Verification Continuum™

VC Verification IP
USB
HDL User Guide

Version R-2021.03, March 2021

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Contents

Preface

About This Manual

This manual contains installation, setup, and usage material for Verilog HDL users of the Synopsys USB VIP, and is for design or verification engineers who want to verify USB operation using a HDL testbench written in Verilog. Readers are assumed to be familiar with USB, Object Oriented Programming (OOP), Verilog, and Universal Verification Methodology (UVM) techniques.

Note From the R-2021.03 release onwards, the documentation updates for the guides that are based on the Verilog HDL designs will be suspended. For more information, see the UVM guides for the current updates on this VIP. Contact Synopsys support for any queries or clarifications.

Web Resources

- Documentation through SolvNetPlus: https://solvnetplus.synopsys.com (Synopsys password required)
- Synopsys Common Licensing (SCL): http://www.synopsys.com/keys

Customer Support

To obtain support for your product, choose one of the following:

1. Go to https://solvnetplus.synopsys.com/ and open a case.
   - Enter the information according to your environment and your issue.
   - For simulation issues, provide a UVM_FULL verbosity log file of the VIP instance and a VPD or FSDB dump file of the VIP interface.
2. Send an e-mail message to support_center@synopsys.com
   - Include the Product name, Sub Product name, and Product version for which you want to register the problem.
3. Telephone your local support center.
   - North America:

     Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
   - All other countries:

     https://www.synopsys.com/support/global-support-centers.html

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

1 Introduction

The VC VIP for USB and the VCS Verification Library, support verification of SoC designs that incorporate USB interfaces.

The VC VIP for Verification IP supports verification of SoC designs that include interfaces implementing the Universal Serial Bus 3.0 Specification. This document describes the use of this VIP in Verilog testbenches.

1.1 Product Overview

The VC VIP for USB can be used in two ways:

- For *command-based* Verilog testbenches. This usage relies on directed tests constructed using the commands and data objects described in this document. In Verilog testbenches, the VIP is brought out as a module with respective interface signals.
- For *object-based* SystemVerilog testbenches as described by the Verification Methodology Manual (VMM). This usage simplifies constrained random test methodologies and can also be used for directed test methodologies. For information about VMM object-based usage, refer to the *VC VIP for USB SystemVerilog/VMM User Manual*.

1.2 USB Feature Support

This document assumes that you are familiar with the USB protocol. For information on the USB protocol, see *Universal Serial Bus 3.0 Specificat*ion, Revision 1.0, November 12, 2008

The USB Verification IP supports the following USB features:

- SuperSpeed (SS), High-Speed (HS), Full-Speed (FS) and Low-Speed (LS)
- Host, Device, and Hub verification
- PHY interface support
- Power Management: USB and USB 2.0 implementations
- SS PIPE3, SS Serial, and 2.0 Serial interfaces
- USB Layer Stack Feature Support
- USB 2.0 OTG and Embedded Host support for serial interfaces

USB 2.0 OTG and Embedded Host support is a BETA feature, and is not LCA ready.

1.2.1 Protocol Layer Features

The USB protocol layer manages the end to end flow of data between a device and its host. The VIP Protocol transactor accepts and processes testbench transfer requests according to the specification.

USB VIP supports the following protocol layer functions:

- Host emulation: Optional control to emulate Hub's downstream-facing hub port
- Device emulation: Optional control to emulate Hub's upstream-facing hub port
  - Emulates up to 128 devices with up to 32 endpoints each
- Transfers: Bulk, Control, ISOC, and Interrupt
- SS Stream protocol
- SOF and ITP packets: Generated automatically or under testbench control
- SS LMP packets: Includes optional automatic generation of LMP capability, configuration or config response packets upon U0 entry
- Endpoint Halt control
  - Automatic endpoint halt upon receipt of STALL response.
  - Ability for VIP Host to attempt 'n' more transfers to a halted endpoint to verify persistence of DUT device halt status.
  - Support for testbench requested endpoint halt status entry and clearing of halt status
- USB 2.0 split traffic
- USB 2.0 LS on FS traffic
- Transaction scheduler: Transfers scheduling in (micro) frames/bus intervals based on multiple criteria

1.2.2 Link Layer Features

The VIP Link Layer transactor emulates the link protocol data flow defined by the USB specifications, according to the bus speed appropriate protocol specification. USB VIP supports the following link layer functions:

