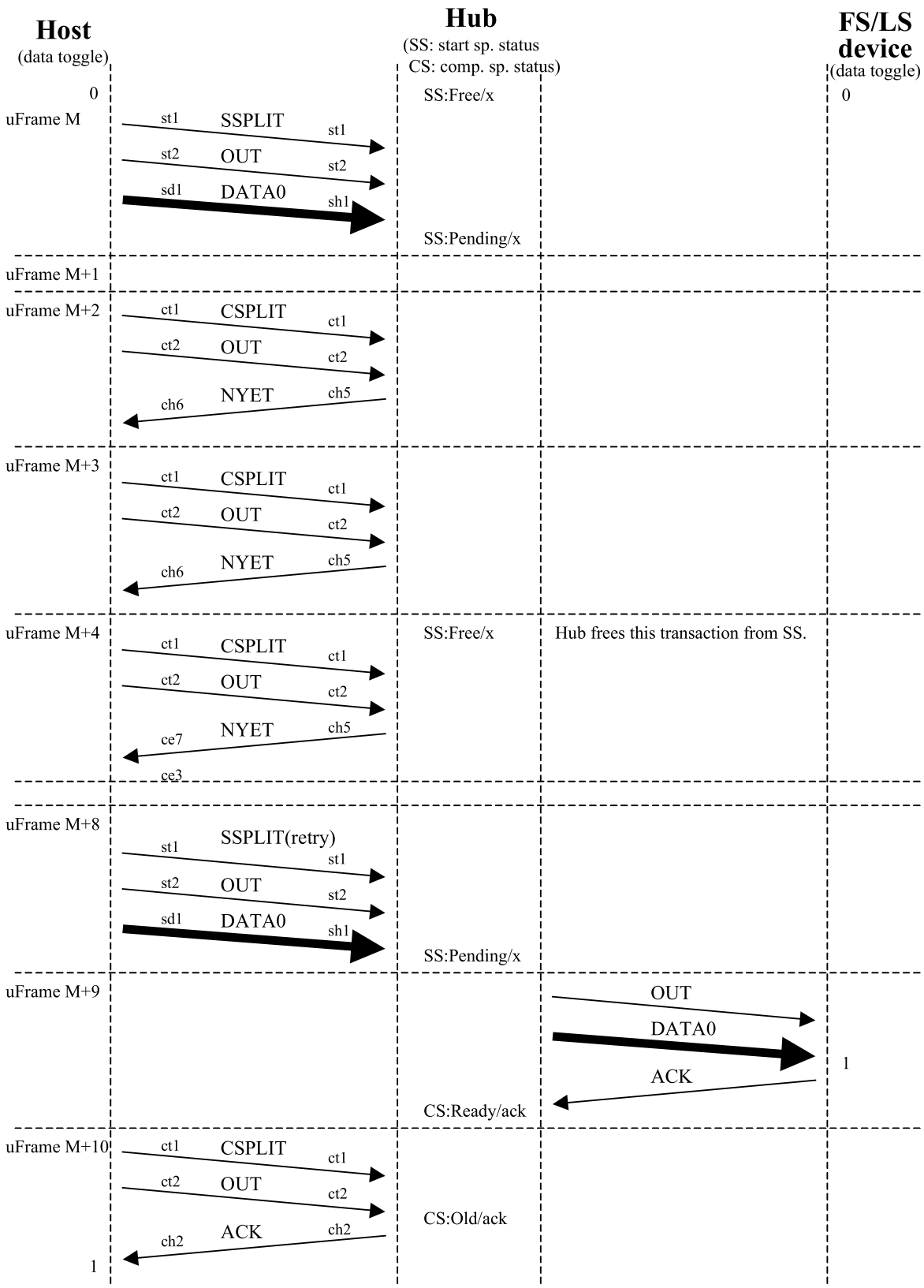



**Figure A-61. Abort and Free Free(FS/LS Transaction is not Started at End of M+3)**

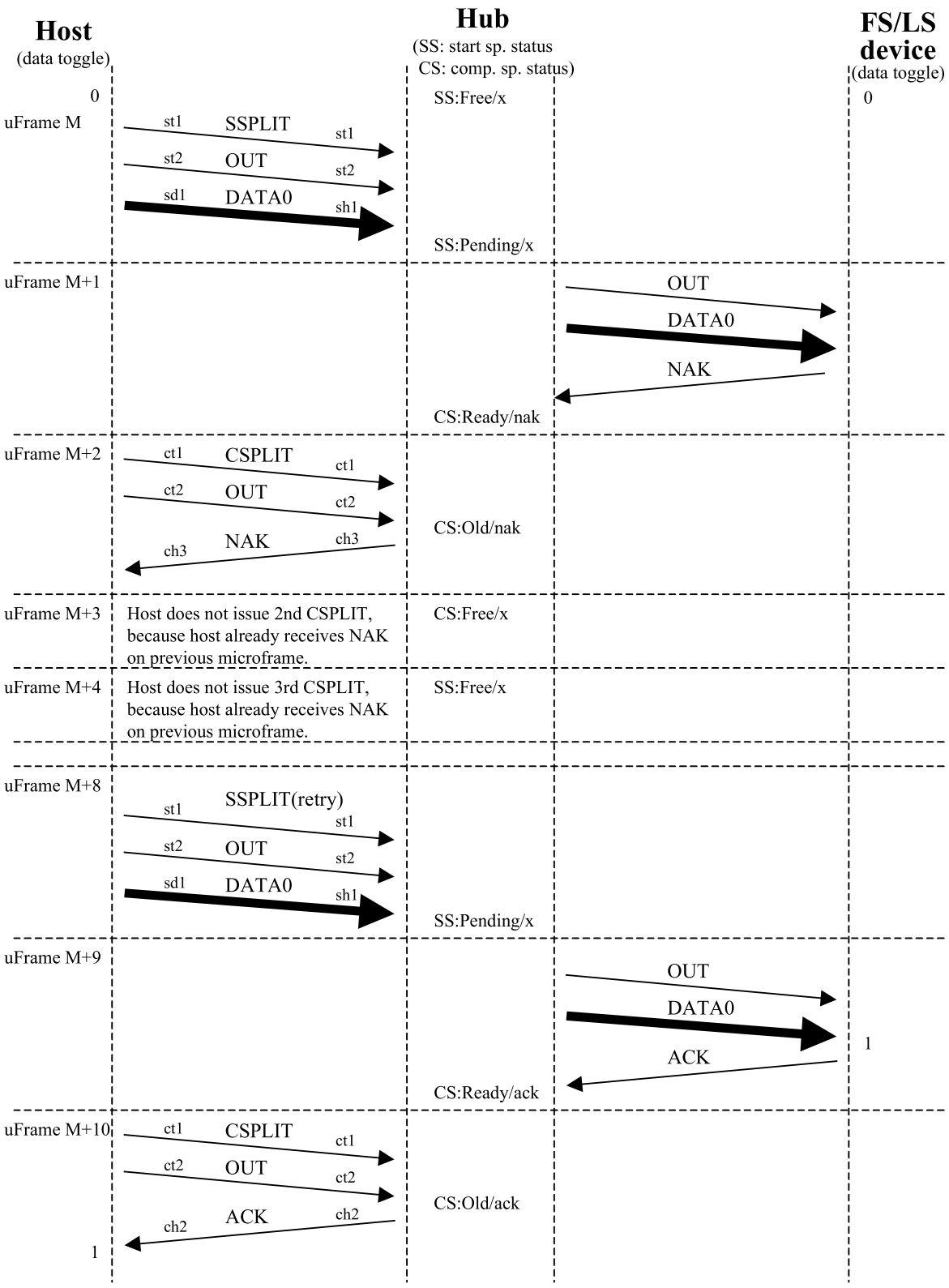



**Figure A-62. Device Busy No Smash(FS/LS NAK)**

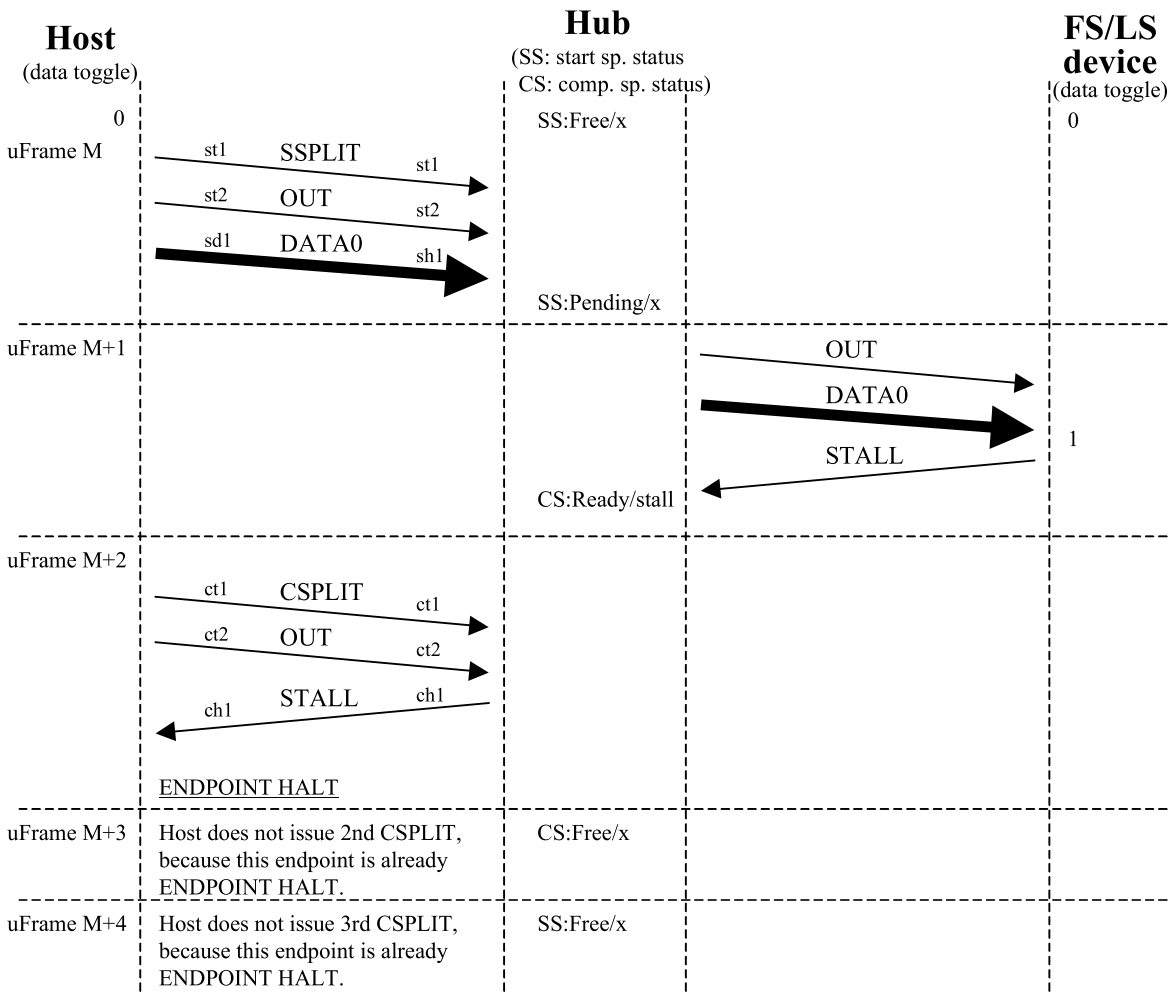



**Figure A-63. Device Stall No Smash(FS/LS STALL)**

## A.4 Interrupt IN Transaction Examples

Legend:

(S): Start Split

(C): Complete Split

Summary of cases for Interrupt OUT transaction

- Normal cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash<br>(FS/LS data packet is on M+1) | Figure A-64 | |
| HS SSPLIT smash | Figure A-65 | |
| HS SSPLIT 3 strikes smash | No figure | |
| HS IN(S) smash | | Figure A-65 |
| HS IN(S) 3 strikes smash | No figure | |
| HS CSPLIT smash | Figure A-66 | |
| HS CSPLIT 3 strikes smash | Figure A-67 | |
| HS IN(C) smash | | Figure A-66 |
| HS IN(C) 3 strikes smash | | Figure A-67 |
| HS DATA0/1 smash | Figure A-68 | |
| HS DATA0/1 3 strikes smash | Figure A-69 | |
| FS/LS IN smash | Figure A-70 | |
| FS/LS IN 3 strikes smash | No figure | |
| FS/LS DATA0/1 smash | Figure A-71 | |
| FS/LS DATA0/1 3 strikes smash | No figure | |
| FS/LS ACK smash | Figure A-72 | |
| FS/LS ACK 3 strikes smash | No figure | |

- Searcing

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash | Figure A-73 | |

- CS(Complete-split transaction) earlier cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash (HS MDATA and FS/LS transaction is on M+1 and M+2) | Figure A-74 | |
| No smash (HS NYET and FS/LS transaction is on M+2) | Figure A-75 | |
| No smash (HS NYET and MDATA and FS/LS transaction is on M+2 and M+3) | Figure A-76 | |
| No smash (HS NYET and FS/LS transaction is on M+3) | Figure A-77 | |
| HS NYET smash | Figure A-78 | |
| HS NYET 3 strikes smash | Figure A-79 | |

- Abort and Free cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash and abort (HS NYETand FS/LS transaction is continued at end of M+3) | Figure A-80 | |
| No smash and free(HS NYETand FS/LS transaction is not started at end of M+3) | Figure A-81 | |

- FS/LS transaction error cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| HS ERR smash | | Figure A-68 |
| HS ERR 3 strikes smash | | Figure A-69 |

- Device busy cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash(HS NAK(C)) | Figure A-82 | |
| HS NAK(C) smash | | Figure A-68 |
| HS NAK(C) 3 strikes smash | | Figure A-69 |
| FS/LS NAK smash | | Figure A-71 |
| FS/LS NAK 3 strikes smash | No figure | |

- Device stall cases

| Case | Reference Figure | Similar Figure |
|---|---|---|
| No smash | Figure A-83 | |
| HS STALL(C) smash | | Figure A-68 |
| HS STALL(C) 3 strikes smash | | Figure A-69 |
| FS/LS STALL smash | | Figure A-71 |
| FS/LS STALL 3 strikes smash | No figure | |

