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

Table 9-6. Standard Feature Selectors

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

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

9.4.1 Clear Feature

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

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

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

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

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

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

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

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

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

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

9.4.2 Get Configuration

This request returns the current device configuration value.

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

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

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

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

Address state: The value zero must be returned.

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

9.4.3 Get Descriptor

This request returns the specified descriptor if the descriptor exists.

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

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

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

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

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

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

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

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

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

9.4.4 Get Interface

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

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

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

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

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

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

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

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

9.4.5 Get Status

This request returns status for the specified recipient.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.4.6 Set Address

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

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

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

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

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

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

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

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

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

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

9.4.7 Set Configuration

This request sets the device configuration.

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

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

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

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

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

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

9.4.8 Set Descriptor

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

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

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

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

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

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

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

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

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

9.4.9 Set Feature

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

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

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

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

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

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

Table 9-7. Test Mode Selectors

Value	Description
00H	Reserved
01H	Test_J
02H	Test_K
03H	Test_SE0_NAK
04H	Test_Packet
05H	Test_Force_Enable
06H-3FH	Reserved for standard test selectors
3FH-BFH	Reserved
C0H-FFH	Reserved for vendor-specific test modes.

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

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

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

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

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

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

9.4.10 Set Interface

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

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

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

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

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

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

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

9.4.11 Synch Frame

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

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

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

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

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

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

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

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

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

9.5 Descriptors

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

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

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

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

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

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

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

9.6 Standard USB Descriptor Definitions

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

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

9.6.1 Device

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

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

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

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

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

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

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

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

Table 9-8. Standard Device Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	DEVICE Descriptor Type
2	*bcdUSB*	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e., 2.10 is 210H). This field identifies the release of the USB Specification with which the device and its descriptors are compliant.
4	*bDeviceClass*	1	Class	Class code (assigned by the USB-IF). If this field is reset to zero, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and FEH, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to FFH, the device class is vendor-specific.
5	*bDeviceSubClass*	1	SubClass	Subclass code (assigned by the USB-IF). These codes are qualified by the value of the *bDeviceClass* field. If the *bDeviceClass* field is reset to zero, this field must also be reset to zero. If the *bDeviceClass* field is not set to FFH, all values are reserved for assignment by the USB-IF.

Table 9-8. Standard Device Descriptor (Continued)

Offset	Field	Size	Value	Description
6	*bDeviceProtocol*	1	Protocol	Protocol code (assigned by the USB-IF). These codes are qualified by the value of the *bDeviceClass* and the *bDeviceSubClass* fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use class-specific protocols on a device basis. However, it may use class-specific protocols on an interface basis. If this field is set to FFH, the device uses a vendor-specific protocol on a device basis.
7	*bMaxPacketSize0*	1	Number	Maximum packet size for endpoint zero (only 8, 16, 32, or 64 are valid)
8	*idVendor*	2	ID	Vendor ID (assigned by the USB-IF)
10	*idProduct*	2	ID	Product ID (assigned by the manufacturer)
12	*bcdDevice*	2	BCD	Device release number in binary-coded decimal
14	*iManufacturer*	1	Index	Index of string descriptor describing manufacturer
15	*iProduct*	1	Index	Index of string descriptor describing product
16	*iSerialNumber*	1	Index	Index of string descriptor describing the device's serial number
17	*bNumConfigurations*	1	Number	Number of possible configurations

9.6.2 Device_Qualifier

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

Table 9-9. Device_Qualifier Descriptor

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

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

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

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

9.6.3 Configuration

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

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

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

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

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

Table 9-10. Standard Configuration Descriptor

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

Table 9-10. Standard Configuration Descriptor (Continued)

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

9.6.4 Other_Speed_Configuration

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

Table 9-11. Other_Speed_Configuration Descriptor

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

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

9.6.5 Interface

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

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

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

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

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

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

Table 9-12. Standard Interface Descriptor

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

Table 9-12. Standard Interface Descriptor (Continued)

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

9.6.6 Endpoint

Each endpoint used for an interface has its own descriptor. This descriptor contains the information required by the host to determine the bandwidth requirements of each endpoint. An endpoint descriptor is always returned as part of the configuration information returned by a GetDescriptor(Configuration) request. An endpoint descriptor cannot be directly accessed with a GetDescriptor() or SetDescriptor() request. There is never an endpoint descriptor for endpoint zero. Table 9-13 shows the standard endpoint descriptor.

Table 9-13. Standard Endpoint Descriptor

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

Table 9-13. Standard Endpoint Descriptor (Continued)

Offset	Field	Size	Value	Description
3	*bmAttributes*	1	Bitmap	This field describes the endpoint's attributes when it is configured using the *bConfigurationValue*. Bits 1..0: Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt If not an isochronous endpoint, bits 5..2 are reserved and must be set to zero. If isochronous, they are defined as follows: Bits 3..2: Synchronization Type 00 = No Synchronization 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4: Usage Type 00 = Data endpoint 01 = Feedback endpoint 10 = Implicit feedback Data endpoint 11 = Reserved Refer to Chapter 5 for more information. All other bits are reserved and must be reset to zero. Reserved bits must be ignored by the host.

Table 9-13. Standard Endpoint Descriptor (Continued)

Offset	Field	Size	Value	Description
4	*wMaxPacketSize*	2	Number	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. For isochronous endpoints, this value is used to reserve the bus time in the schedule, required for the per-(micro)frame data payloads. The pipe may, on an ongoing basis, actually use less bandwidth than that reserved. The device reports, if necessary, the actual bandwidth used via its normal, non-USB defined mechanisms. For all endpoints, bits 10..0 specify the maximum packet size (in bytes). For high-speed isochronous and interrupt endpoints: Bits 12..11 specify the number of additional transaction opportunities per microframe: 00 = None (1 transaction per microframe) 01 = 1 additional (2 per microframe) 10 = 2 additional (3 per microframe) 11 = Reserved Bits 15..13 are reserved and must be set to zero. Refer to Chapter 5 for more information.
6	*bInterval*	1	Number	Interval for polling endpoint for data transfers. Expressed in frames or microframes depending on the device operating speed (i.e., either 1 millisecond or 125 µs units). For full-/high-speed isochronous endpoints, this value must be in the range from 1 to 16. The *bInterval* value is used as the exponent for a $2^{(bInterval-1)}$ value; e.g., a *bInterval* of 4 means a period of 8 ($2^{(4-1)}$). For full-/low-speed interrupt endpoints, the value of this field may be from 1 to 255. For high-speed interrupt endpoints, the *bInterval* value is used as the exponent for a $2^{(bInterval-1)}$ value; e.g., a *bInterval* of 4 means a period of 8 ($2^{(4-1)}$). This value must be from 1 to 16. For high-speed bulk/control OUT endpoints, the *bInterval* must specify the maximum NAK rate of the endpoint. A value of 0 indicates the endpoint never NAKs. Other values indicate at most 1 NAK each *bInterval* number of microframes. This value must be in the range from 0 to 255. See Chapter 5 description of periods for more detail.

The *bmAttributes* field provides information about the endpoint's Transfer Type (bits 1..0) and Synchronization Type (bits 3..2). In addition, the Usage Type bit (bits 5..4) indicate whether this is an endpoint used for normal data transfers (bits 5..4=00B), whether it is used to convey explicit feedback information for one or more data endpoints (bits 5..4=01B) or whether it is a data endpoint that also serves

as an implicit feedback endpoint for one or more data endpoints (bits 5..4=10B). Bits 5..2 are only meaningful for isochronous endpoints and must be reset to zero for all other transfer types.

If the endpoint is used as an explicit feedback endpoint (bits 5..4=01B), then the Transfer Type must be set to isochronous (bits1..0 = 01B) and the Synchronization Type must be set to No Synchronization (bits 3..2=00B).

A feedback endpoint (explicit or implicit) needs to be associated with one (or more) isochronous data endpoints to which it provides feedback service. The association is based on endpoint number matching. A feedback endpoint always has the opposite direction from the data endpoint(s) it services. If multiple data endpoints are to be serviced by the same feedback endpoint, the data endpoints must have ascending ordered–but not necessarily consecutive–endpoint numbers. The first data endpoint and the feedback endpoint must have the same endpoint number (and opposite direction). This ensures that a data endpoint can uniquely identify its feedback endpoint by searching for the first feedback endpoint that has an endpoint number equal or less than its own endpoint number.

Example: Consider the extreme case where there is a need for five groups of OUT asynchronous isochronous endpoints and at the same time four groups of IN adaptive isochronous endpoints. Each group needs a separate feedback endpoint and the groups are composed as shown in Figure 9-7.

OUT Group	Nr of OUT Endpoints	IN Group	Nr of IN Endpoints
1	1	6	1
2	2	7	2
3	2	8	3
4	3	9	4
5	3		

Figure 9-7. Example of Feedback Endpoint Numbers

The endpoint numbers can be intertwined as illustrated in Figure 9-8.

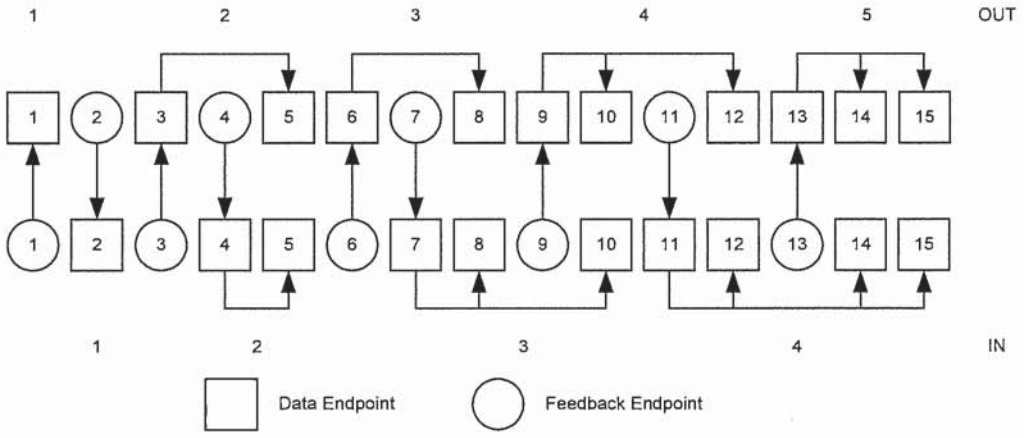

Figure 9-8. Example of Feedback Endpoint Relationships

High-speed isochronous and interrupt endpoints use bits 12..11 of *wMaxPacketSize* to specify multiple transactions for each microframe specified by *bInterval*. If bits 12..11 of *wMaxPacketSize* are zero, the maximum packet size for the endpoint can be any allowed value (as defined in Chapter 5). If bits 12..11 of *wMaxPacketSize* are not zero (0), the allowed values for *wMaxPacketSize* bits 10..0 are limited as shown in Table 9-14.

Table 9-14. Allowed wMaxPacketSize Values for Different Numbers of Transactions per Microframe

wMaxPacketSize bits 12..11	*wMaxPacketSize* bits 10..0 Values Allowed
00	1 – 1024
01	513 – 1024
10	683 – 1024
11	N/A; reserved

For high-speed bulk and control OUT endpoints, the *bInterval* field is only used for compliance purposes; the host controller is not required to change its behavior based on the value in this field.

9.6.7 String

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

String descriptors use UNICODE encodings as defined by *The Unicode Standard, Worldwide Character Encoding, Version 3.0*, The Unicode Consortium, Addison-Wesley Publishing Company, Reading, Massachusetts (URL: http://www.unicode.com). The strings in a USB device may support multiple languages. When requesting a string descriptor, the requester specifies the desired language using a sixteen-bit language ID (LANGID) defined by the USB-IF. The list of currently defined USB LANGIDs can be found at http://www.usb.org/developers/docs.html. String index zero for all languages returns a string descriptor that contains an array of two-byte LANGID codes supported by the device. Table 9-15 shows the LANGID code array. A USB device may omit all string descriptors. USB devices that omit all string descriptors must not return an array of LANGID codes.

The array of LANGID codes is not NULL-terminated. The size of the array (in bytes) is computed by subtracting two from the value of the first byte of the descriptor.

Table 9-15. String Descriptor Zero, Specifying Languages Supported by the Device

Offset	Field	Size	Value	Description
0	*bLength*	1	N+2	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	STRING Descriptor Type
2	*wLANGID[0]*	2	Number	LANGID code zero
...	...	...	...	...
N	*wLANGID[x]*	2	Number	LANGID code *x*

The UNICODE string descriptor (shown in Table 9-16) is not NULL-terminated. The string length is computed by subtracting two from the value of the first byte of the descriptor.

Table 9-16. UNICODE String Descriptor

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

9.7 Device Class Definitions

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

9.7.1 Descriptors

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

9.7.2 Interface(s) and Endpoint Usage

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

9.7.3 Requests

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

Chapter 10
USB Host: Hardware and Software

The USB interconnect supports data traffic between a host and a USB device. This chapter describes the host interfaces necessary to facilitate USB communication between a software client, resident on the host, and a function implemented on a device. The implementation described in this chapter is not required. This implementation is provided as an example to illustrate the host system behavior expected by a USB device. A host system may provide a different host software implementation as long as a USB device experiences the same host behavior.

10.1 Overview of the USB Host

10.1.1 Overview

The basic flow and interrelationships of the USB communications model are shown in Figure 10-1.

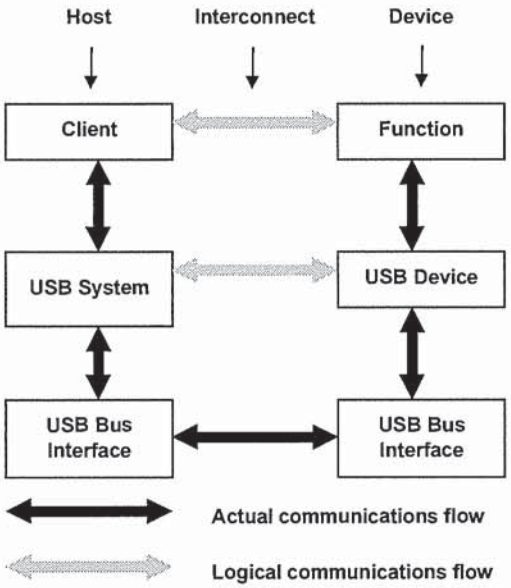

Figure 10-1. Interlayer Communications Model

The host and the device are divided into the distinct layers depicted in Figure 10-1. Vertical arrows indicate the actual communication on the host. The corresponding interfaces on the device are implementation-specific. All communications between the host and device ultimately occur on the physical USB wire. However, there are logical host-device interfaces between each horizontal layer. These communications, between client software resident on the host and the function provided by the device, are typified by a contract based on the needs of the application currently using the device and the capabilities provided by the device.

This client-function interaction creates the requirements for all of the underlying layers and their interfaces.

This chapter describes this model from the point of view of the host and its layers. Figure 10-2 illustrates, based on the overall view introduced in Chapter 5, the host's view of its communication with the device.

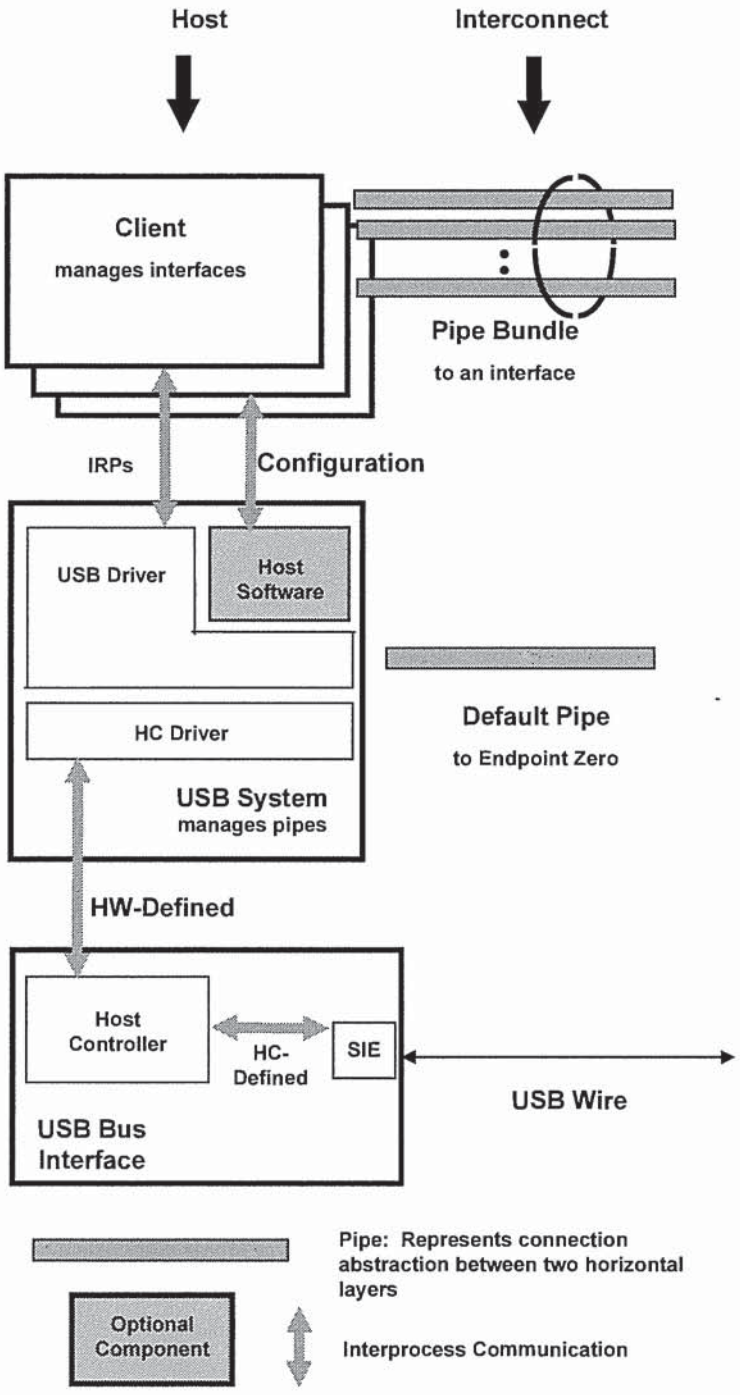

Figure 10-2. Host Communications

There is only one host for each USB. The major layers of a host consist of the following:

- USB bus interface
- USB System
- Client

The USB bus interface handles interactions for the electrical and protocol layers (refer to Chapter 7 and Chapter 8). From the interconnect point of view, a similar USB bus interface is provided by both the USB device and the host, as exemplified by the Serial Interface Engine (SIE). On the host, however, the USB bus interface has additional responsibilities due to the unique role of the host on the USB and is implemented as the Host Controller. The Host Controller has an integrated root hub providing attachment points to the USB wire.

The USB System uses the Host Controller to manage data transfers between the host and USB devices. The interface between the USB System and the Host Controller is dependent on the hardware definition of the Host Controller. The USB System, in concert with the Host Controller, performs the translation between the client's view of data transfers and the USB transactions appearing on the interconnect. This includes the addition of any USB feature support such as protocol wrappers. The USB System is also responsible for managing USB resources, such as bandwidth and bus power, so that client access to the USB is possible.

The USB System has three basic components:

- Host Controller Driver
- USB Driver
- Host Software

The Host Controller Driver (HCD) exists to more easily map the various Host Controller implementations into the USB System, such that a client can interact with its device without knowing to which Host Controller the device is connected. The USB Driver (USBD) provides the basic host interface (USBDI) for clients to USB devices. The interface between the HCD and the USBD is known as the Host Controller Driver Interface (HCDI). This interface is never available directly to clients and thus is not defined by the USB Specification. A particular HCDI is, however, defined by each operating system that supports various Host Controller implementations.

The USBD provides data transfer mechanisms in the form of I/O Request Packets (IRPs), which consist of a request to transport data across a specific pipe. In addition to providing data transfer mechanisms, the USBD is responsible for presenting to its clients an abstraction of a USB device that can be manipulated for configuration and state management. As part of this abstraction, the USBD owns the default pipe (see Chapter 5 and Chapter 9) through which all USB devices are accessed for the purposes of standard USB control. This default pipe represents a logical communication between the USBD and the abstraction of a USB device as shown in Figure 10-2.

In some operating systems, additional non-USB System Software is available that provides configuration and loading mechanisms to device drivers. In such operating systems, the device driver shall use the provided interfaces instead of directly accessing the USBDI mechanisms.

The client layer describes all the software entities that are responsible for directly interacting with USB devices. When each device is attached to the system, these clients might interact directly with the peripheral hardware. The shared characteristics of the USB place USB System Software between the client and its device; that is, a client cannot directly access the device's hardware.

Overall, the host layers provide the following capabilities:

- Detecting the attachment and removal of USB devices
- Managing USB standard control flow between the host and USB devices
- Managing data flow between the host and USB devices
- Collecting status and activity statistics
- Controlling the electrical interface between the Host Controller and USB devices, including the provision of a limited amount of power

The following sections describe these responsibilities and the requirements placed on the USBDI in greater detail. The actual interfaces used for a specific combination of host platform and operating system are described in the appropriate operating system environment guide.

All hubs (see Chapter 11) report internal status changes and their port change status via the status change pipe. This includes a notification of when a USB device is attached to or removed from one of their ports. A USBD client generically known as the hub driver receives these notifications as owner of the hub's Status Change pipe. For device attachments, the hub driver then initiates the device configuration process. In some systems, this hub driver is a part of the host software provided by the operating system for managing devices.

10.1.2 Control Mechanisms

Control information may be passed between the host and a USB device using in-band or out-of-band signaling. In-band signaling mixes control information with data in a pipe outside the awareness of the host. Out-of-band signaling places control information in a separate pipe.

There is a message pipe called the default pipe for each attached USB device. This logical association between a host and a USB device is used for USB standard control flow such as device enumeration and configuration. The default pipe provides a standard interface to all USB devices. The default pipe may also be used for device-specific communications, as mediated by the USBD, which owns the default pipes of all of the USB devices.

A particular USB device may allow the use of additional message pipes to transfer device-specific control information. These pipes use the same communications protocol as the default pipe, but the information transferred is specific to the USB device and is not standardized by the USB Specification.

The USBD supports the sharing of the default pipe, which it owns and uses, with its clients. It also provides access to any other control pipes associated with the device.

10.1.3 Data Flow

The Host Controller is responsible for transferring streams of data between the host and USB devices. These data transfers are treated as a continuous stream of bytes. The USB supports four basic types of data transfers:

- Control transfers
- Isochronous transfers
- Interrupt transfers
- Bulk transfers

For additional information on transfer types, refer to Chapter 5.

Each device presents one or more interfaces that a client may use to communicate with the device. Each interface is composed of zero or more pipes that individually transfer data between the client and a particular endpoint on the device. The USBD establishes interfaces and pipes at the explicit request of the Host Software. The Host Controller provides service based on parameters provided by the Host Software when the configuration request is made.

A pipe has several characteristics based on the delivery requirements of the data to be transferred. Examples of these characteristics include the following:

- The rate at which data needs to be transferred
- Whether data is provided at a steady rate or sporadically
- How long data may be delayed before delivery
- Whether the loss of data being transferred is catastrophic

A USB device endpoint describes the characteristics required for a specific pipe. Endpoints are described as part of a USB device's characterization information. For additional details, refer to Chapter 9.

10.1.4 Collecting Status and Activity Statistics

As a common communicant for all control and data transfers between the host and USB devices, the USB System and the Host Controller are well-positioned to track status and activity information. Such information is provided upon request to the Host Software, allowing that software to manage status and activity information. This specification does not identify any specific information that should be tracked or require any particular format for reporting activity and status information.

10.1.5 Electrical Interface Considerations

The host provides power to USB devices attached to the root hub. The amount of power provided by a port is specified in Chapter 7.

10.2 Host Controller Requirements

In all implementations, Host Controllers perform the same basic duties with regard to the USB and its attached devices. These basic duties are described below.

The Host Controller has requirements from both the host and the USB. The following is a brief overview of the functionality provided. Each capability is discussed in detail in subsequent sections.

State Handling	As a component of the host, the Host Controller reports and manages its states.
Serializer/Deserializer	For data transmitted from the host, the Host Controller converts protocol and data information from its native format to a bit stream transmitted on the USB. For data being received into the host, the reverse operation is performed.
(micro)frame Generation	The Host Controller produces SOF tokens at a period of 1 ms when operating with full-speed devices, and at a period of 125 µs when operating with high-speed devices.
Data Processing	The Host Controller processes requests for data transmission to and from the host.
Protocol Engine	The Host Controller supports the protocol specified by the USB.
Transmission Error Handling	All Host Controllers exhibit the same behavior when detecting and reacting to the defined error categories.
Remote Wakeup	All Host Controllers must have the ability to place the bus into the Suspended state and to respond to bus wakeup events.

Root Hub	The root hub provides standard hub function to link the Host Controller to one or more USB ports.
Host System Interface	Provides a high-speed data path between the Host Controller and host system.

The following sections present a more detailed discussion of the required capabilities of the Host Controller.

10.2.1 State Handling

The Host Controller has a series of states that the USB System manages. Additionally, the Host Controller provides the interface to the following two areas of USB-relevant state:

- State change propagation
- Root hub

The root hub presents to the hub driver the same standard states as other USB devices. The Host Controller supports these states and their transitions for the hub. For detailed discussions of USB states, including their interrelations and transitions, refer to Chapter 9.

The overall state of the Host Controller is inextricably linked with that of the root hub and of the overall USB. Any Host Controller state changes that are visible to attached devices must be reflected in the corresponding device state change information such that the resulting Host Controller and device states are consistent.

USB devices request a wakeup through the use of resume signaling (refer to Chapter 7). The Host Controller must notify the rest of the host of a resume event through a mechanism or mechanisms specific to that system's implementation. The Host Controller itself may cause a resume event through the same signaling method.

10.2.2 Serializer/Deserializer

The actual transmission of data across the physical USB takes places as a serial bit stream. A Serial Interface Engine (SIE), whether implemented as part of the host or a USB device, handles the serialization and deserialization of USB transmissions. On the host, this SIE is part of the Host Controller.