- USB General Support
  - Host and Device support
  - Speed fallback from SS to FS or HS
  - Speed fall-forward from FS or HS to SS
- USB Support
  - LTSSM – Direct LTSSM state change support
  - SS power management
  - ITP support
  - Link advertisements
  - Link command and packet processing and response
- USB 2.0 Support
  - LPM

- Suspend and Resume
- USB2 power management

1.2.3 Physical Layer Features

USB VIP supports the following physical layer functions:

- SS PIPE3 and Serial
  - Data scrambling/descrambling
  - 8b/10b encoding/decoding
  - Rx data and clock recovery
  - Rx error detection and polarity inversion
  - Receiver detect
  - Low frequency Periodic Signal transmission/detection
  - Spread spectrum Clocking allowed on input clock
  - 8/16/32 bit parallel interface
  - PCLK generation (when configured as PHY)
  - 5.0 Gb/s serial interface
- USB 2.0 serial
  - 480Mb/s (HS), 12Mb/s (FS), or 1.5Mb/s (LS) serial interface
  - Bit stuffing/un-stuffiing
  - NRZI encoding/decoding
  - Rx clock and data recovery
  - Rx error detection
  - Receiver detection
  - SYNC/EOP transmission/detection
  - Reset, Resume, Wakeup, and Suspend transmission

2 General Concepts

This chapter describes general concepts that lay the foundation for understanding the operation of the Synsopsys VIP for USB in Verilog testbenches. This chapter includes the following:

- Modules
- Commands
- Data Objects

2.1 Modules

Modules are the USB VIP elements that are instantiated in a user's HDL testbench. Instantiated modules are connected to the wires of the testbench (via port connections), just as an RTL design would be.

Each module has two built-in properties (attributes), as follows:

- **inst:** An instance name (string) that identifies an instance of the module.
- **stream_id:** A numerical (integer) ID that can be assigned to a module instance. This property identifies a data stream to which a transaction belongs.

In addition to these properties, each module contains functionality to support the production and control of log (transcript) messages. This functionality is referred to as the module's logger.

2.1.1 Notifications

Notifications are timing (event) markers supplied by a module. You may call the notify_wait_for command on a VIP module to synchronize to a specific notification. Each notification is identified by a unique name of the form *NOTIFY_<event_name>*, which is passed to the command as a control argument. The notify_wait_for command blocks until the notification point is reached.

2.1.2 Callbacks Notifications

Callback notifications are points during the simulation at which the data objects currently in use by the VIP may be accessed by testbench code. Callback notifications, like notifications, are timing (event) markers. However, there are the following differences:

- A callback notification returns a handle that can be used to access a data object (typically, a transaction), while a notification does not.
- A callback notification requires a handshake from the testbench code to tell the VIP module when the testbench code is done accessing the data object.

Attention The testbench must not allow simulation time to advance while it accesses the data object obtained at a callback point. If simulation time is advanced between the cmd_callback_wait_for call and the cmd_callback_proceed or another cmd_callback_wait_for call, a fatal error is reported.

To synchronize your testbench with a callback so you can access the associated data object, pass the callback notification identifier to either of the following commands:

cmd_callback_wait_for

Blocks testbench execution until the notification is reached, at which time it returns a handle to the data object associated with that callback notification. The testbench may use the returned handle (or handles) to access the properties of the associated data object. When the testbench is done accessing the data properties, it must then perform one of the following:

- call the handshake command (cmd_callback_proceed) with the same identifier.
- call cmd_callback_wait_for again with the same handle (the handle returned via the previous cmd_callback_wait_for) and the callback identifier.

Use cmd_callback_wait_for to proceed from an existing callback if you expect multiple instances of the same callback within a single cycle. This clears the previous block and sets up the new one, insuring that you catch multiple instances.

cmd_callback_proceed

Behaves similarly to cmd_callback_wait_for, but you use this command if you do NOT expect to see multiple instances of the same callback within a cycle. For example, it's possible for multiple transactions to complete simultaneously. In this situation, the testbench should use cmd_callback_wait_for (usually in a loop) to proceed past the current (finished) callback and immediately begin waiting for the next callback point.

Callback points have unique names with the following form:

NOTIFY_CB_<event_name>

example: `cmd_callback_wait_for(is_valid, callback_handle, "prot.NOTIFY_CB_NEW_SS_RESPONSE_TRANSFER");`

2.2 Commands

Commands are procedure calls on a USB VIP module that you use in your testbench code. Commands take control arguments and, in general, return data via reference arguments.