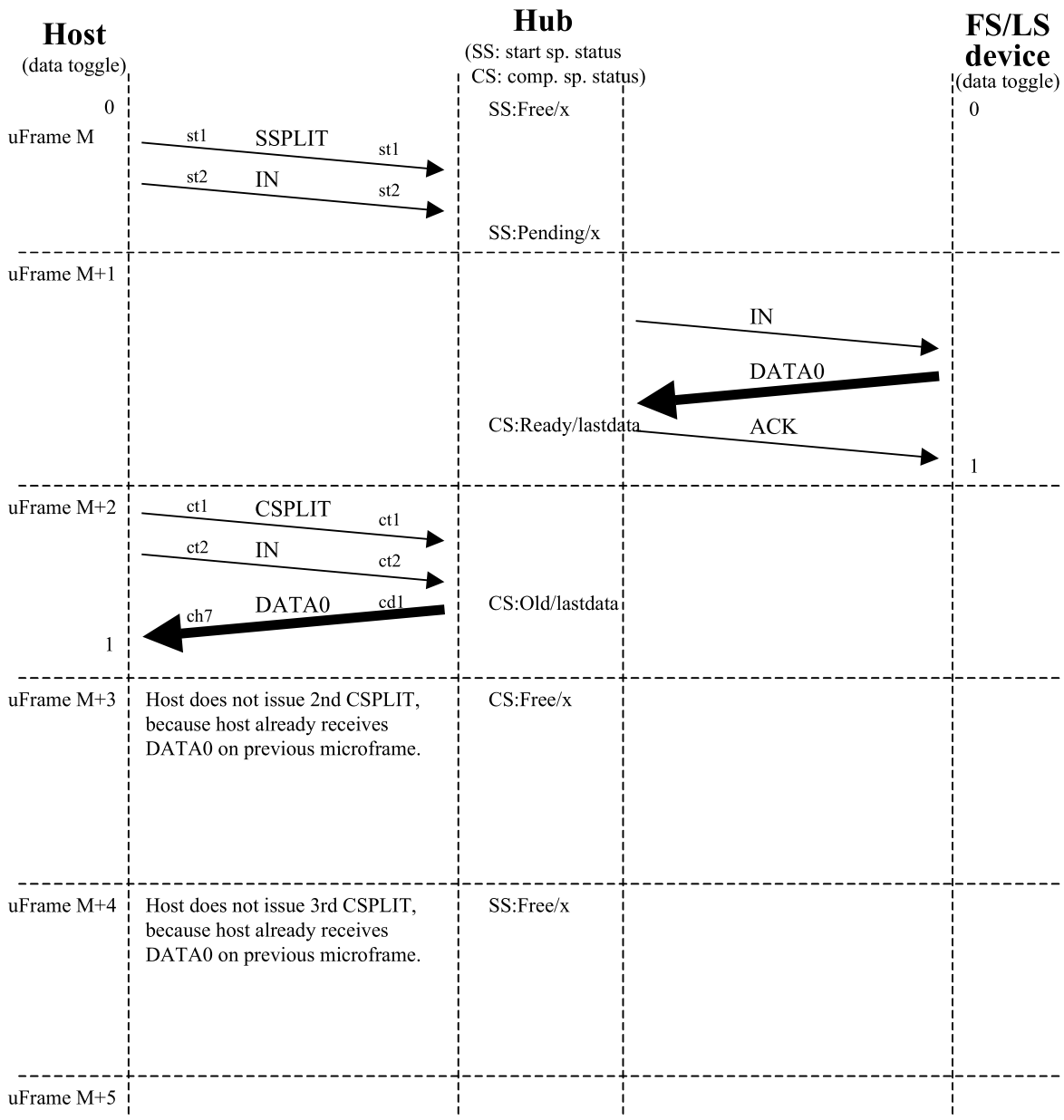



**Figure A-64. Normal No Smash(FS/LS Data Packet is on M+1)**

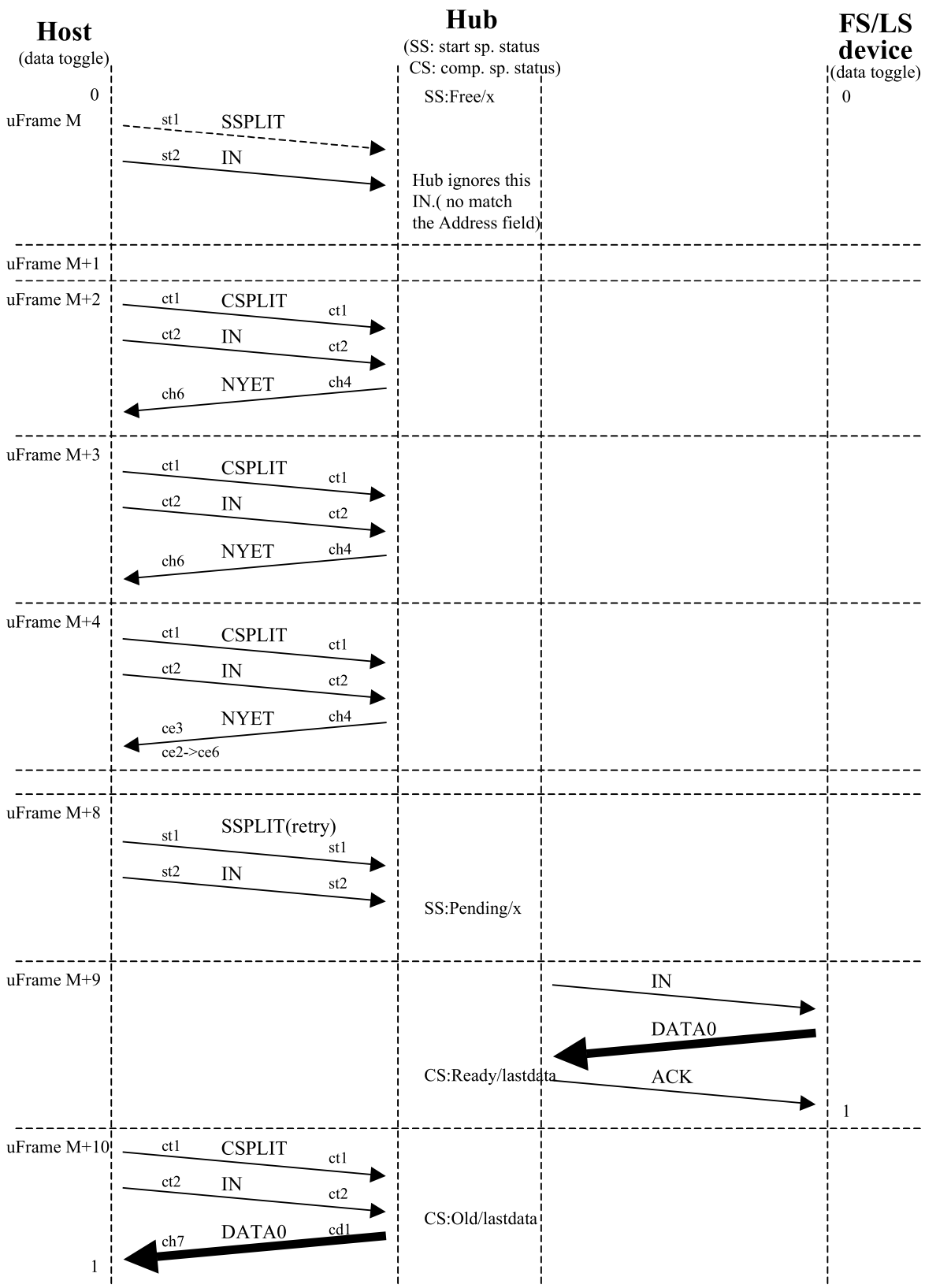



**Figure A-65. Normal HS SSPLIT Smash**

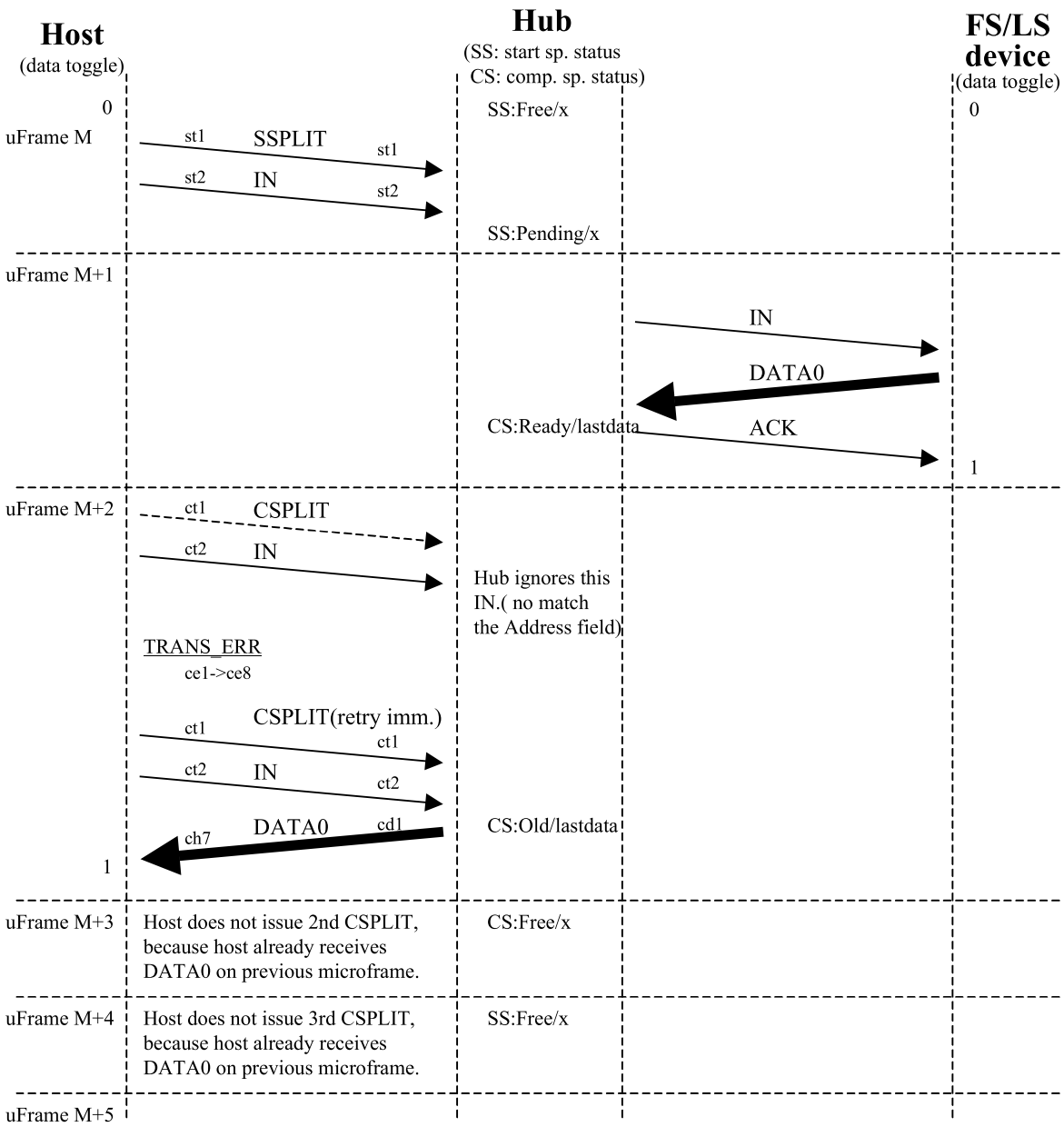



**Figure A-66. Normal HS CSPLIT Smash**

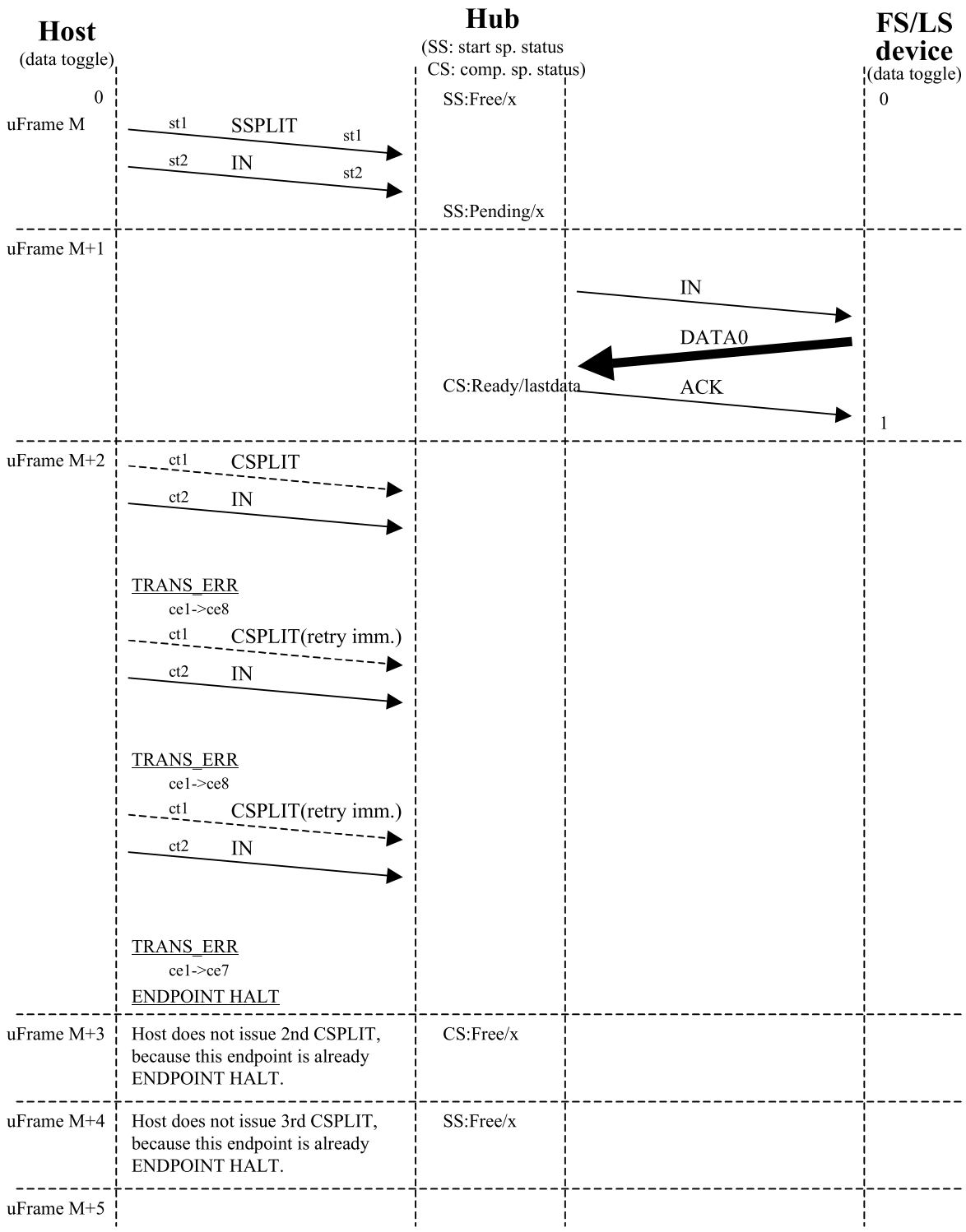



**Figure A-67. Normal HS CSPLIT 3 Strikes Smash**

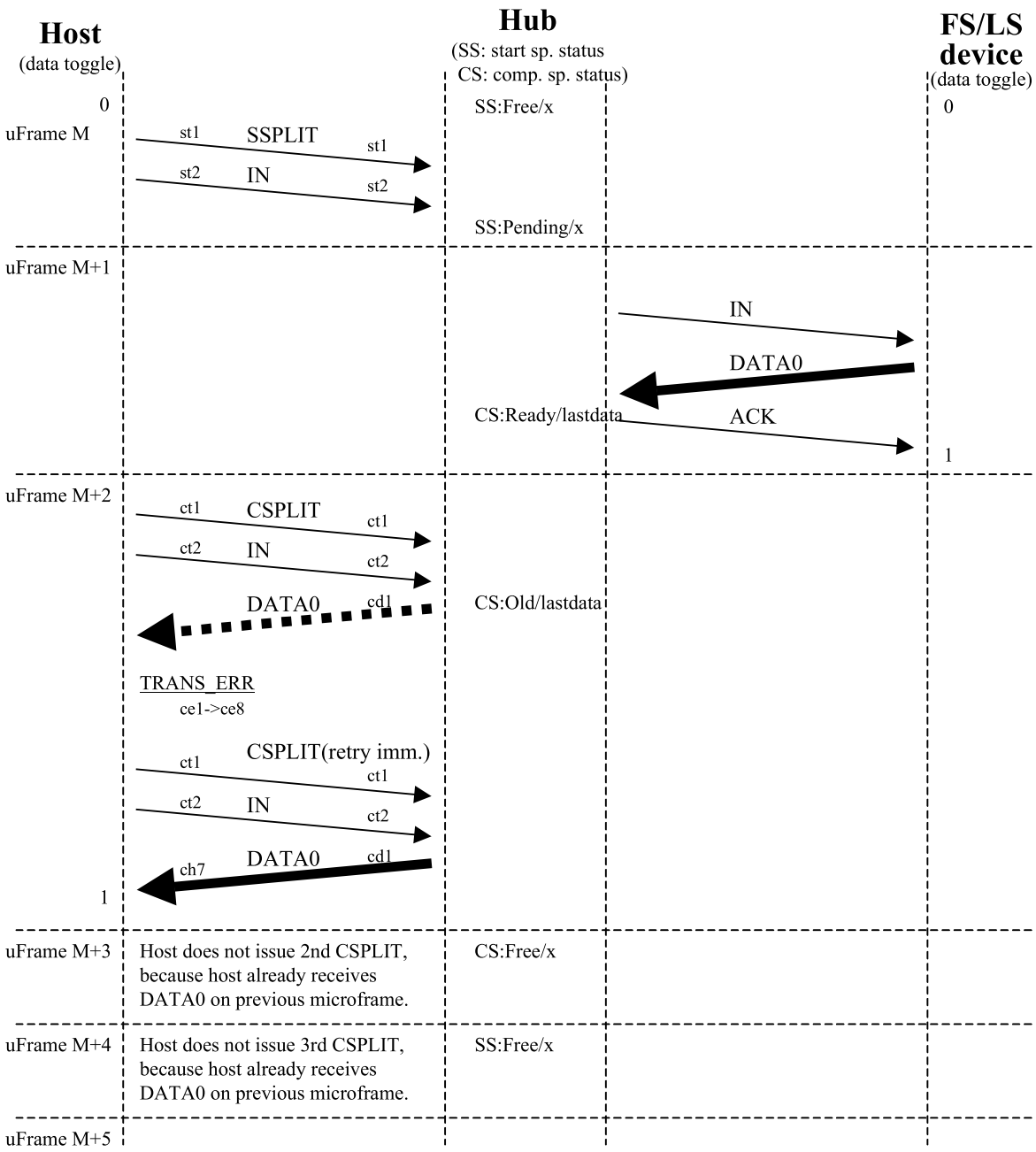



**Figure A-68. Normal HS DATA0/1 Smash**

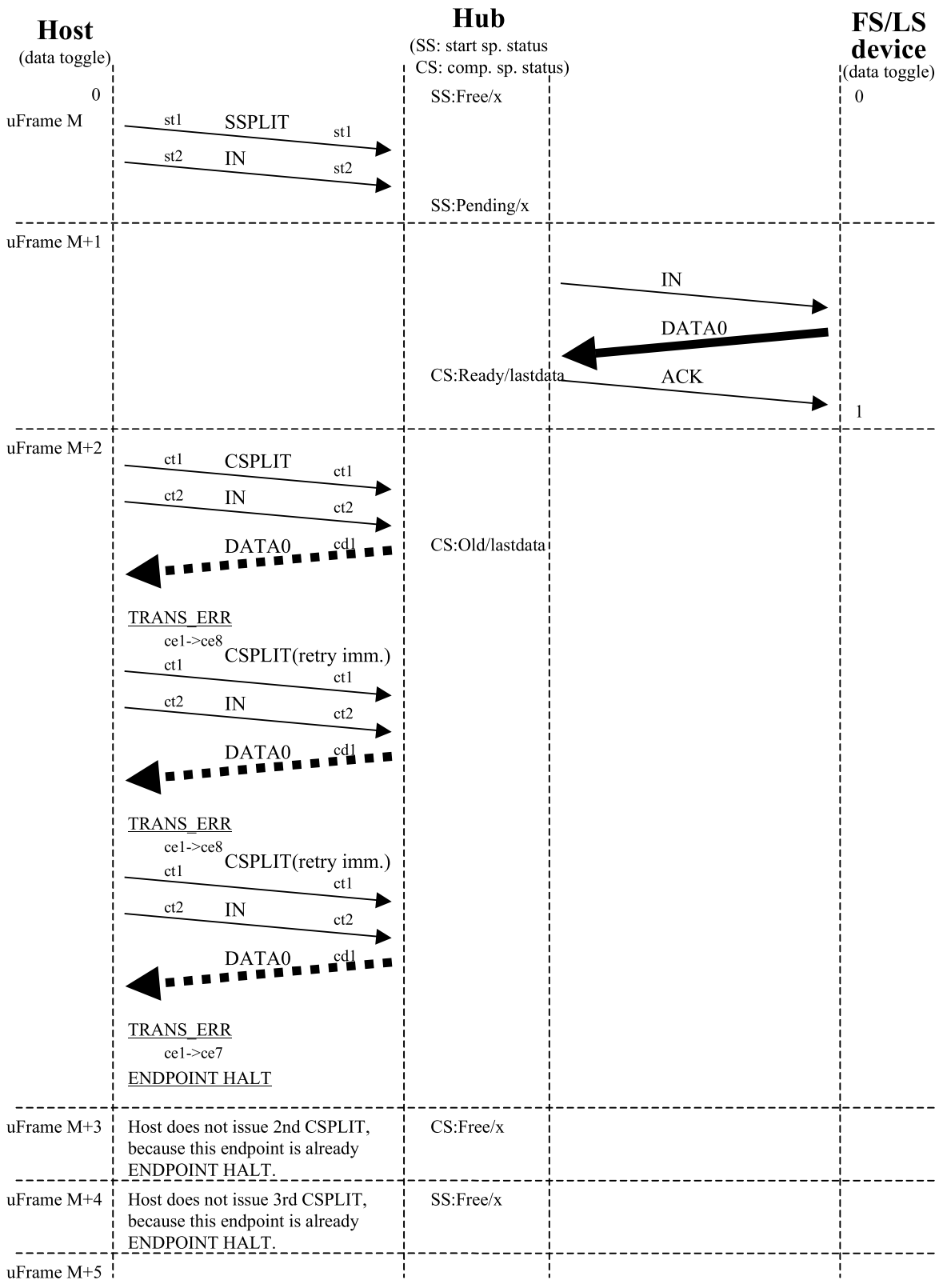



**Figure A-69. Normal HS DATA0/1 3 Strikes Smash**

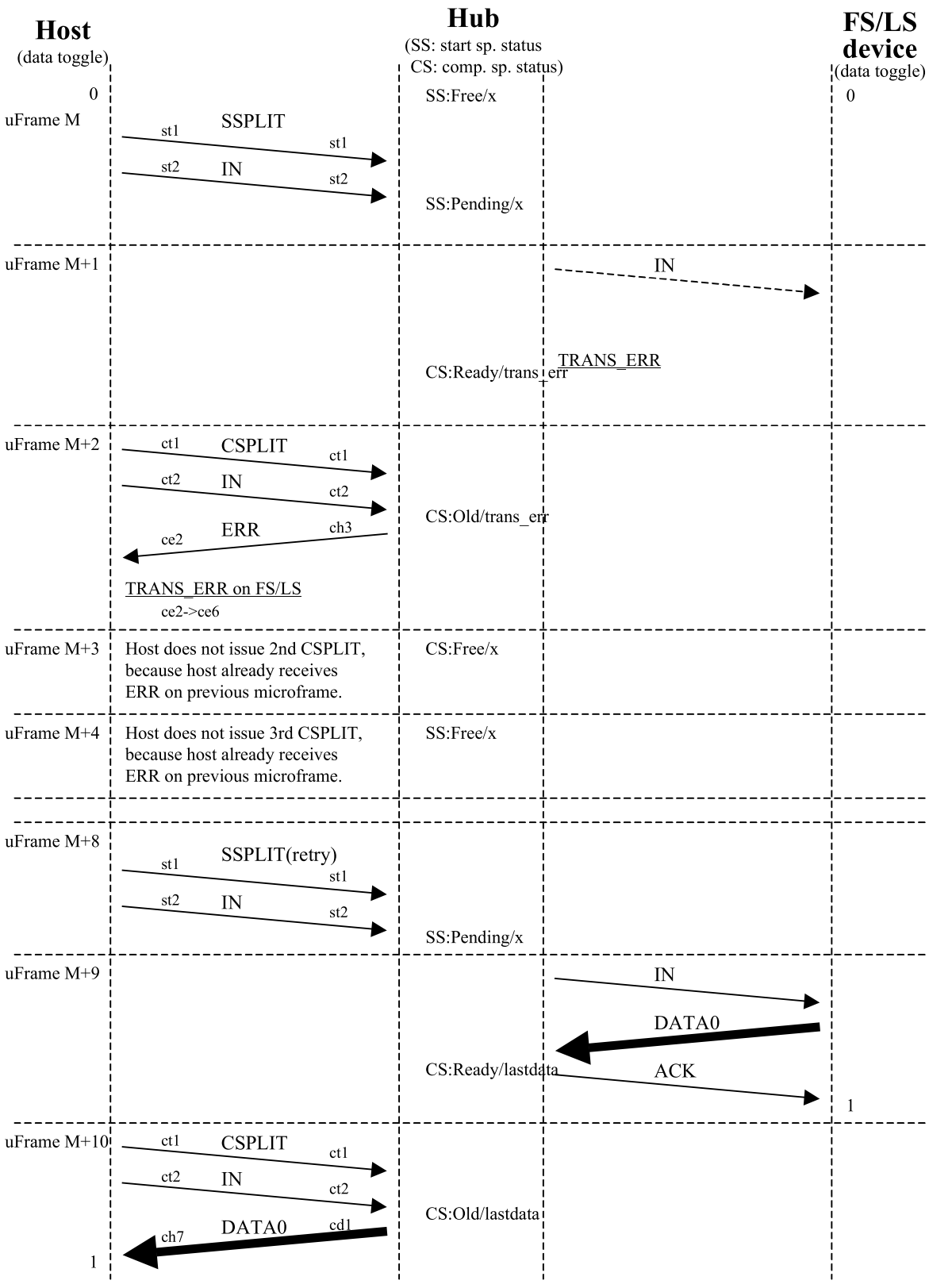



**Figure A-70. Normal FS/LS IN Smash**

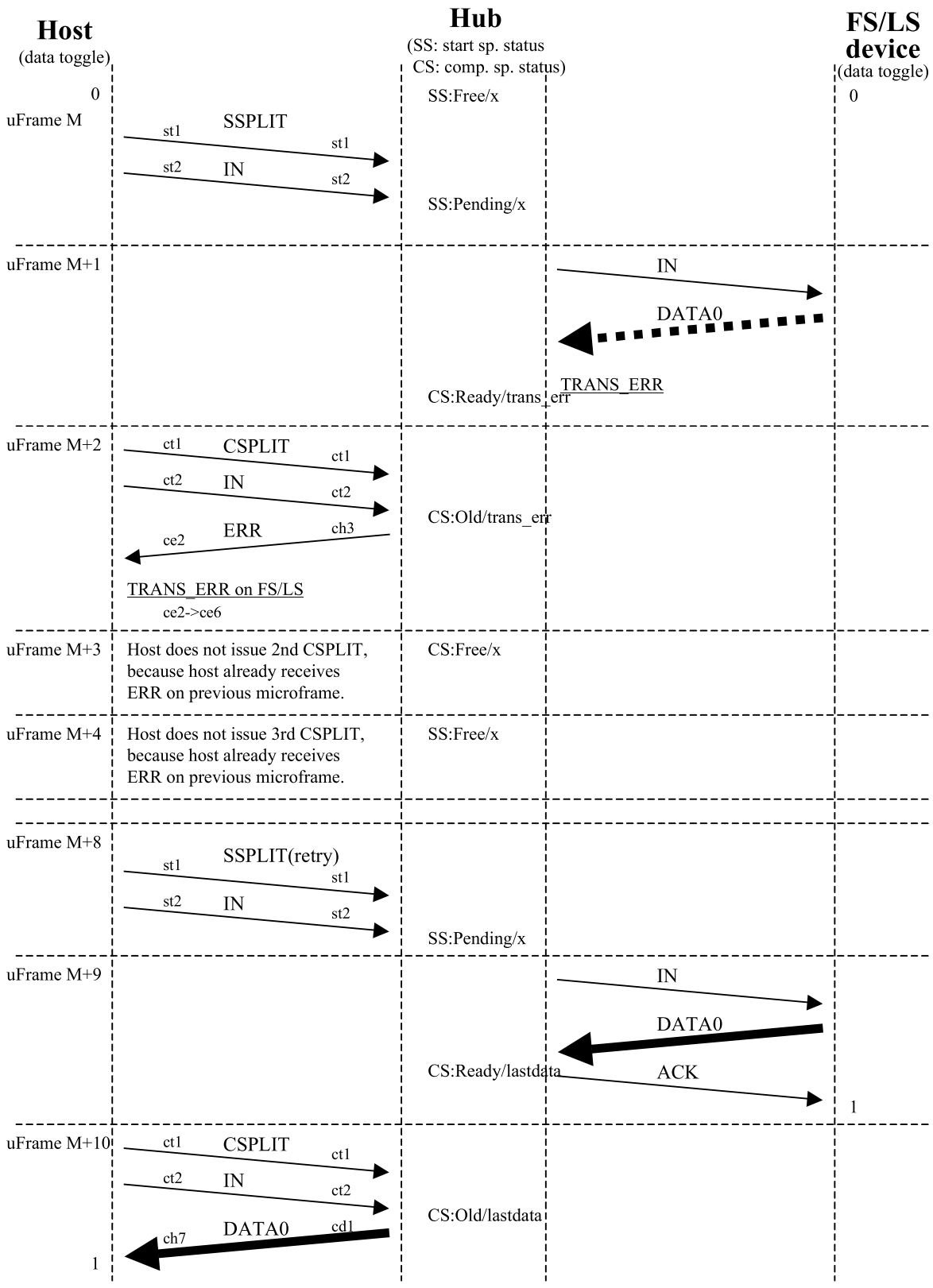



**Figure A-71. Normal FS/LS DATA0/1 Smash**

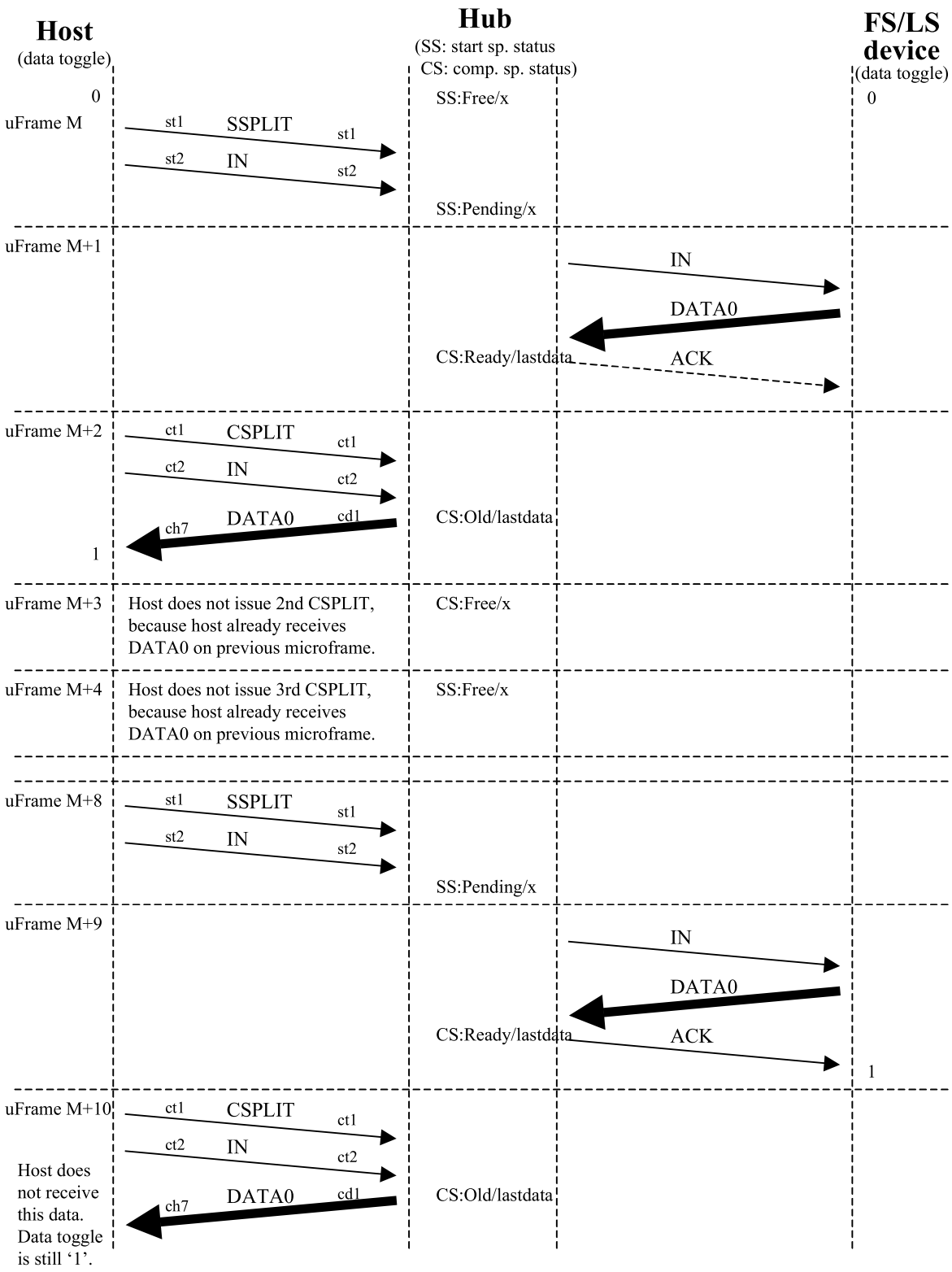



**Figure A-72. Normal FS/LS ACK Smash**

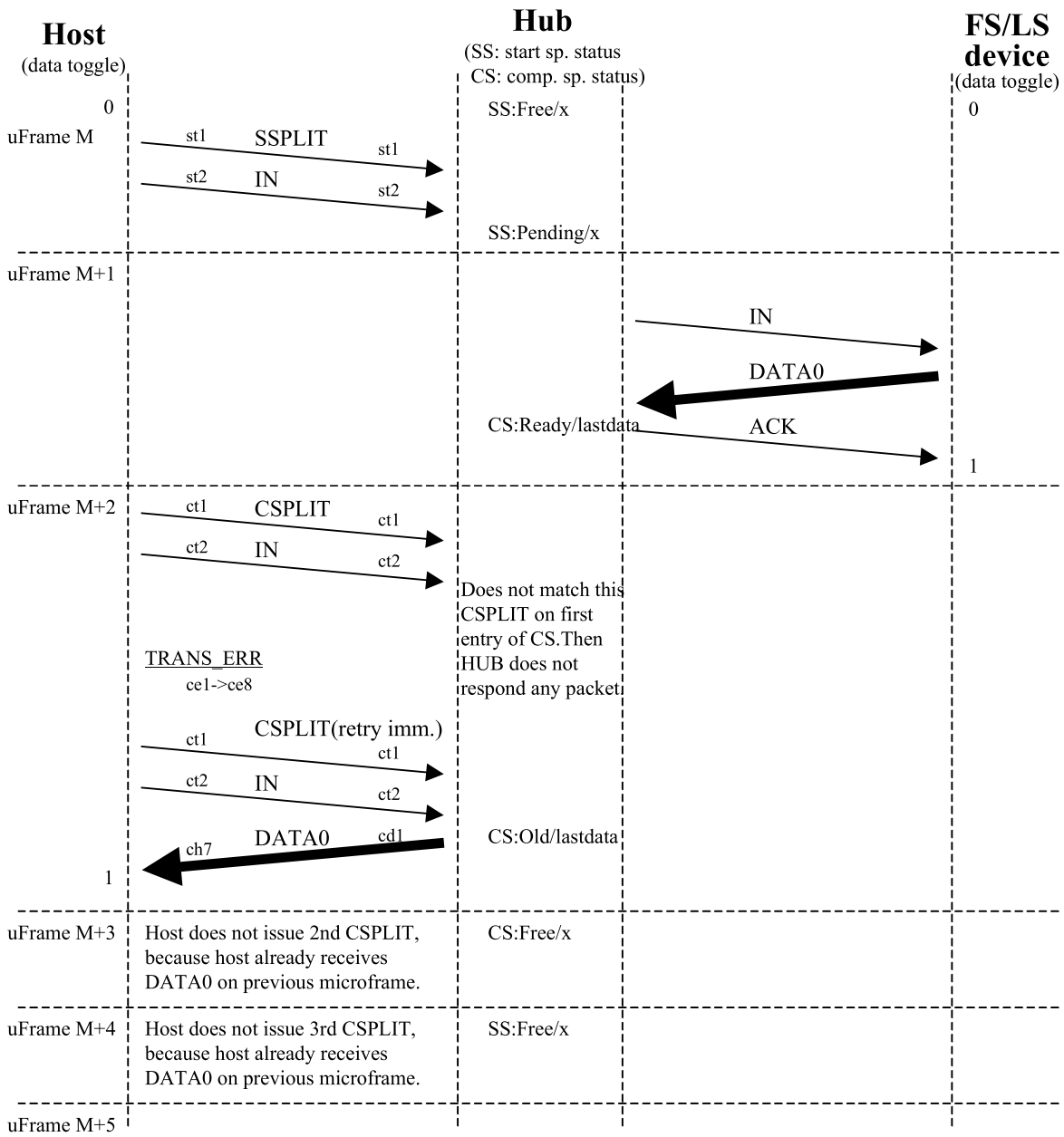



**Figure A-73. Searching No Smash**

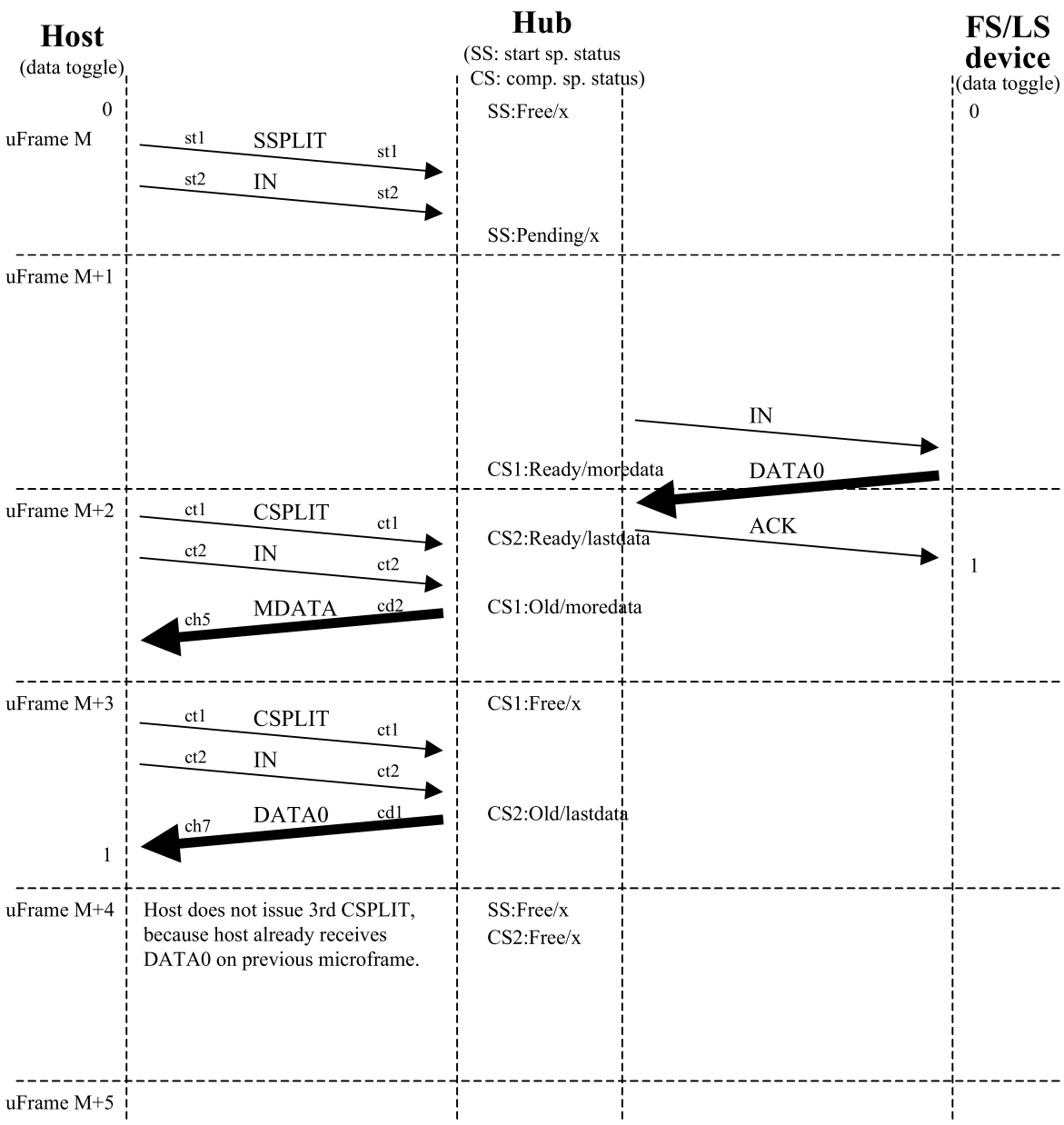