10.2.3 Frame and Microframe Generation

It is the Host Controller's responsibility to partition USB time into quantities called "frames" when operating with full-speed devices, and "microframes" when operating with high-speed devices. Frames and microframes are created by the Host Controller through issuing Start-of-Frame (SOF) tokens as shown in Figure 10-3. The SOF token is the first transmission in the (micro)frame period. Host controllers operating with high-speed devices generate SOF tokens at 125 µs intervals. Host controllers operating with full-speed devices generate SOF tokens at 1.00 ms intervals. After issuing an SOF token, the Host Controller is free to transmit other transactions for the remainder of the (micro)frame period. When the Host Controller is in its normal operating state, SOF tokens must be continuously generated at appropriate periodic rate, regardless of other bus activity or lack thereof. If the Host Controller enters a state where it is not providing power on the bus, it must not generate SOFs. When the Host Controller is not generating SOFs, it may enter a power-reduced state.

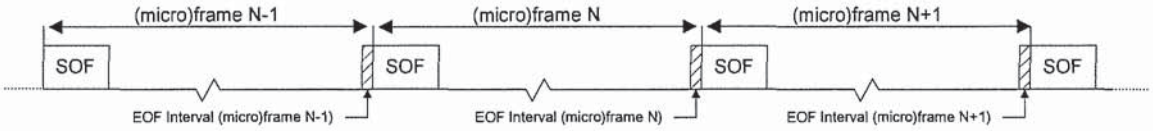

Figure 10-3. Frame and Microframe Creation

The SOF token holds the highest priority access to the bus. Babble circuitry in hubs electrically isolates any active transmitters during the End-of-microframe or End-of-Frame (EOF) interval, providing an idle bus for the SOF transmission.

The Host Controller maintains the current (micro)frame number that may be read by the USB System.

The following apply to the current (micro)frame number maintained by the host:

- Used to uniquely identify one (micro)frame from another
- Incremented at the end of every (micro)frame period
- Valid through the subsequent (micro)frame

Host controllers operating with full-speed devices maintain a current frame number (at least 11 bits) that increments at a 1 ms period. The host transmits the lower 11 bits of the current frame number in each SOF token transmission.

Host controllers operating with high-speed devices maintain a current microframe number (at least 14 bits) that increments at a 125 µs period. The host transmits bits 3 through 13 of the current microframe number in each SOF token transmission. This results in the same SOF packet value being transmitted for eight consecutive microframes before the SOF packet value increments.

When requested from the Host Controller, the current (micro)frame number is the (micro)frame number in existence at the time the request was fulfilled. The current (micro)frame number as returned by the host (Host Controller or HCD) is at least 32 bits, although the Host Controller itself is not required to maintain more than 11 bits when operating with full-speed devices or 14 bits when operating with high-speed devices.

The Host Controller shall cease transmission during the EOF interval. When the EOF interval begins, any transactions scheduled specifically for the (micro)frame that has just passed are retired. If the Host Controller is executing a transaction at the time the EOF interval is encountered, the Host Controller terminates the transaction.

10.2.4 Data Processing

The Host Controller is responsible for receiving data from the USB System and sending it to the USB and for receiving data from the USB and sending it to the USB System. The particular format used for the data communications between the USB System and the Host Controller is implementation specific, within the rules for transfer behavior described in Chapter 5.

10.2.5 Protocol Engine

The Host Controller manages the USB protocol level interface. It inserts the appropriate protocol information for outgoing transmissions. It also strips and interprets, as appropriate, the incoming protocol information.

10.2.6 Transmission Error Handling

The Host Controller must be capable of detecting the following transmission error conditions, which are defined from the host's point of view:

- Timeout conditions after a host-transmitted token or packet. These errors occur when the addressed endpoint is unresponsive or when the structure of the transmission is so badly damaged that the targeted endpoint does not recognize it.
- Data errors resulting in missing or invalid transmissions:
  - The Host Controller is unable to completely send or receive a packet for host specific reasons, for example, a transmission extending beyond EOF or a lack of resources available to the Host Controller.
  - An invalid CRC field on a received data packet.
- Protocol errors:
  - An invalid handshake PID, such as a malformed or inappropriate handshake
  - A false EOP
  - A bit stuffing error

For each bulk, control, and interrupt transaction, the host must maintain an error count tally. Errors result from the conditions described above, not as a result of an endpoint NAKing a request. This value reflects the number of times the transaction has encountered a transmission error. It is recommended that the error count not be incremented when there was an error due to host specific reasons (buffer underrun or overrun), and that whenever a transaction does not encounter a transmission error, the error count is reset to zero.

If the error count for a given transaction reaches three, the host retires the transfer. When a transfer is retired due to excessive errors, the last error type must be indicated. Isochronous transactions are attempted only once, regardless of outcome, and, therefore, no error count is maintained for this type.

10.2.7 Remote Wakeup

If USB System wishes to place the bus in the Suspended state, it commands the Host Controller to stop all bus traffic, including SOFs. This causes all USB devices to enter the Suspended state. In this state, the USB System may enable the Host Controller to respond to bus wakeup events. This allows the Host Controller to respond to bus wakeup signaling to restart the host system.

10.2.8 Root Hub

The root hub provides the connection between the Host Controller and one or more USB ports. The root hub provides the same functionality for dealing with USB topology as other hubs (see Chapter 11), except that the hardware and software interface between the root hub and the Host Controller is defined by the specific hardware implementation.

10.2.8.1 Port Resets

Section 7.1.7.5 describes the requirements of a hub to ensure all upstream resume attempts are overpowered with a long reset downstream. Root hubs must provide an aggregate reset period of at least 50 ms. If the reset duration is controlled in hardware and the hardware timer is <50 ms, the USB System can issue several consecutive resets to accumulate the specified reset duration as described in Section 7.1.7.5.

10.2.9 Host System Interface

The Host Controller provides a high-speed bus-mastering interface to and from main system memory. The physical transfer between memory and the USB wire is performed automatically by the Host Controller. When data buffers need to be filled or emptied, the Host Controller informs the USB System.

10.3 Overview of Software Mechanisms

The HCD and the USBD present software interfaces based on different levels of abstraction. They are, however, expected to operate together in a specified manner to satisfy the overall requirements of the USB System (see Figure 10-2). The requirements for the USB System are expressed primarily as requirements for the USBDI. The division of duties between the USBD and the HCD is not defined. However, the one requirement of the HCDI that must be met is that it supports, in the specified operating system context, multiple Host Controller implementations.

The HCD provides an abstraction of the Host Controller and an abstraction of the Host Controller's view of data transfer across the USB. The USBD provides an abstraction of the USB device and of the data transfers between the client of the USBD and the function on the USB device. Overall, the USB System acts as a facilitator for transmitting data between the client and the function and as a control point for the USB-specific interfaces of the USB device. As part of facilitating data transfer, the USB System provides buffer management capabilities and allows the synchronization of the data transmittal to the needs of the client and the function.

The specific requirements for the USBDI are described later in this chapter. The exact functions that fulfill these requirements are described in the relevant operating system environment guide for the HCDI and the USBDI. The procedures involved in accomplishing data transfers via the USBDI are described in the following sections.

10.3.1 Device Configuration

Different operating system environments perform device configuration using different software components and different sequences of events. The USB System does not assume a specific operating system method. However, there are some basic requirements that must be fulfilled by any USB System implementation. In some operating systems, existing host software provides these requirements. In others, the USB System provides the capabilities.

The USB System assumes a specialized client of the USBD, called a hub driver, that acts as a clearinghouse for the addition and removal of devices from a particular hub. Once the hub driver receives such notifications, it will employ additional host software and other USBD clients, in an operating system specific manner, to recognize and configure the device. This model, shown in Figure 10-4, is the basis of the following discussion.

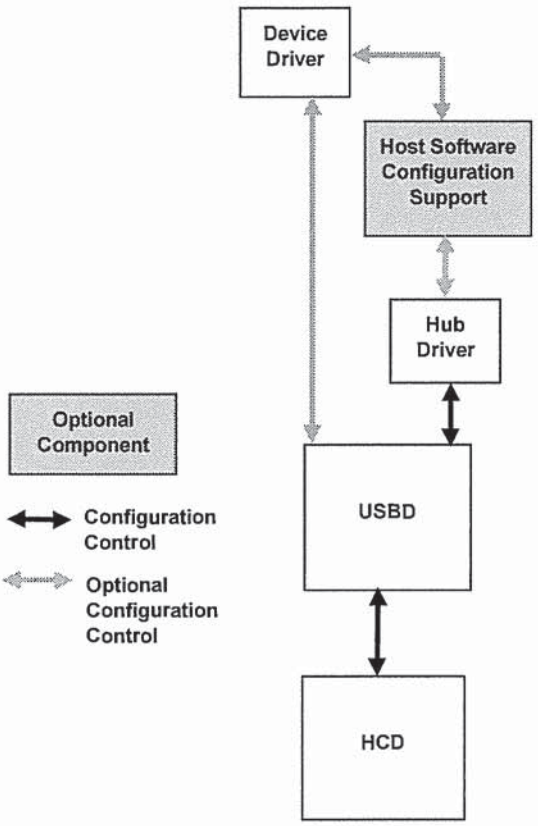

Figure 10-4. Configuration Interactions

When a device is attached, the hub driver receives a notification from the hub detecting the change. The hub driver, using the information provided by the hub, requests a device identifier from the USBD. The USBD in turn sets up the default pipe for that device and returns a device identifier to the hub driver.

The device is now ready to be configured for use. For each device, there are three configurations that must be complete before that device is ready for use:

1. Device Configuration: This includes setting up all of the device's USB parameters and allocating all USB host resources that are visible to the device. This is accomplished by setting the configuration value on the device. A limited set of configuration changes, such as alternate settings, is allowed without totally reconfiguring the device. Once the device is configured, it is, from its point of view, ready for use.
2. USB Configuration: In order to actually create a USBD pipe ready for use by a client, additional USB information, not visible to the device, must be specified by the client. This information, known as the Policy for the pipe, describes how the client will use the pipe. This includes such items as the maximum amount of data the client will transfer with one IRP, the maximum service interval the client will use, the client's notification identification, and so on.
3. Function Configuration: Once configuration types 1 and 2 have been accomplished, the pipe is completely ready for use from the USB's point of view. However, additional vendor- or class-specific setup may be required before the client can actually use the pipe. This configuration is a private matter between the device and the client and is not standardized by the USBD.

The following paragraphs describe the device and USB configuration requirements.

The responsible configuring software performs the actual device configuration. Depending on the particular operating system implementation, the software responsible for configuration can include the following:

- The hub driver
- Other host software
- A device driver

The configuring software first reads the device descriptor, then requests the description for each possible configuration. It may use the information provided to load a particular client, such as a device driver, which initially interacts with the device. The configuring software, perhaps with input from that device driver, chooses a configuration for the device. Setting the device configuration sets up all of the endpoints on the device and returns a collection of interfaces to be used for data transfer by USBD clients. Each interface is a collection of pipes owned by a single client.

This initial configuration uses the default settings for interfaces and the default bandwidth for each endpoint. A USBD implementation may additionally allow the client to specify alternate interfaces when selecting the initial configuration. The USB System will verify that the resources required for the support of the endpoint are available and, if so, will allocate the bandwidth required. Refer to Section 10.3.2 for a discussion of resource management.

The device is now configured, but the created pipes are not yet ready for use. The USB configuration is accomplished when the client initializes each pipe by setting a Policy to specify how it will interact with the pipe. Among the information specified is the client's maximum service interval and notification information. Among the actions taken by the USB System, as a result of setting the Policy, is determining the amount of buffer working space required beyond the data buffer space provided by the client. The size of the buffers required is based upon the usage chosen by the client and upon the per-transfer needs of the USB System.

The client receives notifications when IRPs complete, successfully or due to errors. The client may also wake up independently of USB notification to check the status of pending IRPs.

The client may also choose to make configuration modifications, such as enabling an alternate setting for an interface or changing the bandwidth allocated to a particular pipe. In order to perform these changes, the interface or pipe, respectively, must be idle.

10.3.2 Resource Management

Whenever a pipe is setup by the USBD for a given endpoint, the USB System must determine if it can support the pipe. The USB System makes this determination based on the requirements stated in the endpoint descriptor. One of the endpoint requirements, which must be supported in order to create a pipe for an endpoint, is the bandwidth necessary for that endpoint's transfers. There are two stages to check for available bandwidth. First the maximum execution time for a transaction is calculated. Then the (micro)frame schedule is consulted to determine if the indicated transaction will fit.

The allocation of the guaranteed bandwidth for isochronous and interrupt pipes, and the determination of whether a particular control or bulk transaction will fit into a given (micro)frame, can be determined by a software heuristic in the USB System. If the actual transaction execution time in the Host Controller exceeds the heuristically determined value, the Host Controller is responsible for ensuring that (micro)frame integrity is maintained (refer to Section 10.2.3). The following discussion describes the requirements for the USB System heuristic.

In order to determine if bandwidth can be allocated, or if a transaction can be fit into a particular (micro)frame, the maximum transaction execution time must be calculated. The calculation of the maximum transaction execution time requires that the following information be provided: (Note that an agent other than the client may provide some of this information.)

- Number of data bytes (*wMaxPacketSize*) to be transmitted.
- Transfer type.
- Depth in the topology. If less precision is allowed, the maximum topology depth may be assumed.

This calculation must include the bit transmission time, the signal propagation delay through the topology, and any implementation-specific delays, such as preparation or recovery time required by the Host Controller itself. Refer to Chapter 5 for examples of formulas that can be used for such calculations.

10.3.3 Data Transfers

The basis for all client-function communication is the interface: a bundle of related pipes associated with a particular USB device.

Exactly one client on the host manages a given interface. The client initializes each pipe of an interface by setting the Policy for that pipe. This includes the maximum amount of data to be transmitted per IRP and the maximum service interval for the pipe. A service interval is stated in milliseconds and describes the interval over which an IRP's data will be transmitted for an isochronous pipe. It describes the polling interval for an interrupt pipe. The client is notified when a specified request is completed. The client manages the size of each IRP such that its duty cycle and latency constraints are maintained. Additional Policy information includes the notification information for the client.

The client provides the buffer space required to hold the transmitted data. The USB System uses the Policy to determine the additional working space it will require.

The client views its data as a contiguous serial stream, which it manages in a similar manner to those streams provided over other types of bus technologies. Internally, the USB System may, depending on its own Policy and any Host Controller constraints, break the client request down into smaller requests to be sent across the USB. However, two requirements must be met whenever the USB System chooses to undertake such division:

- The division of the data stream into smaller chunks is not visible to the client.
- USB samples are not split across bus transactions.

When a client wishes to transfer data, it will send an IRP to the USBD. Depending on the direction of data transfer, a full or empty data buffer will be provided. When the request is complete (successfully or due to an error condition), the IRP and its status is returned to the client. Where relevant, this status is also provided on a per-transaction basis.

10.3.4 Common Data Definitions

In order to allow the client to receive request results as directly as possible from its device, it is desirable to minimize the amount of processing and copying required between the device and the client. To facilitate this, some control aspects of the IRP are standardized such that different layers in the stack may directly use the information provided by the client. The particular format for this data is dependent on the actualization of the USBDI in the operating system. Some data elements may in fact not be directly visible to the client at all but are generated as a result of the client request.

The following data elements define the relevant information for a request:

- Identification of the pipe associated with the request. Identifying this pipe also describes information such as transfer type for this request.
- Notification identification for the particular client.
- Location and length of data buffer that is to be transmitted or received.
- Completion status for the request. Both the summary status and, as required, detailed per-transaction status must be provided.
- Location and length of working space. This is implementation-dependent.

The actual mechanisms used to communicate requests to the USBD are operating system-specific. However, beyond the requirements stated above for what request-related information must be available, there are also requirements on how requests will be processed. The basic requirements are described in Chapter 5. Additionally, the USBD provides a mechanism to designate a group of isochronous IRPs for which the transmission of the first transaction of each IRP will occur in the same (micro)frame. The USBD also provides a mechanism for designating an uninterruptable set of vendor- or class-specific requests to a default pipe. No other requests to that default pipe, including standard, class, or vendor request may be inserted in the execution flow for such an uninterruptable set. If any request in this set fails, the entire set is retired.

10.4 Host Controller Driver

The Host Controller Driver (HCD) is an abstraction of Host Controller hardware and the Host Controller's view of data transmission over the USB. The HCDI meets the following requirements:

- Provides an abstraction of the Host Controller hardware.
- Provides an abstraction for data transfers by the Host Controller across the USB interconnect.
- Provides an abstraction for the allocation (and de-allocation) of Host Controller resources to support guaranteed service to USB devices.
- Presents the root hub and its behavior according to the hub class definition. This includes supporting the root hub such that the hub driver interacts with the root hub exactly as it would for any hub. In particular, even though a root hub can be implemented in a combination of hardware and software, the root hub responds initially to the default device address (from a client perspective), returns descriptor information, supports having its device address set, and supports the other hub class requests. However, bus transactions may or may not need to be generated to accomplish this behavior given the close integration possible between the Host Controller and the root hub.

The HCD provides a software interface (HCDI) that implements the required abstractions. The function of the HCD is to provide an abstraction, which hides the details of the Host Controller hardware. Below the Host Controller hardware is the physical USB and all the attached USB devices.

The HCD is the lowest tier in the USB software stack. The HCD has only one client: the Universal Serial Bus Driver (USBD). The USBD maps requests from many clients to the appropriate HCD. A given HCD may manage many Host Controllers.

The HCDI is not directly accessible from a client. Therefore, the specific interface requirements for the HCDI are not discussed here.

10.5 Universal Serial Bus Driver

The USBD provides a collection of mechanisms that operating system components, typically device drivers, use to access USB devices. The only access to a USB device is that provided by the USBD. The USBD implementations are operating system-specific. The mechanisms provided by the USBD are implemented, using as appropriate and augmenting as necessary, the mechanisms provided by the operating system environment in which the USB runs. The following discussion centers on the basic capabilities

required for all USBD implementations. For specifics of the USBD operation within a specific environment, see the relevant operating system environment guide for the USBD. A single instance of the USBD directs accesses to one or more HCDs that in turn connect to one or more Host Controllers. If allowed, how USBD instancing is managed is dependent upon the operating system environment. However, from the client's point of view, the USBD with which the client communicates manages all of the attached USB devices.

10.5.1 USBD Overview

Clients of USBD direct commands to devices or move streams of data to or from pipes. The USBD presents two groups of software mechanisms to clients: command mechanisms and pipe mechanisms.

Command mechanisms allow clients to configure and control USBD operation as well as to configure and generically control a USB device. In particular, command mechanisms provide all access to the device's default pipe.

Pipe mechanisms allow a USBD client to manage device specific data and control transfers. Pipe mechanisms do not allow a client to directly address the device's default pipe.

Figure 10-5 presents an overview of the USBD structure.

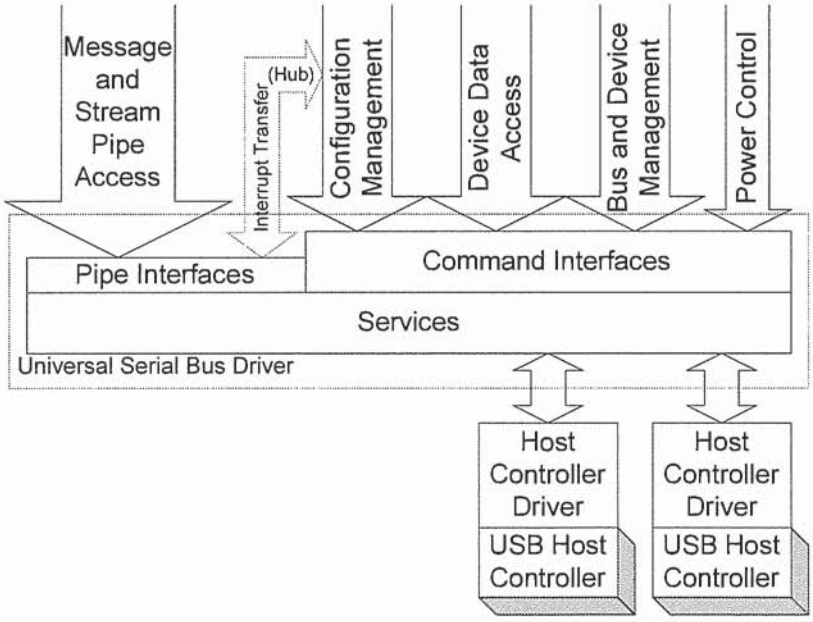

Figure 10-5. Universal Serial Bus Driver Structure

10.5.1.1 USBD Initialization

Specific USBD initialization is operating system-dependent. When a particular USB managed by USBD is initialized, the management information for that USB is also created. Part of this management information is the default address device and its default pipe used to communicate to a newly reset device.

When a device is attached to a USB, it responds to a special address known as the default address (refer to Chapter 9) until its unique address is assigned by the bus enumerator. In order for the USB System to interact with the new device, the default device address and the device's default pipe must be available to the hub driver when a device is attached. During device initialization, the default address is changed to a unique address.

10.5.1.2 USBD Pipe Usage

Pipes are the method by which a device endpoint is associated with a Host Software entity. Pipes are owned by exactly one such entity on the host. Although the basic concept of a pipe is the same no matter who the owner, some distinction of capabilities provided to the USBD client occurs between two groups of pipes:

- Default pipes, which are owned and managed by the USBD
- All other pipes, which are owned and managed by clients of the USBD

Default pipes are never directly accessed by clients, although they are often used to fulfill some part of client requests relayed via command mechanisms.

10.5.1.2.1 Default Pipes

The USBD is responsible for allocating and managing appropriate buffering to support transfers on the default pipe that are not directly visible to the client such as setting a device address. For those transfers that are directly visible to the client, such as sending vendor and class commands or reading a device descriptor, the client must provide the required buffering.

10.5.1.2.2 Client Pipes

Any pipe not owned and managed by the USBD can be owned and managed by a USBD client. From the USBD viewpoint, a single client owns the pipe. In fact, a cooperative group of clients can manage the pipe, provided they behave as a single coordinated entity when using the pipe.

The client is responsible for providing the amount of buffering it needs to service the data transfer rate of the pipe within a service interval attainable by the client. Additional buffering requirements for working space are specified by the USB System.

10.5.1.3 USBD Service Capabilities

The USBD provides services in the following categories:

- Configuration via command mechanisms
- Transfer services via both command and pipe mechanisms
- Event notification
- Status reporting and error recovery

10.5.2 USBD Command Mechanism Requirements

USBD command mechanisms allow a client generic access to a USB device. Generally, these commands allow the client to make read or write accesses to one of potentially several device data and control spaces. The client provides as little as a device identifier and the relevant data or empty buffer pointer.

USBD command transfers do not require that the USB device be configured. Many of the device configuration facilities provided by the USBD are command transfers.

Following are the specific requirements on the command mechanisms provided.

10.5.2.1 Interface State Control

USBD clients must be able to set a specified interface to any settable pipe state. Setting an interface state results in all of the pipes in that interface moving to that state. Additionally, all of the pipes in an interface may be reset or aborted.

10.5.2.2 Pipe State Control

USBD pipe state has two components:

- Host status
- Reflected endpoint status

Whenever the pipe status is reported, the value for both components will be identified. The pipe status reflected from the endpoint is the result of the endpoint being in a particular state. The USBD client manages the pipe state as reported by the USBD. For any pipe state reflected from the endpoint, the client must also interact with the endpoint to change the state.

A USBD pipe is in exactly one of the following states:

- Active: The pipe's Policy has been set and the pipe is able to transmit data. The client can query as to whether any IRPs are outstanding for a particular pipe. Pipes for which there are no outstanding IRPs are still considered to be in the Active state as long as they are able to accept new IRPs.
- Halted: An error has occurred on the pipe. This state may also be a reflection of the corresponding Halted endpoint on the device.

A pipe and endpoint are considered active when the device is configured and the pipe and/or endpoint is not stalled. Clients may manipulate pipe state in the following ways:

- Aborting a Pipe: All of the IRPs scheduled for a pipe are retired immediately and returned to the client with a status indicating they have been aborted. Neither the host state nor the reflected endpoint state of the pipe is affected.
- Resetting a Pipe: The pipe's IRPs are aborted. The host state is moved to Active. If the reflected endpoint state needs to be changed, that must be commanded explicitly by the USBD client.
- Clearing a Halted pipe: The pipe's state is cleared from *Halted* to *Active*.
- Halting a Pipe: The pipe's state is set to *Halted*.

10.5.2.3 Getting Descriptors

The USBDI must provide a mechanism to retrieve standard device, configuration, and string descriptors, as well as any class- or vendor-specific descriptors.

10.5.2.4 Getting Current Configuration Settings

The USBDI must provide a facility to return, for any specified device, the current configuration descriptor. If the device is not configured, no configuration descriptor is returned. This action is equivalent to returning the configuration descriptor for the current configuration by requesting the specific configuration descriptor. It does not, however, require the client to know the identifier for the current configuration. This will return all of the configuration information, including the following:

- All of the configuration descriptor information as stored on the device, including all of the alternate settings for all of the interfaces
- Indicators for which of the alternate settings for interfaces are active
- Pipe handles for endpoints in the active alternate settings for interfaces
- Actual *wMaxPacketSize* values for endpoints in the active alternate settings for interfaces

Additionally, for any specified pipe, the USBDI must provide a facility to return the *wMaxPacketSize* that is currently being used by the pipe.

10.5.2.5 Adding Devices

The USBDI must provide a mechanism for the hub driver to inform USBD of the addition of a new device to a specified USB and to retrieve the USBD ID of the new USB device. The USBD tasks include assigning the device address and preparing the device's default pipe for use.

10.5.2.6 Removing Devices

The USBDI must provide a facility for the hub driver to inform the USBD that a specific device has been removed.

10.5.2.7 Managing Status

The USBDI must provide a mechanism for obtaining and clearing device-based status on a device, interface, or pipe basis.

10.5.2.8 Sending Class Commands

This USBDI mechanism is used by a client, typically a class-specific or adaptive driver, to send one or more class-specific commands to a device.

10.5.2.9 Sending Vendor Commands

This USBDI mechanism is used by a client to send one or more vendor-specific commands to a device.

10.5.2.10 Establishing Alternate Settings

The USBDI must provide a mechanism to change the alternate setting for a specified interface. As a result, the pipe handles for the previous setting are released and new pipe handles for the interface are returned. For this request to succeed, the interface must be idle; i.e., no data buffers may be queued for any pipes in the interface.