Attention Commands are described in the USB HDL HTML Reference located at:

`$DESIGNWARE_HOME/vip/svt/usb_svt/latest/doc/usb_svt_hdl_reference/html/index.html`

2.3 Data Objects

Data objects are abstractions of information used or produced by USB VIP modules. Also, data objects are sets of related data.

Within the USB VIP modules, data objects represent testbench elements such as configuration information. Two types of data objects that control configuration and transaction data:

- transaction objects control and communicate transaction information.
- configuration objects control and communicate configuration information.
- exception objects w control error injection information.
- status objects log the status and state of VIP component in processing transmitted and received transaction objects.

Each type of data object is represented by a unique name that is known to all modules in the USB VIP suite. During operation, data objects in use by a USB VIP module are assigned integer *handles* for identification. These handles allow testbench code to reference the corresponding internal data objects and to access their properties by name.

Data objects are *typically* temporary. That is, they are valid only at the current simulation timestep. If a command operates on a temporary data object, it must be done in zero simulation time with respect to when the original data object was created. If simulation time advances, the *handle* that refers to a temporary data object is no longer valid. The underlying data object is effectively discarded and memory is freed up.

2.3.1 General Session

- A new data object handle is obtained by the testbench using one of the following commands:
  - get_data_prop
  - new_data
  - cmd_callback_wait_for
- The VC VIP for USB is instructed to start using the data object when the following command is called:
  - apply_data
  - cmd_callback_proceed (makes implicit apply_data calls)
  - cmd_callback_wait_for (makes implicit apply_data calls)

There are no restrictions on read access (such as through get_data_prop), but write access (such as through set_data_prop) is limited to *edit sessions*, which are described below. Note that at the end of each edit session the data object must be valid; otherwise, a fatal error is reported.

2.3.2 Edit Session

The command interface supports the concept of an edit session for a data object manipulated through the command interface.

An *edit session* is initiated when the set_data_prop command is called with a handle referencing a data object for which an edit session is not already in progress. The handle originates with a call to one of the following commands:

- get_data_prop (provides a data object handle)
- new_data (provides a new data object handle)

- cmd_callback_wait_for (provides a new data object handle)

Attention A testbench's edit session must not consume simulation time; if it does, a fatal error is reported.

An edit session is terminated when the apply_data commands is called.

Each time an edit session is terminated, a validity check is performed on the data object. If the data object is not valid at that time, a fatal error will be reported.

2.3.3 Data Object Properties

A *property* (in the context of a data object) is a named data item contained by the data object. This implies a name/value pair. Properties may be integers, bit-fields, strings, enumerated types, or sub-objects. Also, a property may be an array of any of these types.

3 Modules

3.1 Modules

A USB VIP sub-environment (subenv) is a module that is composed of a transactor stack and a port interface. Sub-environments are implemented through modules. Each sub-environment module is capable of performing as a USB host or device.

A module is instantiated into the VIP environment, then configured through edit sessions, as described in "General Session". After configuration is complete, the module is connected to other testbench modules through ports.

The following section describes USB VIP modules, their configuration, and their usage.

Attention Commands are described in the USB HDL HTML Reference located at:

```
$DESIGNWARE_HOME/vip/usb_svt/latest/doc/usb_svt_hdl/html/index.html
```

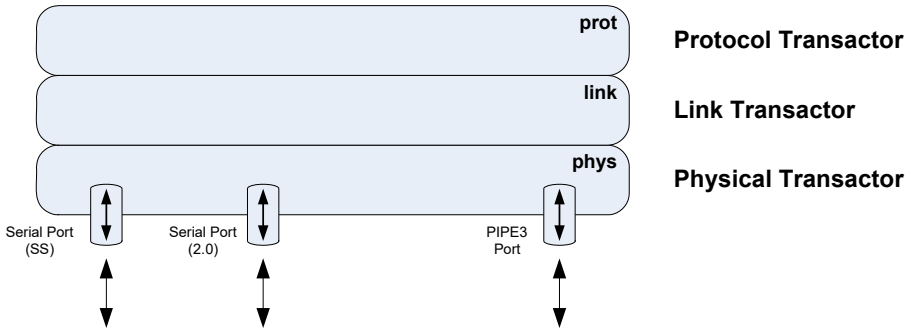
3.1.1 Transactor Stacks

USB VIP transactors are code blocks that implement a specific layer of the USB protocol. The USB VIP defines three transactors:

- Protocol transactor
- Link transactor
- Physical transactor

The transactor stack is a module element that consists of instantiated transactors. A valid stack consists of up to four transactors. The transactor set is specified by the module definition and a configuration top_xactor property. Figure 3-1 displays an example protocol stack.