**Figure A-74. CS Earlier No Smash(HS MDATA and FS/LS Data Packet is on M+1 and M+2)**

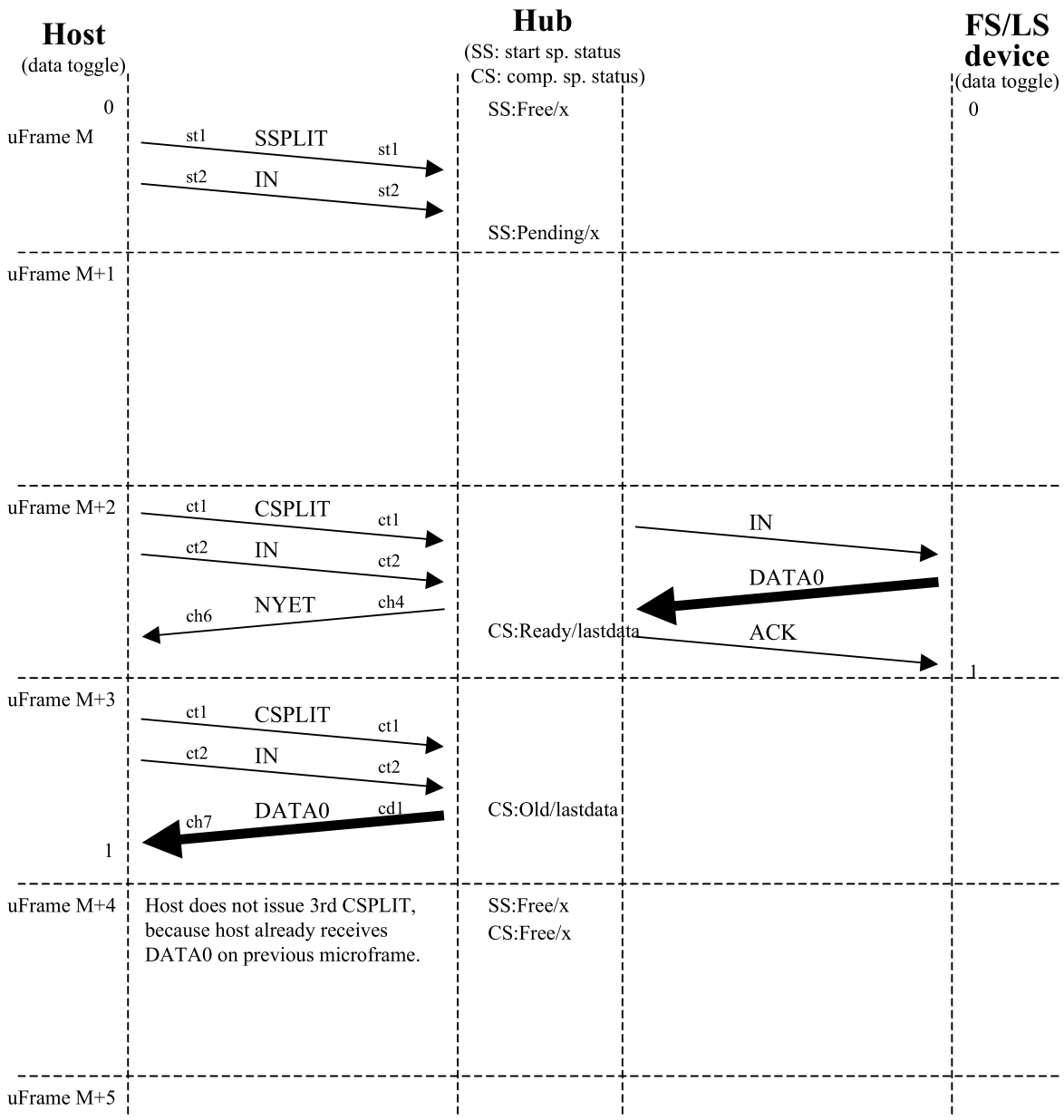



**Figure A-75. CS Earlier No Smash(HS NYET and FS/LS Data Packet is on M+2)**

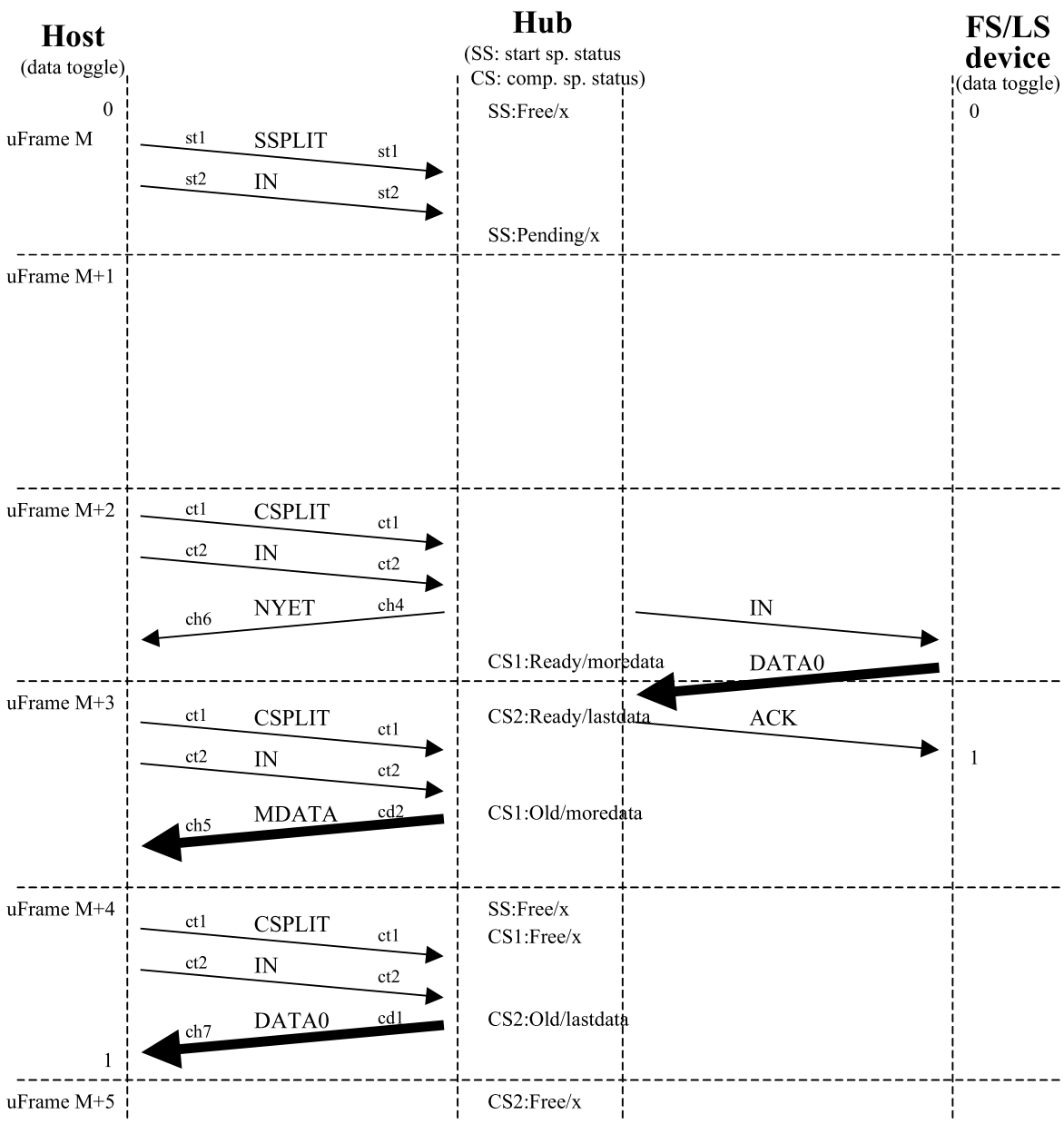



**Figure A-76. CS Earlier No Smash(HS NYET and MDATA and FS/LS Data Packet is on M+2 and M+3)**

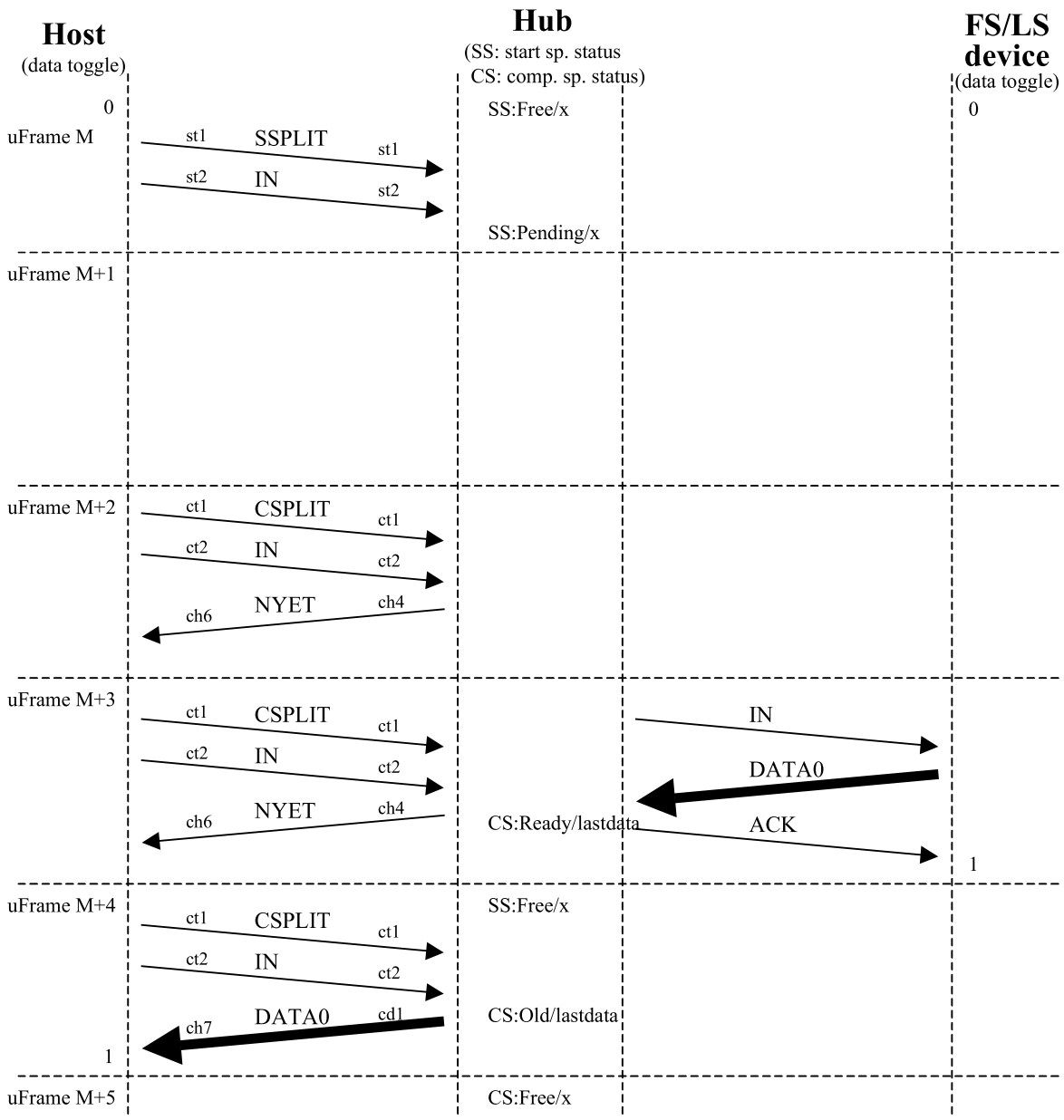



**Figure A-77. CS Earlier No Smash(HS NYET and FS/LS Data Packet is on M+3)**

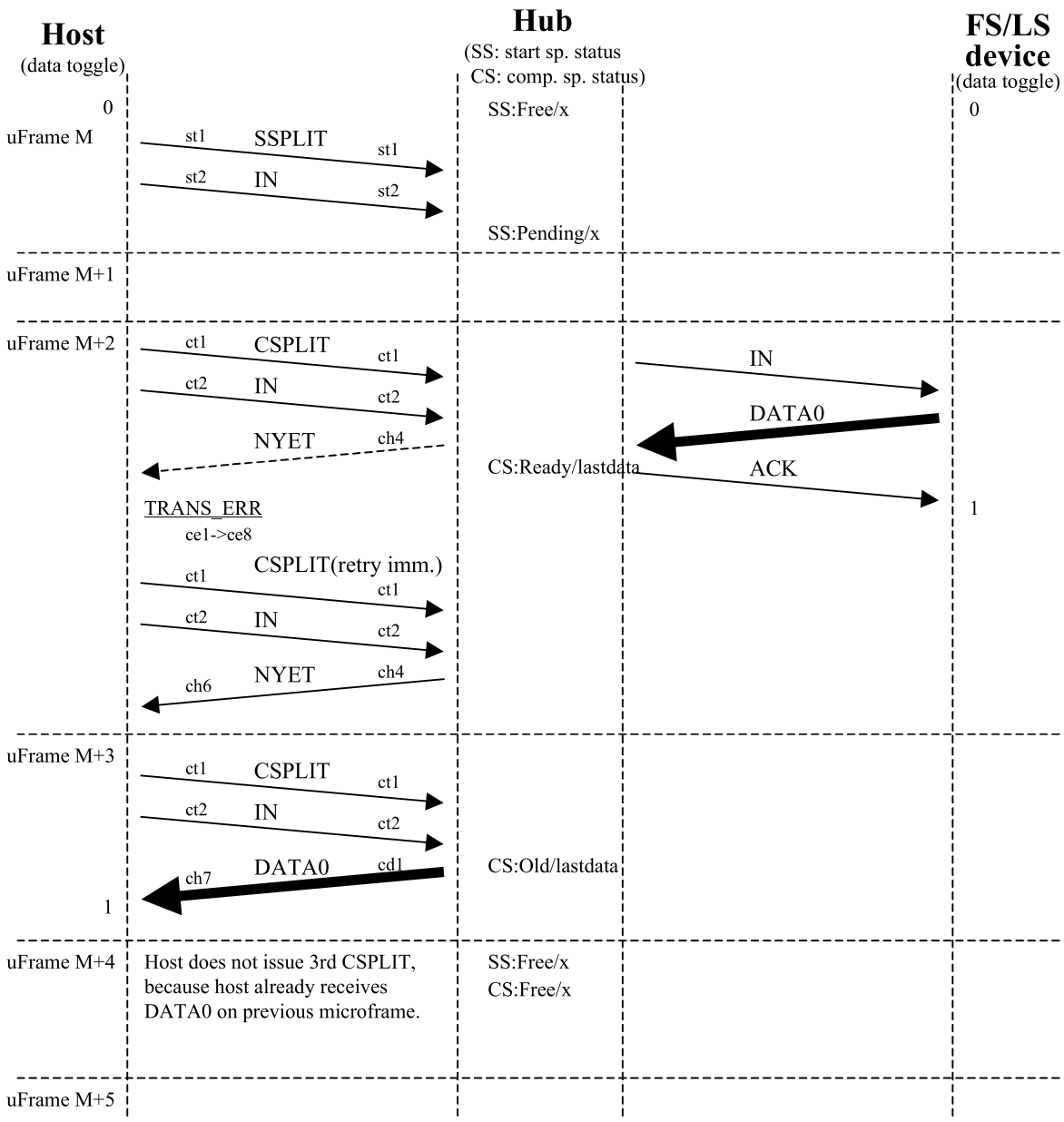



**Figure A-78. CS Earlier HS NYET Smash**

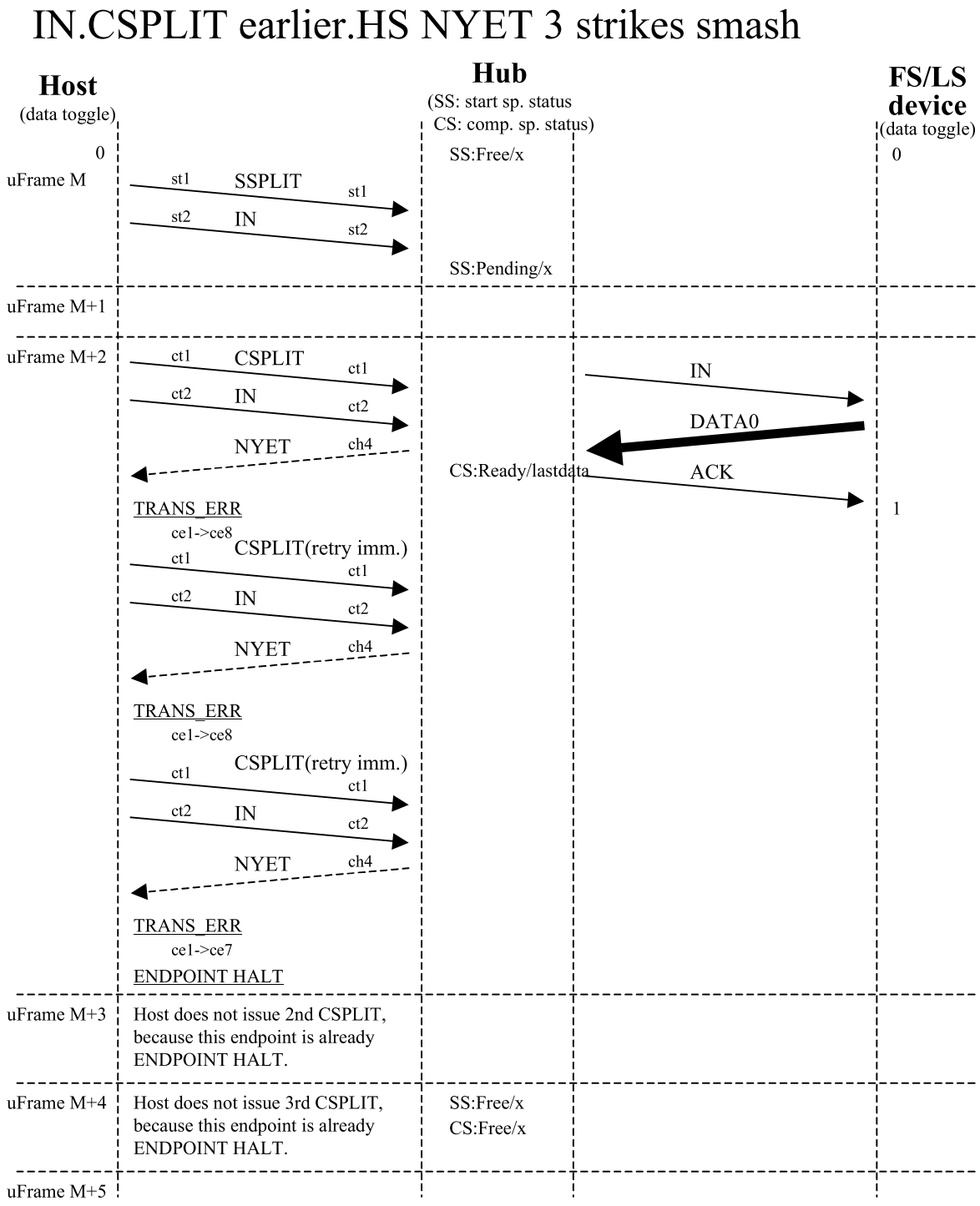



**Figure A-79. CS Earlier HS NYET 3 Strikes Smash**

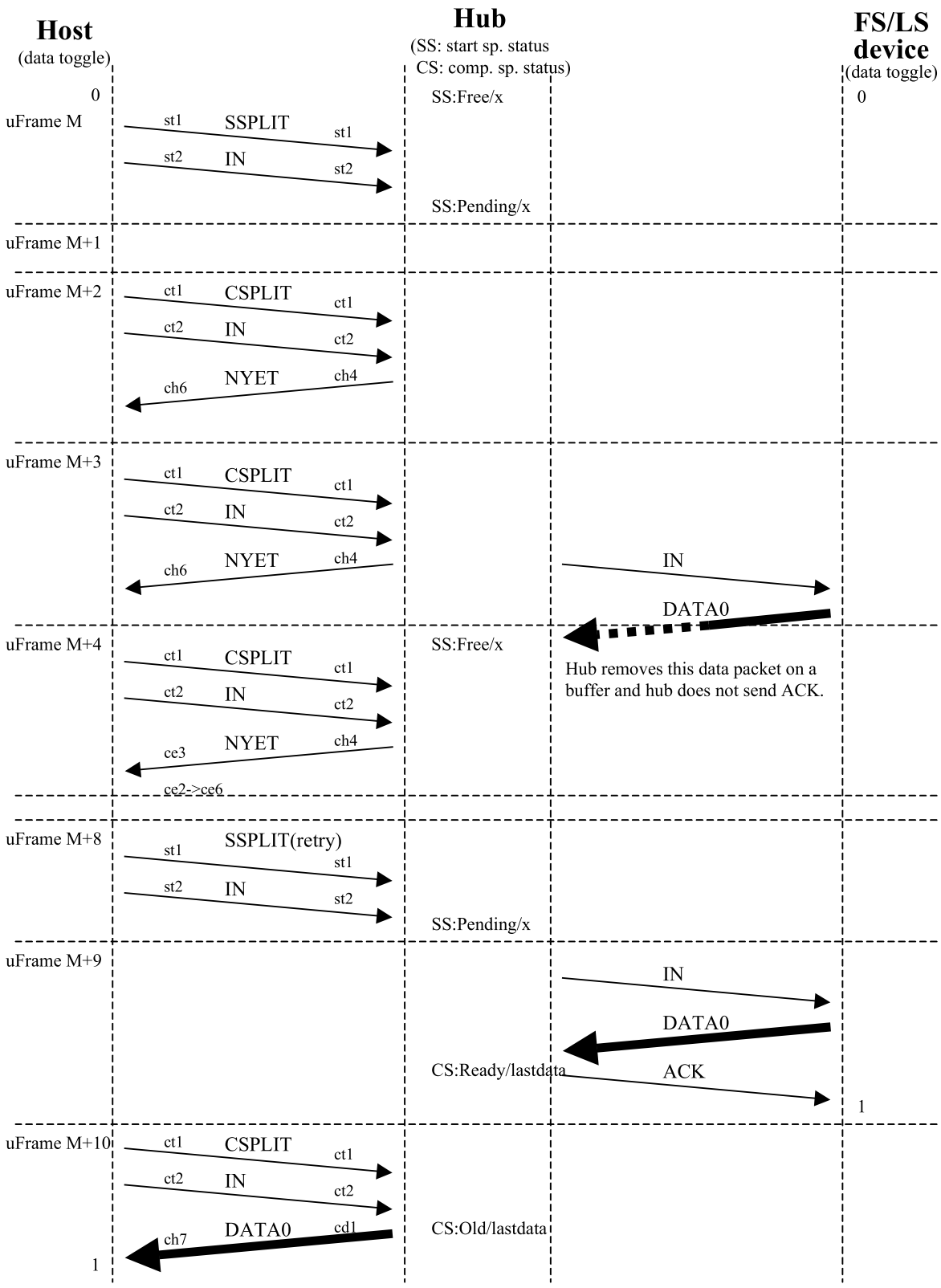



**Figure A-80. Abort and Free Abort(HS NYET and FS/LS Transaction is Continued at End of M+3)**

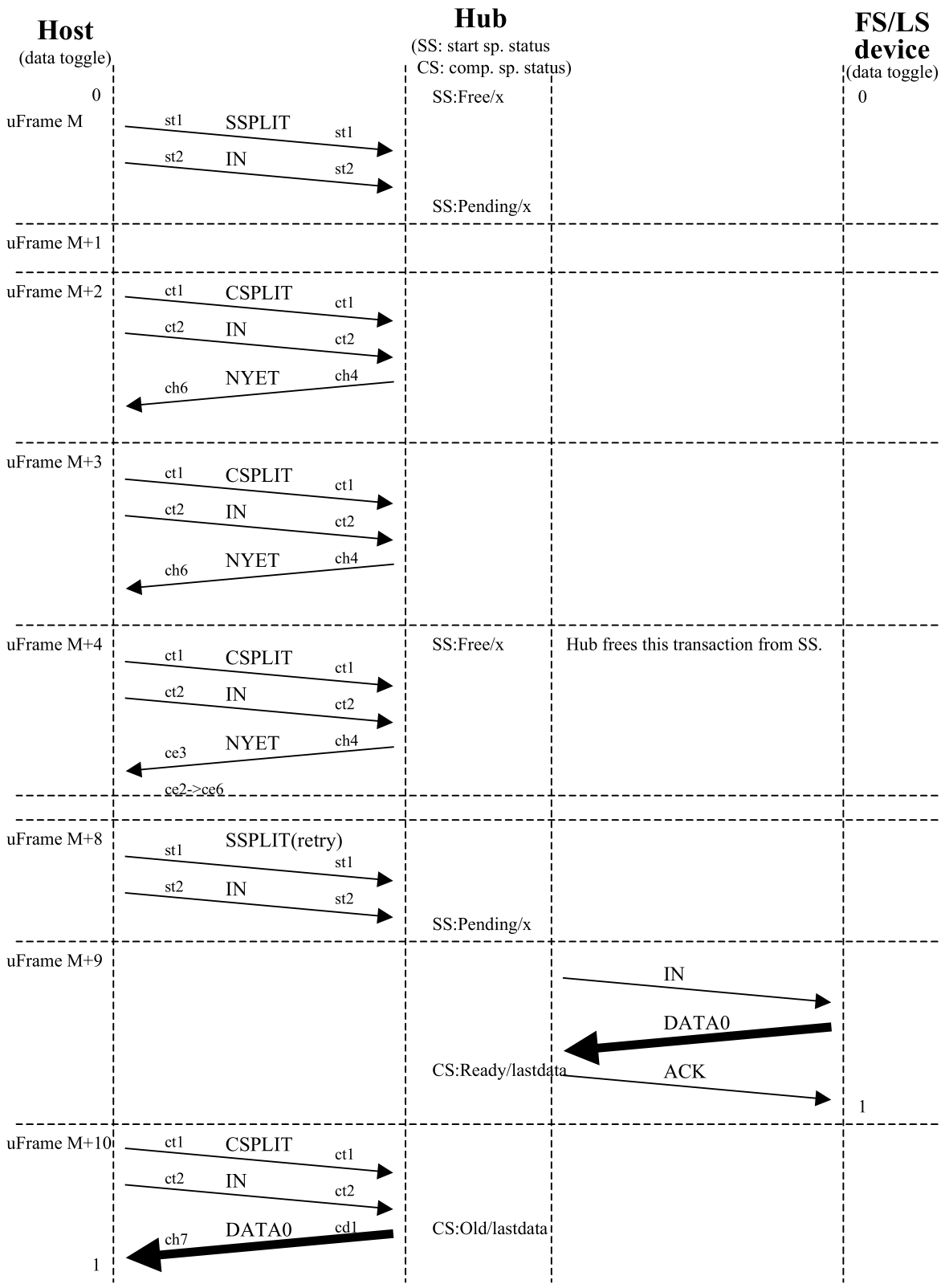



**Figure A-81. Abort and Free Free(HS NYET and FS/LS Transaction is not Started at End of M+3)**

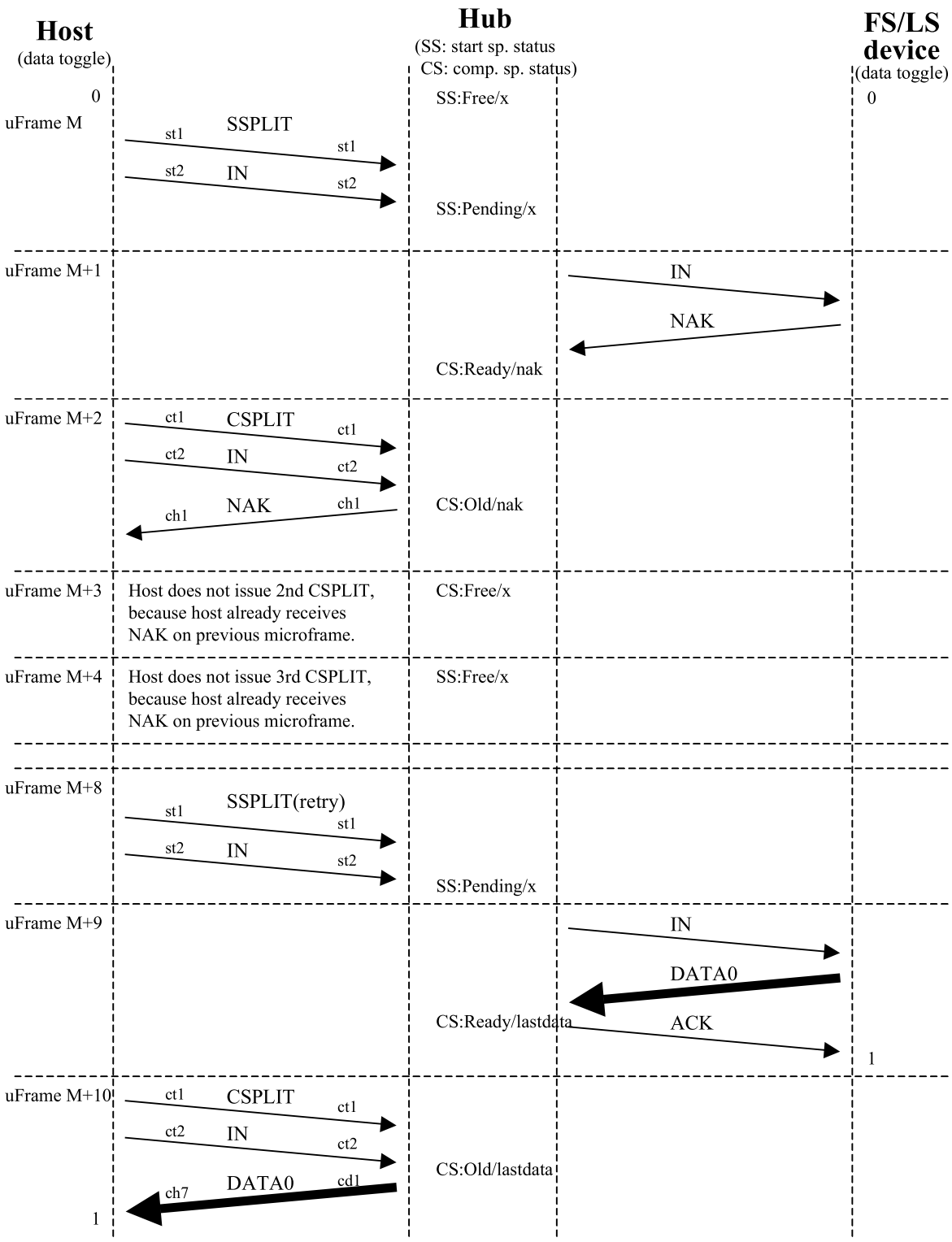