10.5.2.11 Establishing a Configuration

Configuring software requests a configuration by passing a buffer containing the configuration information to the USBD. The USBD requests resources for the endpoints in the configuration, and if all resource requests succeed, the USBD sets the device configuration and returns interface handles with corresponding pipe handles for all of the active endpoints. The default values are used for all alternate settings for interfaces.

Note: The interface implementing the configuration may require specific alternate settings to be identified.

10.5.2.12 Setting Descriptors

For devices supporting this behavior, the USBDI allows existing descriptors to be updated or new descriptors to be added.

10.5.3 USBD Pipe Mechanisms

This part of the USBDI offers clients the highest-speed, lowest overhead data transfer services possible. Higher performance is achieved by shifting some pipe management responsibilities from the USBD to the client. As a result, the pipe mechanisms are implemented at a more primitive level than the data transfer services provided by the USBD command mechanisms. Pipe mechanisms do not allow access to a device's default pipe.

USBD pipe transfers are available only after both the device and USB configuration have completed successfully. At the time the device is configured, the USBD requests the resources required to support all

device pipes in the configuration. Clients are allowed to modify the configuration, constrained by whether the specified interface or pipe is idle.

Clients provide full buffers to outgoing pipes and retrieve transfer status information following the completion of a request. The transfer status returned for an outgoing pipe allows the client to determine the success or failure of the transfer.

Clients provide empty buffers to incoming pipes and retrieve the filled buffers and transfer status information from incoming pipes following the completion of a request. The transfer status returned for an incoming pipe allows a client to determine the amount and the quality of the data received.

10.5.3.1 Supported Pipe Types

The four types of pipes supported, based on the four transfer types, are described in the following sections.

10.5.3.1.1 Isochronous Data Transfers

Each buffer queued for an isochronous pipe is required to be viewable as a stream of samples. As with all pipe transfers, the client establishes a Policy for using this isochronous pipe, including the relevant service interval for this client. Lost or missing bytes, which are detected on input, and transmission problems, which are noted on output, are indicated to the client.

The client queues a first buffer, starting the pipe streaming service. To maintain the continuous streaming transfer model used in all isochronous transfers, the client queues an additional buffer before the current buffer is retired.

The USBD is required to be able to provide a sample stream view of the client's data stream. In other words, using the client's specified method of synchronization, the precise packetization of the data is hidden from the client. Additionally, a given transaction is always contained completely within some client data buffer.

For an output pipe, the client provides a buffer of data. The USBD allocates the data across the (micro)frames for the service period using the client's chosen method of synchronization.

For an input pipe, the client must provide an empty buffer large enough to hold the maximum number of bytes the client's device will deliver in the service period. Where missing or invalid bytes are indicated, the USBD may leave the space that the bytes would have occupied in place in the buffer and identify the error. One of the consequences of using no synchronization method is that this reserved space is assumed to be the maximum packet size. The buffer-retired notification occurs when the IRP completes. Note that the input buffer need not be full when returned to the client.

The USBD may optionally provide additional views of isochronous data streams. The USBD is also required to be able to provide a packet stream view of the client's data stream.

10.5.3.1.2 Interrupt Transfers

The Interrupt out transfer originates in the client of the USBD and is delivered to the USB device. The Interrupt in transfer originates in a USB device and is delivered to a client of the USBD. The USB System guarantees that the transfers meet the maximum latency specified by the USB endpoint descriptor.

The client queues a buffer large enough to hold the interrupt transfer data (typically a single USB transaction). When all of the data is transferred, or if the error threshold is exceeded, the IRP is returned to the client.

10.5.3.1.3 Bulk Transfers

Bulk transfers may originate either from the device or the client. No periodicity or guaranteed latency is assumed. When all of the data is transferred, or if the error threshold is exceeded, the IRP is returned to the client.

10.5.3.1.4 Control Transfers

All message pipes transfer data in both directions. In all cases, the client outputs a setup stage to the device endpoint. The optional data stage may be either input or output and the final status is always logically presented to the host. For details of the defined message protocol, refer to Chapter 8.

The client prepares a buffer specifying the command phase and any optional data or empty buffer space. The client receives a buffer-retired notification when all phases of the control transfer are complete, or an error notification, if the transfer is aborted due to transmission error.

10.5.3.2 USBD Pipe Mechanism Requirements

The following pipe mechanisms are provided.

10.5.3.2.1 Aborting IRPs

The USBDI must allow IRPs for a particular pipe to be aborted.

10.5.3.2.2 Managing Pipe Policy

The USBDI must allow a client to set and clear the Policy for an individual pipe or for an entire interface. Any IRPs made by the client prior to successfully setting a Policy are rejected by the USBD.

10.5.3.2.3 Queuing IRPs

The USBDI must allow clients to queue IRPs for a given pipe. When IRPs are returned to the client, the request status is also returned. A mechanism is provided by the USBD to identify a group of isochronous IRPs whose first transactions will all occur in the same (micro)frame.

10.5.4 Managing the USB via the USBD Mechanisms

Using the provided USBD mechanisms, the following general capabilities are supported by any USB System.

10.5.4.1 Configuration Services

Configuration services operate on a per-device basis. The configuring software tells the USBD when to perform device configuration. A hub driver has a special role in device management and provides at least the following capabilities:

- Device attach/detach recognition, driven by an interrupt pipe owned by the hub driver
- Device reset, accomplished by the hub driver by resetting the hub port upstream of the device
- Tells the USBD to perform device address assignment
- Power control

The USBDI additionally provides the following configuration facilities, which may be used by the hub driver or other configuring software available on the host:

- Device identification and access to configuration information (via access to descriptors on the device)
- Device configuration via command mechanisms

When the hub driver informs the USBD of a device attachment, the USBD establishes the default pipe for the new device.

10.5.4.1.1 Configuration Management

Configuration management services are provided primarily as a set of specific interface commands that generate USB transactions on the default pipe. The notable exception is the use of an additional interrupt pipe that delivers hub status directly to the hub driver.

Every hub initiates an interrupt transfer when there is a change in the state of one of the hub ports. Generally, the port state change will be the connection or removal of a downstream USB device. (Refer to Chapter 11 for more information.)

10.5.4.1.2 Initial Device Configuration

The device configuration process begins when a hub reports, via its status change pipe, the connection of a new USB device.

Configuration management services allow configuring software to select a USB device configuration from the set of configurations listed in the device. The USBD verifies that adequate power is available and the data transfer rates given for all endpoints in the configuration do not exceed the capabilities of the USB with the current schedule before setting the device configuration.

10.5.4.1.3 Modifying a Device Configuration

Configuration management services allow configuring software to replace a USB device configuration with another configuration from the set of configurations listed in the device. The operation succeeds if adequate power is available and the data transfer rates given for all endpoints in the new configuration fit within the capabilities of the USB with the current schedule. If the new configuration is rejected, the previous configuration remains.

Configuration management services allow configuring software to return a USB device to a Not Configured state.

10.5.4.1.4 Device Removal

Error recovery and/or device removal processing begins when a hub reports via its status change pipe that the USB device has been removed.

10.5.4.2 Power Control

There are two cooperating levels of power management for the USB: bus and device level management. This specification provides mechanisms for managing power on the USB bus. Device classes may define class-specific power control capabilities.

All USB devices must support the Suspended state (refer to Chapter 9). The device is placed into the Suspended state via control of the hub port to which the device is attached. Normal device operation ceases in the Suspend State; however, if the device is capable of wakeup signaling and the device is enabled for remote wakeup, it may generate resume signaling in response to external events.

The power management system may transition a device to the Suspended state or power-off the device in order to control and conserve power. The USB provides neither requirements nor commands for the device state to be saved and restored across these transitions. Device classes may define class-specific device state save-and-restore capabilities.

The USB System coordinates the interaction between device power states and the Suspended state.

It is recommended that while a device is not being used by the system (i.e., no transactions are being transmitted to or from the device besides SOF tokens), the device be suspended as soon as possible by selectively suspending the port to which the device is attached. Suspending inactive devices reduces reliability issues due to high currents passing through a transceiver operating in high-speed mode in the presence of short circuit conditions described in Section 7.1.1. Some of these short circuit conditions are not detectable in the absence of transactions to the device. Suspending the unused device will place the

transceiver interface into full-speed mode which has a greater reliability in the presence of short circuit conditions.

10.5.4.3 Event Notifications

USBD clients receive several kinds of event notifications through a number of sources:

- Completion of an action initiated by a client.
- Interrupt transfers over stream pipes can deliver notice of device events directly to USBD clients. For example, hubs use an interrupt pipe to deliver events corresponding to changes in hub status.
- Event data can be embedded by devices in streams.
- Standard device interface commands, device class commands, vendor-specific commands, and even general control transfers over message pipes can all be used to poll devices for event conditions.

10.5.4.4 Status Reporting and Error Recovery Services

The command and pipe mechanisms both provide status reporting on individual requests as they are invoked and completed.

Additionally, USB device status is available to USBD clients using the command mechanisms.

The USBD provides clients with pipe error recovery mechanisms by allowing pipes to be reset or aborted.

10.5.4.5 Managing Remote Wakeup Devices

The USB System can minimize the resume power consumption of a suspended USB tree. This is accomplished by explicitly enabling devices capable of resume signaling and controlling propagation of resume signaling via selectively suspending and/or disabling hub ports between the device and the nearest self-powered, awake hub.

In some error-recovery scenarios, the USB System will need to re-enumerate sub-trees. The sub-tree may be partially or completely suspended. During error-recovery, the USB System must avoid contention between a device issuing resume signaling and simultaneously driving reset down the port. Avoidance is accomplished via management of the devices' remote wakeup feature and the hubs' port features. The rules are as follows:

- Issue a SetDeviceFeature(DEVICE_REMOTE_WAKEUP) request to the leaf device, only just prior to selectively suspending any port between where the device is connected and the root port (via a SetPortFeature(PORT_SUSPEND) request).
- Do not reset a suspended port that has had a device enabled for remote wakeup without first enabling that port.
- Verify that after a remote wakeup, the devices in the subtree affected by the remote wakeup are still present. This will typically be done as part of determining which potential remote wakeup device was the source of the wakeup. This should be done to ensure that a suspended device is not disconnected (and possibly reconnected) or reset (e.g., by noise) during a suspend/resume process.

10.5.5 Passing USB Preboot Control to the Operating System

A single software driver owns the Host Controller. If the host system implements USB services before the operating system loads, the Host Controller must provide a mechanism that disables access by the preboot software and allows the operating system to gain control. Preboot USB configuration is not passed to the operating system. Once the operating system gains control, it is responsible to fully configure the bus. If the operating system provides a mechanism to pass control back to the preboot environment, the bus will be in an unknown state. The preboot software should treat this event as a powerup.

10.6 Operating System Environment Guides

As noted previously, the actual interfaces between the USB System and host software are specific to the host platform and operating system. A companion specification is required for each combination of platform and operating system with USB support. These specifications describe the specific interfaces used to integrate the USB into the host. Each operating system provider for the USB System identifies a compatible Universal USB Specification revision.

Chapter 11
Hub Specification

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

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

11.1 Overview

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

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

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

11.1.1 Hub Architecture

Figure 11-1 shows a hub and the locations of its upstream and downstream facing ports. A hub consists of a Hub Repeater section, a Hub Controller section, and a Transaction Translator section. The hub must operate at high-speed when its upstream facing port is connected at high-speed. The hub must operate at full-speed when its upstream facing port is connected at full-speed.

The Hub Repeater is responsible for managing connectivity between upstream and downstream facing ports which are operating at the same speed. The Hub Repeater supports full-/low-speed connectivity and high-speed connectivity. The Hub Controller provides status and control and permits host access to the hub. The Transaction Translator takes high-speed split transactions and translates them to full-/low-speed transactions when the hub is operating at high-speed and has full-/low-speed devices attached. The operating speed of a device attached on a downstream facing port determines whether the Routing Logic connects a port to the Transaction Translator or hub repeater sections.

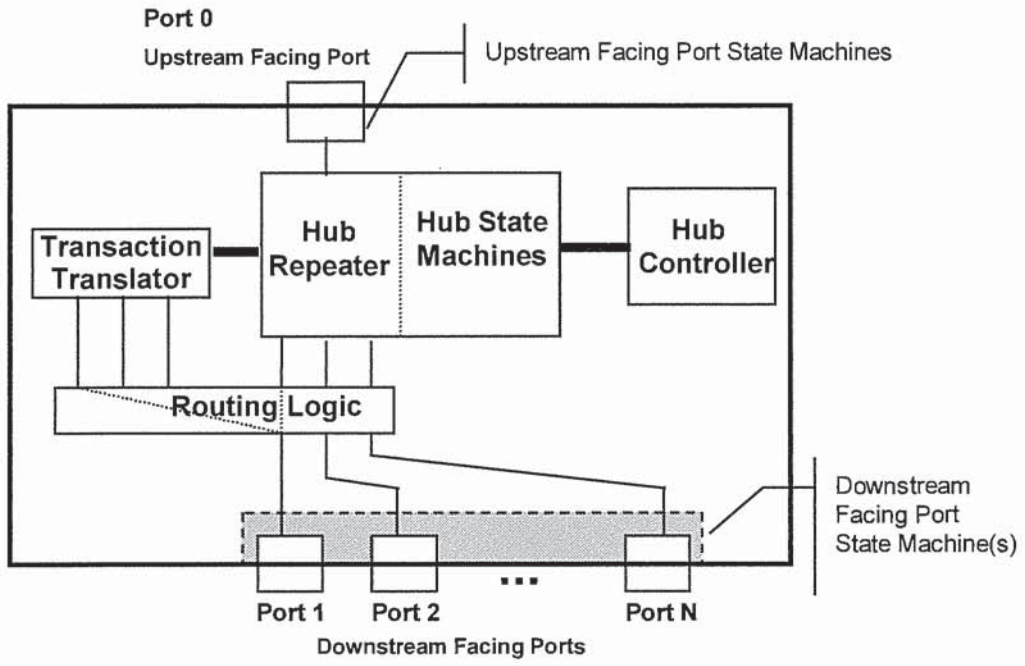

Figure 11-1. Hub Architecture

When a hub'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.

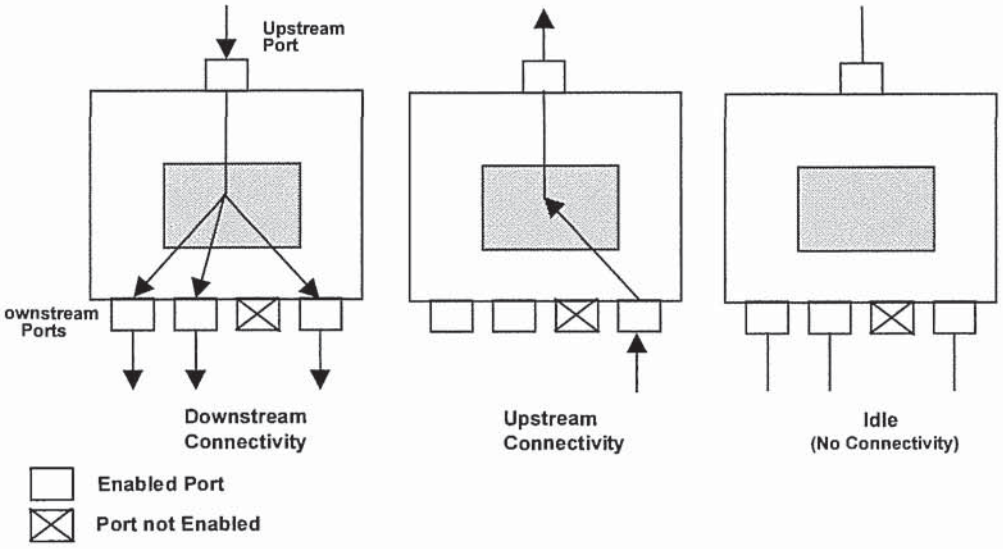

Figure 11-2. Hub Signaling Connectivity

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.

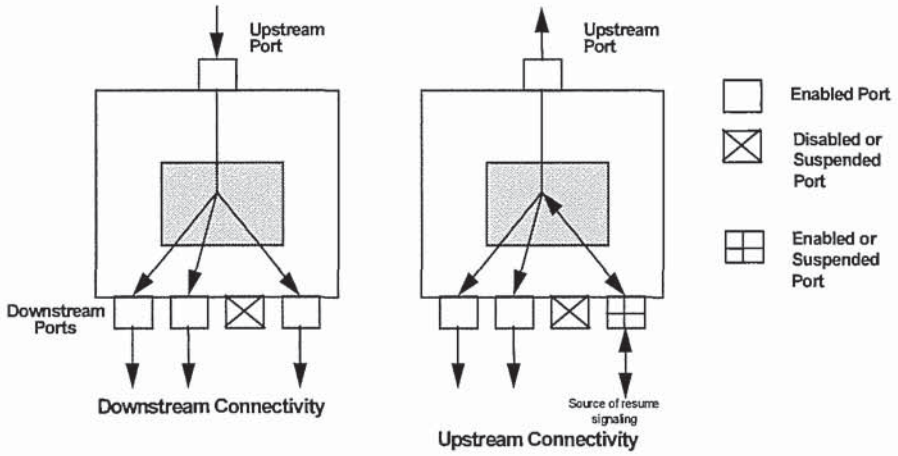

Figure 11-3. Resume Connectivity

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) +/-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).

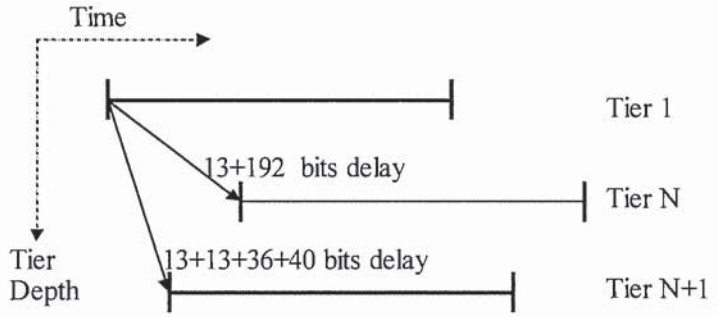

Figure 11-4. Example High-speed EOF Offsets Due to Propagation Delay Without EOF Advancement

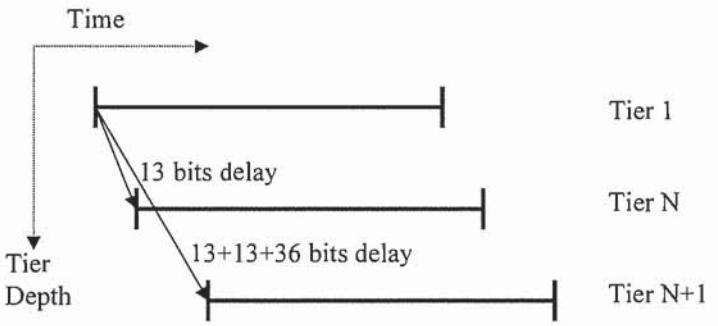

Figure 11-5. Example High-speed EOF Offsets Due to Propagation Delay With EOF Advancement

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.

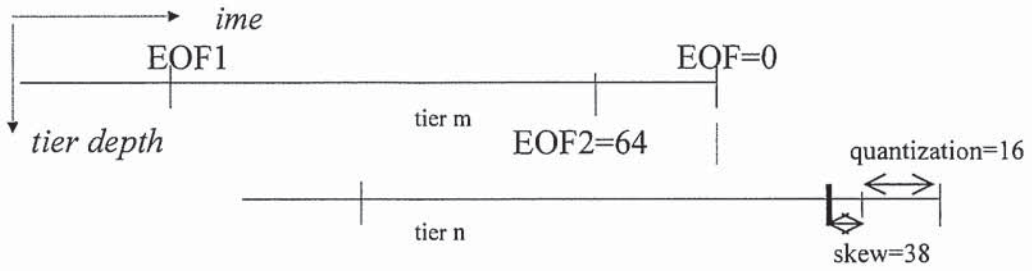

Figure 11-6. High-speed EOF2 Timing Point

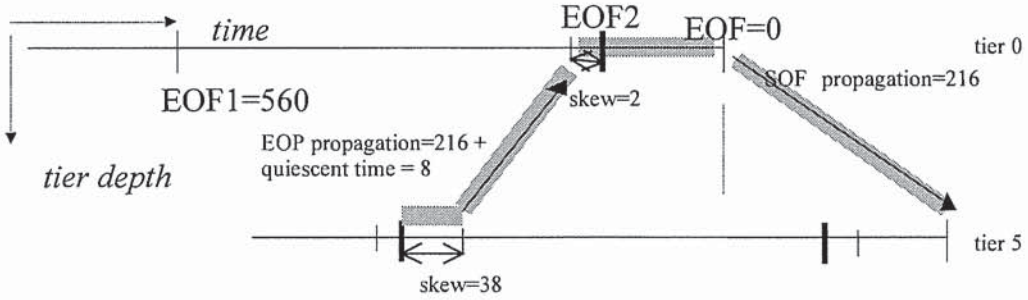

Figure 11-7. High-speed EOF1 Timing Point

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.

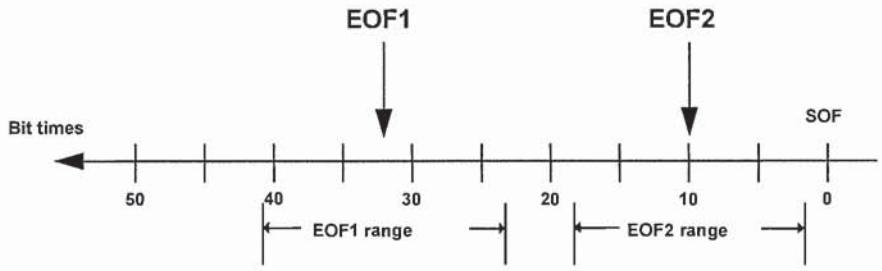

Figure 11-8. Full-speed EOF Timing Points

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 $\pm 1 \pm 2 \pm 3 = \pm 6$ 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.

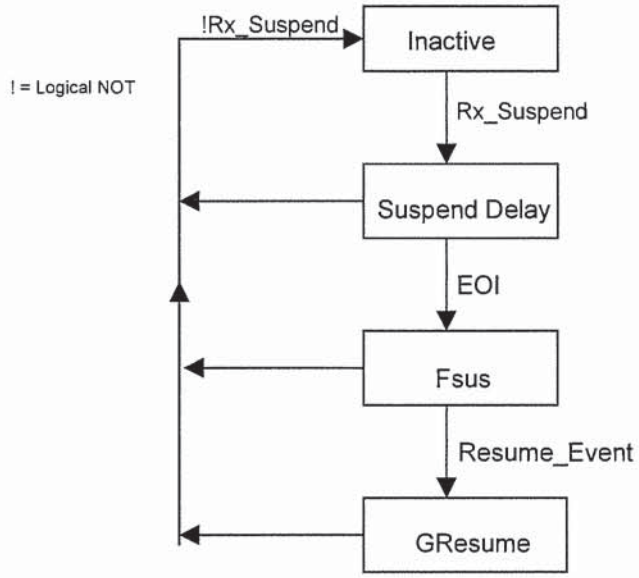

Figure 11-9. Internal Port State Machine

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.

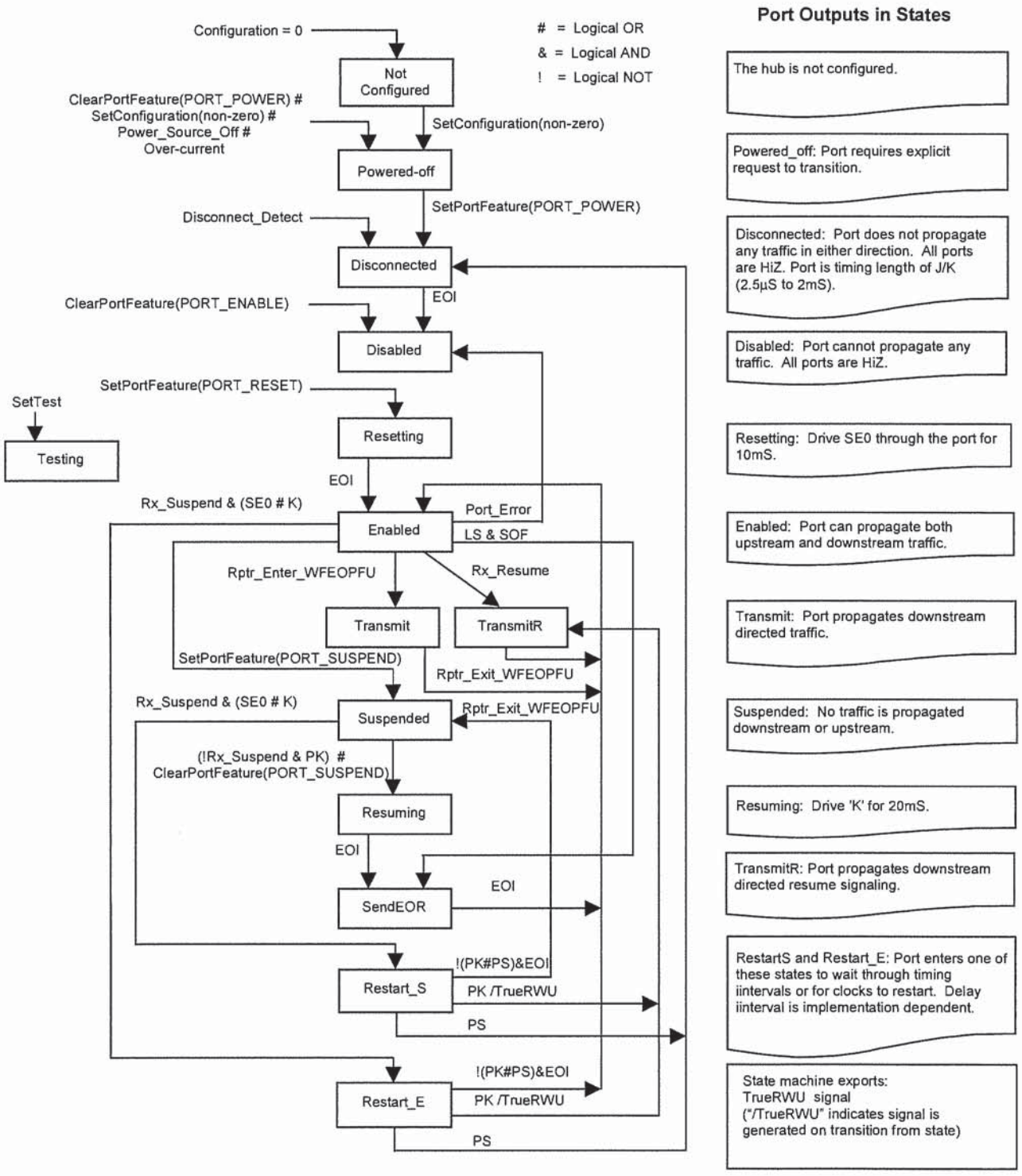

Figure 11-10. Downstream Facing Hub Port State Machine

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

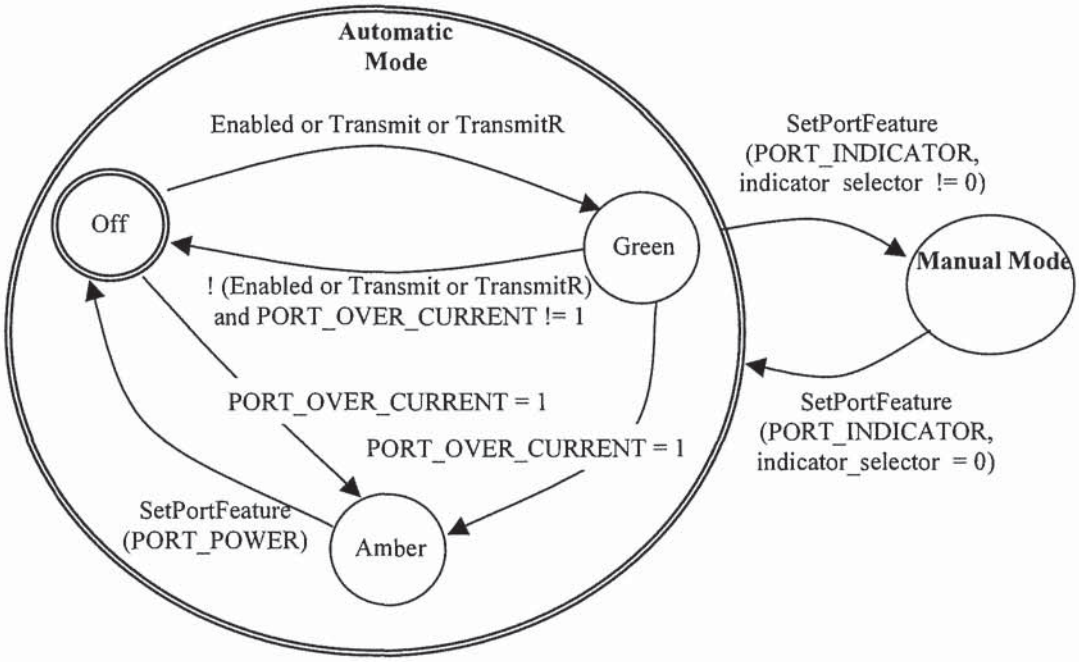

Figure 11-11. Port Indicator State Diagram

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.