Figure 3-1 USB Protocol Stack Example



The set of transactors within a stack are contiguous. The VIP defines four transactor attributes that correspond to the four transactors that a module can. The following lists the four transactor attributes, listed in order from top to bottom:

- **prot:** a transactor that emulates the protocol layer of the local USB stack
- **link:** a transactor that emulates the link layer of the local USB stack
- **phys:** a transactor that emulates the physical layer of the local USB stack
- **remote_phys:** a transactor that emulates the physical layer of the remote USB stack.

The top three transactors reside on the same side of the USB bus, relative to the serial bus connecting the two USB entities (host-device). The remote_phys transactor resides on the opposite side of the serial bus from the other transactors.

The module specifies the bottom transactor in the stack. A configuration top_xactor property (which can be set through the set_data_prop command) specifies the top transactor in the stack. The top layer of the transactor stack receives stimulus objects that drive the verification through a channel interface. While the most typical stacks install the Protocol transactor at the top of the stack, the link or physical transactors may also be defined as the top transactor. Stimulus data that is input into the stack must be consistent with that accepted by the top layer of the defined stack.

The bottom layer is a Physical transactor that connects to the DUT through a signal interface. The USB VIP defines serial and PIPE3 interfaces, allowing the Physical transactor to connect with a USB controller through a PIPE3 interface, a USB PHY through a PIPE3 or serial interface. The VIP supplies SS and 2.0 serial interfaces.

3.1.2 Protocol Transactor SuperSpeed Link Callback, and Notification Flows

This section provides detailed information about the sequence and content of callbacks provided by the VIP.

3.1.2.1 Protocol SuperSpeed Callback Flow when the VIP is Acting as a Host

3.1.2.1.1 Non-isochronous Transfers

Sequence of operations:

1. VIP gets transfer from transfer_in input channel.
2. After pulling a USB transfer descriptor out of the input channel, VIP issues *NOTIFY_CB_POST_TRANSFER_IN_CHAN_GET* and then acts on the descriptor.
3. VIP issues *NOTIFY_CB_TRANSFER_IN_CHAN_COV* to allow the testbench to collect functional coverage information from a USB transfer received in the transfer_in input channel.

After performing the above steps, the transactor can modify and process the object as required.

Send Phase

In the send phase, the VIP protocol transactor layer performs the following actions:

1. VIP issues *NOTIFY_CB_PRE_TRANSACTION* when a protocol processor thread is ready to begin a USB Transaction. This callback can be extended to collect functional coverage data, or to check the transaction against a scoreboard, and other such operations.
2. VIP issues *NOTIFY_CB_PRE_USB_SS_PACKET_OUT_CHAN_PUT* before putting a USB packet descriptor into the SS packet output channel.
3. VIP issues *NOTIFY_CB_TRANSFER_IN_CHAN_COV* to enable the testbench to collect functional coverage information from a USB packet about to be sent to the link layer through the USB SS packet output channel.

Receive Phase

In the receive phase, the VIP protocol transactor layer performs the following actions:

1. VIP pulls a USB packet descriptor out of the USB SuperSpeed input channel (link layer), issues *NOTIFY_CB_POST_USB_SS_PACKET_IN_CHAN_GET* and then acts on the descriptor.
2. The VIP then performs actions based on one of the following conditions:
   - ✦ No endpoint claims the packet
   - ✦ The packet is dropped

No endpoint claims the packet
VIP calls *unclaimed_ss_packet(svt_usb_protocol xactor, svt_usb_packet packet, ref bit drop);* and then issues an error message stating that an unclaimed USB packet that came through the USB SuperSpeed Packet input channel (from the link layer). This is called if the packet cannot be handled by the applicable protocol block.
If the packet is dropped
VIP calls d*iscarded_ss_packet(svt_usb_protocol xactor, svt_usb_packet packet);* and then discards the USB packet descriptor that came through the USB SuperSpeed Packet input channel (from the link layer). This is called if the packet cannot be handled by the applicable protocol block.

3. VIP issues *NOTIFY_CB_PRE_TRANSACTION* when a protocol processor thread is ready to begin a USB Transaction. This callback can be extended to collect functional coverage data, or to check the transaction against a scoreboard, or other such operations.
4. VIP then performs actions based on the type of packet.