**Figure A-82. Device Busy No Smash(FS/LS NAK)**

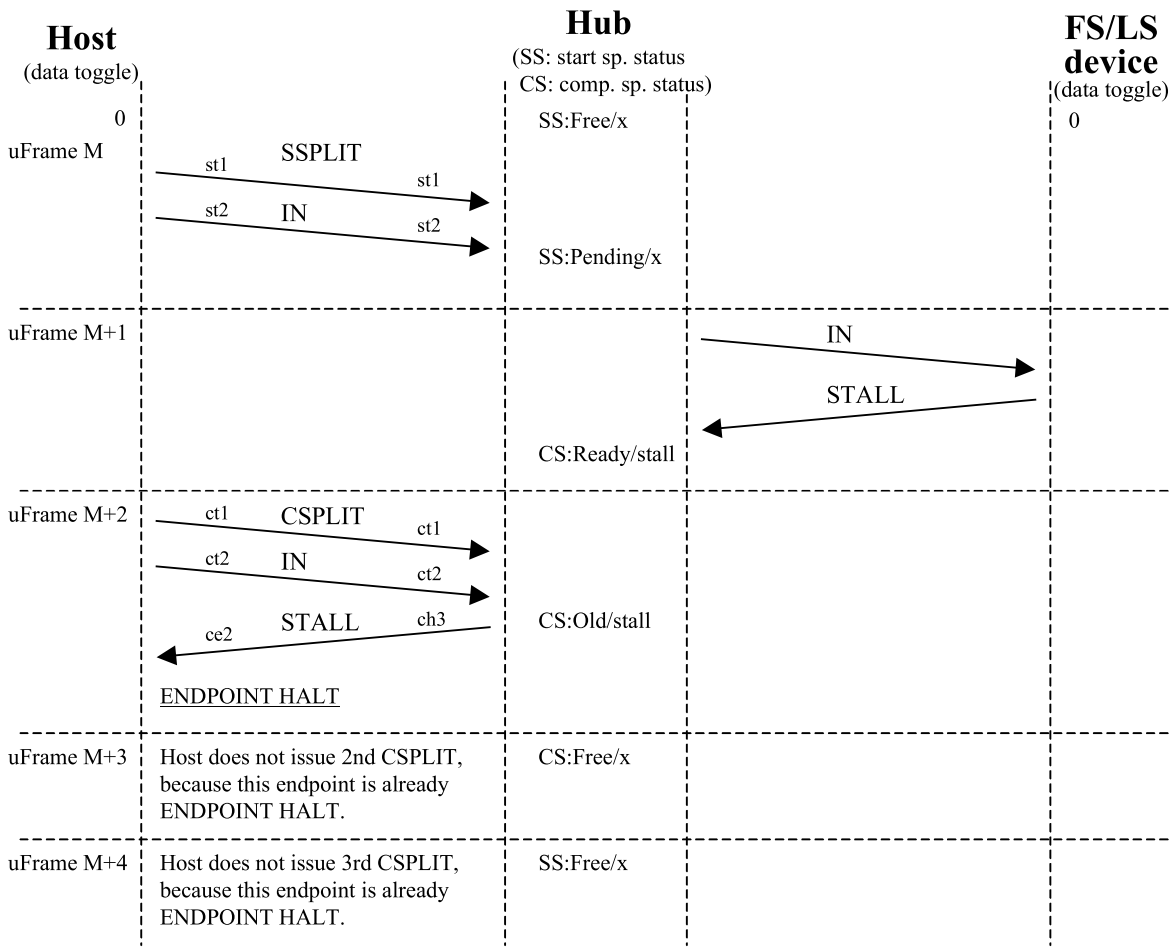
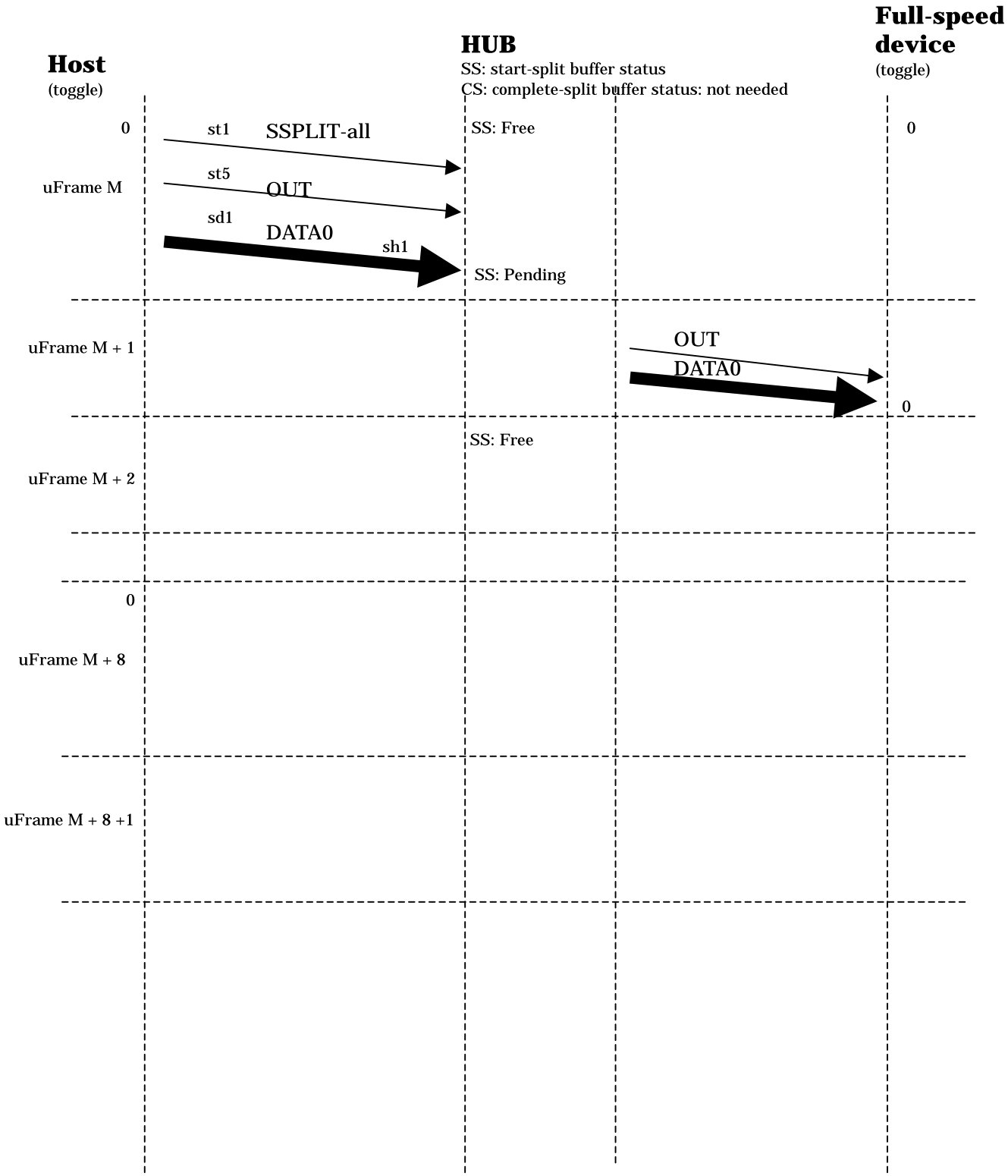



**Figure A-83. Device Stall No Smash(FS/LS STALL)**

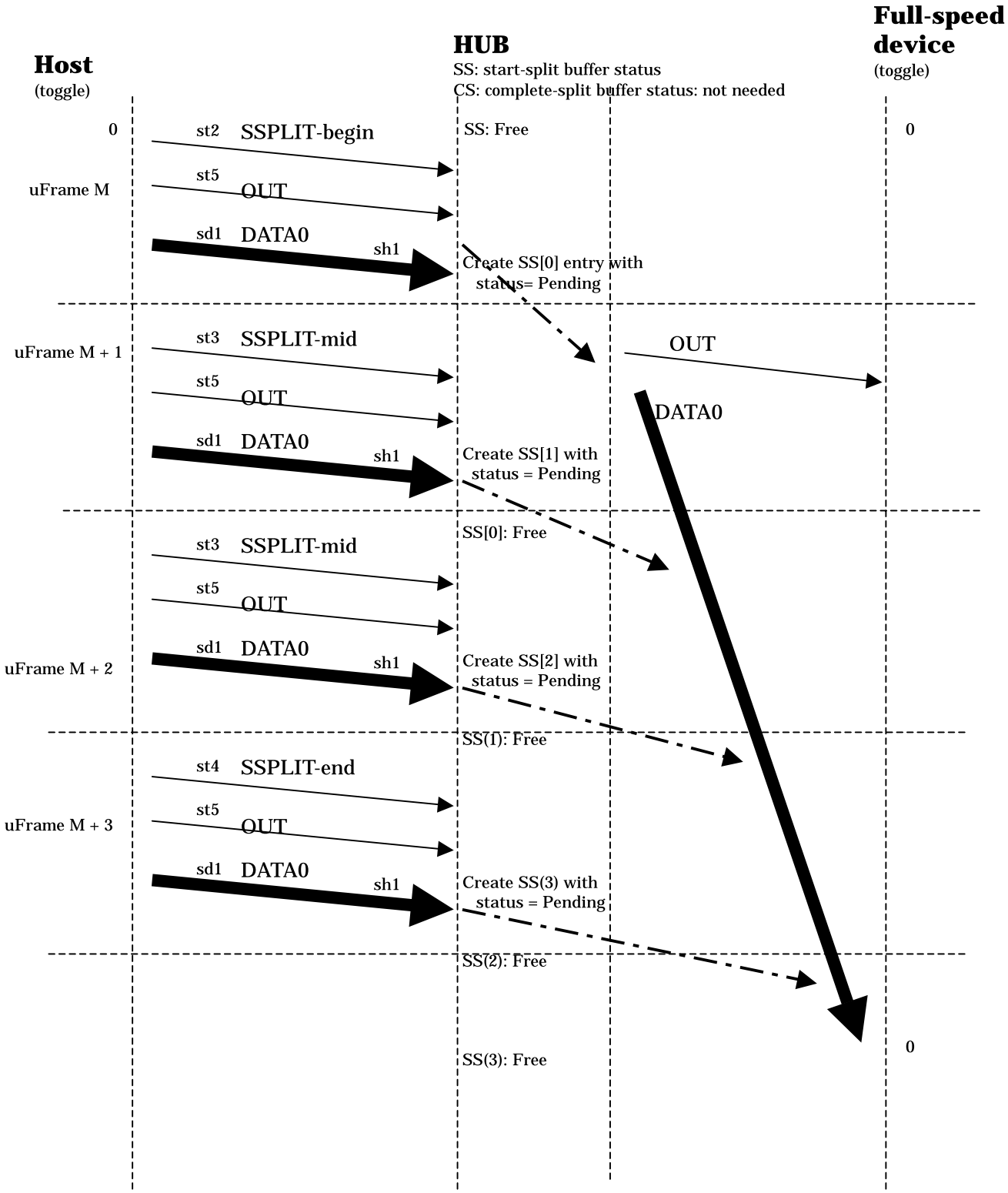
## A.5 Isochronous OUT Split-transaction Examples

| Case | Reference Figure |
|---|---|
| Normal: small payload (<=188) | 1 |
| Normal: large payload (> 188) | 2 |
| HS SSPLIT-all corrupted, HS OUT corrupted | 3 |
| HS DATA0 corrupted (small payload) | 4 |
| HS SSPLIT-begin corrupted | 5 |
| HS OUT after the HS SSPLIT-begin is corrupted | 6 |
| HS DATA0 corrupted (large payload) | 7 |
| HS SSPLIT-mid or OUT or DATA0 corrupted | 8 |

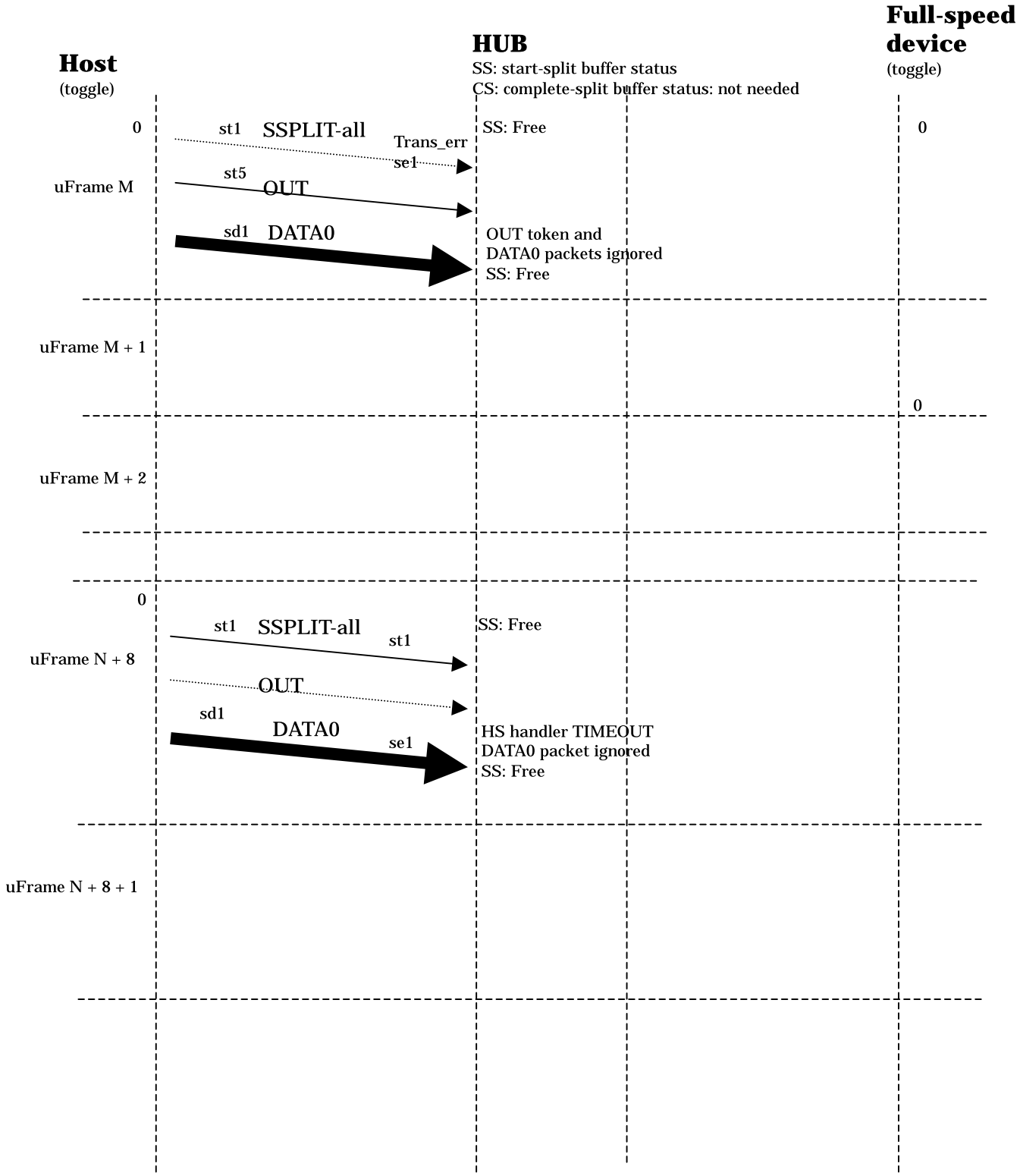
1) Normal. Payload <= 188 bytes:



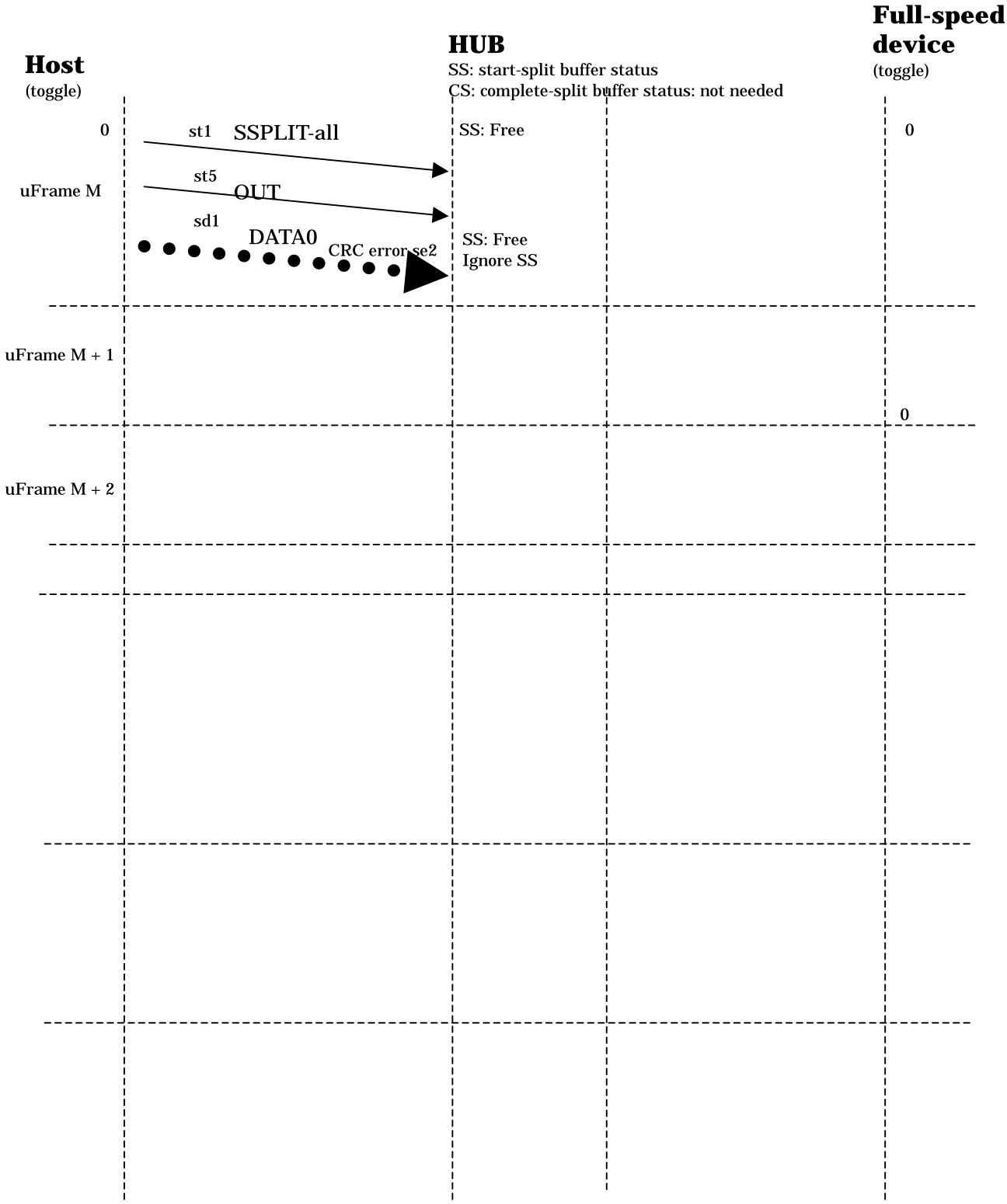
## 2) Normal. Payload > 188 Bytes



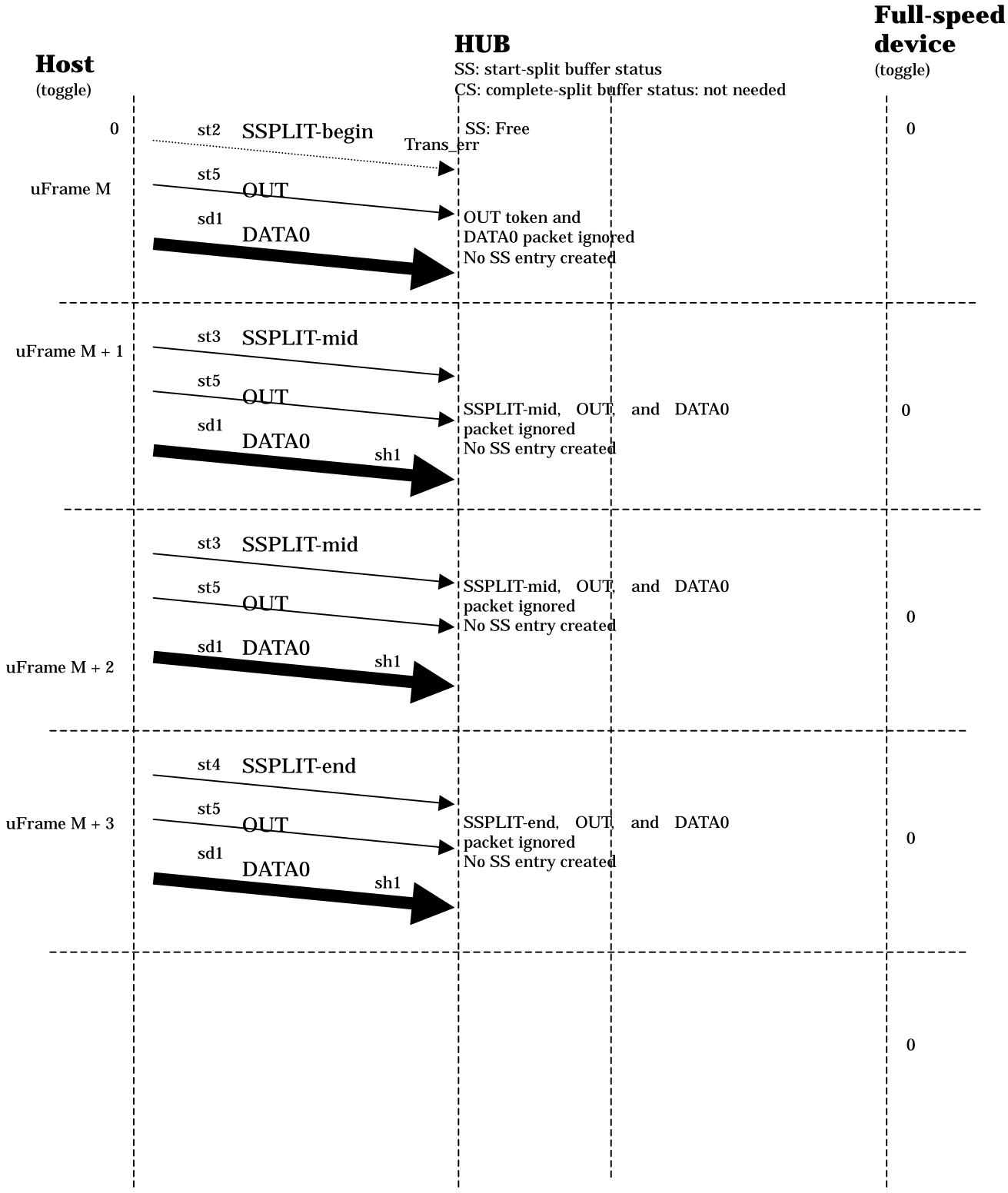
## 3) HS SSPLIT-all corrupted (missing or CRC error etc.) HS OUT corrupted