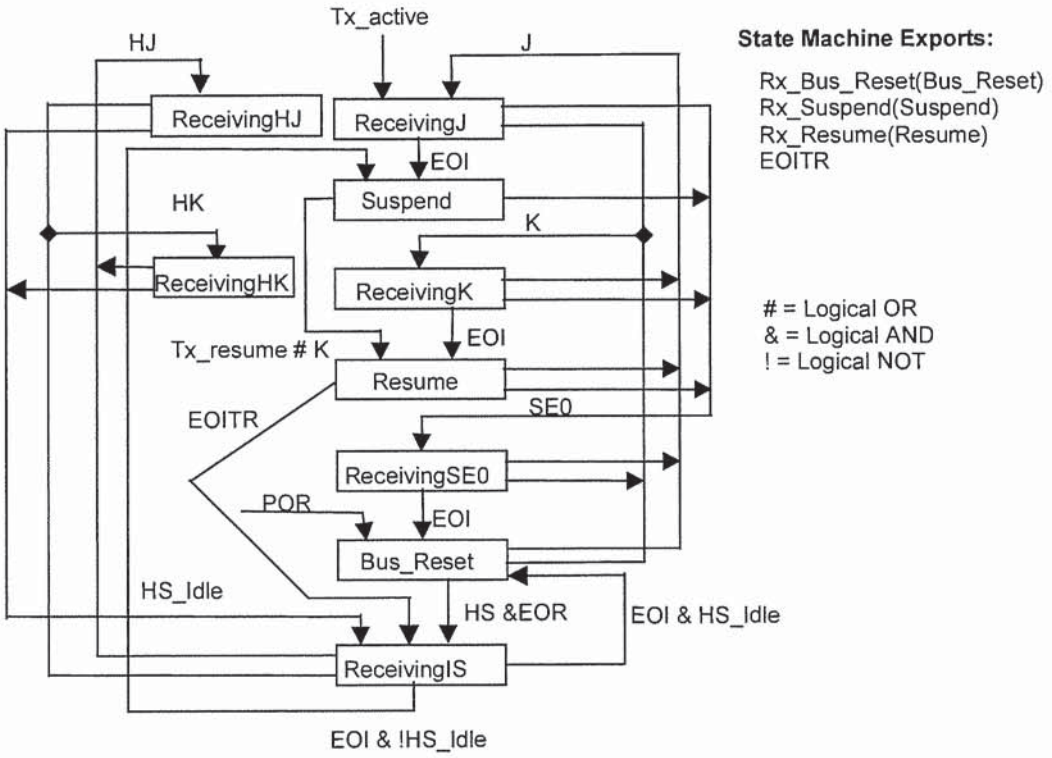

Figure 11-12. Upstream Facing Port Receiver State Machine

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.

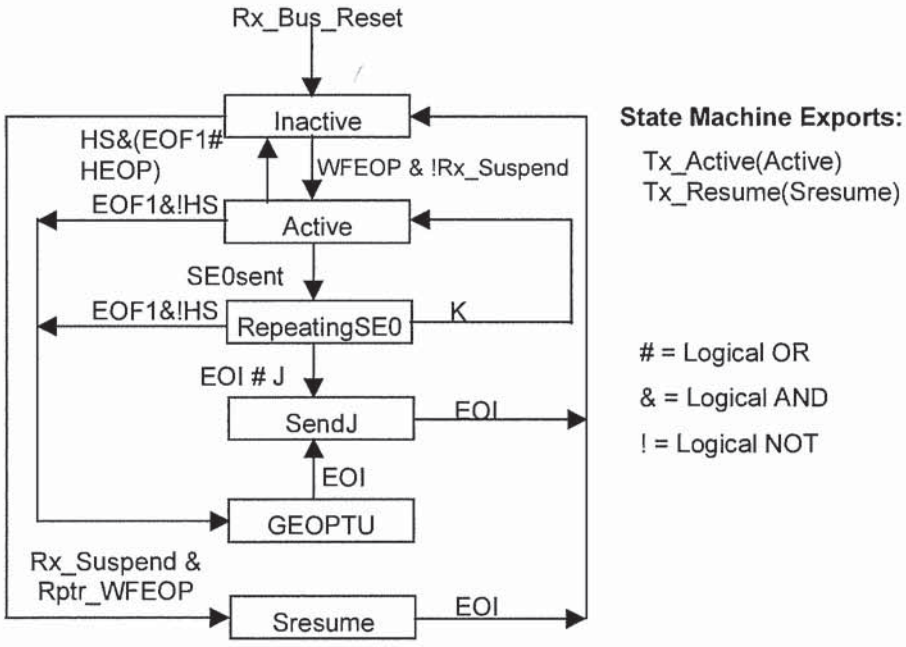

Figure 11-13. Upstream Facing Port Transmitter State Machine

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.

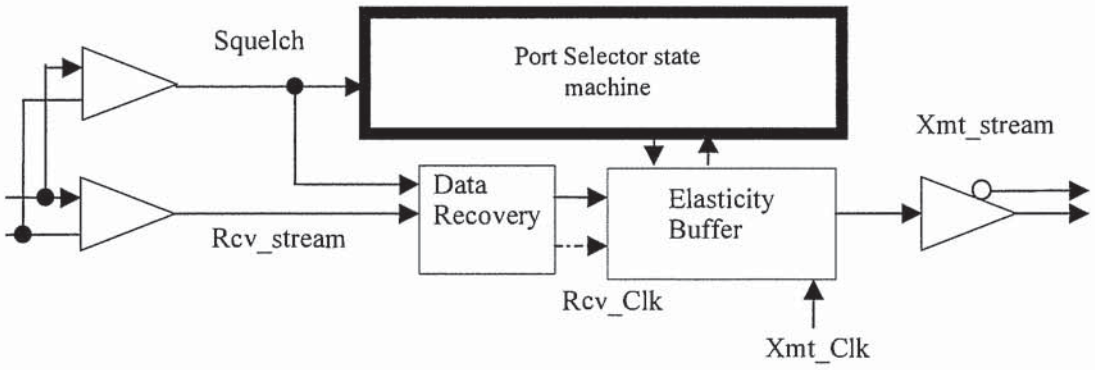

Figure 11-14. Example Hub Repeater Organization

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.)

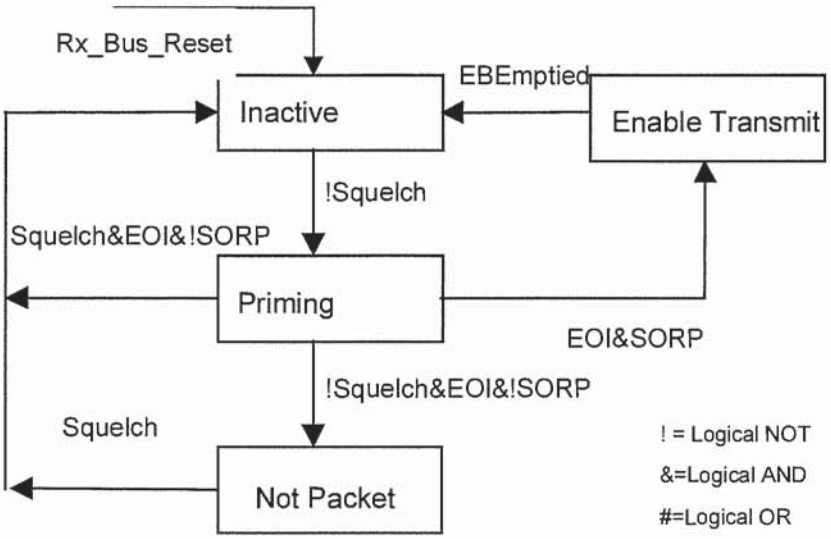

Figure 11-15. High-speed Port Selector State Machine

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

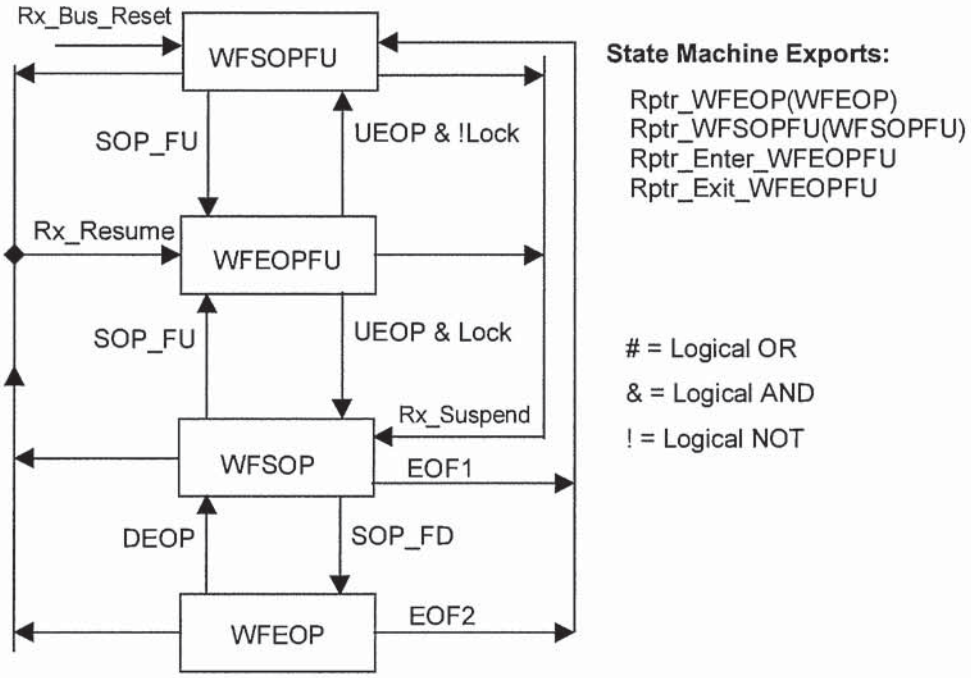

Figure 11-16. Hub Repeater State Machine

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: EBEmptied signal from port selector state machine OR transition at EOI from SendEOR state in downstream facing port state machine OR 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.

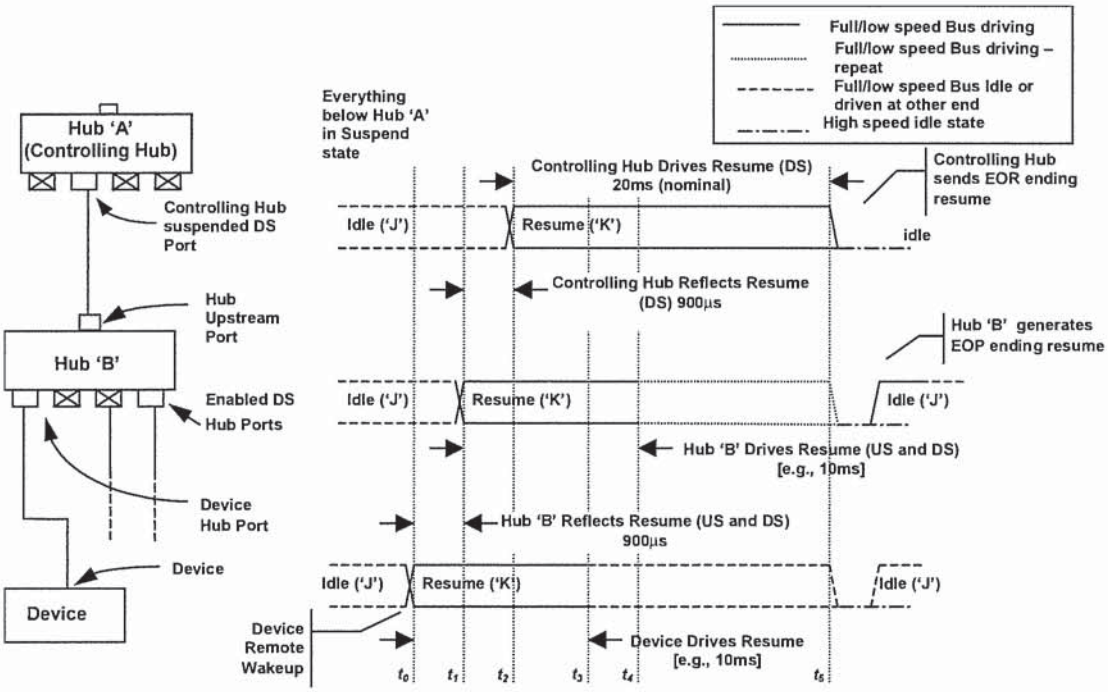

Figure 11-17. Example Remote-wakeup Resume Signaling With Full-/low-speed Device

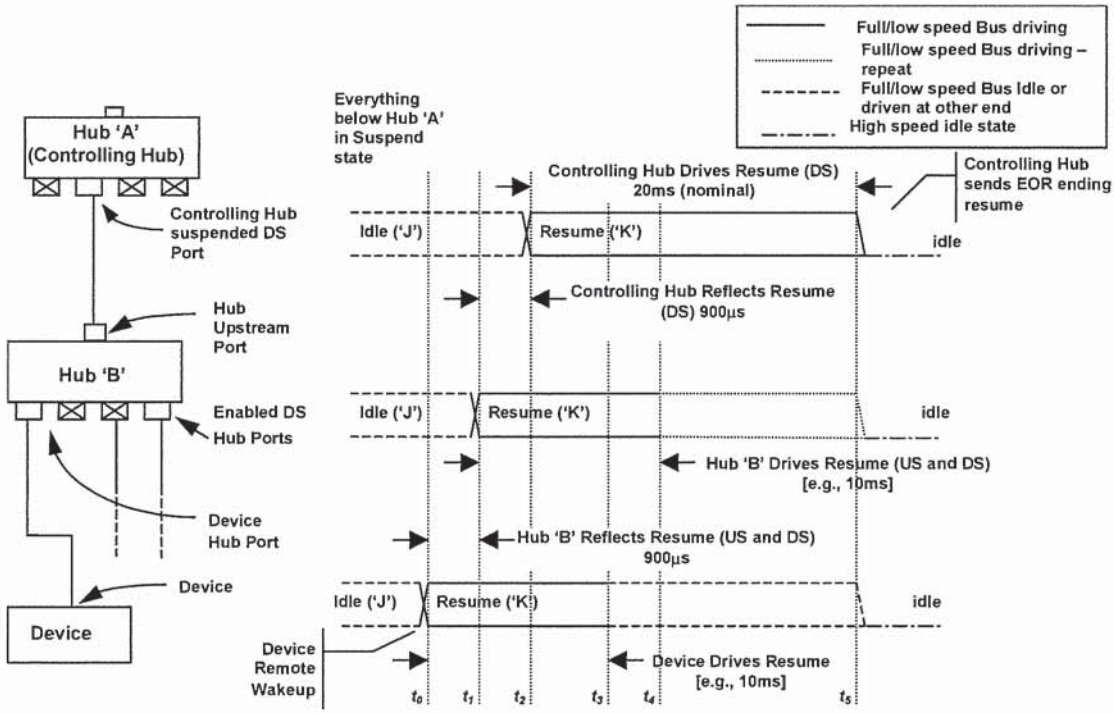

Figure 11-18. Example Remote-wakeup Resume Signaling With High-speed Device

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.

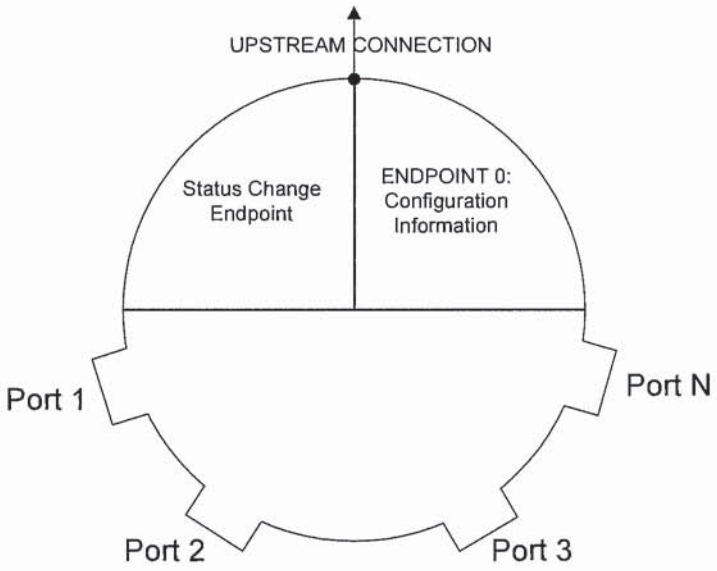

Figure 11-19. Example Hub Controller Organization

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.

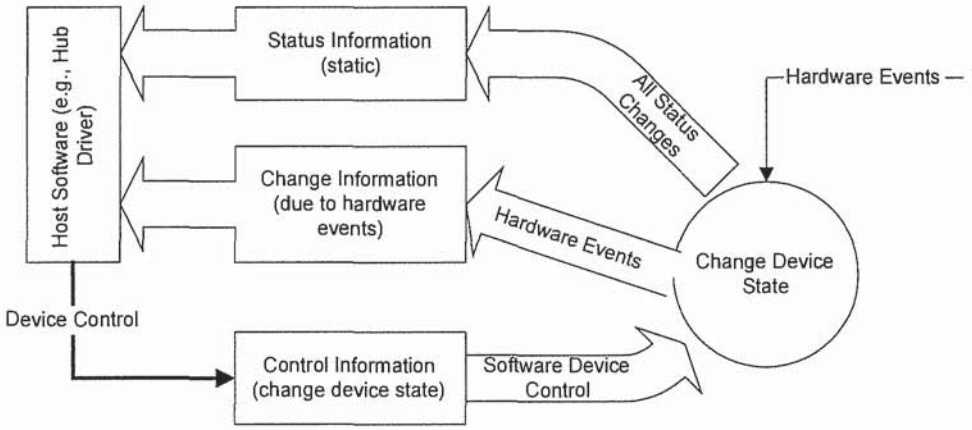

Figure 11-20. Relationship of Status, Status Change, and Control Information to Device States

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).

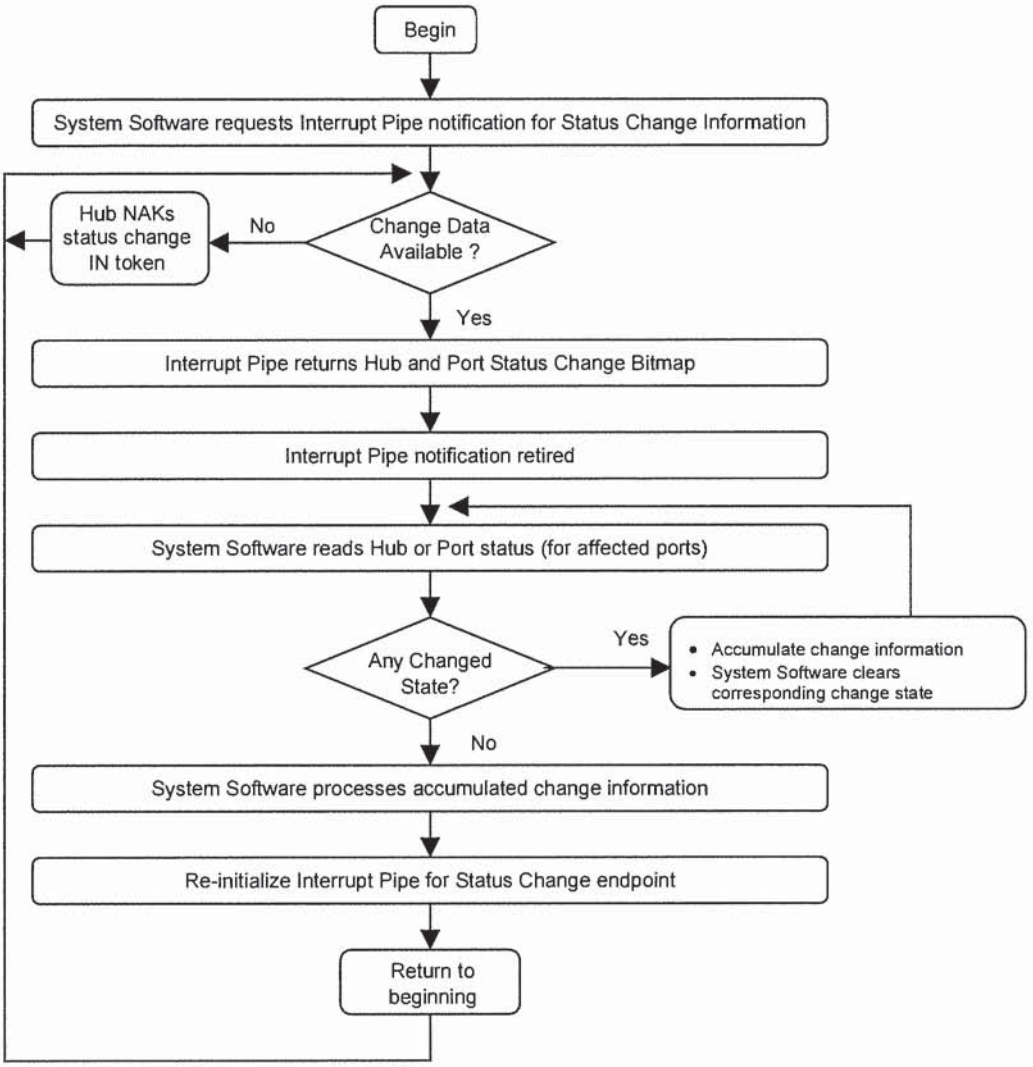

Figure 11-21. Port Status Handling Method

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).

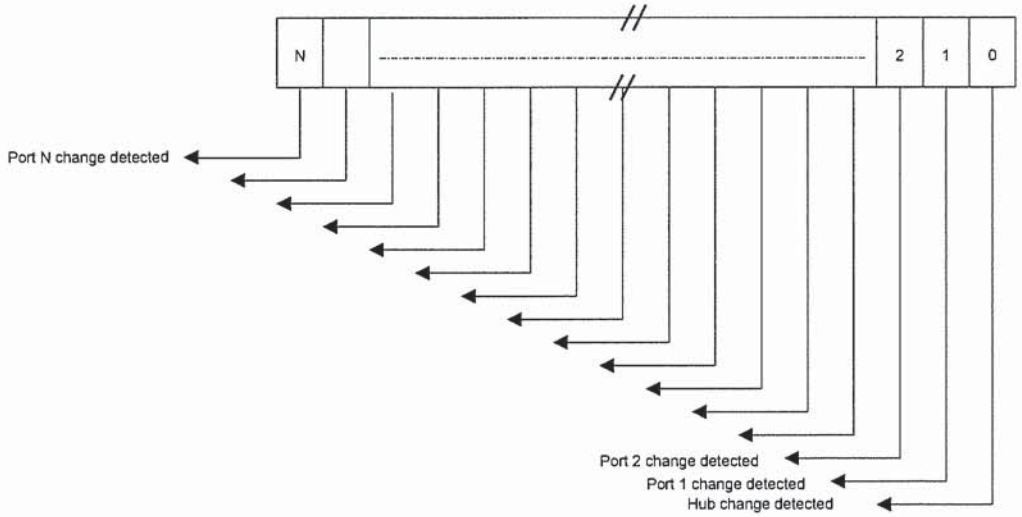

Figure 11-22. Hub and Port Status Change Bitmap

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.