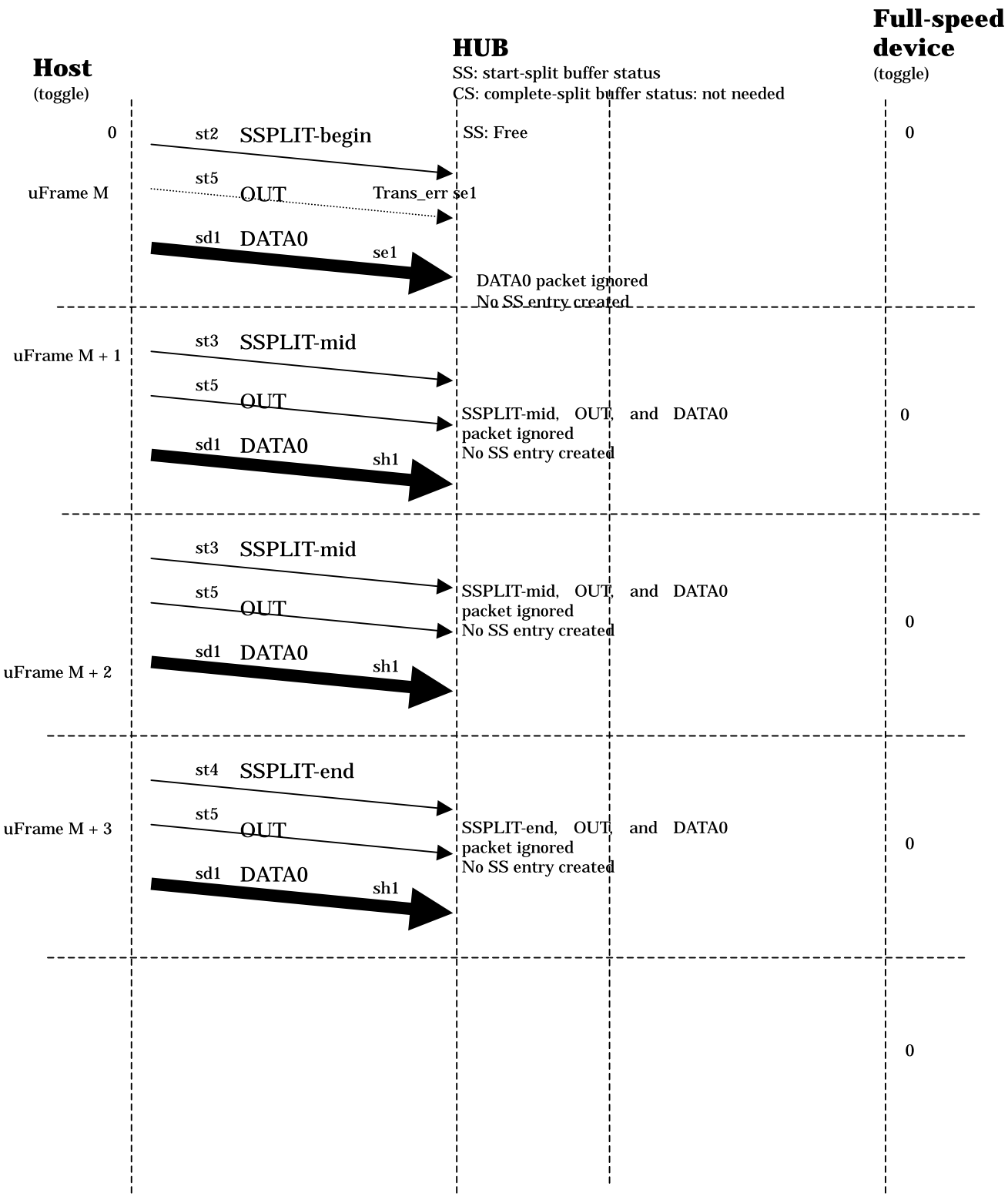
## 4) HS DATA0 corrupted



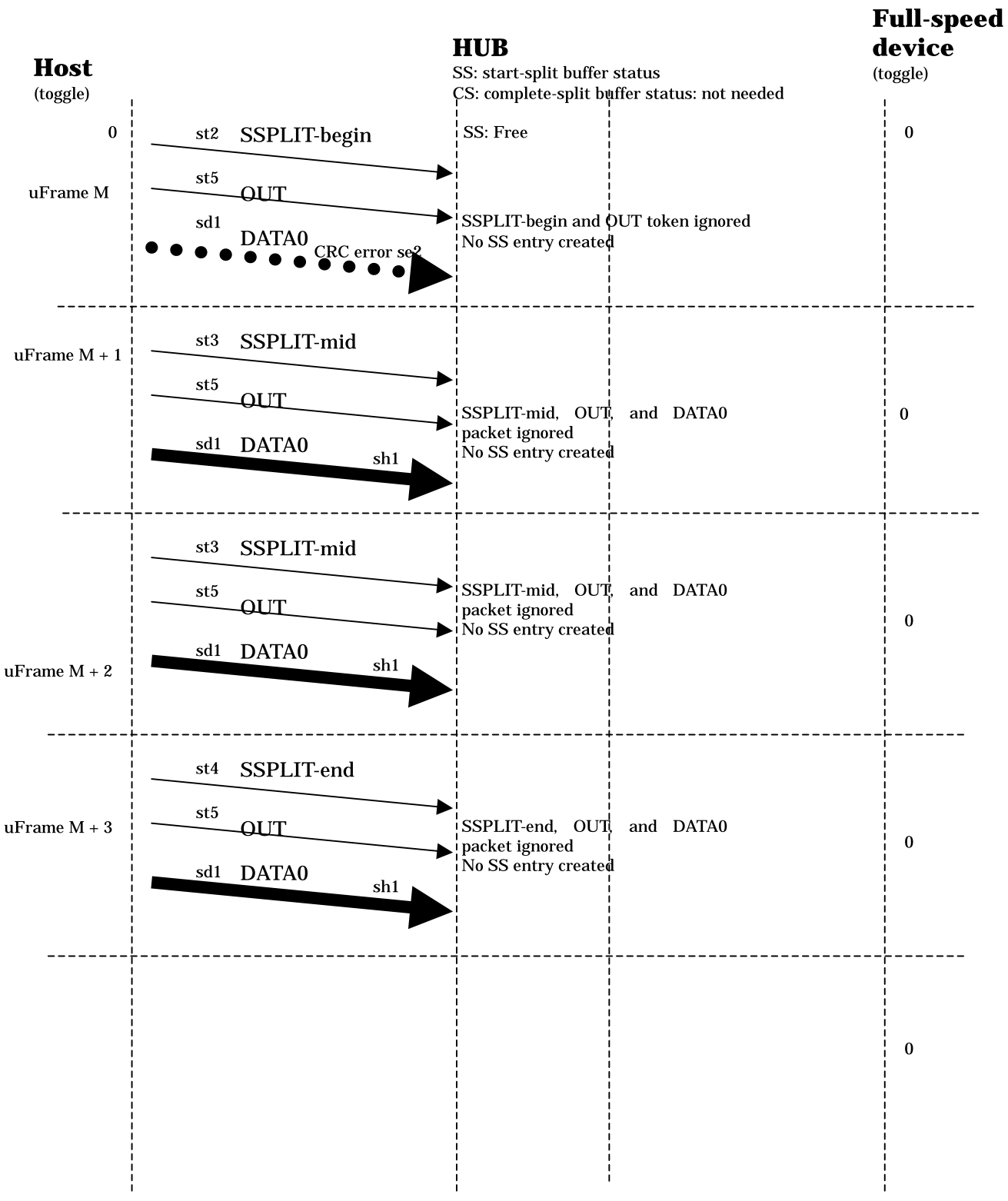
# 5) HS SSPLIT-begin corrupted (missing or CRC error etc.)



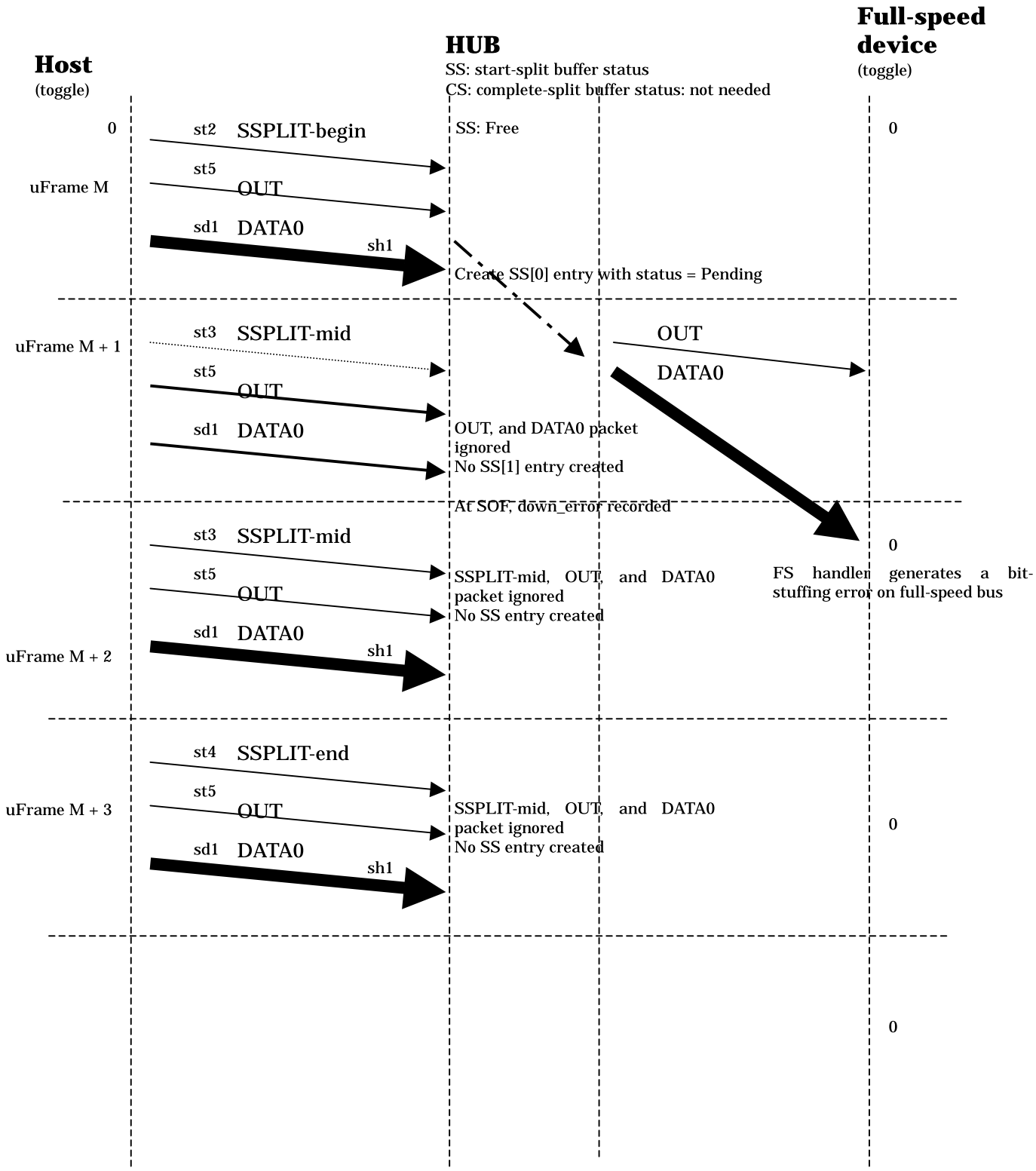
# 6) HS OUT after the HS SSPLIT-begin is corrupted



## 7) HS DATA0 corrupted



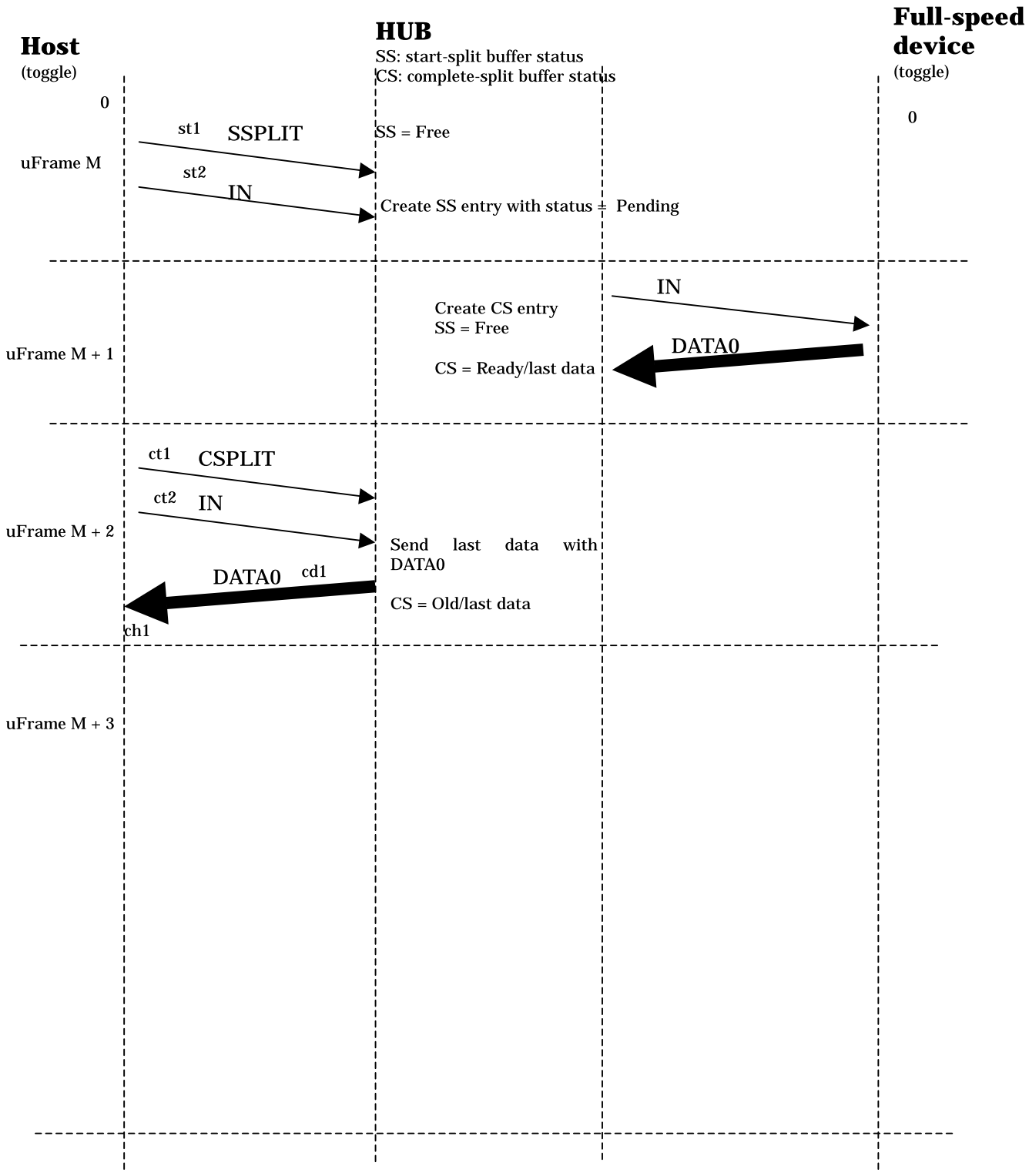
# 8) HS SSPLIT-mid or OUT token or DATA0 packet after it is corrupted



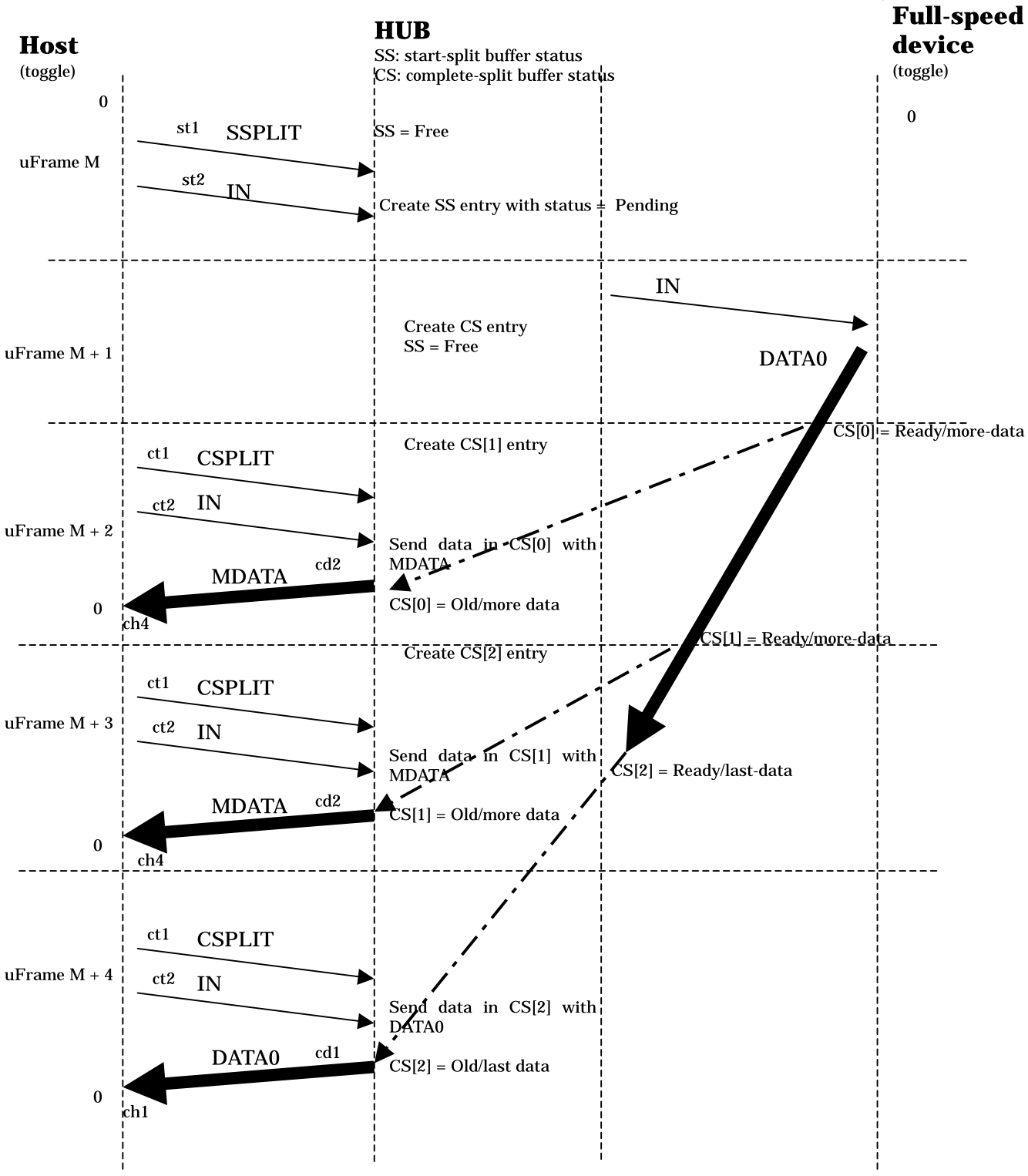
## A.6 Isochronous IN Split-transaction Examples

| Case | Reference Figure |
|---|---|
| Normal: full-speed bus transaction does not cross microframe boundary | 1 |
| Normal: full-speed bus transaction crosses microframe boundary | 2 |
| HS SSPLIT corrupted | 3 |
| IN after HS SSPLIT corrupted | 4 |
| HS CSPLIT corrupted | 5 |
| Consecutive HS CSPLIT corrupted | 6 |
| HS IN corrupted | 7 |
| Consecutive HS IN corrupted | 8 |
| HS data corrupted (case 1) | 9 |
| HS data corrupted (case 2) | 10 |
| TT has more data than HS expects | 11 |
| HS CS too early (full-speed data not available yet) | 12 |
| Full-speed timeout or CRC error | 13 |

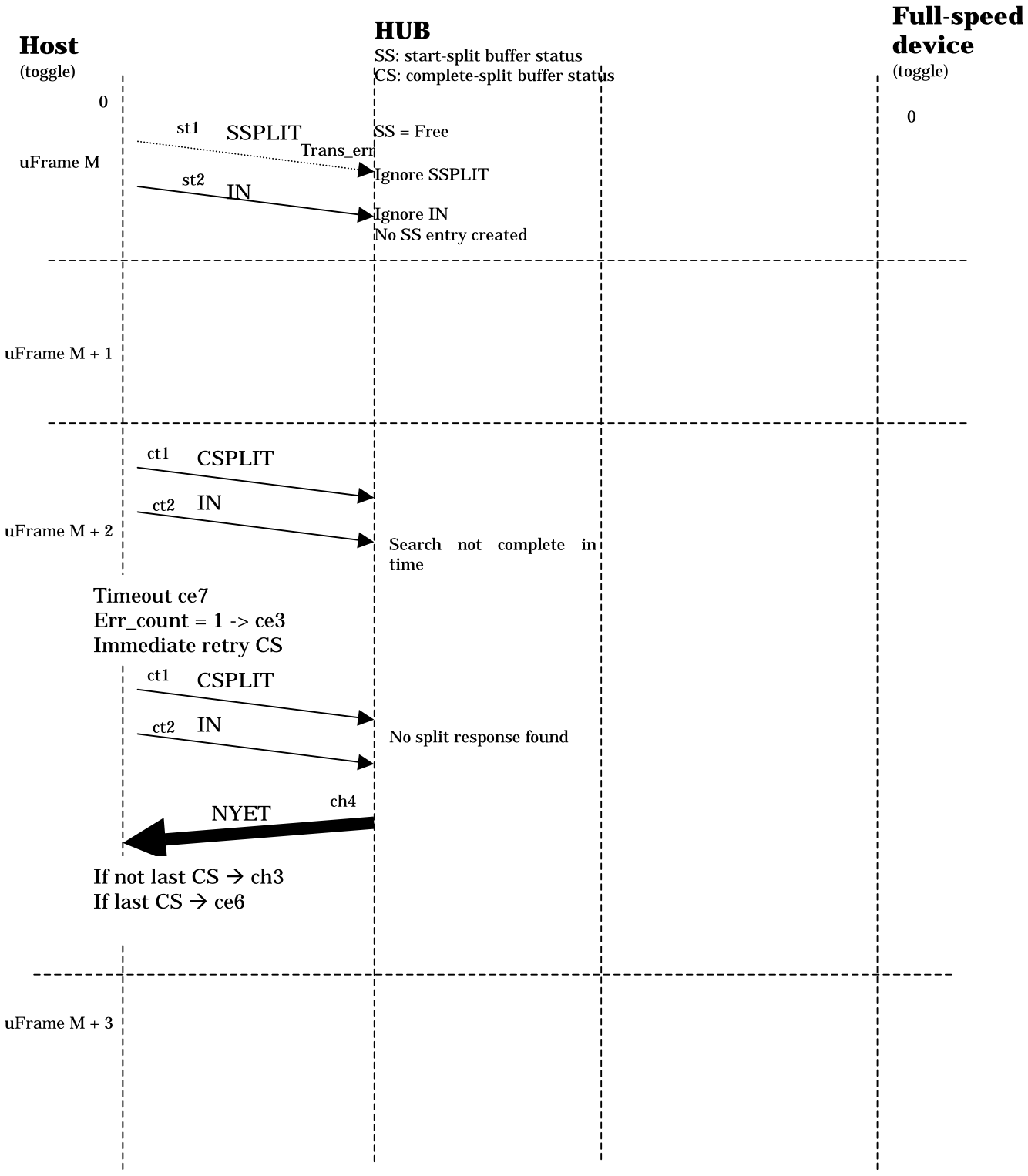
1) Normal: full-speed bus transaction does not cross microframe boundary