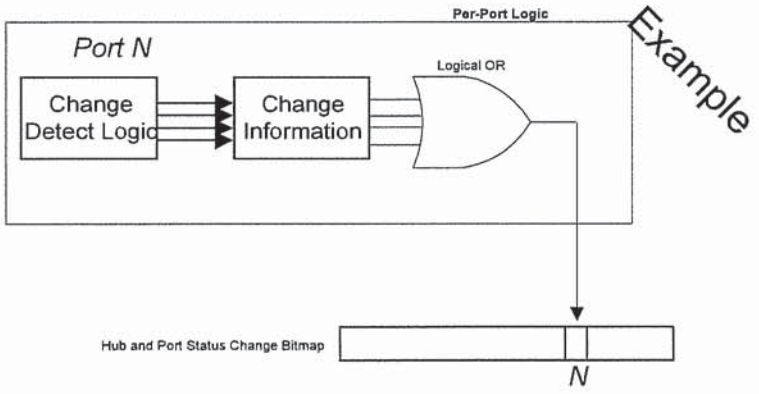

Figure 11-23. Example Hub and Port Change Bit Sampling

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 This is an illegal set of information.
0	1	0	N/A A device which is only self-powered, but does not have local power cannot connect to the bus and communicate.
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.

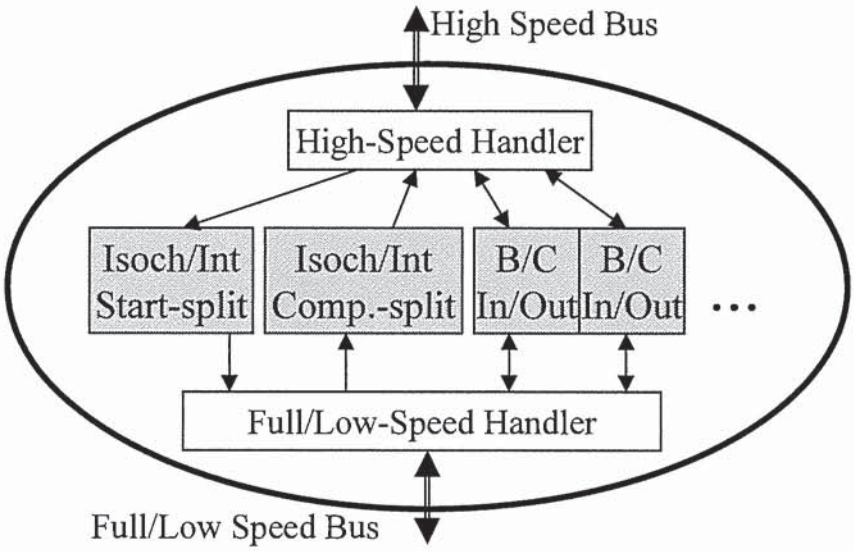

Figure 11-24. Transaction Translator Overview

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.

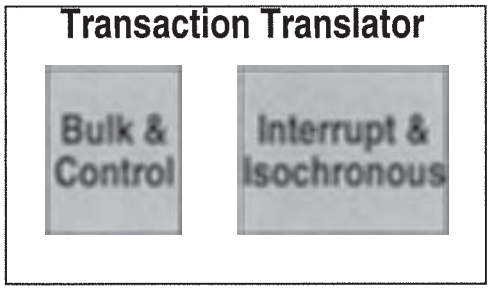

Figure 11-25. Periodic and Non-periodic Buffer Sections of TT

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.

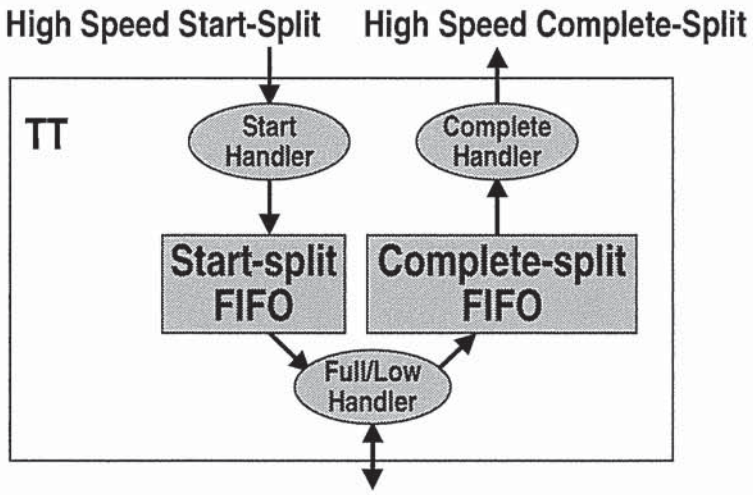

Figure 11-26. TT Microframe Pipeline for Periodic Split Transactions

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.

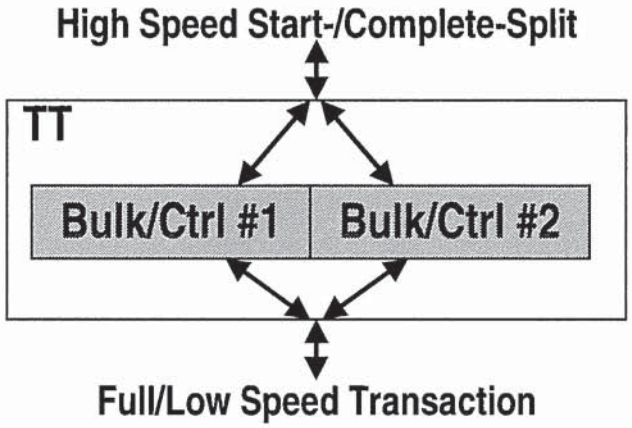

Figure 11-27. TT Nonperiodic Buffering

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.

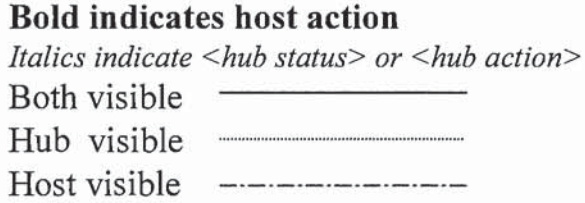

Figure 11-29. Flow Sequence Legend

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.

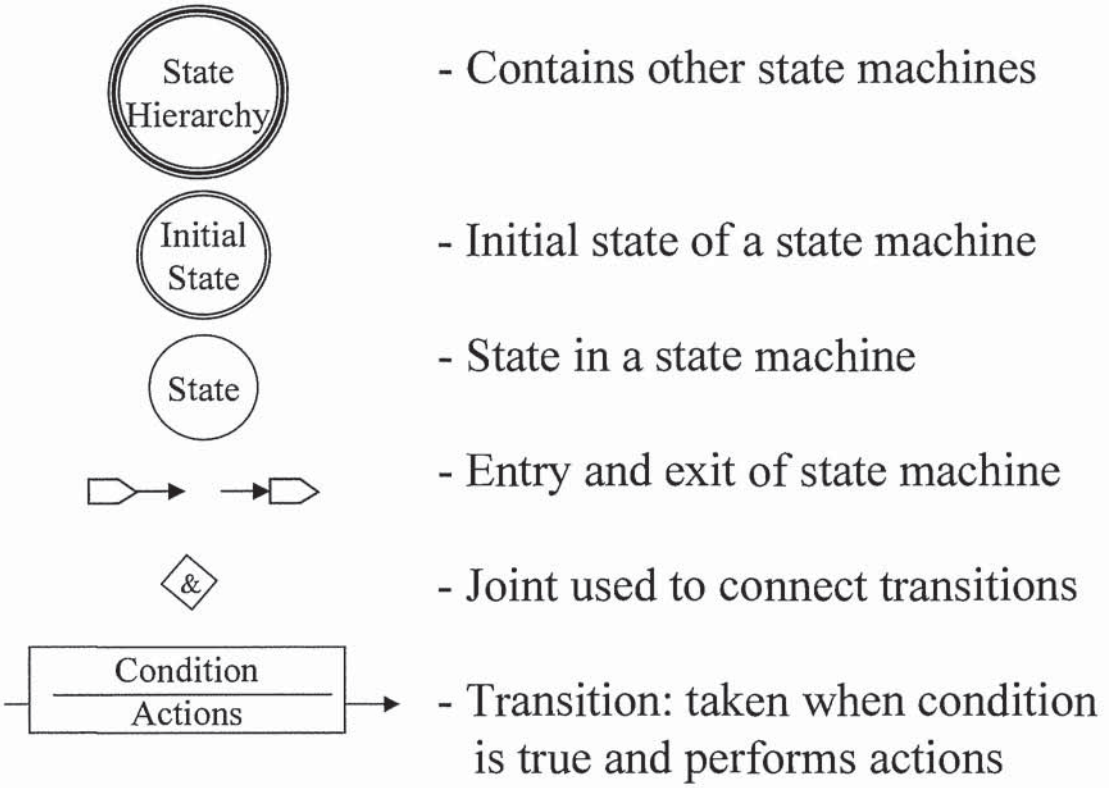

Figure 11-30. Legend for State Machines

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.

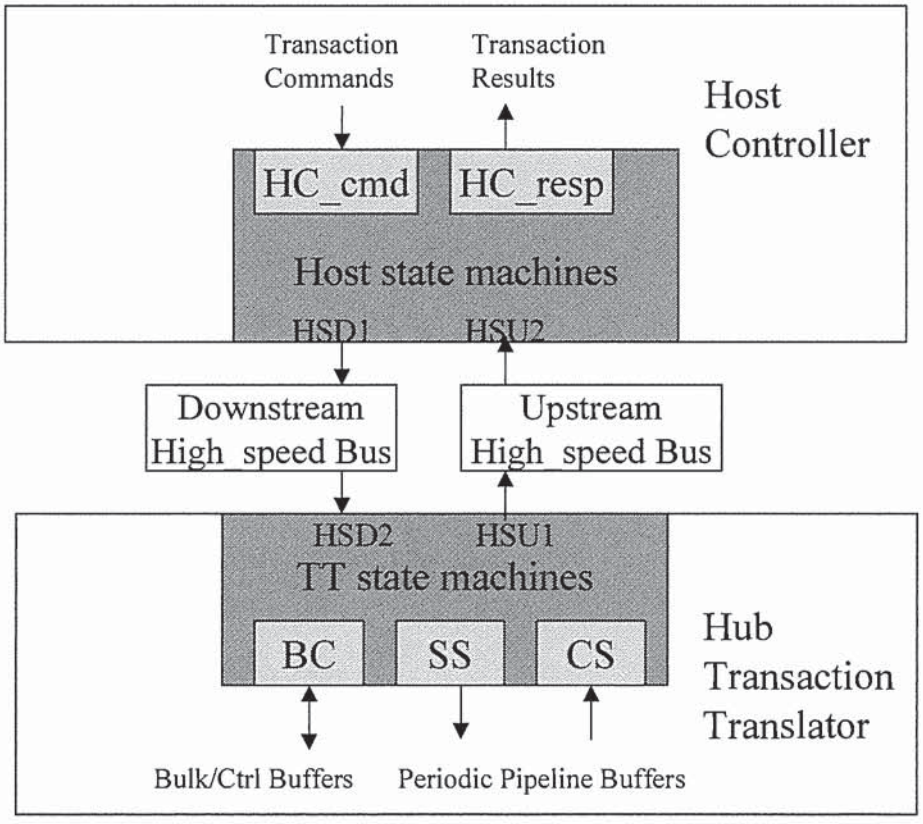

Figure 11-31. State Machine Context Overview

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.