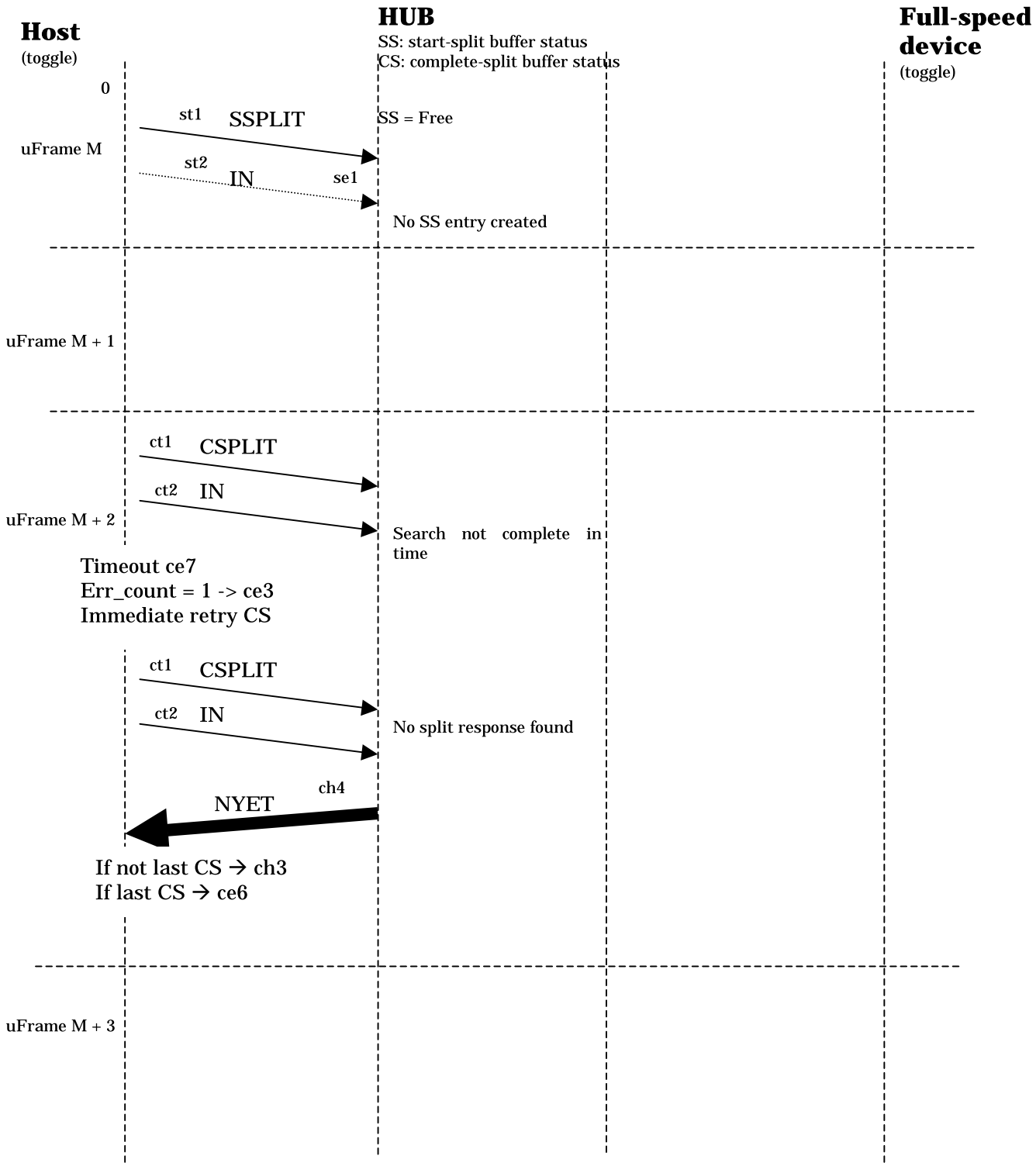
## 2) Normal: full-speed bus transaction crosses microframe boundary



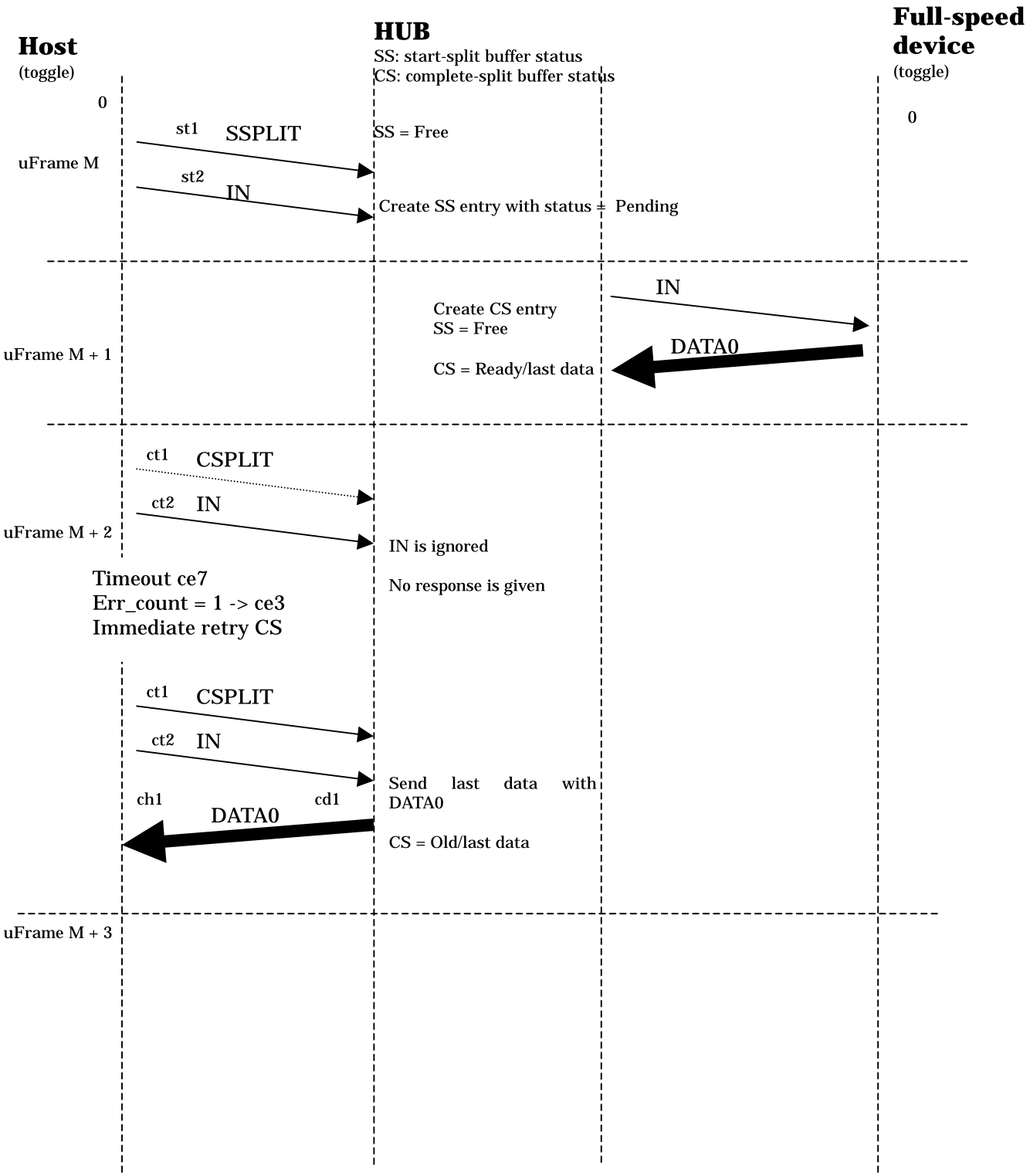
## 3) HS SSPLIT corrupted



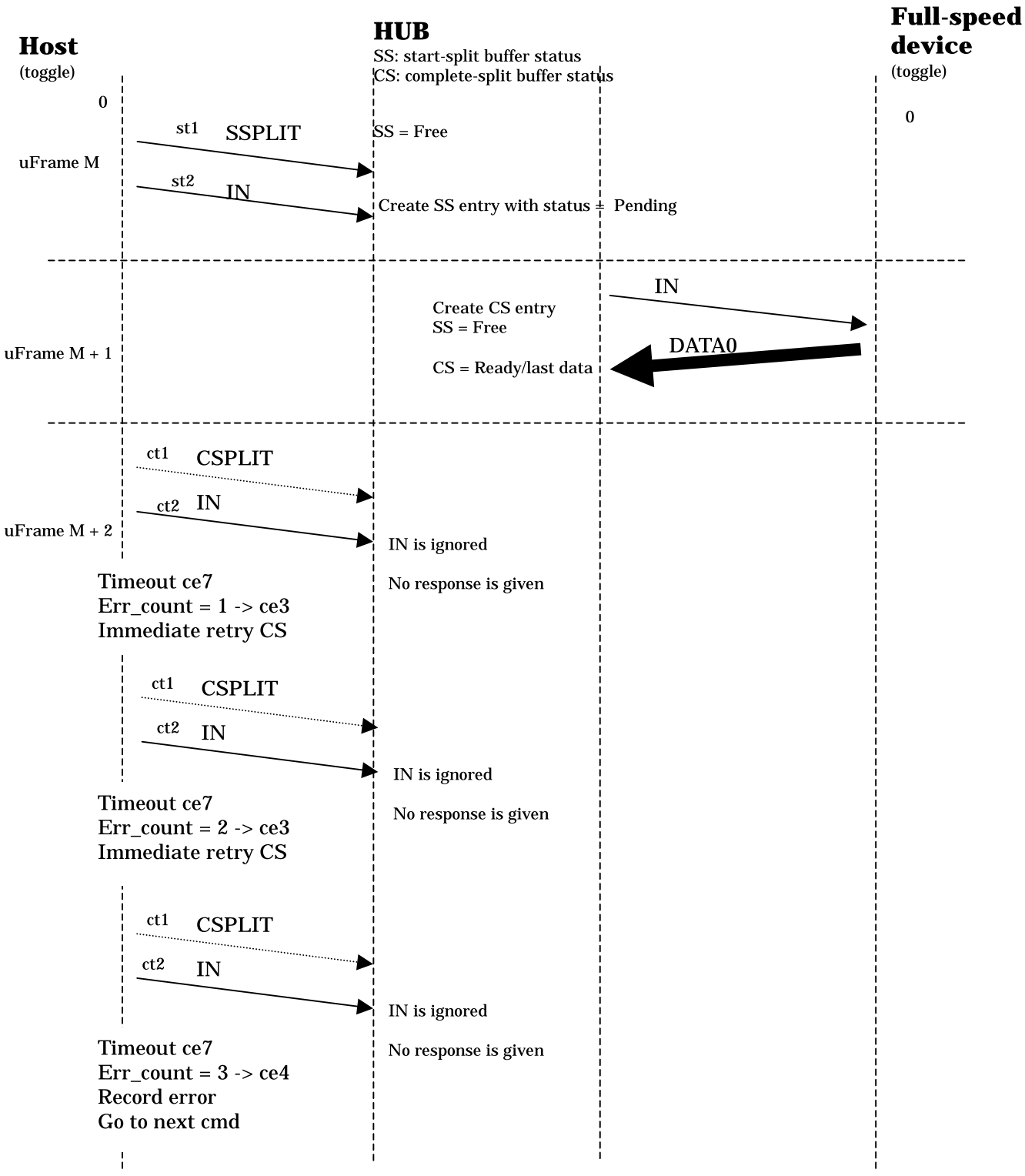
4) IN after HS SSPLIT corrupted



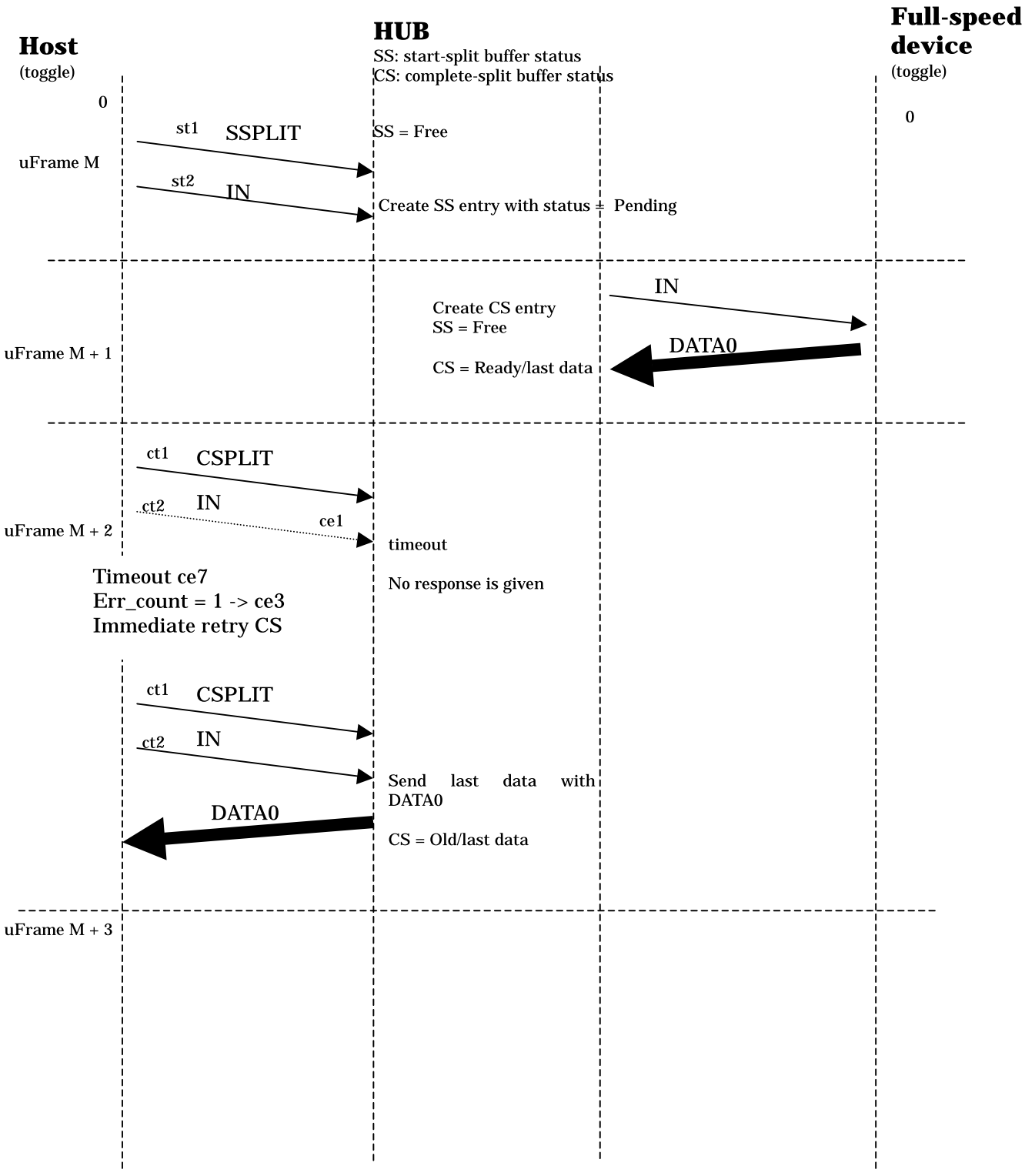
5) HS CSPLIT corrupted



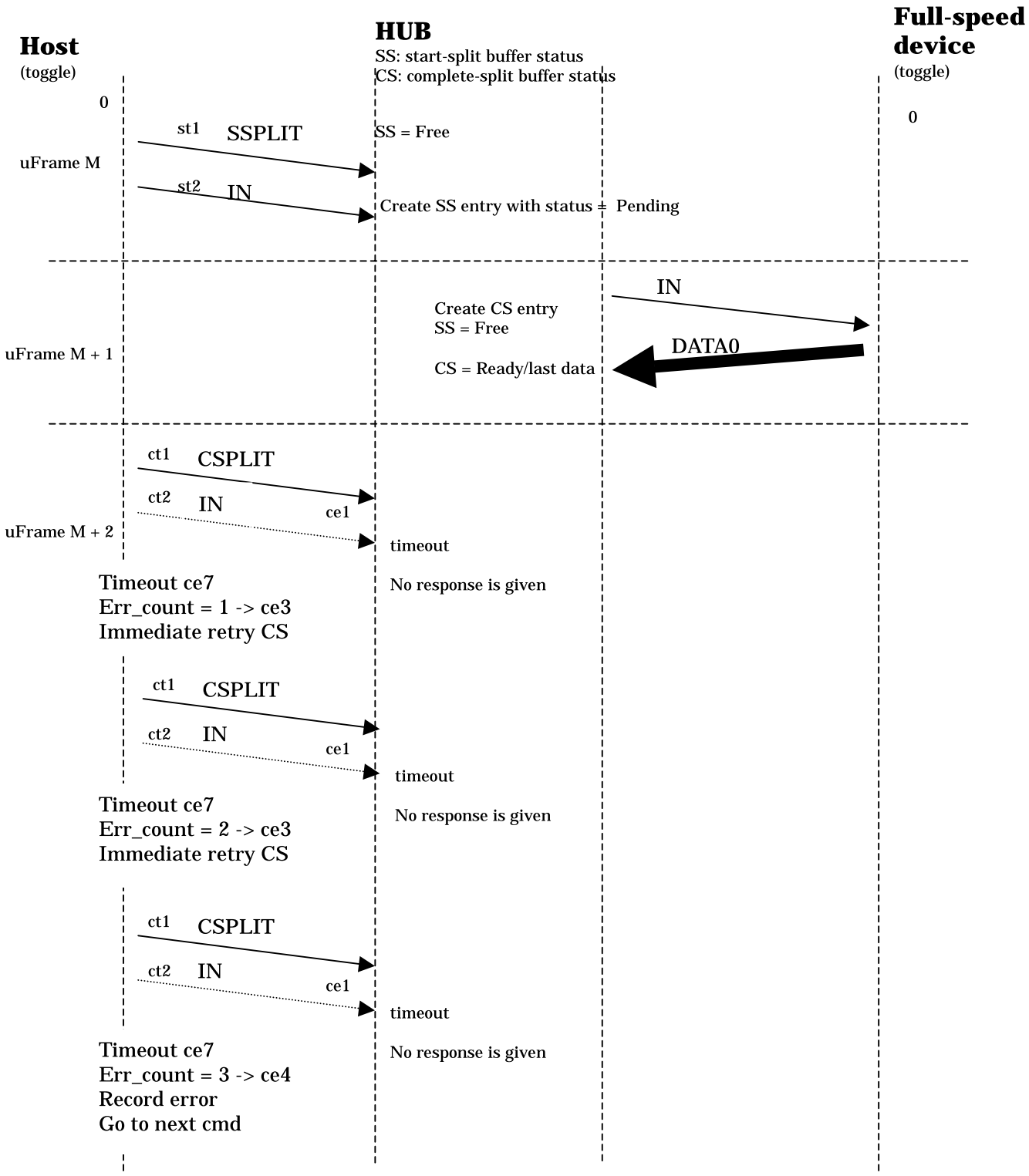
## 6) Consecutive HS CSPLIT corrupted



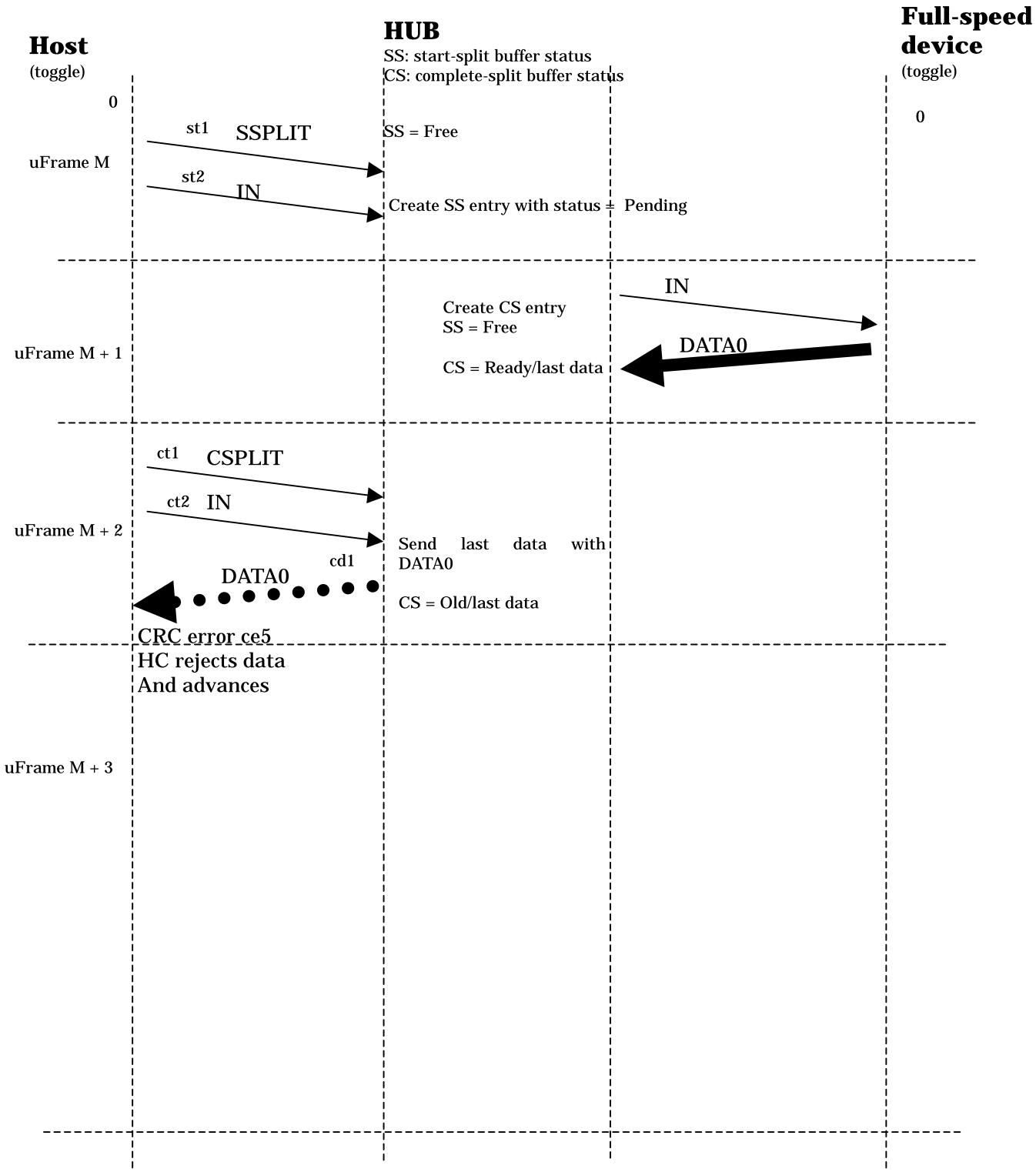
## 7) HS IN corrupted



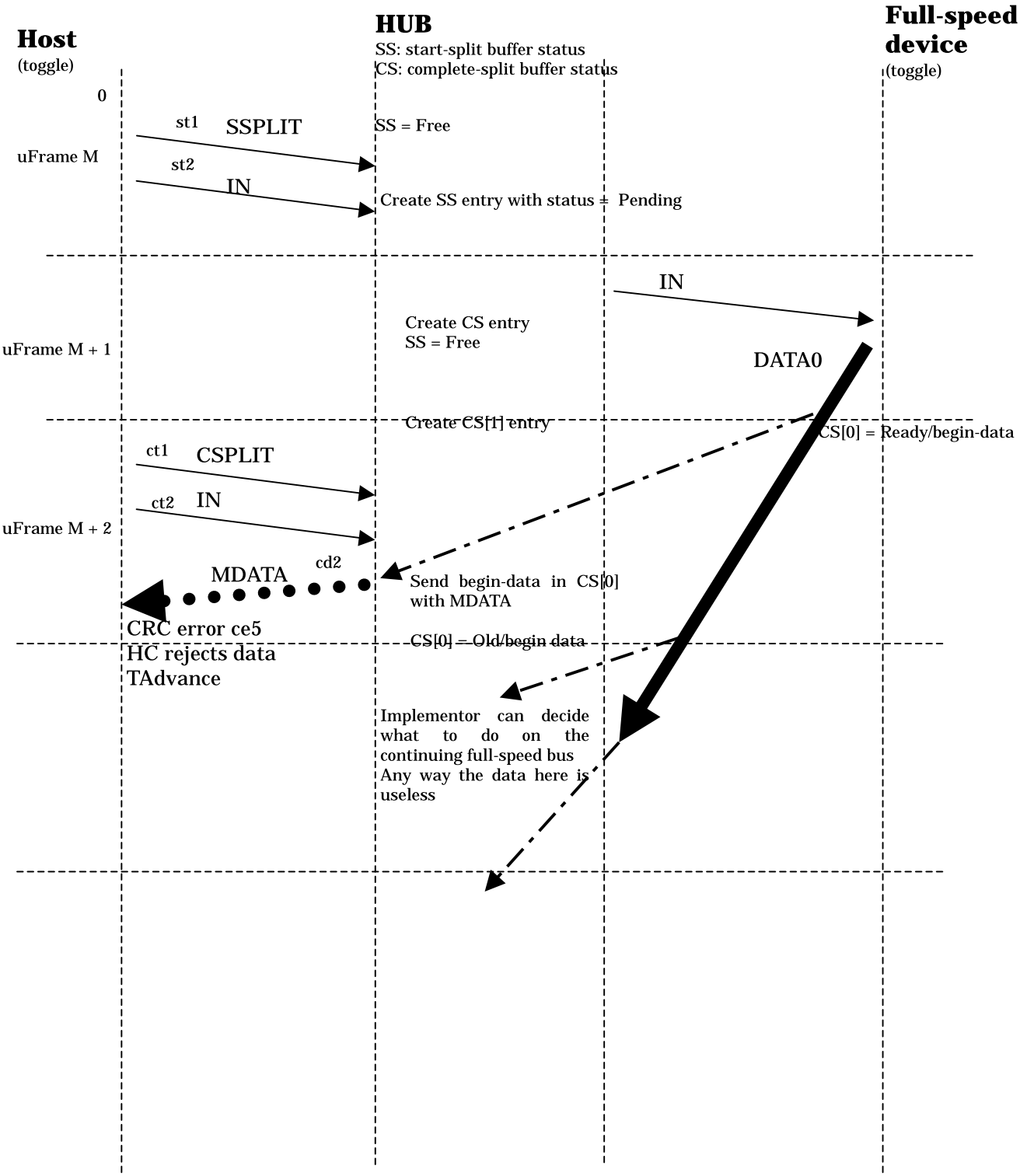
## 8) Consecutive HS IN corrupted



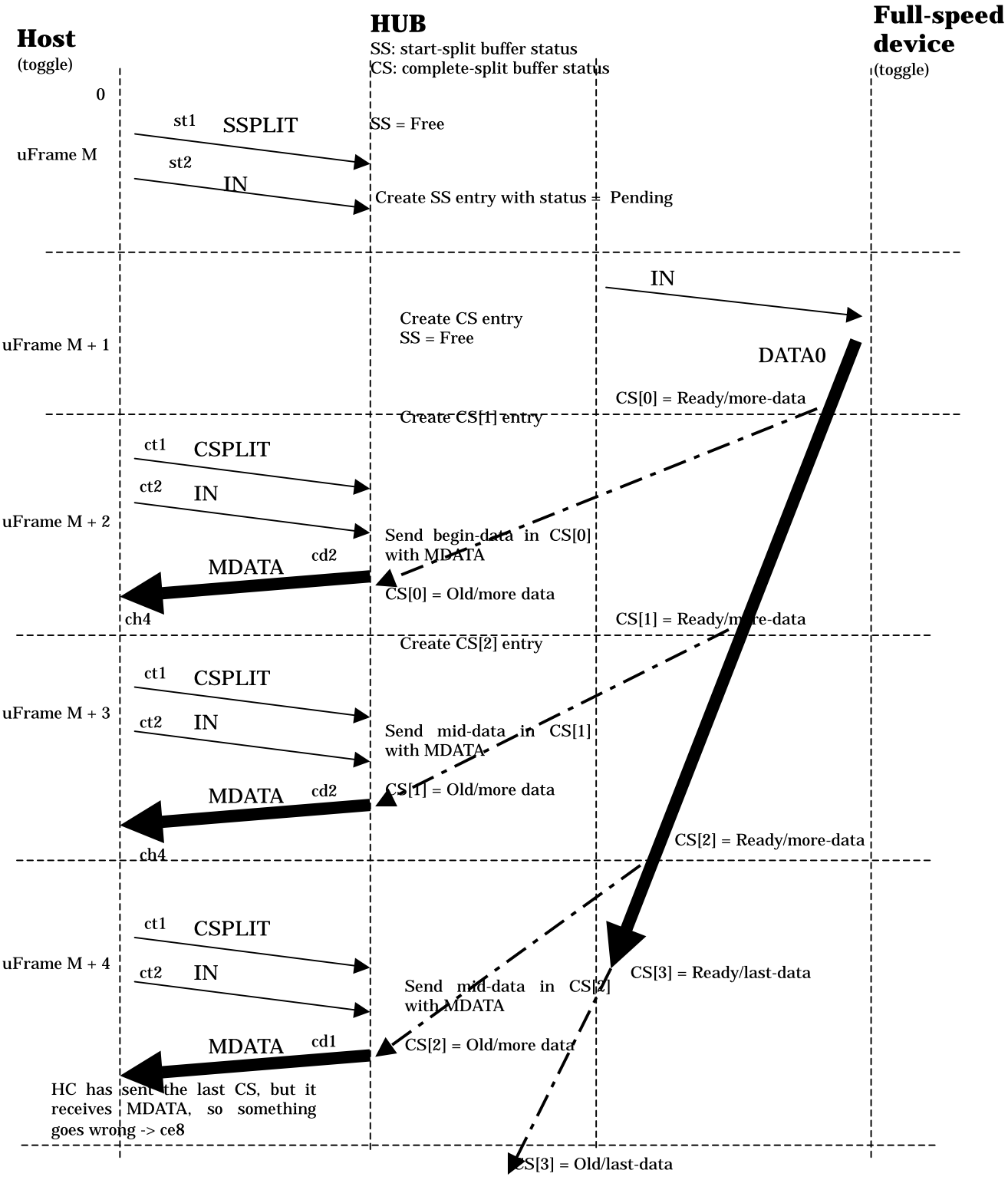
9) HS data corrupted (case 1)