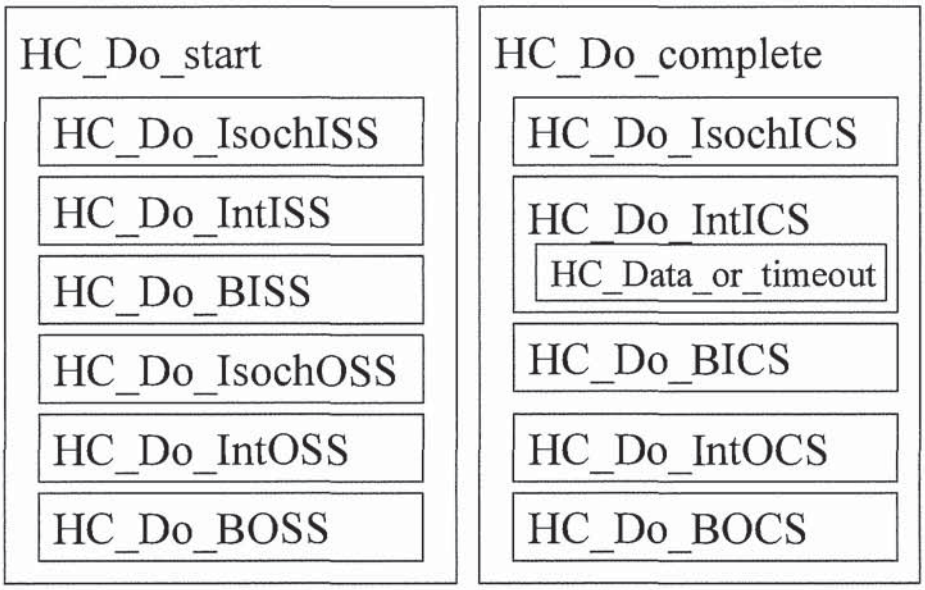

Figure 11-32. Host Controller Split Transaction State Machine Hierarchy Overview

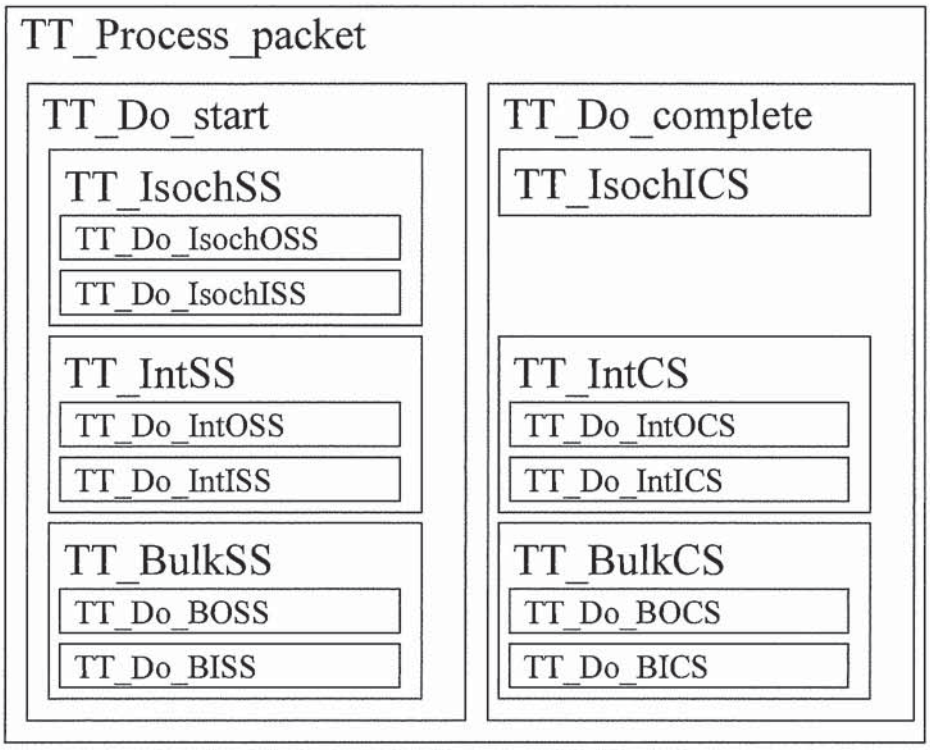

Figure 11-33. Transaction Translator State Machine Hierarchy Overview

11.16.1 Host Controller State Machine

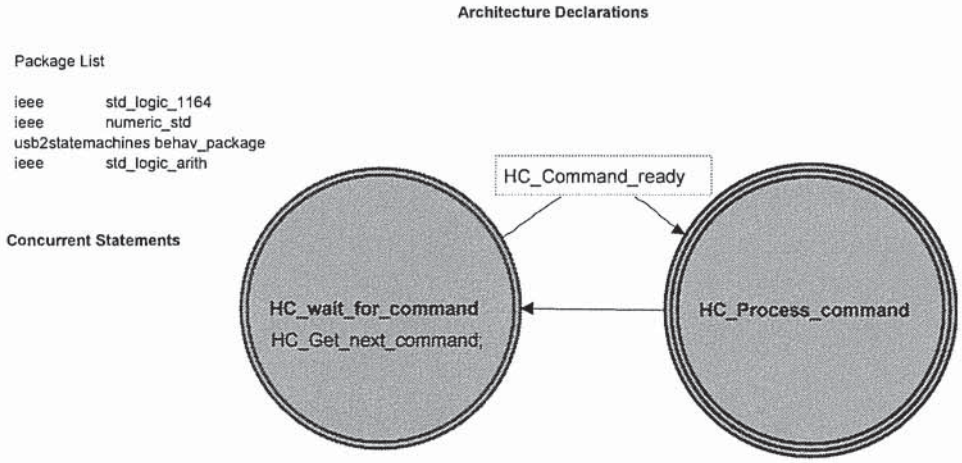

Figure 11-34. Host Controller

11.16.1.1 HC_Process_command State Machine

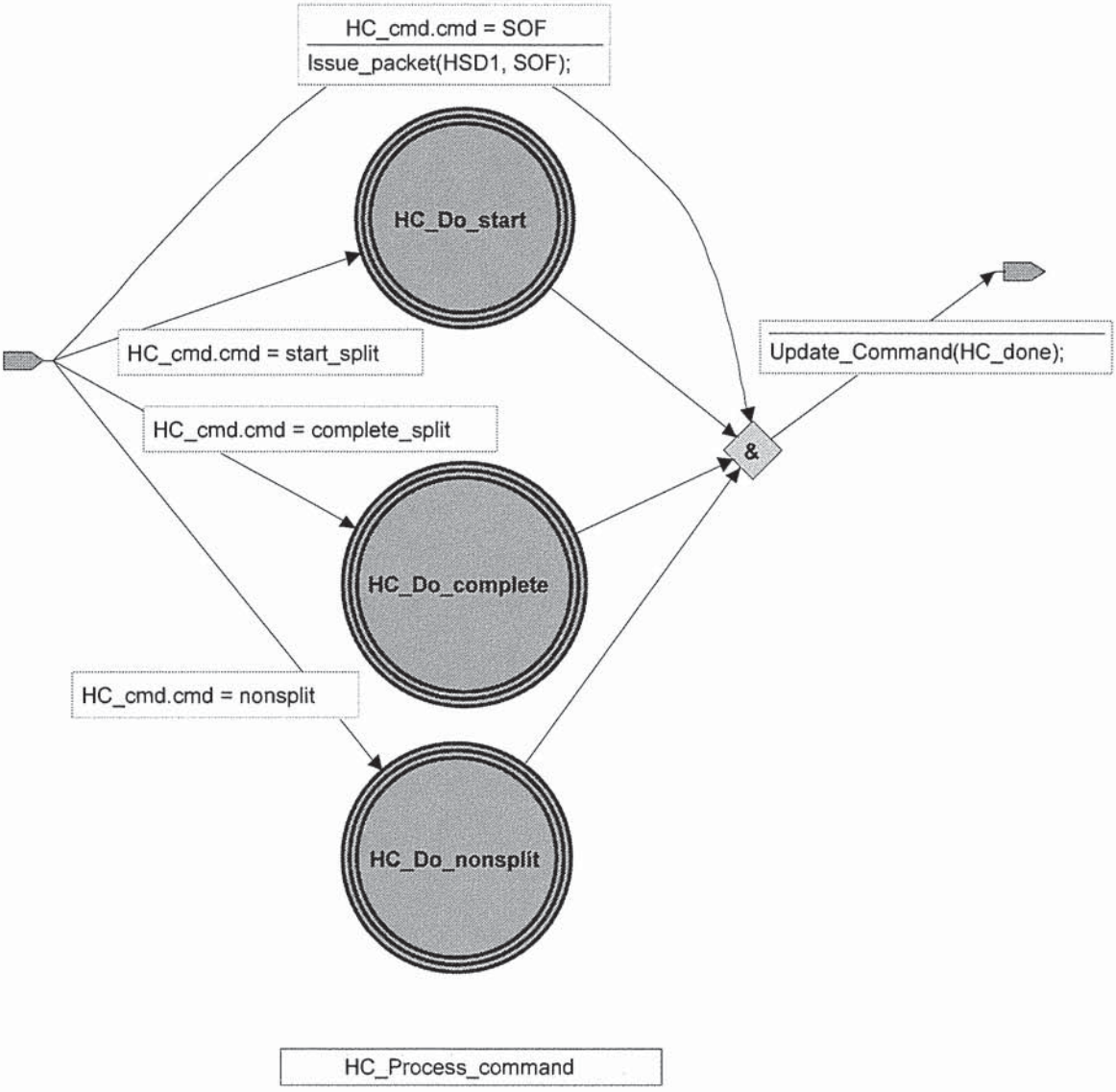

Figure 11-35. HC_Process_Command

11.16.1.1.1 HC_Do_start State Machine

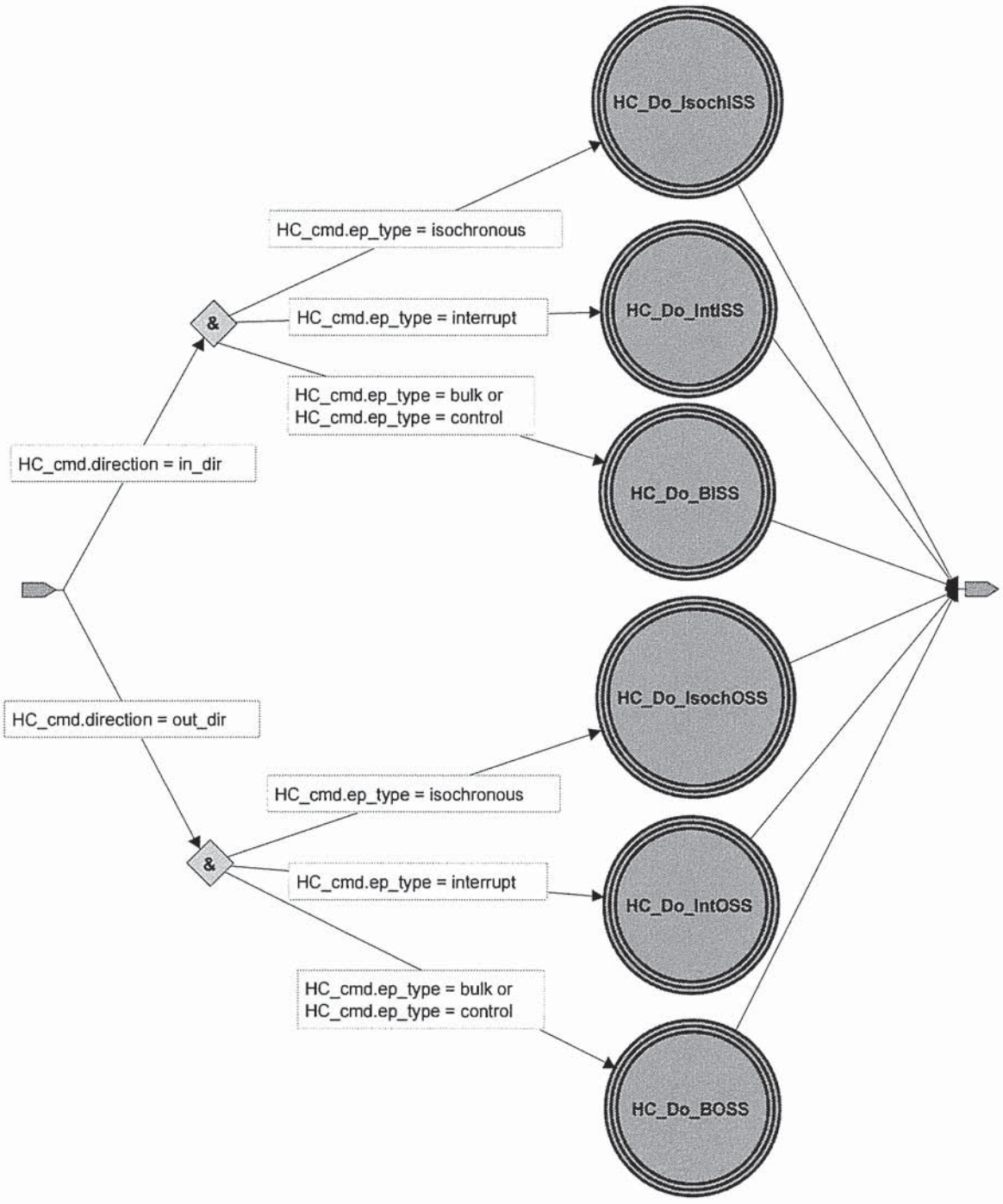

Figure 11-36. HC_Do_Start

11.16.1.1.2 HC_Do_complete State Machine

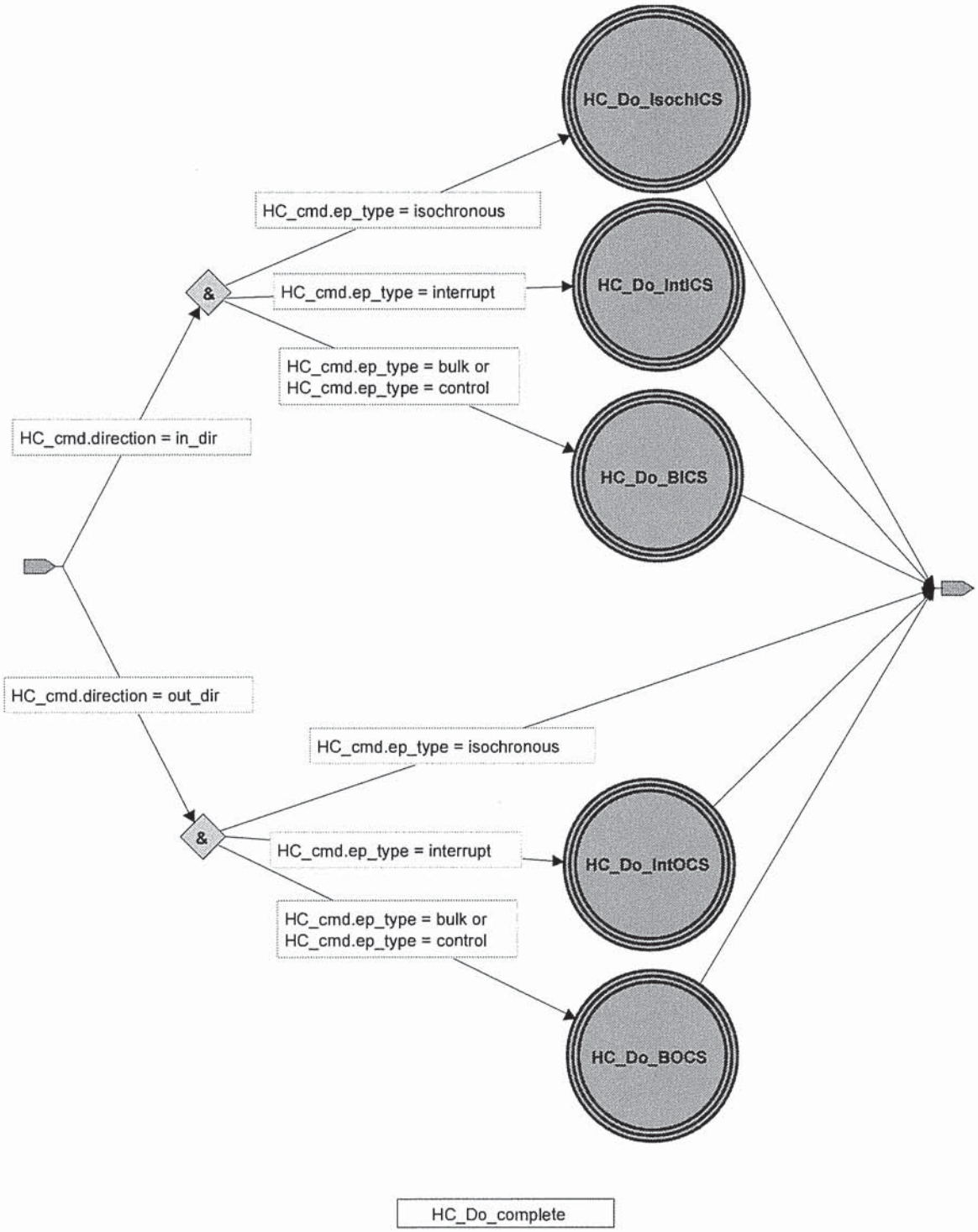

Figure 11-37. HC_Do_Complete

11.16.2 Transaction Translator State Machine

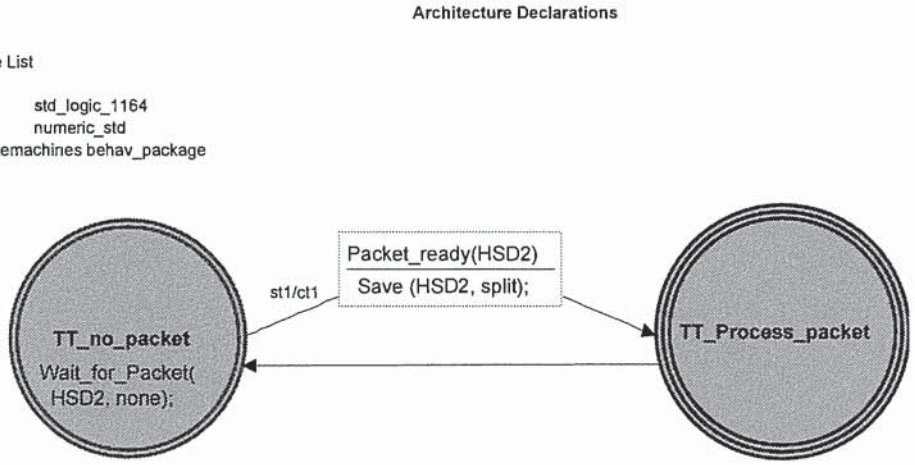

Figure 11-38. Transaction Translator

11.16.2.1 TT_Process_packet State Machine

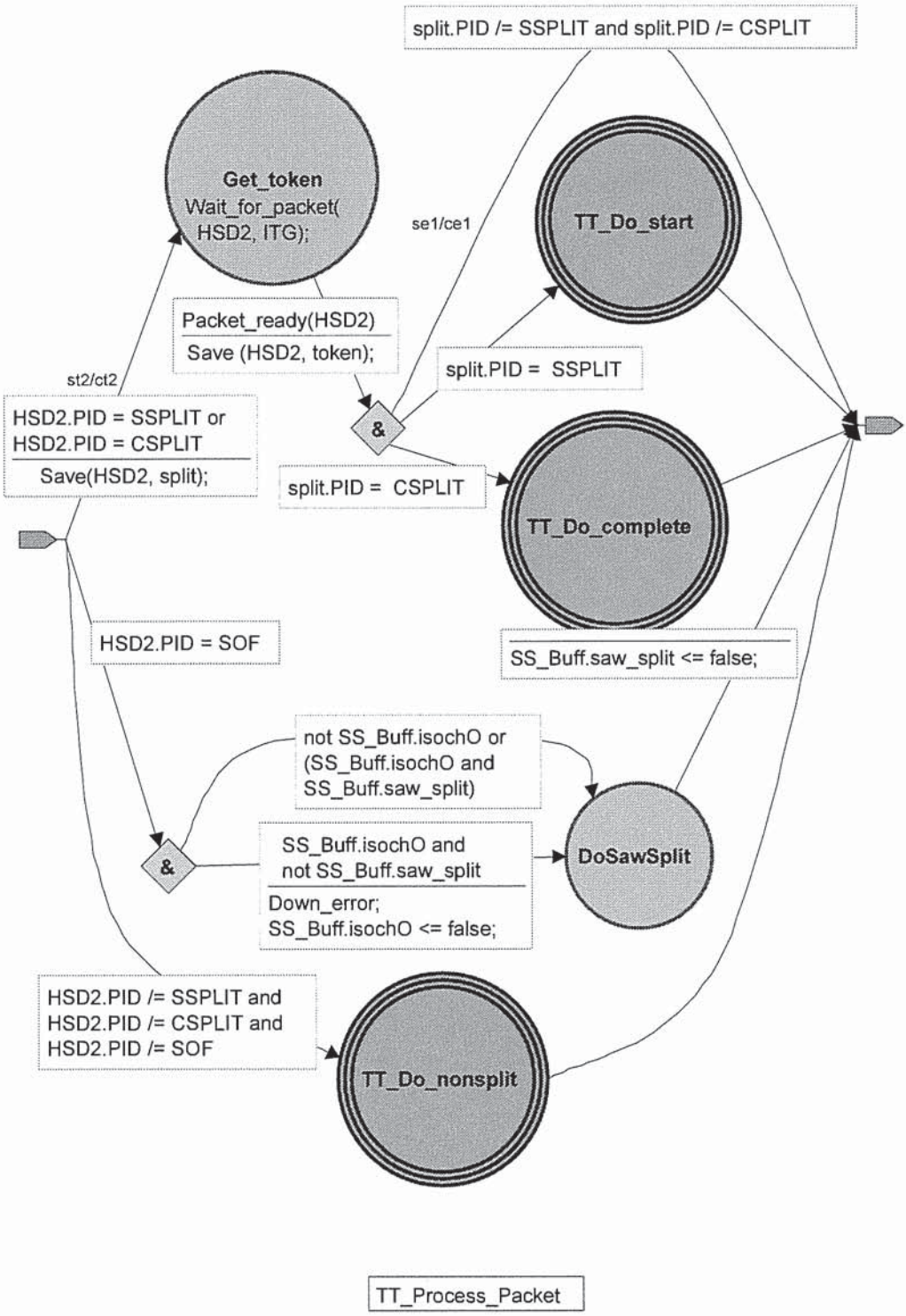

Figure 11-39. TT_Process_Packet

11.16.2.1.1 TT_Do_Start State Machine

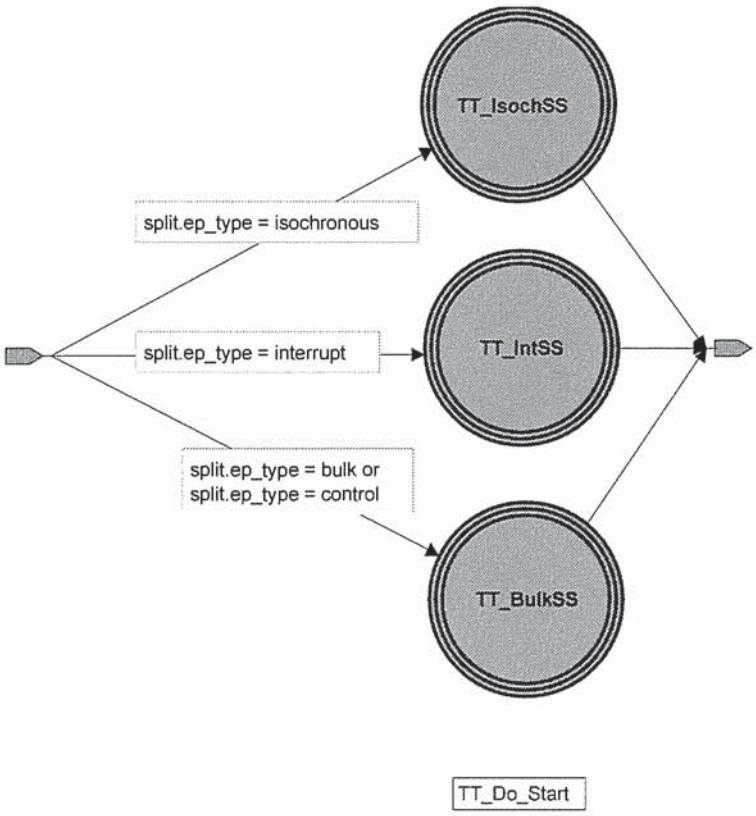

Figure 11-40. TT_Do_Start

11.16.2.1.2 TT_Do_Complete State Machine

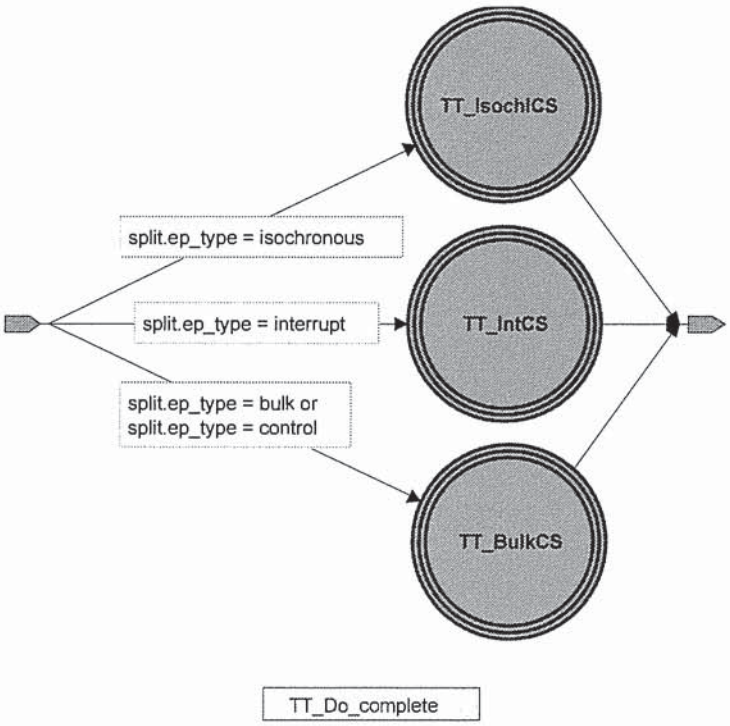

Figure 11-41. TT_Do_Complete

11.16.2.1.3 TT_BulkSS State Machine

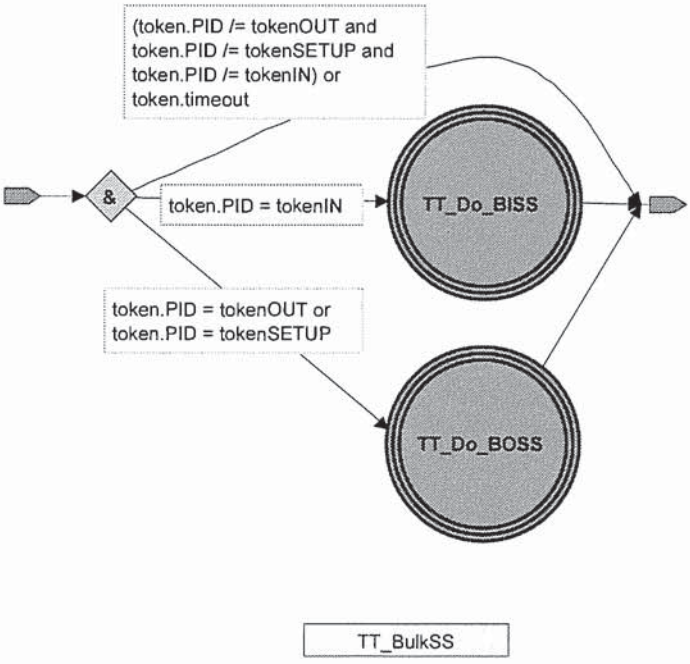

Figure 11-42. TT_BulkSS

11.16.2.1.4 TT_BulkCS State Machine

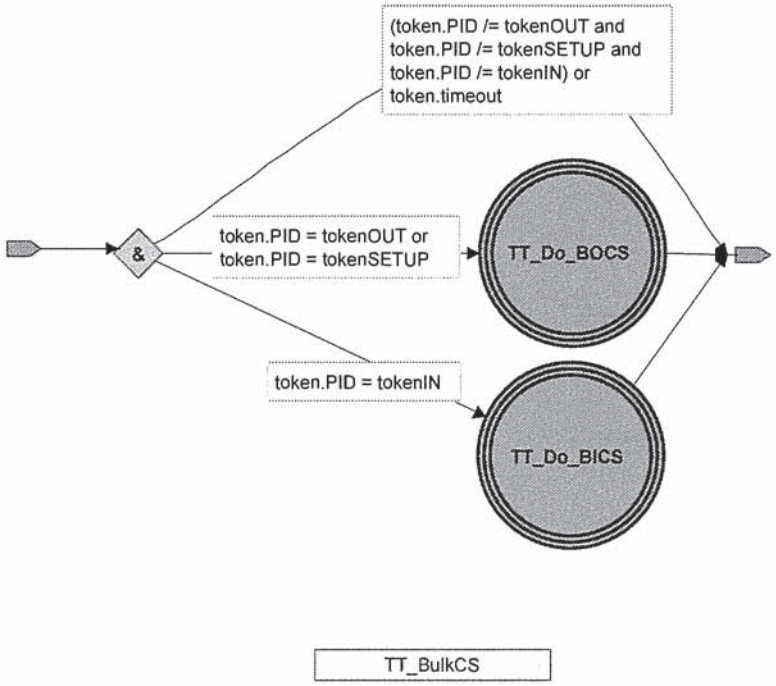

Figure 11-43. TT_BulkCS

11.16.2.1.5 TT_IntSS State Machine

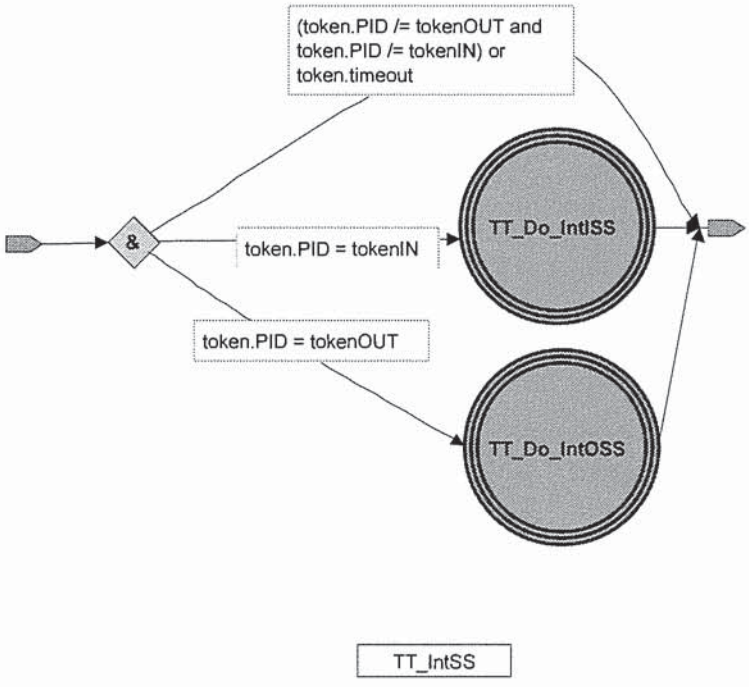

Figure 11-44. TT_IntSS

11.16.2.1.6 TT_IntCS State Machine

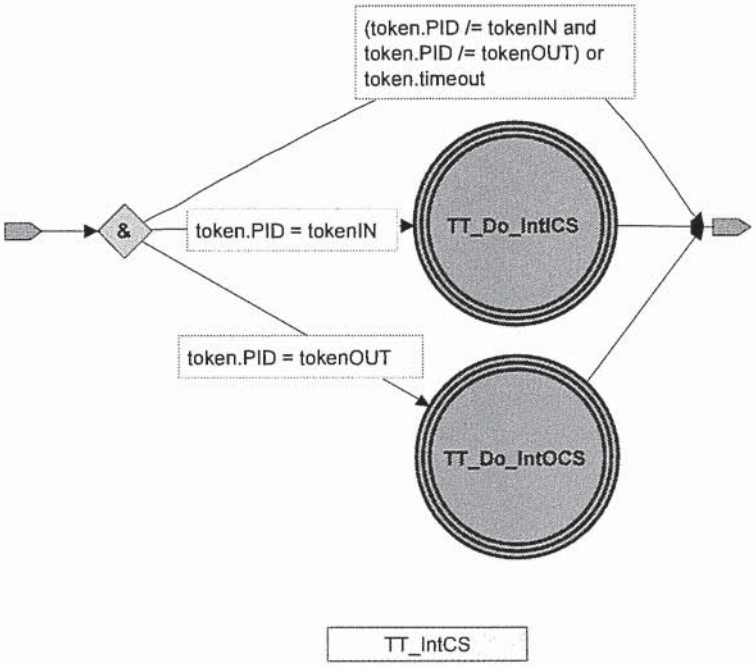

Figure 11-45. TT_IntCS

11.16.2.1.7 TT_IsochSS State Machine

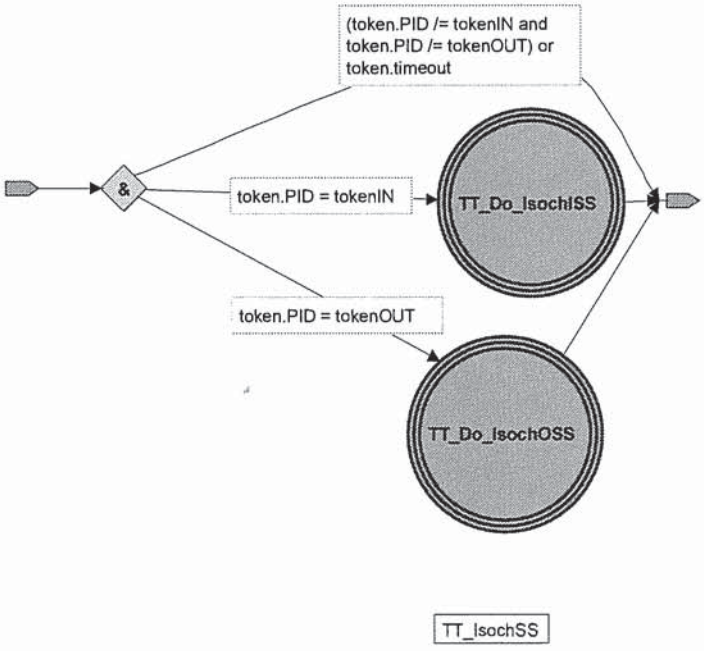

Figure 11-46. TT_IsochSS

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.

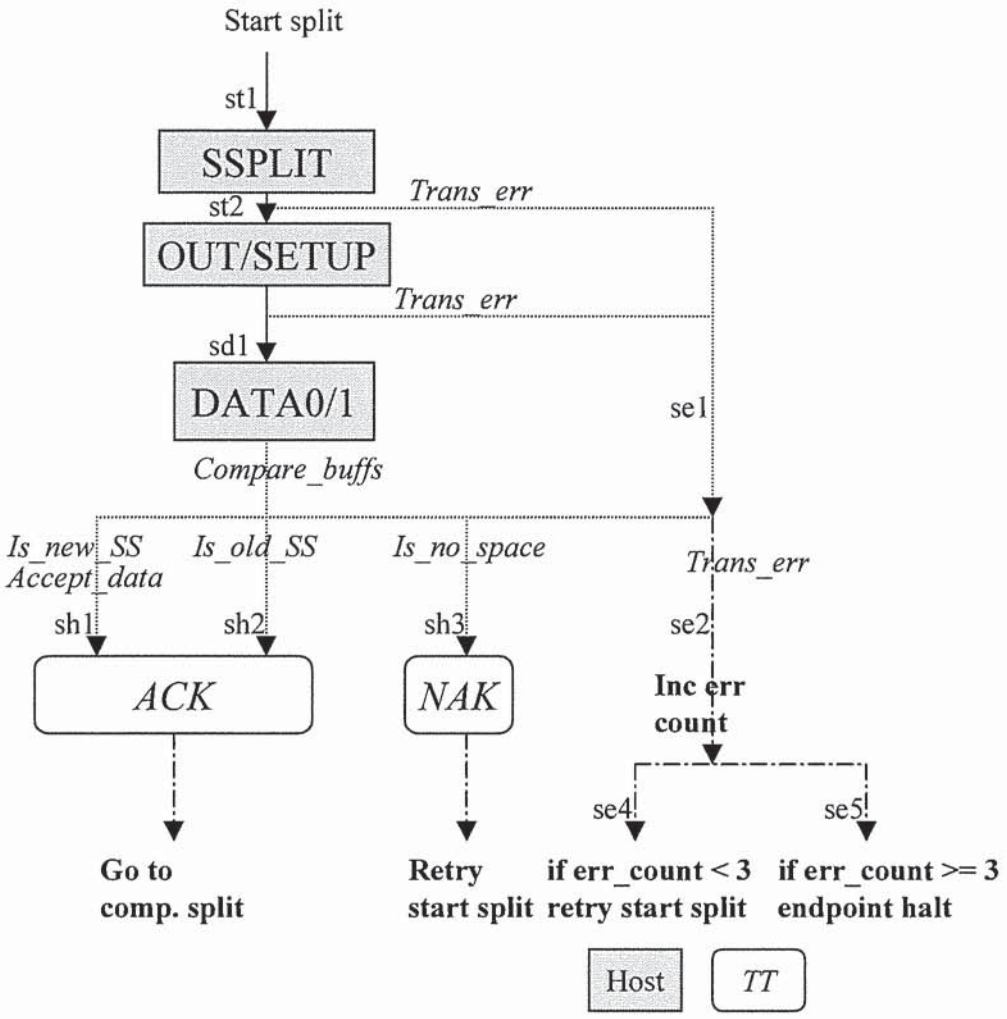

Figure 11-48. Bulk/Control OUT Start-split Transaction Sequence

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.

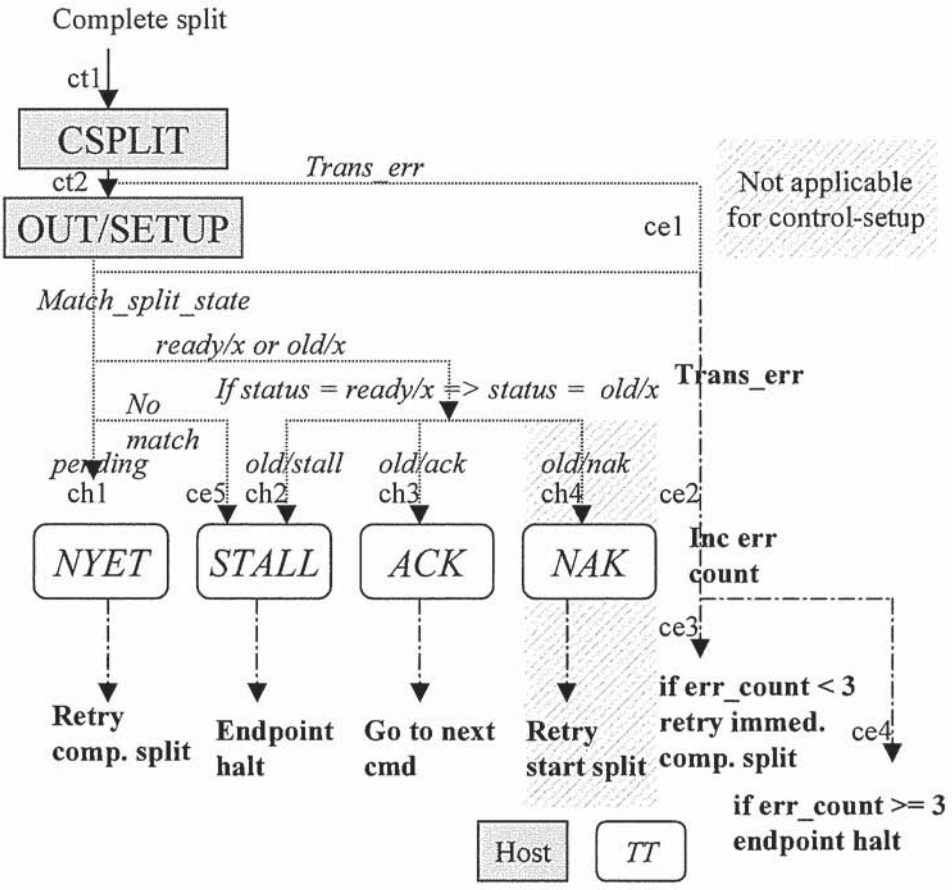

Figure 11-49. Bulk/Control OUT Complete-split Transaction Sequence

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.

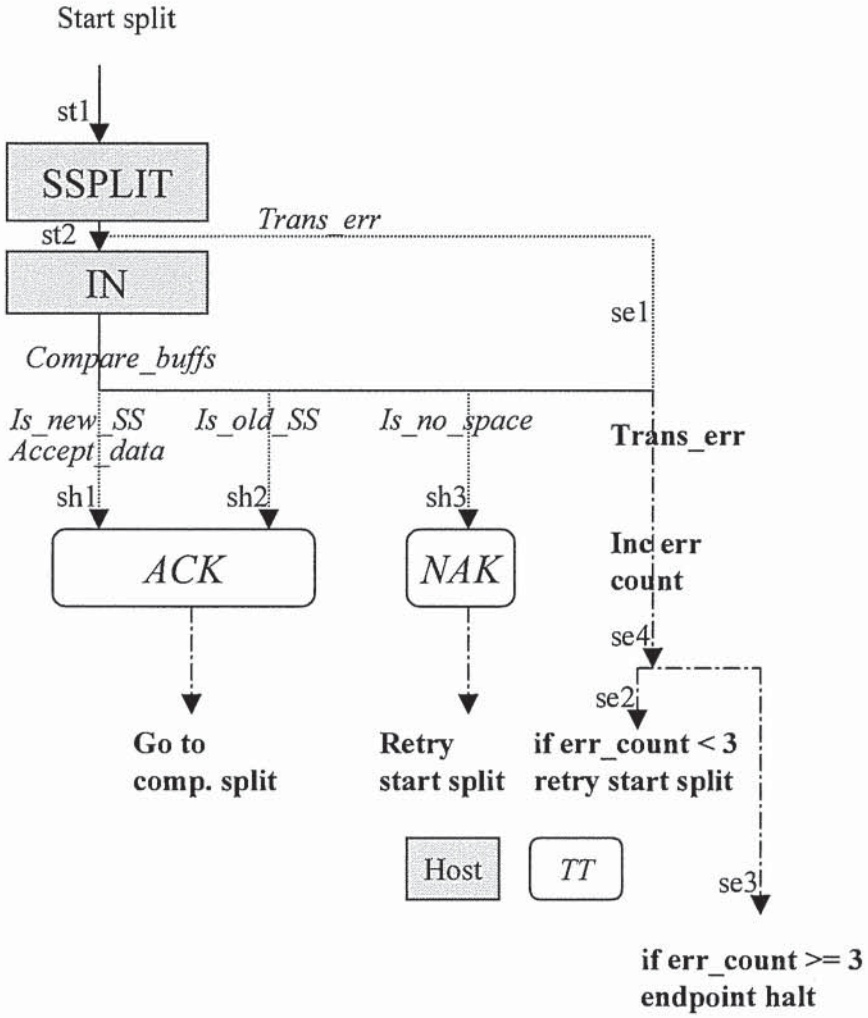

Figure 11-50. Bulk/Control IN Start-split Transaction Sequence

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.

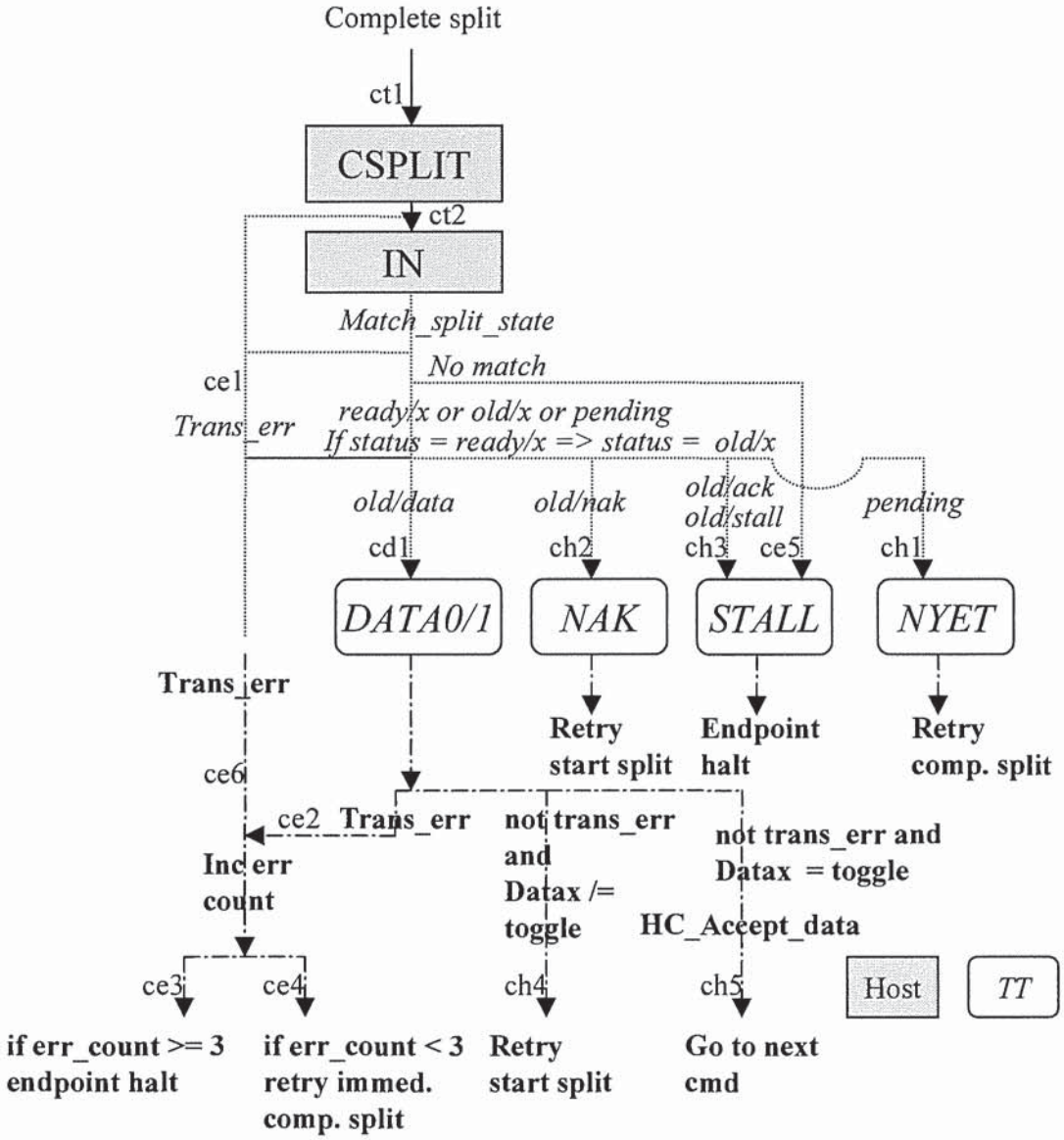

Figure 11-51. Bulk/Control IN Complete-split Transaction Sequence

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.