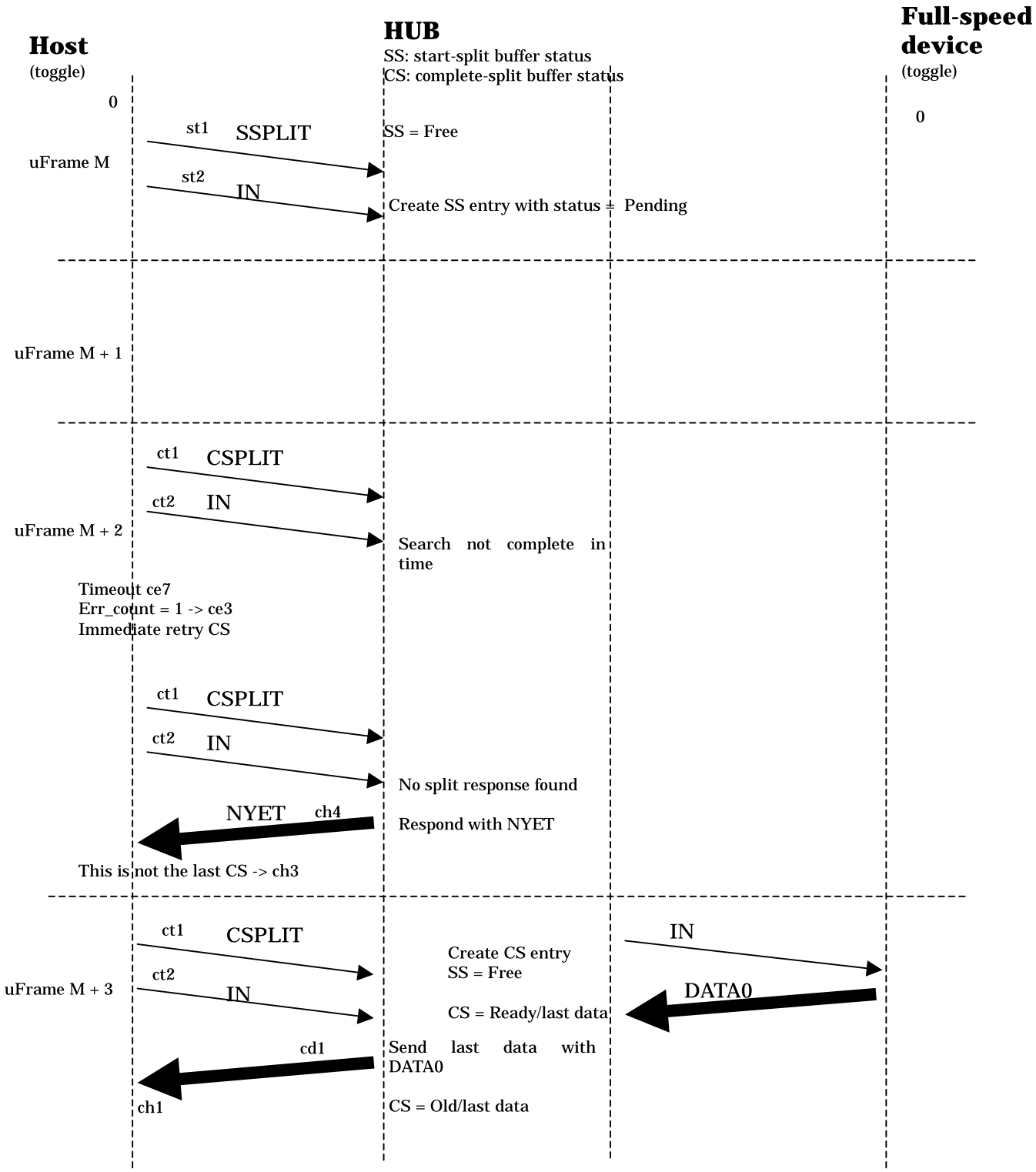
10) HS data corrupted (case 2)



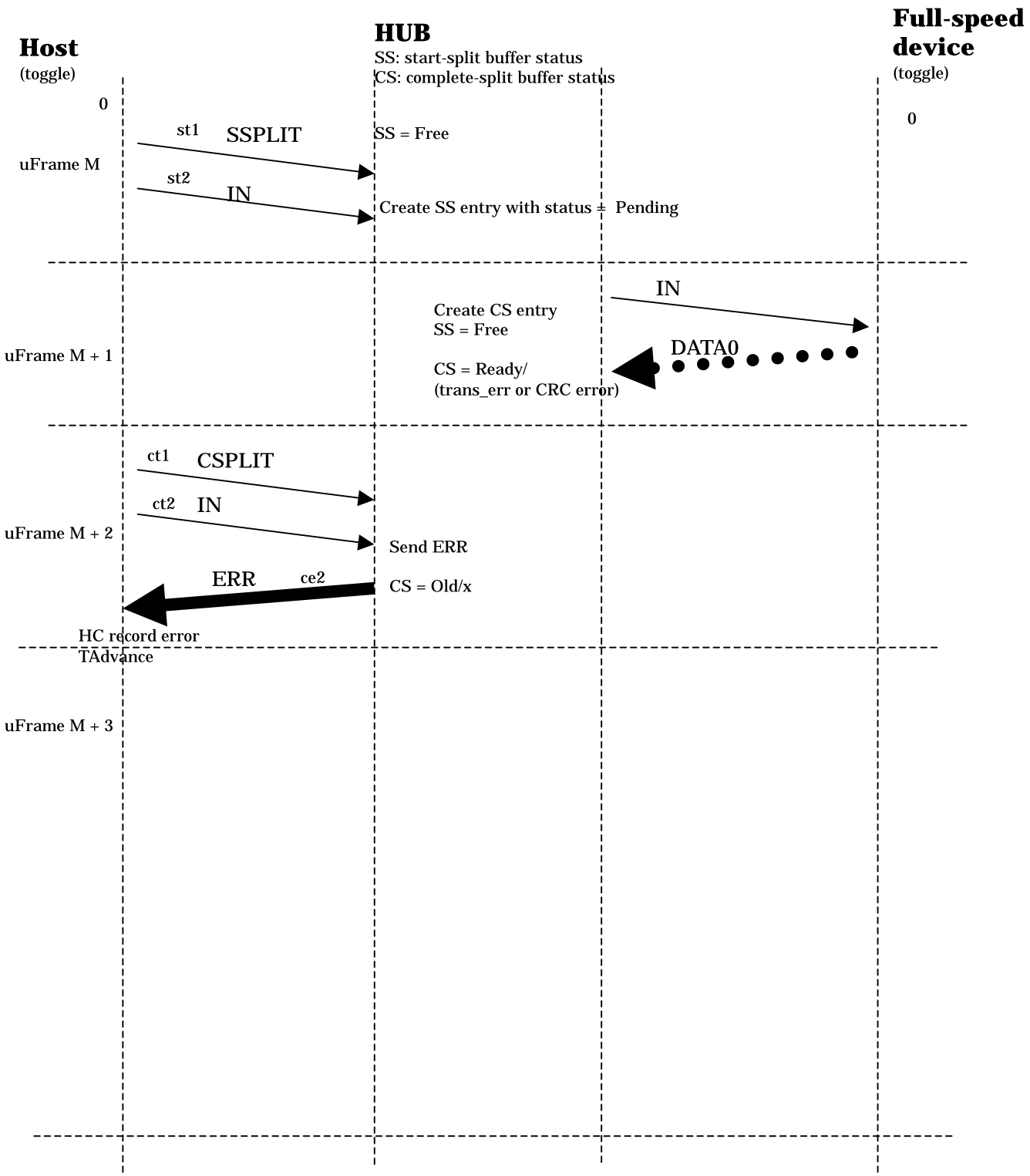
11) TT has more data than HC expects



12) HS CS too early (full-speed data not available yet)



13) Full-speed timeout or CRC error



# Appendix B
# Example Declarations for State Machines

This appendix contains example declarations used in the construction of the state machines in Chapters 8 and 11. These declarations may help in understanding some aspects of the state machines. There are three sets of declarations: global declarations, host controller specific declarations, and transaction translator declarations.

## B.1 Global Declarations

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

PACKAGE behav_package IS

    CONSTANT FIFO_DEPTH  : INTEGER := 3;     -- Size of bulk buffer.
                                             -- Determines how many outstanding
                                             -- Split transactions are allowed.

    CONSTANT ERROR_INJECT_DEPTH  : INTEGER := 16;     -- Size of Error Inject FIFO.

    TYPE ep_types IS (bulk, control, isochronous, interrupt);  -- endpoint types

    TYPE directions IS (in_dir, out_dir);  -- data transfer directions

    TYPE pids IS (NAK, ACK, STALL,         -- possible packet PIDs
      tokenIN, tokenOUT, tokenSETUP,
      SOF, ping,
      MDATA,
      DATAx,                      -- represents both DATA0 and DATA1
      CSPLIT, SSPLIT,
      NYET, ERR,
      TRANS_ERR);               -- pseudo PIDs for error cases

    TYPE cmds IS (start_split, complete_split, nonsplit, SOF);       -- HC commands

    TYPE data_choices IS (alldata, begindata, enddata, middata);
      -- isochronous data part for an HC command

    TYPE HCresponses IS (      -- what HC should do next for this command
      do_start,                -- do start-split transaction
      do_complete,             -- do complete-split transaction
      do_complete_immediate,
          -- do complete-split immediately before doing a different transaction
      do_halt,
          -- do endpoint halt processing for the endpoint of this command
      do_next_cmd,             -- do next command for this endpoint
                               -- advance data pointer appropriately
      do_same_cmd,             -- do same command over again
      do_comp_immed_now,
          -- do complete-split imediately within same microframe
      do_next_complete,
          -- do next complete-split in next microframe (periodic)
      do_next_ping,
      do_ping,
      do_out,
      do_idle                  -- Response not active - Used for Simulation
    );

    TYPE Devresponses IS (
      do_next_data,
      do_nothing
    );
```

```
TYPE waits IS (
  ITG,                    -- wait up to an inter packet (intra transaction) gap
                          --  for the next packet.
  none);                  -- wait forever for next packet

TYPE CRCs IS (bad, ok);

TYPE states IS (old, pending, ready, no_match, match_busy);
-- states of a buffer

TYPE results IS (          -- full/low speed transaction result in a buffer
        r_ack,
        r_nak,
        r_trans_err,
        r_stall,
        r_badcrc,
        r_lastdata,
        r_moredata,
        r_data);

TYPE epinfo_rec IS RECORD
        space_avail     : boolean;
        data_avail      : boolean;
        ep_type         : ep_types;
        ep_trouble      : boolean;
        toggle          : boolean;
        END RECORD;

TYPE epinfo_array IS ARRAY(1 DOWNTO 0) OF epinfo_rec;

TYPE device_rec IS RECORD
        ep              : epinfo_array;
        HS             : BOOLEAN;
        END RECORD;


TYPE match_rec IS RECORD  -- result of matching a high-speed complete-split
        state           : states;
        down_result     : results;
        END RECORD;

TYPE HS_bus_rec IS RECORD
-- partial high speed transaction state from a high speed bus
        ep_type         : ep_types;
        PID             : pids;
        dev_addr        : INTEGER RANGE 0 TO 127;
        endpt           : INTEGER RANGE 0 TO 15;
        CRC16           : CRCs;
        direction       : directions;
        x               : boolean;
        datapart        : data_choices;
        ready           : boolean;
        timeout         : boolean;
        END RECORD;

TYPE command_rec IS RECORD          -- command state that the HC must act upon
        ep_type         : ep_types;
        cmd             : cmds;
        setup           : boolean;        -- true is control setup
        ping            : boolean;
        HS              : boolean;
        dev_addr        : INTEGER RANGE 0 TO 127;
        endpt           : INTEGER RANGE 0 TO 15;
        CRC16           : CRCs;
        direction       : directions;
        datapart        : data_choices;
        toggle            : boolean;
        last              : boolean;
        END RECORD;
```

```
    TYPE bc_buf_status IS (OLD,NU,NOSPACE);     -- Responses from Compare_BC_buff.

    TYPE BC_buff_rec IS RECORD       -- (partial) state of a bulk/control buffer
           match           : match_rec;
           index           : INTEGER RANGE 0 TO (FIFO_DEPTH-1);
           status          : bc_buf_status;
           END RECORD;

    TYPE CS_buff_rec IS RECORD
      -- (partial) state of a periodic complete-split buffer
           match           : match_rec;
           store           : hs_bus_rec;
           END RECORD;

    TYPE SS_buff_rec IS RECORD
           saw_split: boolean;
           isochO:    boolean;  -- was the last transaction an isochronous OUT SS
           lastdata:  data_choices;
             -- if isochO is true, then what was the last data portion
           END RECORD;

    TYPE cam_rec IS RECORD       -- Information stored in the bulk/control Buffer.
           store           : hs_bus_rec;
           match           : match_rec;
           END RECORD;

    TYPE phases IS (SPLIT, TOKEN, DATA);        -- Error Inject phases.

    TYPE err_inject_rec IS RECORD               -- Error Injection FIFO record.
           phase           : phases;
           timeout         : boolean;
           crc             : CRCs;
           pid             : boolean;
           END RECORD;

    TYPE err_inject_type IS ARRAY((ERROR_INJECT_DEPTH - 1) DOWNTO 0)
      of err_inject_rec;

    TYPE cam_type IS ARRAY((FIFO_DEPTH - 1) DOWNTO 0) OF cam_rec;

    --returns true when there is a packet ready to receive from a bus

    FUNCTION Packet_ready(HS_bus_in: HS_bus_rec) RETURN boolean;

    -- wait until there is a packet ready on a bus
    PROCEDURE Wait_for_packet(HS_bus_in: HS_bus_rec; wait_type: waits);

    PROCEDURE RespondDev(dr: devresponses);

    PROCEDURE HC_Accept_data;

    PROCEDURE HC_Reject_data;

    PROCEDURE Dev_Accept_data;

    PROCEDURE Dev_Record_error;
END behav_package;
```

## B.2 Host Controller Declarations

```
    shared VARIABLE ErrorCount     :  integer :=0;
    shared VARIABLE HC_response_v  : HCresponses;
    shared VARIABLE rd_ptr         : integer RANGE 0 TO (ERROR_INJECT_DEPTH-1) := 0;
    SIGNAL HSU2_ready              : boolean;
    SIGNAL HC_command_ready        : boolean := FALSE;
    SIGNAL HC_cmd                  : command_rec;
    SIGNAL HCresponse              : HCresponses;
    SIGNAL err_inject_fifo         : err_inject_type;
    SIGNAL wr_ptr                  : integer RANGE 0 TO (ERROR_INJECT_DEPTH-1) := 0;

    ------------------------------------------------------------------------------
    -- Issue a packet onto the HS bus.
    ------------------------------------------------------------------------------
    PROCEDURE Issue_packet(SIGNAL HS_bus_out : OUT HS_bus_rec;
                               pid        : pids) IS
    BEGIN

        HS_bus_out.ep_type  <= HC_cmd.ep_type;
        HS_bus_out.endpt    <= HC_cmd.endpt;
        HS_bus_out.dev_addr <= HC_cmd.dev_addr;
        HS_bus_out.direction <= HC_cmd.direction;
        HS_bus_out.datapart <= HC_cmd.datapart;

        HS_bus_out.x        <= HC_cmd.toggle;     -- ????

        -- Check for Error injection when FIFO is not empty.
        IF (wr_ptr /= rd_ptr) THEN

            -- Insert an error during SPLIT phase ?
            IF ((err_inject_fifo(rd_ptr).phase = SPLIT AND (pid = SSPLIT OR pid =
CSPLIT)) OR
            -- Insert an error during Token phase ?
                (err_inject_fifo(rd_ptr).phase = TOKEN AND
                  (pid = tokenIN OR pid = tokenOUT OR pid = tokenSETUP)) OR
            -- Insert an error during Data phase ?
                (err_inject_fifo(rd_ptr).phase = DATA AND (pid = MDATA OR pid = DATAx)))
THEN

                HS_bus_out.crc16   <= err_inject_fifo(rd_ptr).crc;
                HS_bus_out.timeout <= err_inject_fifo(rd_ptr).timeout;
                IF (err_inject_fifo(rd_ptr).pid) THEN
                    HS_bus_out.pid     <= TRANS_ERR;
                ELSE
                    HS_bus_out.pid      <= pid;
                END IF;

                -- Update read pointer.
                IF (rd_ptr = (ERROR_INJECT_DEPTH-1)) THEN
                   rd_ptr := 0;
                ELSE
                    rd_ptr := rd_ptr + 1;
                END IF;
            ELSE
                -- Otherwise issue packet with no errors.
                HS_bus_out.crc16    <= ok;
                HS_bus_out.timeout  <= FALSE;
                HS_bus_out.pid      <= pid;

            END IF;

        -- Otherwise issue packet with no errors.
        ELSE
            HS_bus_out.crc16    <= ok;
            HS_bus_out.timeout  <= FALSE;
            HS_bus_out.pid      <= pid;
        END IF;
        HS_bus_out.ready    <= TRUE;
        HS_bus_out.ready    <= FALSE after 500 ps;
```

```
END Issue_packet;

-------------------------------------------------------------------------------
-- Get next command for HC to execute.
-- NOT USED FOR THIS IMPLEMENTATION !!!
-------------------------------------------------------------------------------
PROCEDURE HC_Get_next_command IS
BEGIN
END;

-------------------------------------------------------------------------------
-- Tells HC what happened to this command.
-------------------------------------------------------------------------------
PROCEDURE RespondHC (HCresponse : HCresponses) IS
BEGIN
    HC_response_v := HCresponse;
END;

-------------------------------------------------------------------------------
-- Update command status for the next time the command will be executed by HC.
-- NOT USED FOR THIS IMPLEMENTATION !!!
-------------------------------------------------------------------------------
PROCEDURE Update_command (SIGNAL HCdone : OUT boolean) IS
BEGIN
    HCdone <= TRUE;
END;

-------------------------------------------------------------------------------
-- Increment "3 strikes" error count for endpoint transaction.
-------------------------------------------------------------------------------
PROCEDURE IncError IS
BEGIN
    ErrorCount := ErrorCount + 1;
END;

-------------------------------------------------------------------------------
-- Record Error for current command.
-- NOT USED FOR THIS IMPLEMENTATION !!!
-------------------------------------------------------------------------------
PROCEDURE Record_error IS
BEGIN
    ErrorCount := 0;

END;
```

## B.3 Transaction Translator Declarations

```
shared VARIABLE cam          : cam_type;         -- TT buffer.
shared VARIABLE BC_buff      : BC_buff_rec;
shared VARIABLE CS_Buff      : CS_buff_rec;
shared VARIABLE rd_ptr       : integer RANGE 0 TO (ERROR_INJECT_DEPTH-1) := 0;
shared VARIABLE derror_v     : boolean;
shared VARIABLE ss_avail_v   : boolean;
shared VARIABLE periodic     : boolean := FALSE;
shared VARIABLE error_time   :time := 1000000000 ns;
SIGNAL split                 : HS_bus_rec;       -- Stored Shared Split Token
SIGNAL token                 : HS_bus_rec;       -- Stored Token
SIGNAL SS_Buff               : SS_buff_rec;
SIGNAL CS_Buff_sig           : CS_buff_rec;
SIGNAL mem                   : cam_rec;
SIGNAL memwrite              : boolean;
SIGNAL err_inject_fifo       : err_inject_type;
SIGNAL wr_ptr                : integer RANGE 0 TO (ERROR_INJECT_DEPTH-1) := 0;
SIGNAL derror                : boolean;
SIGNAL ss_avail              : boolean;

-------------------------------------------------------------------------------
-- Is_no_space - Returns true when there is no space in the Bulk/Control buffers
--               for the current start-split.
-------------------------------------------------------------------------------
function Is_no_space(BC_buff: BC_buff_rec) return boolean is
  variable result:boolean:=FALSE;
begin
  IF (BC_buff.status = NOSPACE) THEN
      result := TRUE;
  END IF;
  return result;
end Is_no_space;

-------------------------------------------------------------------------------
-- Is_new_SS - Returns true when the current high speed start-split is new.
-------------------------------------------------------------------------------
function Is_new_SS(BC_buff: BC_buff_rec) return boolean is
  variable result:boolean:=FALSE;
begin
  IF (BC_buff.status = NU) THEN
      result := TRUE;
  END IF;
  return result;
end Is_new_SS;

-------------------------------------------------------------------------------
-- IS_old_SS - Returns true when the current high speed start-split is a retry.
-------------------------------------------------------------------------------
function Is_old_SS(BC_buff: BC_buff_rec) return boolean is
  variable result:boolean:=FALSE;
begin
  IF (BC_buff.status = OLD) THEN
      result := TRUE;
  END IF;
  return result;
end Is_old_SS;

-------------------------------------------------------------------------------
-- Issue_packet - Issue a packet onto the HS bus.
-------------------------------------------------------------------------------
procedure Issue_packet(signal HS_bus_out : out HS_bus_rec;
                               pid       : pids) IS
begin

  -- Setup HS packet based on whether its periodic or bulk.
    IF (periodic = TRUE) THEN
        HS_bus_out.ep_type      <= CS_Buff.store.ep_type;
        HS_bus_out.endpt        <= CS_Buff.store.endpt;
        HS_bus_out.dev_addr     <= CS_Buff.store.dev_addr;
```

```
        HS_bus_out.direction    <= CS_Buff.store.direction;
        HS_bus_out.datapart     <= CS_Buff.store.datapart;
        HS_bus_out.x            <= CS_Buff.store.x;     -- ????
    ELSE
        HS_bus_out.ep_type      <= cam(BC_buff.index).store.ep_type;
        HS_bus_out.endpt        <= cam(BC_buff.index).store.endpt;
        HS_bus_out.dev_addr     <= cam(BC_buff.index).store.dev_addr;
        HS_bus_out.direction    <= cam(BC_buff.index).store.direction;
        HS_bus_out.datapart     <= cam(BC_buff.index).store.datapart;
        HS_bus_out.x            <= cam(BC_buff.index).store.x;     -- ????