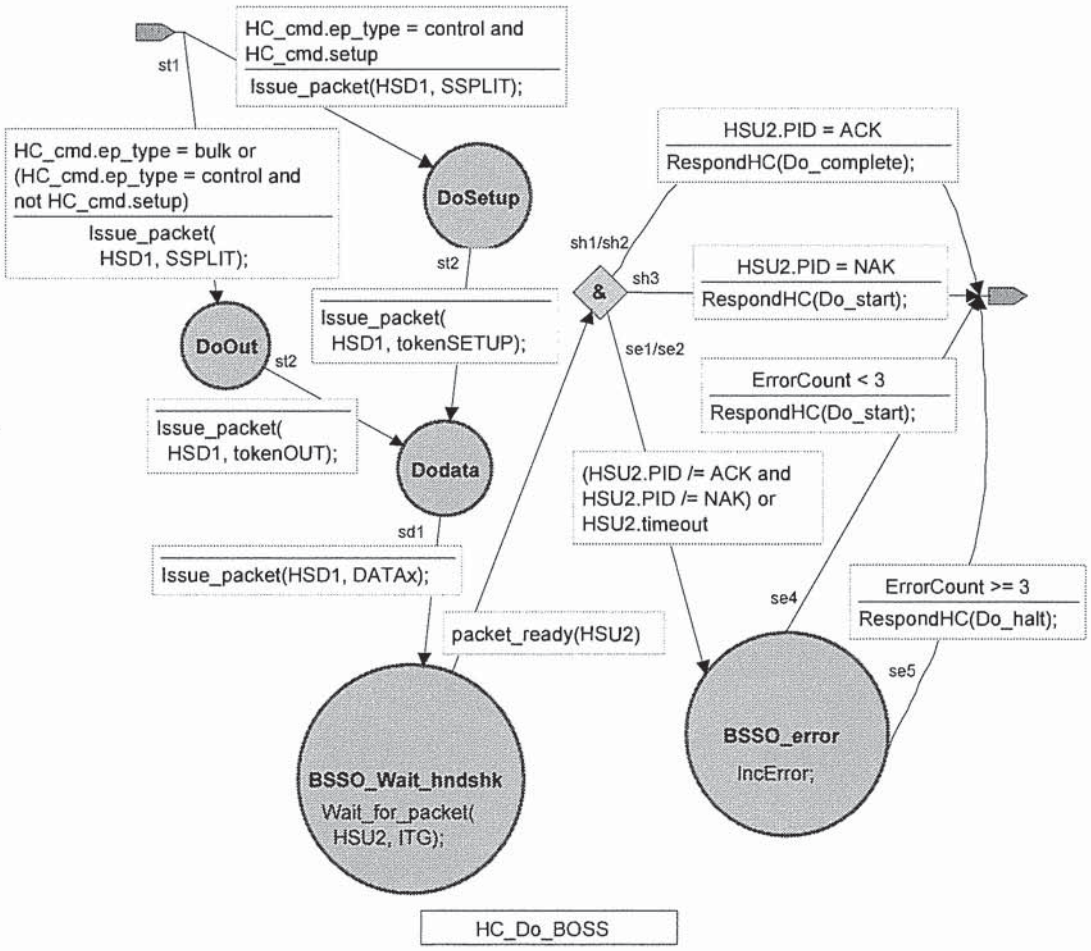

Figure 11-52. Bulk/Control OUT Start-split Transaction Host State Machine

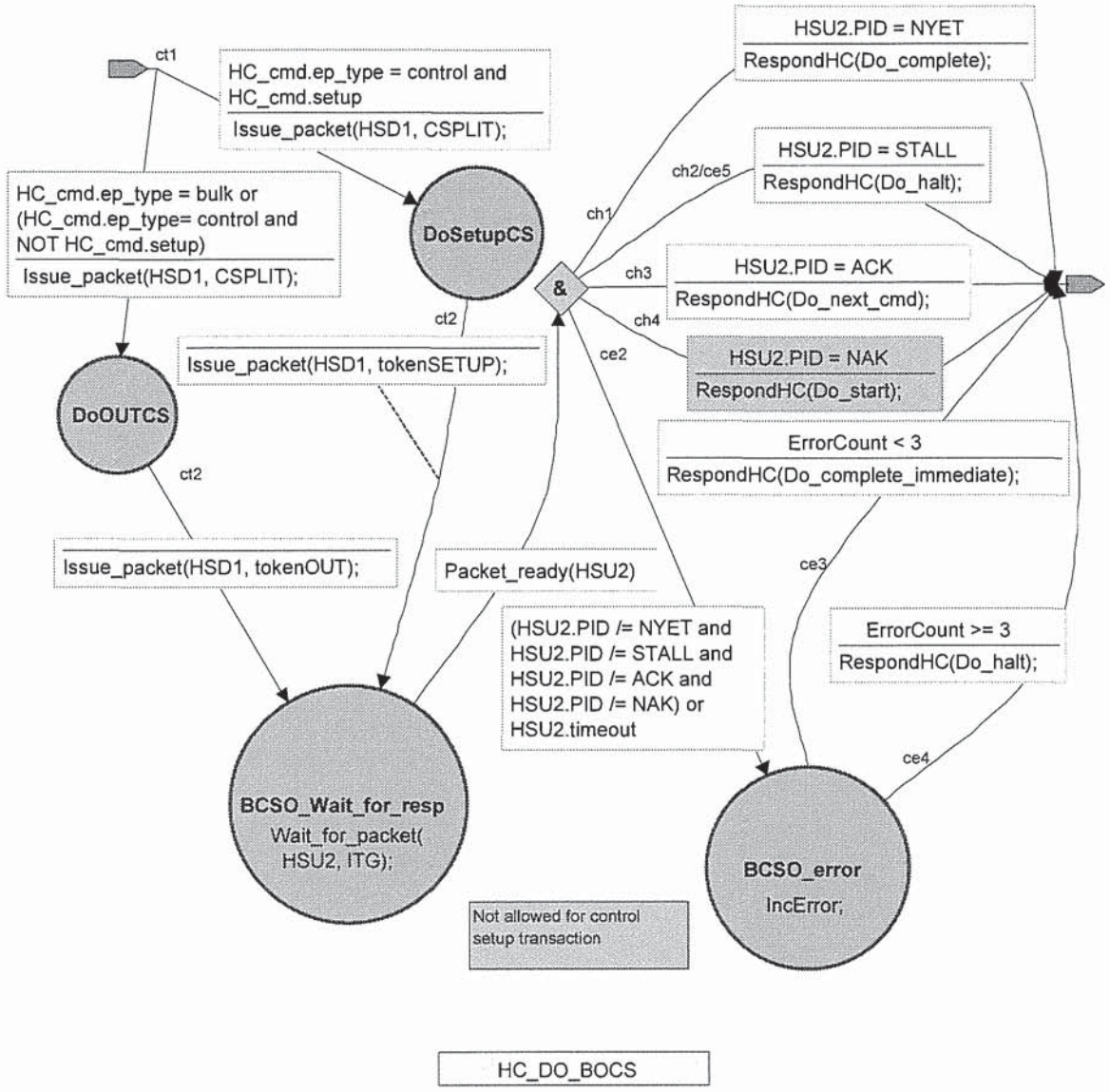

Figure 11-53. Bulk/Control OUT Complete-split Transaction Host State Machine

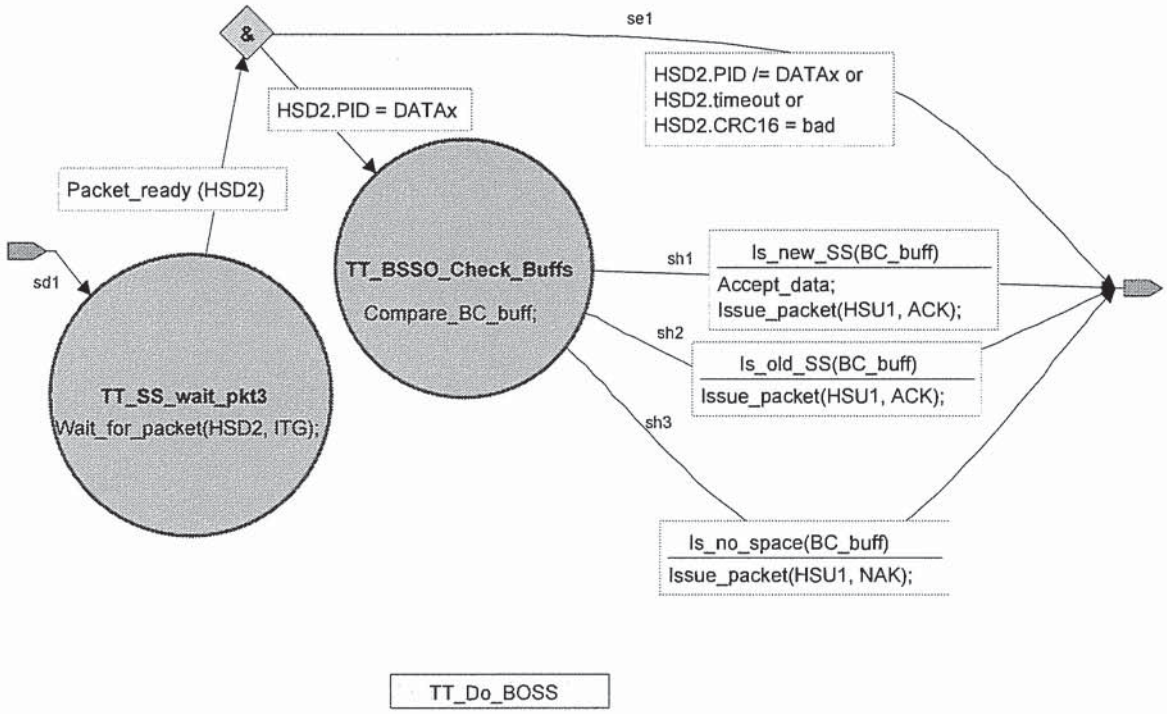

Figure 11-54. Bulk/Control OUT Start-split Transaction TT State Machine

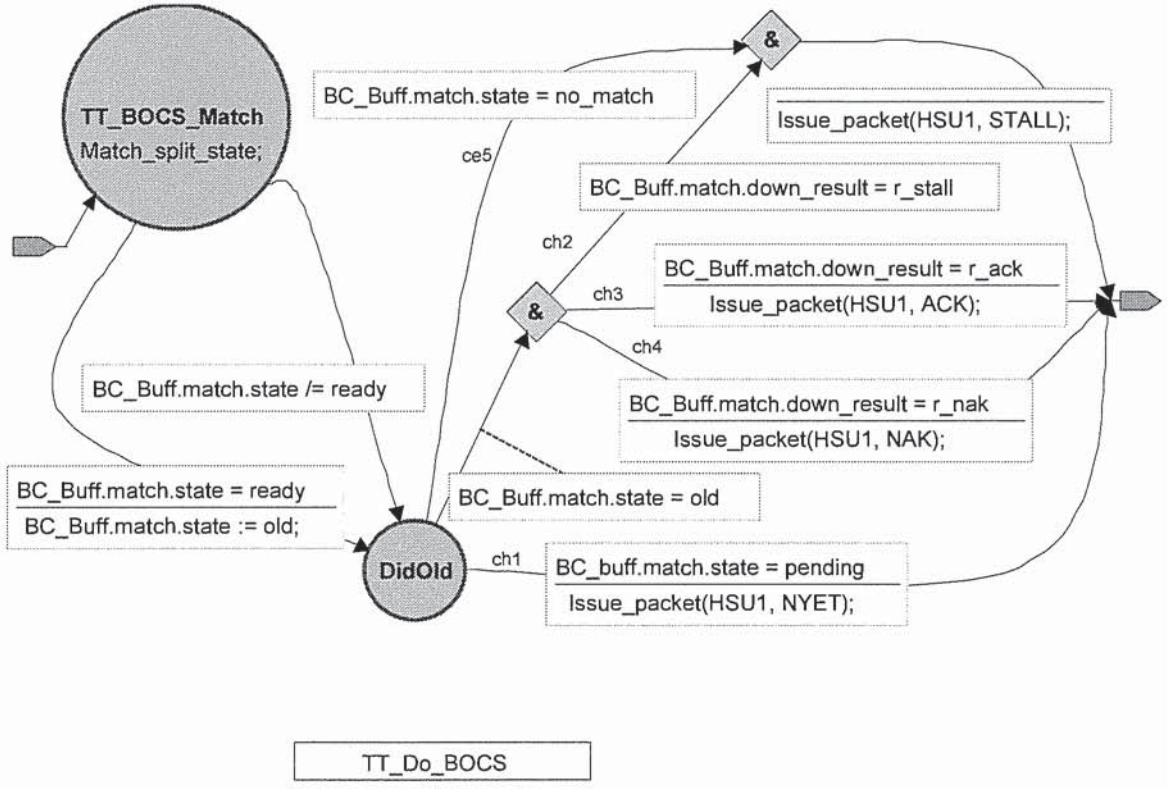

Figure 11-55. Bulk/Control OUT Complete-split Transaction TT State Machine

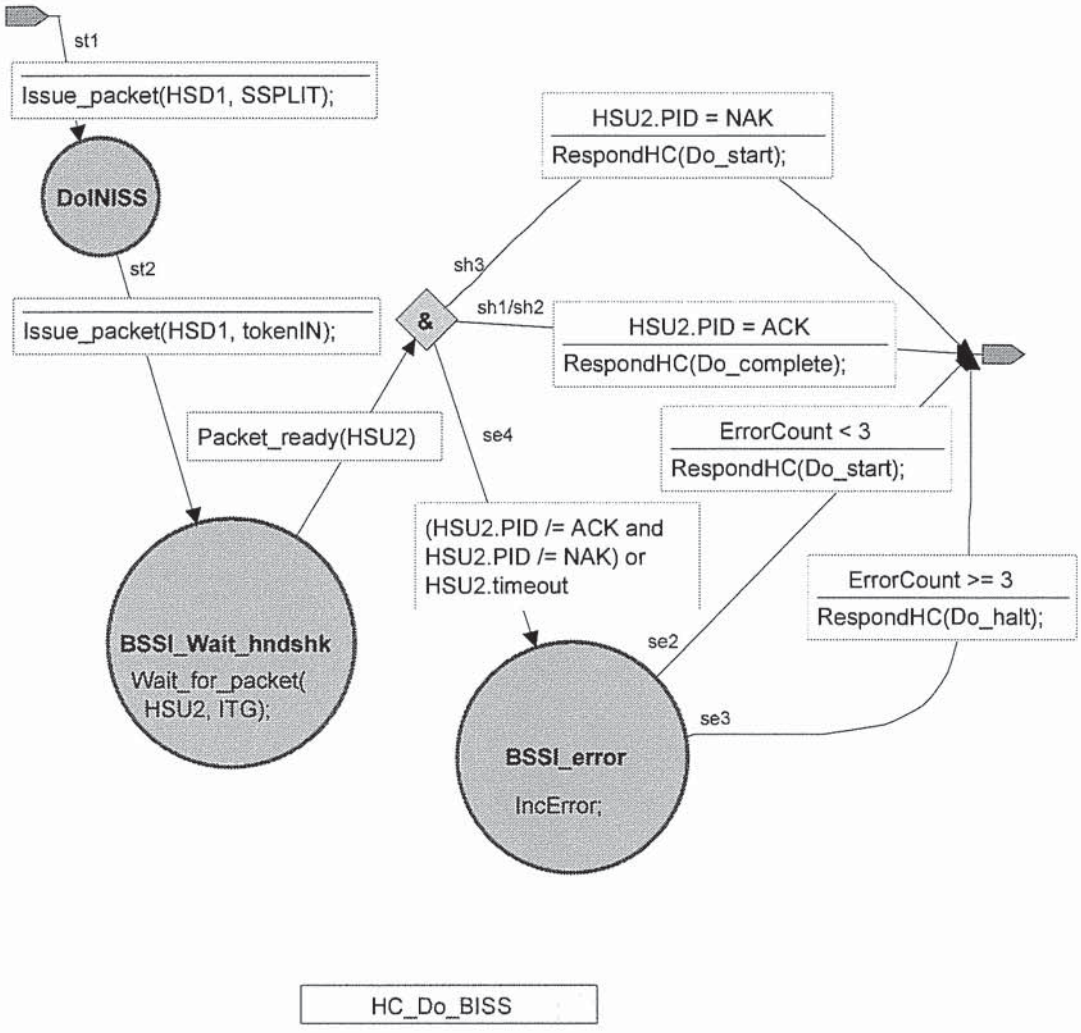

Figure 11-56. Bulk/Control IN Start-split Transaction Host State Machine

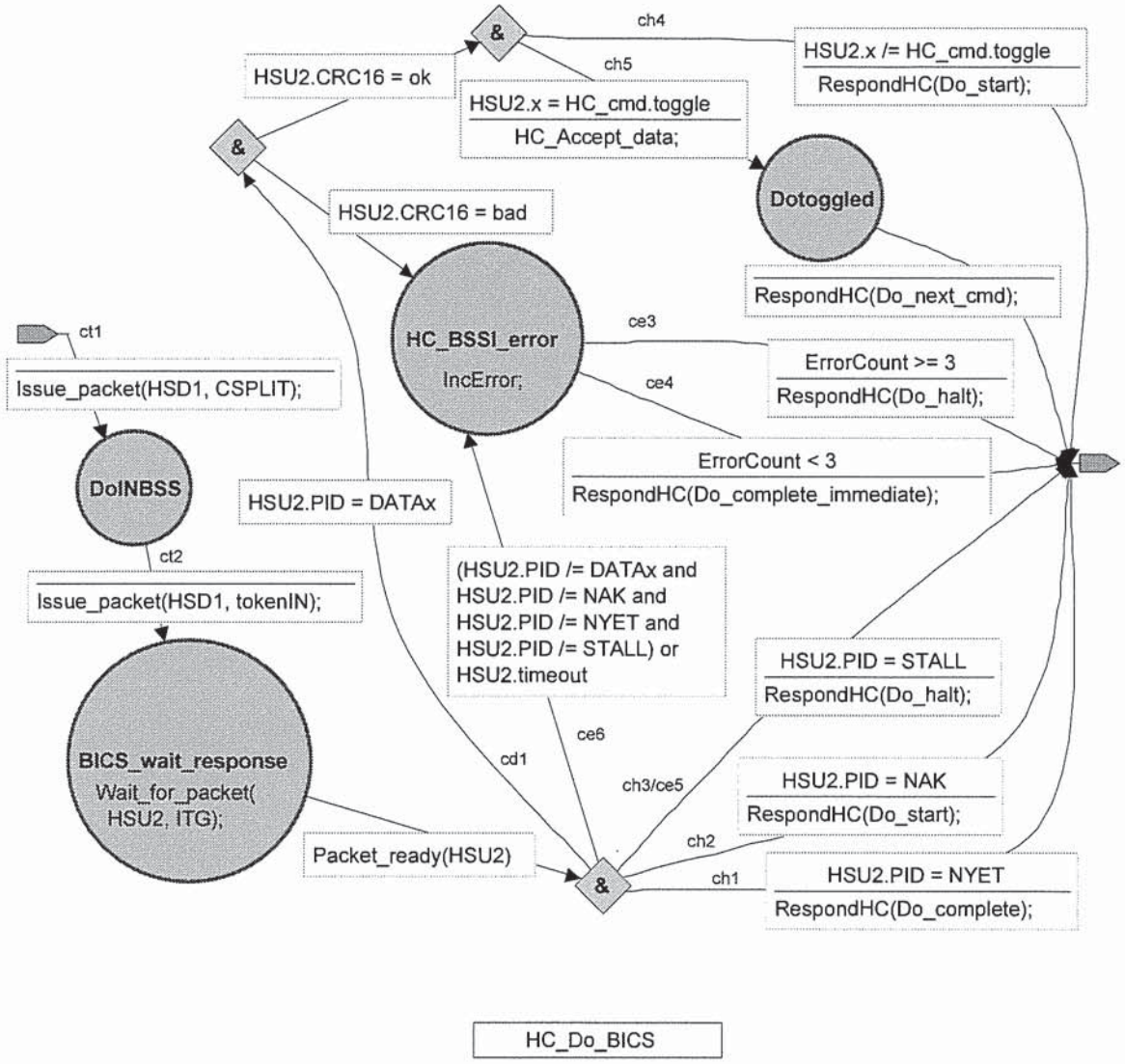

Figure 11-57. Bulk/Control IN Complete-split Transaction Host State Machine

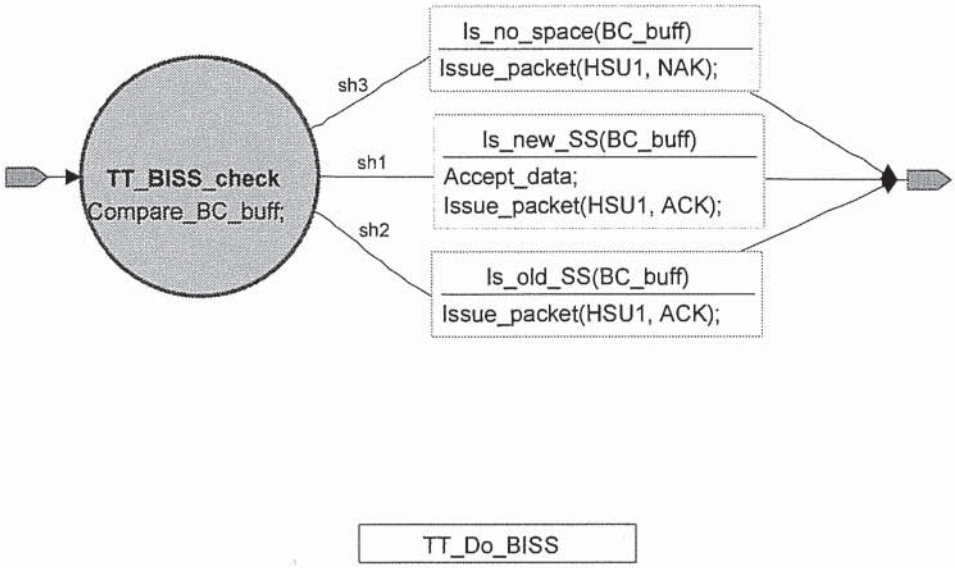

Figure 11-58. Bulk/Control IN Start-split Transaction TT State Machine

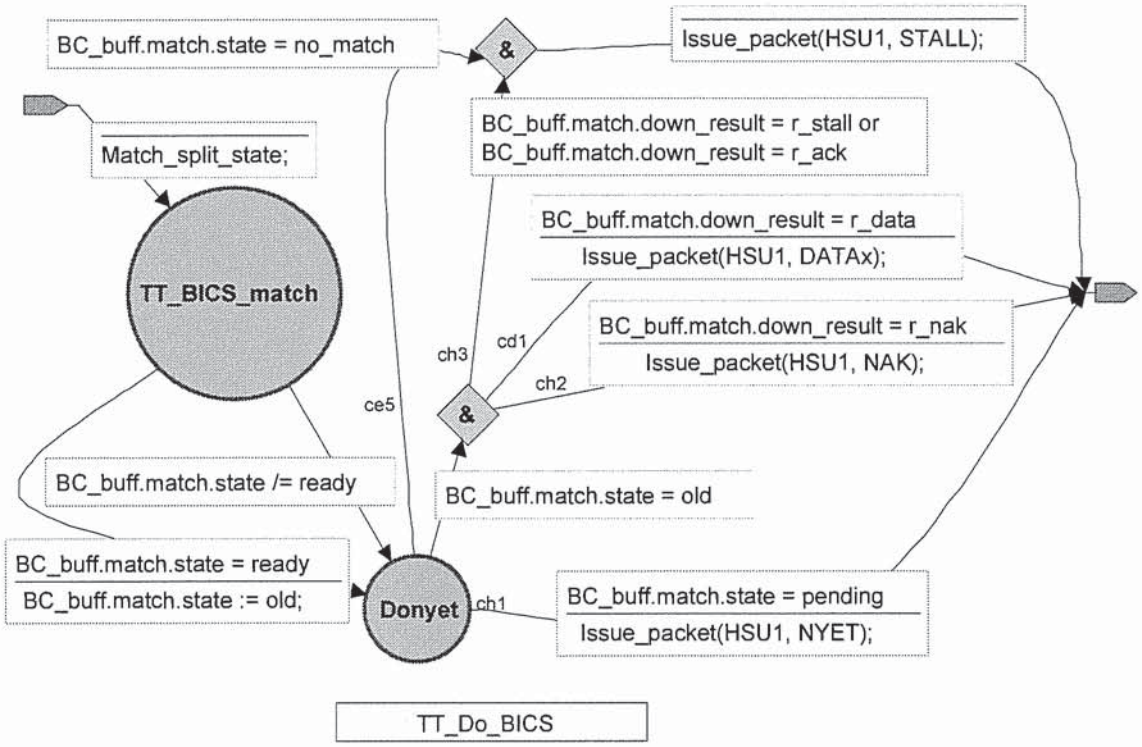

Figure 11-59. Bulk/Control IN Complete-split Transaction TT State Machine

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

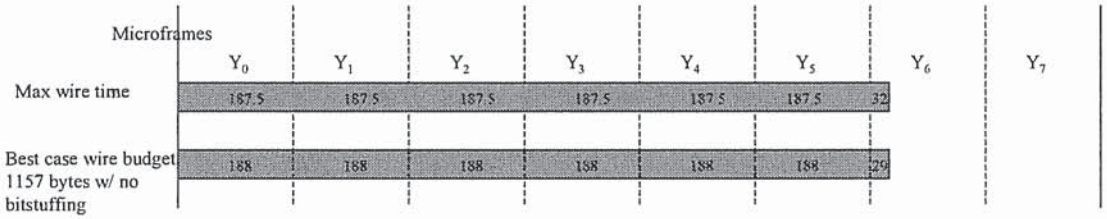

Figure 11-60. Best Case Budgeted Full-speed Wire Time With No Bit Stuffing

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.

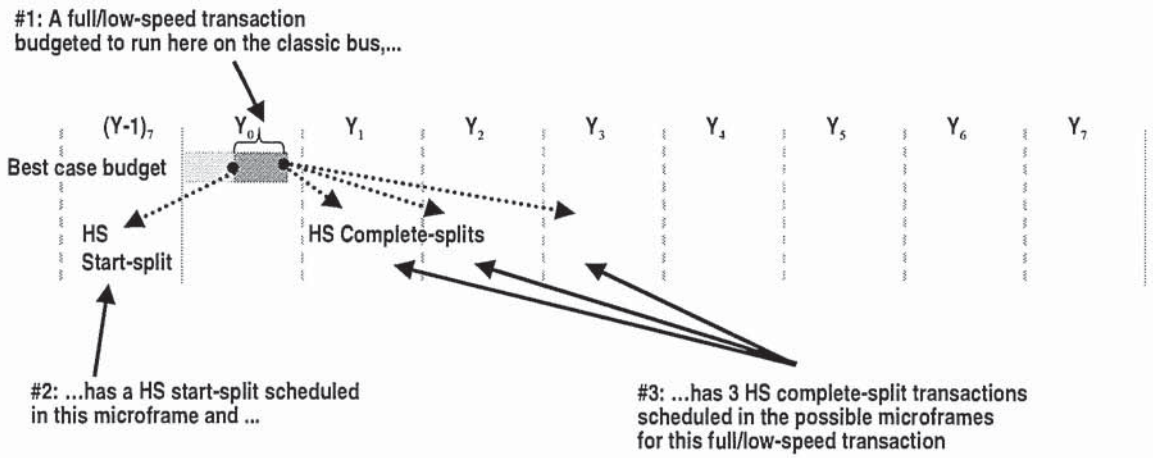

Figure 11-61. Scheduling of TT Microframe Pipeline

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.

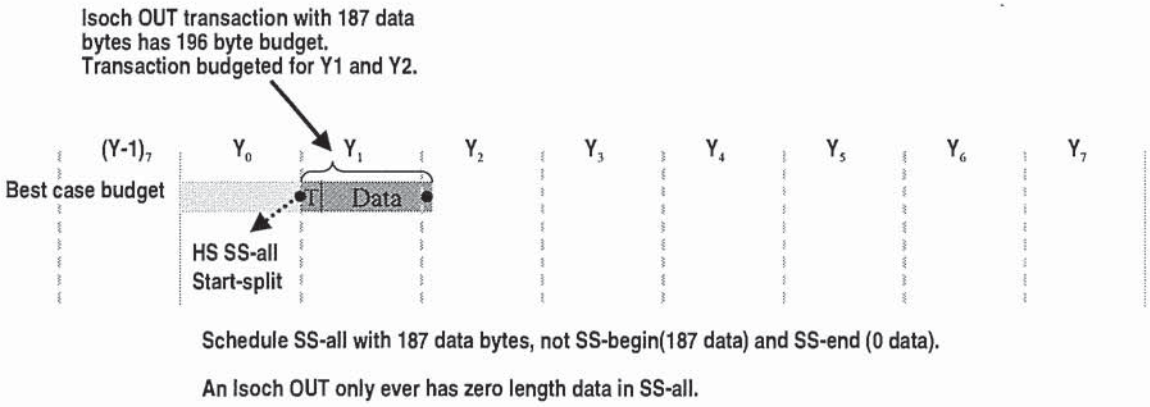

Figure 11-62. Isochronous OUT Example That Avoids a Start-split-end With Zero Data

   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.

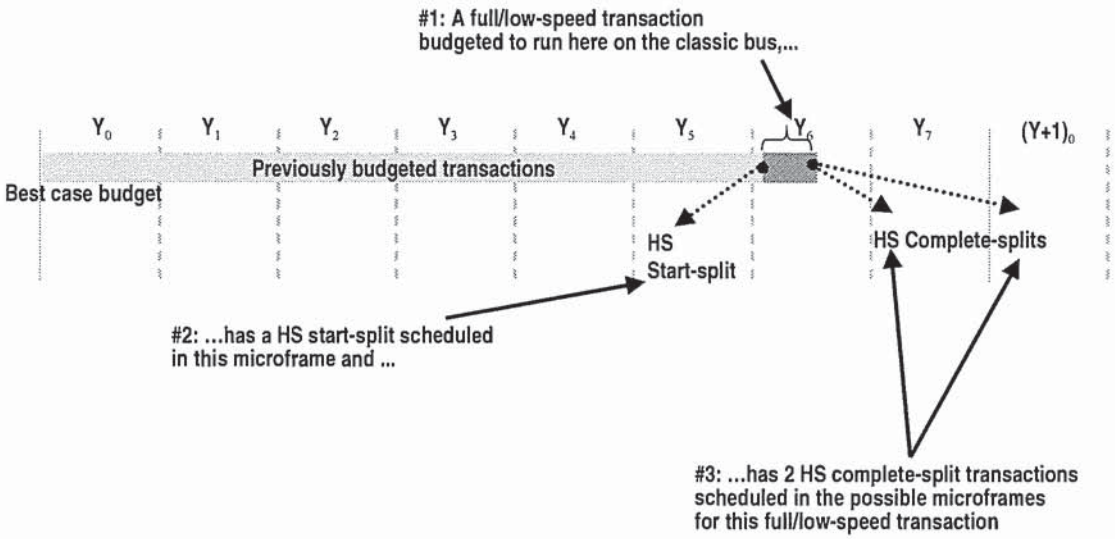

Figure 11-63. End of Frame TT Pipeline Scheduling Example

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.

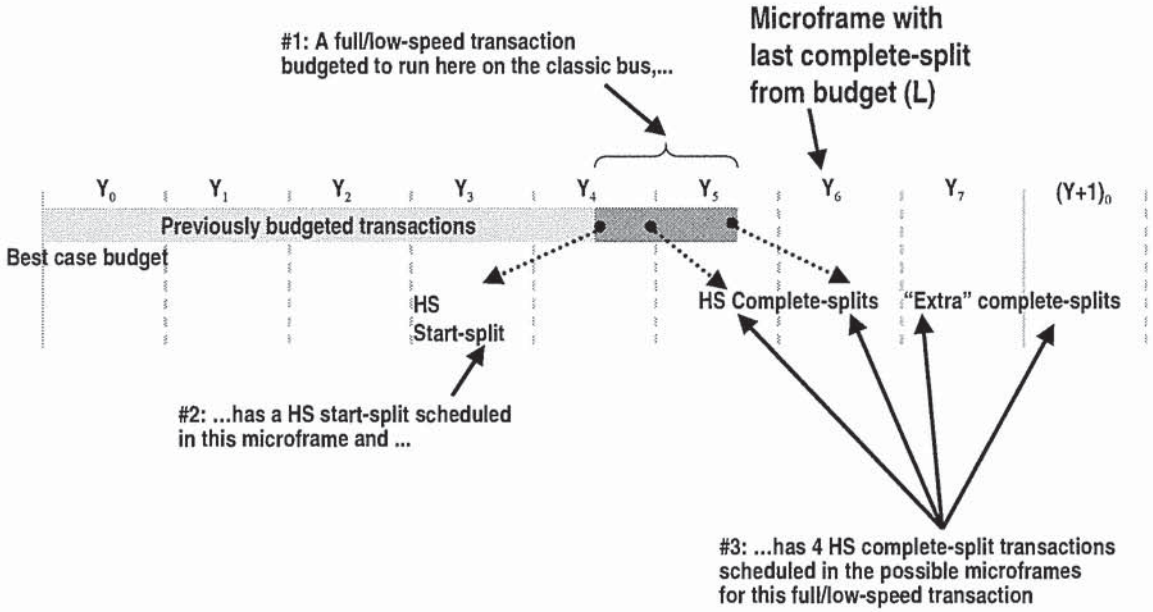

Figure 11-64. Isochronous IN Complete-split Schedule Example at $L=Y_6$

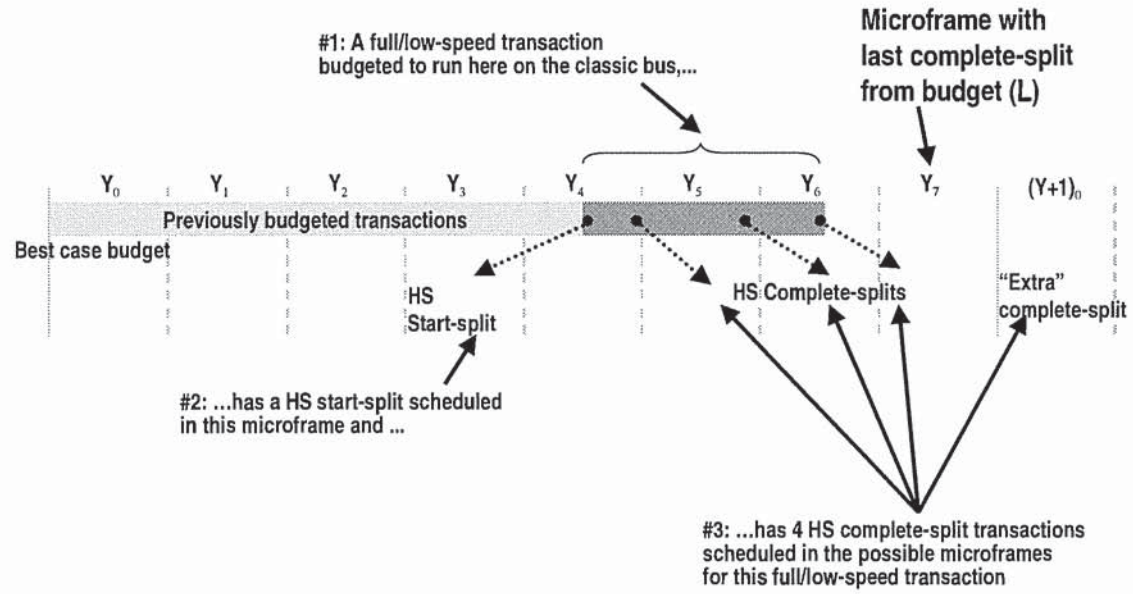

Figure 11-65. Isochronous IN Complete-split Schedule Example at $L=Y_7$

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-1),
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.

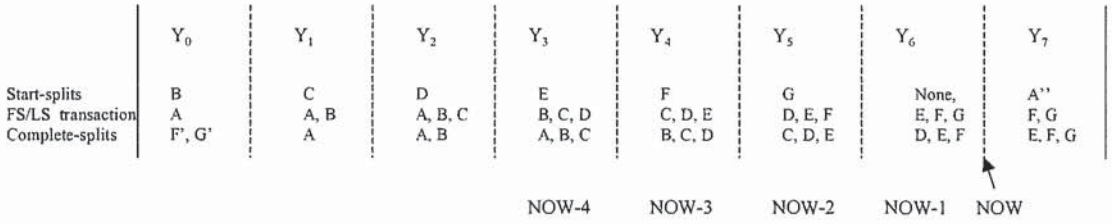

Figure 11-66. Microframe Pipeline

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.

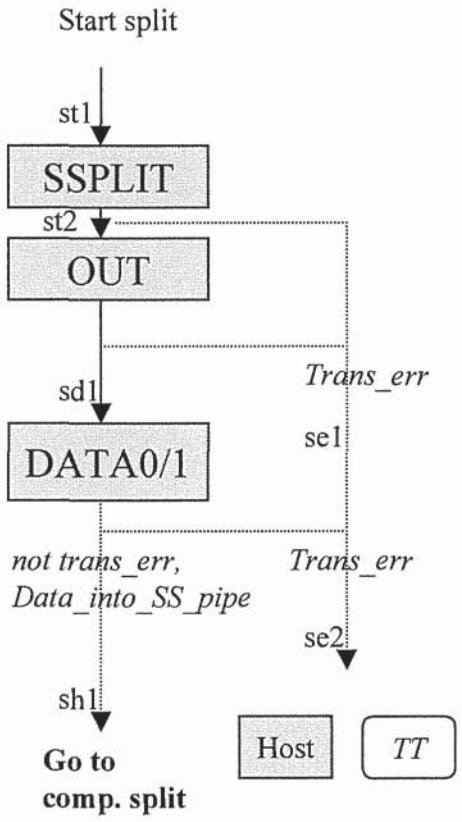

Figure 11-68. Interrupt OUT Start-split Transaction Sequence

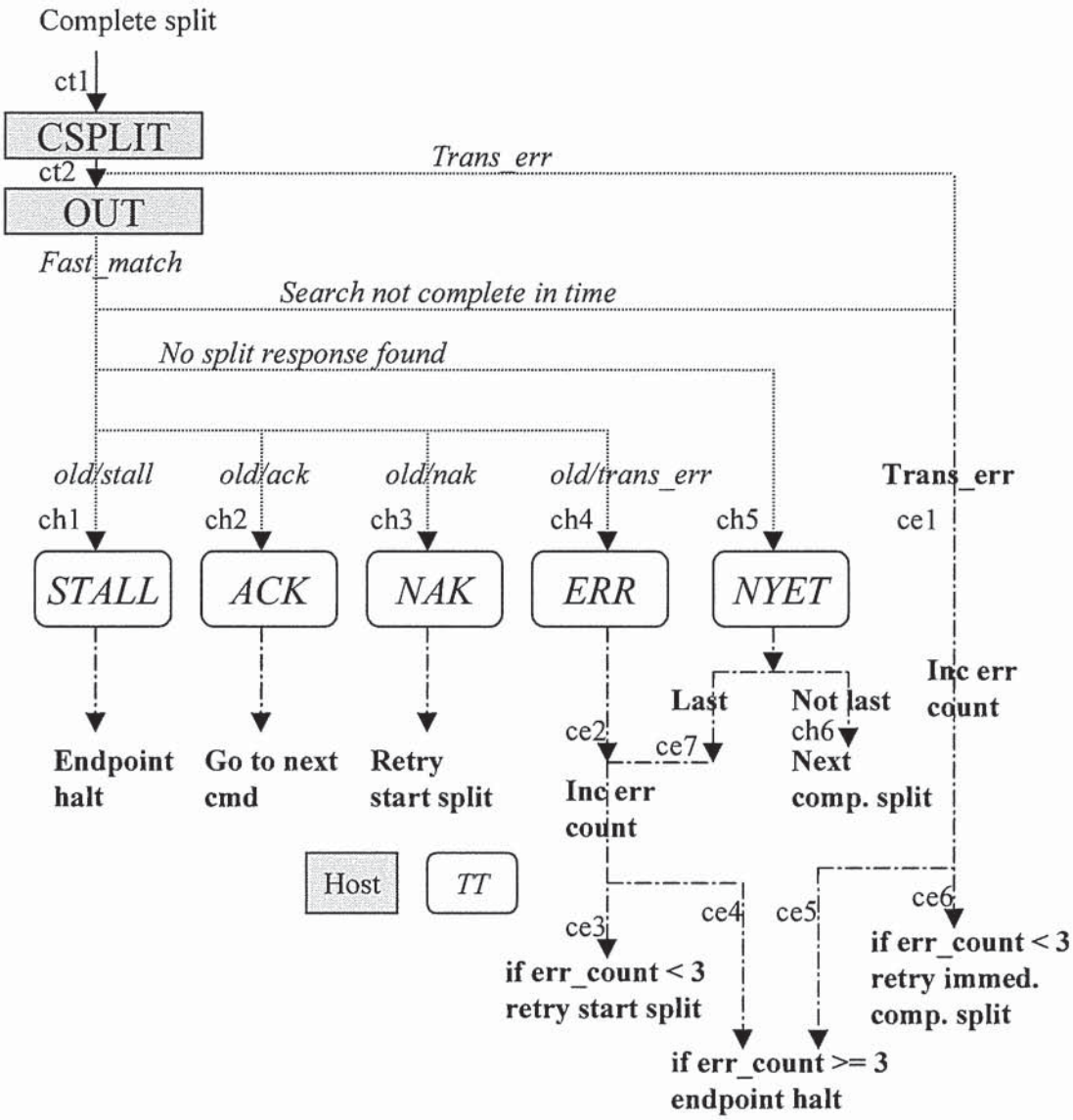

Figure 11-69. Interrupt OUT Complete-split Transaction Sequence

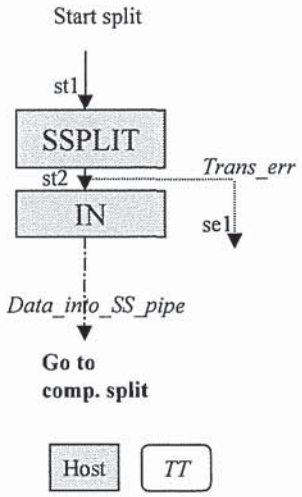

Figure 11-70. Interrupt IN Start-split Transaction Sequence

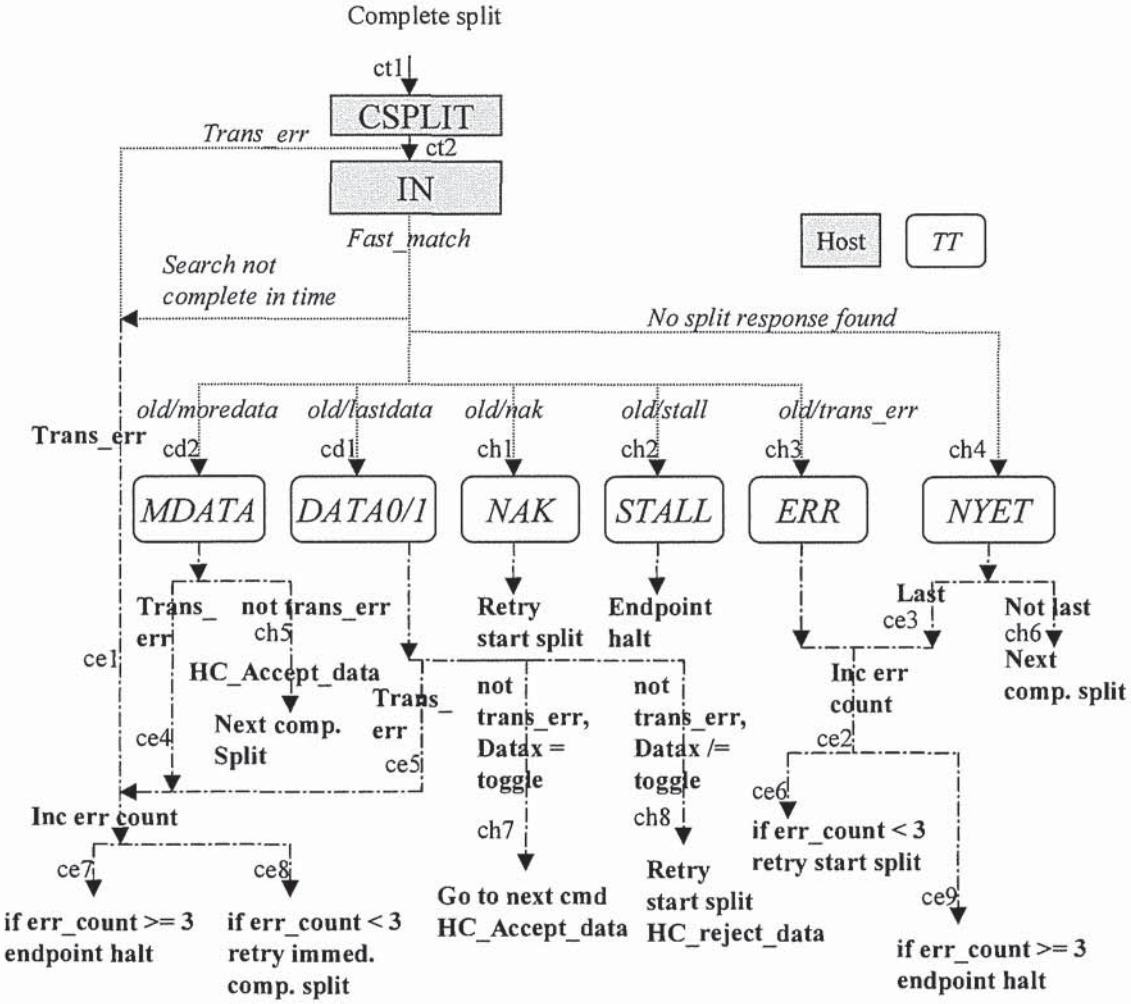

Figure 11-71. Interrupt IN Complete-split Transaction Sequence