        -- Update bulk/control with state information which may have been updated
        -- by the complete-split state machines.
        cam(BC_buff.index).match.state := BC_buff.match.state;
    END IF;


    -- Check for Error injection when FIFO is not empty.
    IF (wr_ptr /= rd_ptr) THEN

        HS_bus_out.crc16   <= err_inject_fifo(rd_ptr).crc;
        HS_bus_out.timeout <= err_inject_fifo(rd_ptr).timeout;
        IF (err_inject_fifo(rd_ptr).pid) THEN
            HS_bus_out.pid     <= TRANS_ERR;
        ELSE
            HS_bus_out.pid      <= pid;
        END IF;

        --IF (now > error_time ) THEN
            -- Update read pointer.
            IF (rd_ptr = (ERROR_INJECT_DEPTH-1)) THEN
               rd_ptr := 0;
            ELSE
                rd_ptr := (rd_ptr + 1);
            END IF;
        --END IF;

        error_time := now;

    -- Otherwise issue packet with no errors.
    ELSE
        HS_bus_out.crc16    <= ok;
        HS_bus_out.timeout  <= FALSE;
        HS_bus_out.pid      <= pid;
    END IF;

    HS_bus_out.ready        <= TRUE;
    HS_bus_out.ready        <= FALSE after 500 ps;


end Issue_packet;

-- returns true when wrong combination of split start and last isoch out transaction
FUNCTION Bad_IsochOut (SS_Buff : SS_Buff_rec;
                       split : HS_bus_rec) RETURN boolean IS
  VARIABLE result:boolean:=FALSE;
BEGIN

  result := ((split.datapart = enddata OR split.datapart = middata) AND
             NOT(SS_Buff.lastdata = begindata OR SS_Buff.lastdata = middata)) OR
            ((split.datapart = begindata OR split.datapart = alldata) AND
              SS_Buff.isochO) OR
            ((split.datapart = middata OR split.datapart = enddata) AND NOT
              SS_Buff.isochO);

  RETURN result;
END Bad_IsochOut;

--------------------------------------------------------------------------------
-- Save - Save the Packet for use later.
--------------------------------------------------------------------------------
```

```
procedure Save(hs_bus_in : IN HS_bus_rec;
               SIGNAL hs_bus_out: OUT HS_bus_rec) IS
begin
    hs_bus_out <= hs_bus_in;
end Save;

-------------------------------------------------------------------------------
--  Compare_BC_buff - This procedure is used to look at the BC buffer to determine
--                    whether the packet should be stored. Compare_BC_buff will
--                    initialize BC_buff with the buffer location information.
-------------------------------------------------------------------------------
procedure Compare_BC_buff IS
  variable match:boolean:=FALSE;
begin

   -- Assume nospace and intialize index to 0.
   BC_buff.status := NOSPACE;
   BC_buff.index  := 0;

   FOR i IN 0 to FIFO_DEPTH-1 LOOP
       IF NOT match THEN
           -- Re-use buffer with same Device Address/End point.
           IF (token.endpt = cam(i).store.endpt AND
                token.dev_addr = cam(i).store.dev_addr AND
               ((token.direction = cam(i).store.direction AND
                 split.ep_type /= CONTROL) OR
                split.ep_type = CONTROL)) THEN

             -- If The buffer is already pending/ready this must be a retry.
             IF (cam(i).match.state = READY OR cam(i).match.state = PENDING) THEN
                 BC_buff.status := OLD;
             ELSE
                 BC_buff.status := NU;
             END IF;
             BC_buff.index := i;
             match := TRUE;

           -- Otherwise use the buffer if it's old.
           ELSIF (cam(i).match.state = OLD) THEN
             BC_buff.status := NU;
             BC_buff.index := i;
           END IF;
       END IF;
   END LOOP;

   BC_buff.match.state := cam(BC_buff.index).match.state;

end Compare_BC_buff;

-------------------------------------------------------------------------------
-- Accept_data - Store start-split into bulk/control buffer. Index is setup
--               in a previous call to Compare_BC_buff.
-------------------------------------------------------------------------------
procedure Accept_data IS
begin
    cam(BC_buff.index).store := token;
    cam(BC_buff.index).match.state := PENDING;
    BC_buff.match.state := PENDING;
end Accept_data;

-------------------------------------------------------------------------------
-- Match_split_state - This procedure finds the BC buffer location which matches
--                     the current complete-split.
-------------------------------------------------------------------------------
procedure Match_split_state IS
  variable match:boolean:=FALSE;
begin
   BC_buff.match.state  := NO_MATCH;
   BC_buff.index  := 0;

   FOR i IN 0 to FIFO_DEPTH-1 LOOP
```

```
        IF NOT match THEN
            -- Is this the buffer used for the start-split
            --  corresponding to this complete-split?
            -- If it is... store information into BC_buff and
            -- indicate match was found.

            IF (token.endpt = cam(i).store.endpt AND
                token.dev_addr = cam(i).store.dev_addr AND
                token.direction = cam(i).store.direction) THEN

              BC_buff.match.state       := cam(i).match.state;
              BC_buff.match.down_result := cam(i).match.down_result;
              BC_buff.index := i;
              match := TRUE;
            END IF;
        END IF;
    END LOOP;

    periodic := FALSE;                -- Setup Issue Packet.

end Match_split_state;

-------------------------------------------------------------------------------
-- Record an error in the SS pipeline for fowarding on the downstream bus.
-------------------------------------------------------------------------------
PROCEDURE Down_error IS
BEGIN
    derror_v := TRUE;
END Down_error;

-------------------------------------------------------------------------------
--
-------------------------------------------------------------------------------
procedure Data_into_SS_pipe IS
begin
    CS_Buff.match.state := MATCH_BUSY;
    ss_avail_v := TRUE;
end Data_into_SS_pipe;

-------------------------------------------------------------------------------
--
-------------------------------------------------------------------------------
procedure Fast_match IS
begin

   periodic := TRUE;                -- Setup Issue Packet.
end Fast_match;
```

# Appendix C
# Reset Protocol State Diagrams

This appendix presents state diagrams that provide implementation examples for the reset protocol as described in Section 7.1.7.5. These state diagrams should be considered as an example to guide implementers; the description of the reset protocol and the high-speed reset handshake in Section 7.1.7.5 is the complete required behavior. By necessity, state diagrams incorporate some implementation dependent parts that, although describing the reset protocol correctly, can also be implemented in a different way yielding similar behavior.

Any timer used in these state diagrams should have a resolution that allows it to always keep to the allowed time frame. For instance, if a timer times out between a time $T_{TIMER}$(min) and $T_{TIMER}$(max), the timer should have a minimal resolution of at least 1 clocktick in the range of $T_{TIMER}$. In a number of places, a time $T_{TIMER}$ is mentioned in a state diagram; while in the tables in Section 7.3, a range is given for this time. In that case, the time represents a chosen value in the range such that it is at least 1 clocktick of the associated timer away from the upper boundary of that range. Under these conditions, a state in the state diagrams will never miss a branch because the associated timer overstepped the time-out condition.

In the state diagrams in this appendix, a timer can be either Run, Started, or Cleared. If a timer is Run, it will update itself every clocktick. If a timer is Cleared, it is stopped and its contents are reset to zero. A timer that is Started is first cleared and then immediately run. Stopping of a timer is never done explicitly in the state diagrams.

## C.1 Downstream Facing Port State Diagram

This section describes the reset protocol state diagram for the downstream facing port.

The state diagram shown in Figure C-1 shows all the necessary and required behavior of a downstream facing port in case of a reset. As this is the initiating party in the reset protocol, the hub enters the Resetting state through a request from the host (the SetPortFeature(PORT_RESET) command). The downstream facing port then drives an SE0 to initiate the reset and at the same time starts a timer T0 to time the whole reset procedure.

If the attached device is low-speed, then the only way that reset ends is when the timer T0 times out ($T_{DRST}$) and the bus returns to idle. Whether a device is low-speed is determined prior to entering the Resetting state in the status bit PORT_LOW_SPEED. This is described in more detail in Section 11.8.2. When reset has completed, the hub enters the low-speed Enabled state.

If the attached device is full-speed and not high-speed capable, it will end reset when timer T0 expires ($T_{DRST}$) and the hub has not detected a valid upstream chirp (continuous Chirp K). It will then enter the full-speed enabled state.

Last, if the attached device is high-speed capable, it will send back an upstream chirp some time after the SE0 has been asserted on the bus. The actual time before the upstream chirp starts depends on whether the attached device was suspended or awake at the time the reset started. The loop between the blocks with “Clear timer T1” and “Run timer T1” represents the 2.5 μs ($T_{FILT}$) filtering the reset protocol asks for.

Note: The timer T1 is required to be reset after an interruption of 16 high-speed bit-times of the continuous Chirp K that makes up the upstream chirp. It may be reset by any shorter interruption.

If the filtering of the upstream chirp takes too much time, the downstream facing port may not be able to finish its downstream chirp in time to be able to end the reset procedure in time. Therefore, when timer T0 reaches beyond the time $T_{UCHEND}$ (time to detect an upstream chirp), the hub is put in a wait state, which it leaves after the timer has timed out the complete reset protocol ($T_{DRST}$). It will then enter the full-speed enabled state.

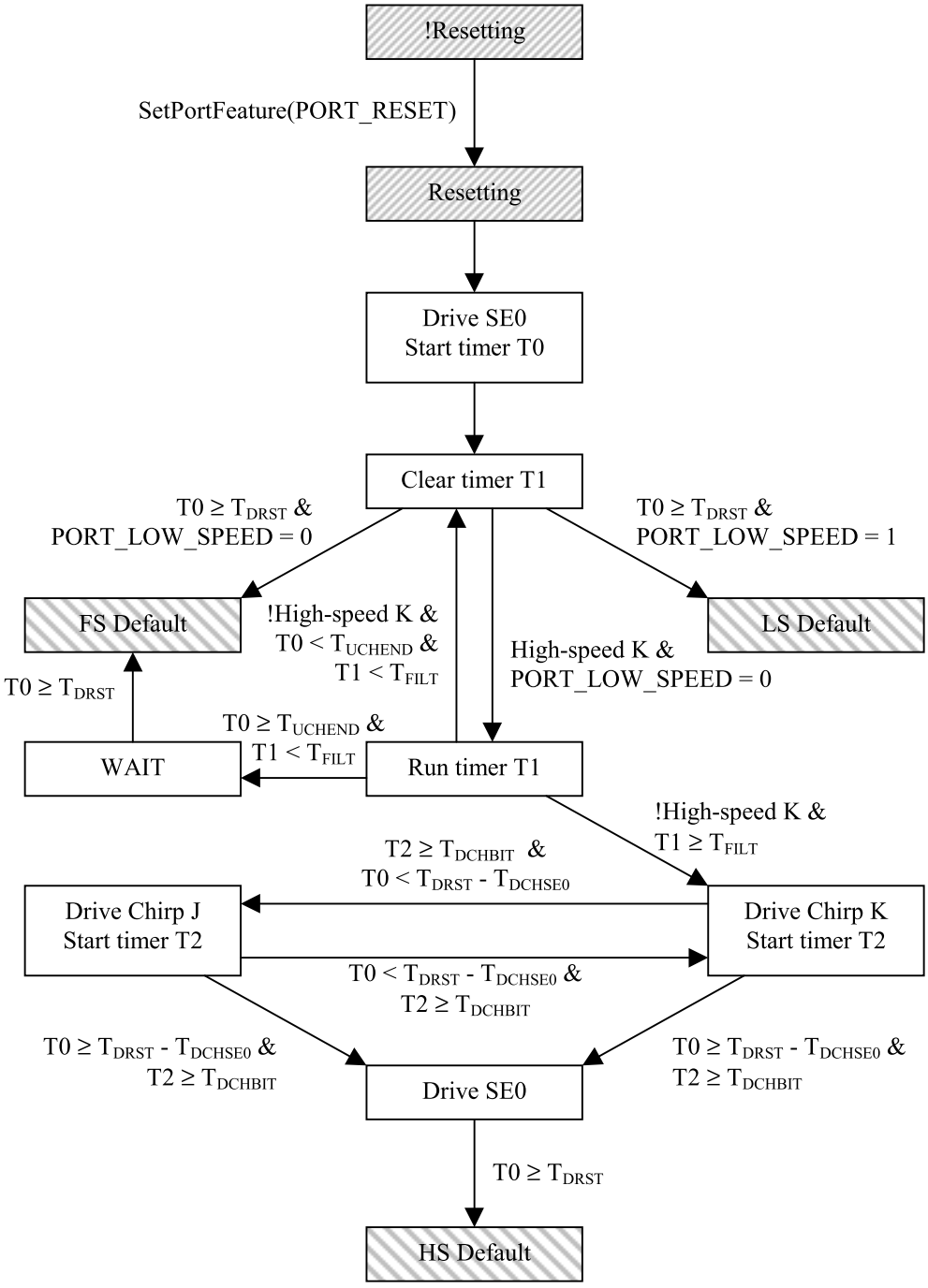



**Figure C-1. Downstream Facing Port Reset Protocol State Diagram**

When the downstream-facing port has successfully detected an upstream chirp, it will start transmitting the downstream chirp as soon as it has seen the bus leave the Chirp K state. This end of the upstream chirp will return the bus to the SE0 state. So immediately (actually within 100 µs ($T_{WTDCH}$) after the end of the upstream chirp according to Section 7.1.7.5), the hub drives a Chirp K for 40 to 60 µs ($T_{DCHBIT}$), then a Chirp J for 40 to 60 µs, then a Chirp K, etc. It continues with this alternating sequence until timer T0 has come within 100 to 500 µs ($T_{DCHSE0}$) of the end of reset ($T_{DRST}$). When this time is reached, the downstream-facing port finishes the

40 to 60 µs of continuous signaling it was busy with when the timer T0 exceeds the value of $T_{DRST} - T_{DCHSE0}$ before driving SE0 until the end of reset.

## C.2 Upstream Facing Port State Diagram

This section describes the reset protocol state diagrams for the upstream facing port. The state diagram for the upstream facing port is more complicated than the diagram for the downstream facing port as the device can be in any possible state when it receives a reset signal. Therefore, the state diagram has been split into two parts:

- The reset detection state diagram which describes the way a device reacts to reset signaling on its upstream facing port (see Figure C-2)
- The reset handshake state diagram that explains how a high-speed capable device performs a handshake procedure with the hub upstream to communicate each others high-speed capabilities and have both enter a high-speed state at the end of reset (see Figure C-3)

Therefore, all of these states must be covered in the diagram. Also, the fact that for a high-speed capable device a suspend is initially indistinguishable from a reset requires that the state diagram for the upstream facing port addresses the suspend procedure as well.

At the start of the reset, we can be any possible state, but we can collect them into three groups, where each group is handled differently, but all states in the same group handle reset in the same way. The states are as follows:

- Suspended
- Powered, FS Default, FS Address, and FS Configured
- HS Default, HS Address, and HS Configured

These groups of states correspond to an identical list of possibilities as described in Section 7.1.7.5 under item 3 of the reset protocol.

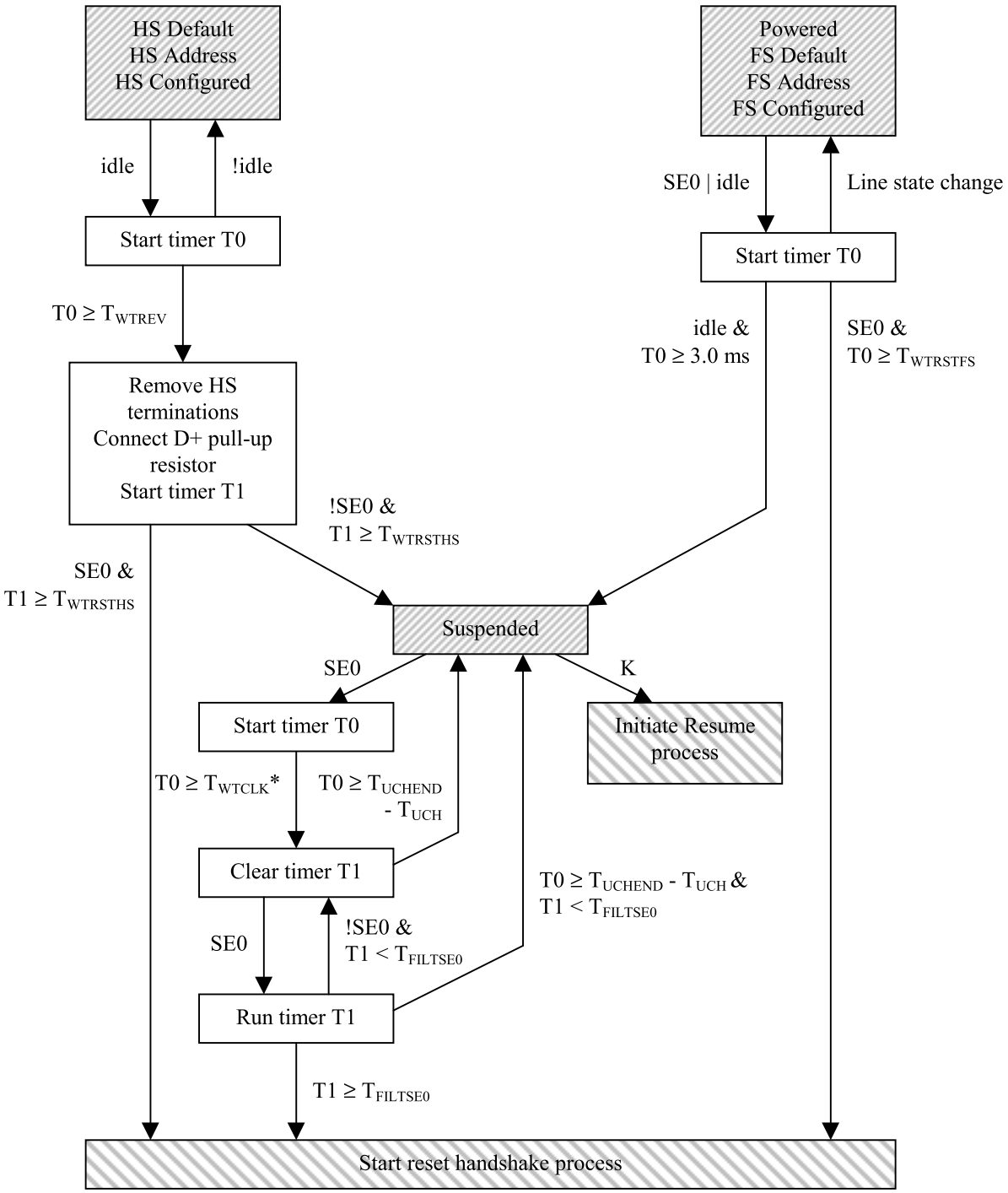
### C.2.1 Reset From Suspended State

As can be seen from Figure C-2, the device wakes up from the Suspended state as soon as it sees a K or an SE0 on the bus. A J would be indistinguishable from idle on the bus that a suspended device sees normally. On seeing a K, the device will initiate a resume process. For the details of this process, see Section 7.1.7.7. On seeing an SE0, the device could enter the reset handshake procedure, so it starts timer T0.

The actual reset handshake is only started after seeing a continuous assertion of SE0 for at least 2.5 µs ($T_{FILTSE0}$). The loop between the blocks with “Clear timer T1” and “Run timer T1” represents this filtering. If the device has not detected a continuous SE0 before timer T0 exceeds the value of $T_{UCHEND} - T_{UCH}$, the device goes back into the Suspended state.

A device coming from suspend most probably had its high-speed clock stopped to meet the power requirements for a suspended device (see Section 7.2.3). Therefore, it may take some time to let the clock settle to a level of operation where it is able to perform the reset detection and handshake with enough precision. In the state diagram, a time symbol $T_{WTCLK}$ is used to have the device wait for a stable clock. This symbol is not part of the USB 2.0 specification and does not appear in Chapter 7. It is an implementation specific detail of the reset detection state diagram for the upstream facing port, where it is marked with a asterisk (*). $T_{WTCLK}$ should have a value somewhere between 0 and 5.0 ms. This allows at least 1.0 ms time to detect the continuous SE0.

If the device has seen an SE0 signal on the bus for at least $T_{FILTSE0}$, then it can safely assume to have detected a reset and can start the reset handshake.




**(*)** **Note:** $T_{WTCLK}$ is a symbol that is only used in this state diagram. It is not part of the USB 2.0 specification and does not appear in Chapter 7. It is an implementation specific detail of this state diagram. See Section C.2.1 for a detailed description.

**Figure C-2. Upstream Facing Port Reset Detection State Diagram**

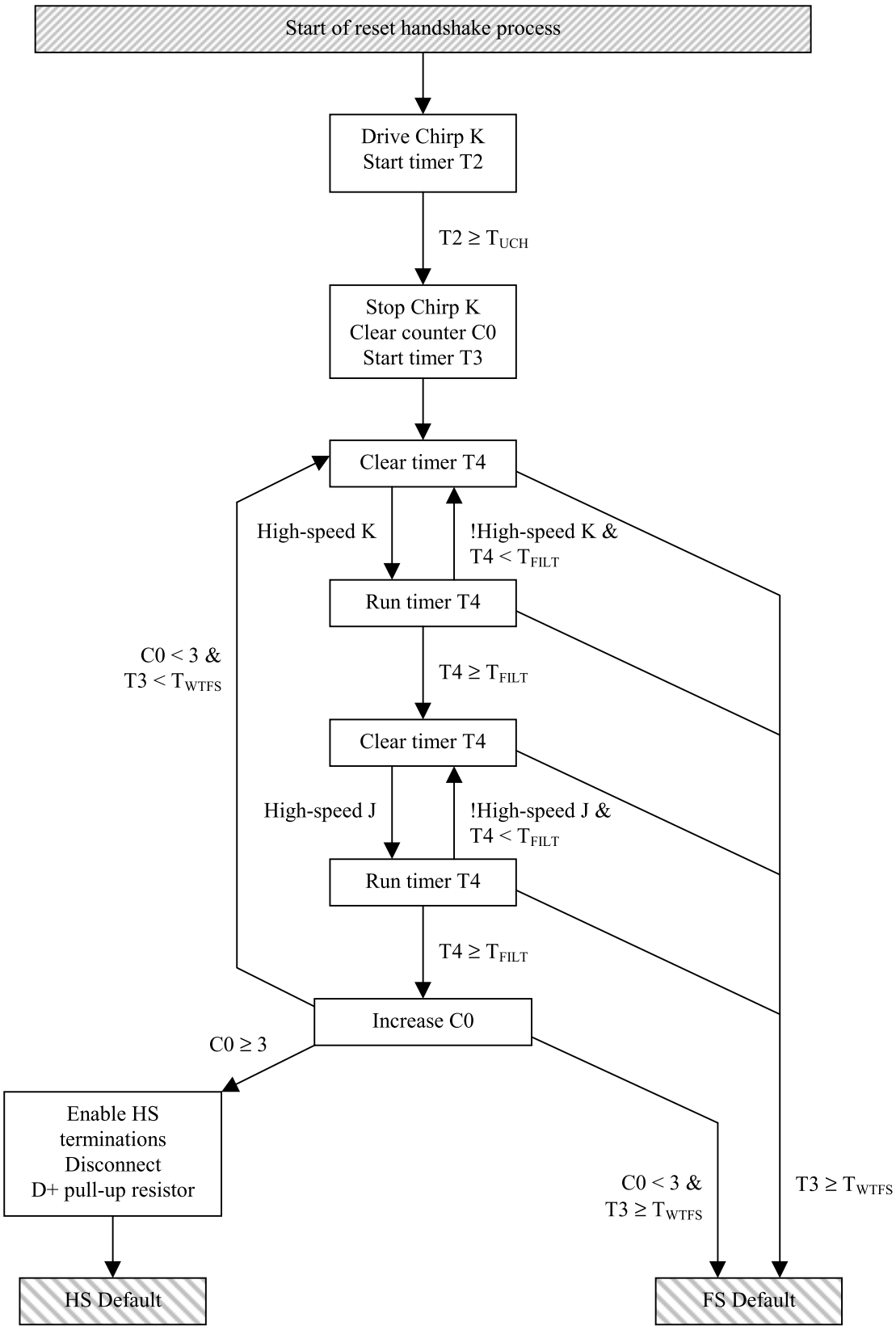



**Figure C-3. Upstream Facing Port Reset Handshake State Diagram**

## C.2.2 Reset From Full-speed Non-suspended State

Timer T0 is started when seeing an SE0 or idle state from a full-speed Non-suspended state.

If a J (idle) is detected and the timer T0 exceeds the value of 3.0 ms while no change has been detected in the state of the bus, the device is suspended.

If an SE0 is detected and the timer T0 times out the value of $T_{WTRSTFS}$ (between 2.5 µs minimum and 3.0 ms maximum) while no change has been detected in the SE0 state, the device can start the reset handshake. On any line state change, the device aborts the detection of reset or suspend from upstream and returns to its previous state.

## C.2.3 Reset From High-speed Non-suspended State

Timer T0 is started when seeing a high-speed idle on the bus from a high-speed Non-suspended state. If anything else than idle is detected on the bus, the device aborts detection of a reset and returns to its previous state. When timer T0 exceeds the value of $T_{WTREV}$ (between 3.0 ms minimally and 3.125 ms maximally), the device reverts to full-speed by switching off its high-speed terminations and connecting the D+ pull-up resistor to the D+ line.

The reset protocol allows some time for debouncing and settling of the lines in the new state ($T_{WTRSTHS}$). After this time, the line should be sampled to see whether the device should be suspended (on detecting a full-speed idle) or reset (on detecting SE0).

If an idle was detected, the device should suspend; if an SE0 was detected, the device can start the reset handshake.

If something other than an idle or an SE0, in other words, a K, was detected, the device will also enter the suspended state. However, on seeing the K, the device will immediately resume, effectively returning to the high-speed state.

## C.2.4 Reset Handshake

At this point, the behavior of devices has become independent of the initial state they were in when the reset started. The reset handshake is started by the device, when it sends an upstream chirp that is at least 1.0 ms long and stops before the timer T0 hits the 7.0 ms mark. Note: This is the same timer T0 that was started in the reset detection state diagram in Figure C-2.

A choice of implementation is available here. The one presented in the state diagram in Figure C-3 is where a timer T2 is started when the Chirp K is asserted to time the minimum required duration of the upstream chirp. The Chirp K is stopped when timer T2 exceeds the value of $T_{UCH}$. Another approach would be to wait until the timer T0 exceeds the value of $T_{UCHEND}$, before ending the upstream chirp. Both conform to the requirements of the reset protocol in Section 7.1.7.5, and the choice may depend on the particular application.

As soon as the upstream chirp has ended, the device starts listening for the downstream chirp. In order to detect at least a K-J-K-J-K-J pattern, it first starts looking for a continuously asserted Chirp K. The method employed in this state diagram is counting the number of K-J transitions. Here K and J are actually Chirp K and Chirp J, respectively, asserted continuously for at least 2.5 µs ($T_{FILT}$).

Continuous assertion is determined by the loop between the “Clear timer T4” and “Run timer T4”. This is similar to the method used in the downstream facing port state diagram in Figure C-1 to detect the upstream chirp. After this, a continuous Chirp J is detected in the same manner, most likely, even using the same hardware. Now we have detected one K-J transition. Until we have detected three K-J transitions in the same way, we will not revert to high-speed.

The whole procedure of detecting the downstream chirp is timed by timer T3 which requires the device to perform the detection of the K-J-K-J-K-J for at least 1.0 ms, but at most 2.5 ms. If the device is unable to detect a sufficient number of K-J transitions before the timer T3 times out at $T_{WTFS}$, the device enters the full-speed default state. Reset ends when the bus state changes from SE0 to idle. The time $T_{WTFS}$ is given a wide range to

allow sufficient leverage for a device which has awoke from suspend to use its (possible not yet stable) clock to time this duration reliably.

Reversion to high-speed when the device has detected the K-J-K-J-K-J pattern is accomplished by enabling the high-speed terminations and disconnecting the pull-up resistor from the D+-line. According to Section 7.1.7.5, you may wait up to 500 μs before actually reverting to high-speed, but in this state diagram, this reversion is done immediately after detection of three K-J transitions. After this switching of terminations and pull-up, the device enters the high-speed Default state. The end of reset is signified by the first packet that is received, most likely an SOF packet.

# Index

## B

## C

## D

## E

## F

## G

## H

## I

## J



## K



## L

## M

# N

# O



# P

## Q



## R

## S

## T

## U

## V



## W

## Z