If the received packet is not TP_ERDY
VIP issues *NOTIFY_CB_PRE_RX_PACKET* when a protocol processor thread is ready to begin processing a USB Packet received as part of a USB transfer.
If the received packet is for IN endpoint processor and retry_pending in transfer is set to zero
VIP issues *NOTIFY_CB_TRANSACTION_ENDED* when a USB transaction is completed.

If the received packet is for IN endpoint processor
VIP issues *NOTIFY_CB_RECEIVED_DATA_PACKET* when an error free data packet that is receiving data for a transfer has just been received.
For all other packets
VIP issues *NOTIFY_CB_TRANSFER_ENDED* when a USB transfer is complete.

3.1.2.1.2 Ping Operation for OUT Transfers

Sequence of operations:

1. VIP gets the USB transfer descriptor from the transfer_in input channel.
2. After pulling the USB transfer descriptor from the transfer_in input channel, VIP issues *NOTIFY_CB_POST_TRANSFER_IN_CHAN_GET.*
3. VIP issues *NOTIFY_CB_TRANSFER_IN_CHAN_COV* to allow the testbench to collect functional coverage information from a USB transfer received from the transfer_in input channel.
4. VIP issues *NOTIFY_CB_PRE_TRANSACTION* when a protocol processor thread is ready to begin a USB Transaction. This callback can be extended to collect functional coverage data, or to check the transaction against a scoreboard, or other such operations.
5. VIP then issues *NOTIFY_CB_PRE_USB_SS_PACKET_OUT_CHAN_PUT* before putting a USB packet descriptor into the SS output channel.
6. VIP then issues *NOTIFY_CB_TRANSFER_IN_CHAN_COV* to enable the testbench to collect functional coverage information from a USB packet that is going to be sent to the link layer through the USB SS output channel.

3.1.2.1.3 Ping Response Operation for OUT Transactions

3.1.2.1.4 Isochronous Transfer Operation

Sequence of operations:

1. VIP gets transfer from transfer_in input channel.
2. After pulling a USB transfer descriptor out of the input channel, VIP issues *NOTIFY_CB_POST_TRANSFER_IN_CHAN_GET* and then acts on the descriptor.
3. VIP issues *NOTIFY_CB_TRANSFER_IN_CHAN_COV* to allow the testbench to collect functional coverage information from a USB transfer is received in the transfer_in input channel.

After performing the above steps, the transactor can modify and process the object as required.

Send Phase

In the send phase, the VIP protocol transactor layer performs the following actions:

1. VIP issues *NOTIFY_CB_PRE_TRANSACTION* when a protocol processor thread is ready to begin a USB Transaction.

Receive Phase

Sequence of operations:

1. VIP issues *NOTIFY_CB_PRE_USB_SS_PACKET_OUT_CHAN_PUT* before it puts a USB packet descriptor into the SS packet output channel.
2. VIP issues *NOTIFY_CB_TRANSFER_IN_CHAN_COV* to enable the testbench to collect functional coverage information from a USB packet that is about to be sent to the link layer through the USB SS packet output channel.
3. VIP issues *NOTIFY_CB_TRANSACTION_ENDED* when a USB transaction is complete.

3.1.2.2 Protocol SuperSpeed Callback Flow when the VIP is Acting as a Device

High Level Sequence of Operations

1. VIP receives the packet from packet_in_chan.
2. VIP retrieves packets from the link layer through the appropriate packet input channel (see Receive Phase).
3. VIP locates device object for packet's device_address and determines if deferral is required based on the value of USB_SEND_DEFERRED_PACKETS protocol service.
4. If deferral is required, VIP does the following:
   a. Creates a copy of the Rx packet (using the Rx packet's allocate() function), assigning the copy to downstream_pkt.
   b. Sets downstream_pkt.was_deferred = 1.
   c. Uses downstream_pkt as factory to create a copy assigning to upstream_pkt.
   d. Places upstream_pkt in deferred_pkt_queue to be reflected back sequentially to the host. Before each individual packet is reflected back, the packet's deferred_pkt_reflection_delay time is inserted.
   e. Places downstream_pkt in rx_pkt_ppd_queue.
   f. After the deferral activity is complete, process each packet in the rx_pkt_ppd_queue sequentially. Each packet is delayed rx_pkt_pre_processing_delay amount of time before it is passed to the applicable protocol processor to create a new transfer or continue an existing transfer.
5. If deferral is not required, VIP does the following:
   a. Sets downstream_pkt equal to Rx packet.
   b. After the deferral activity is complete, process each packet in the rx_pkt_ppd_queue sequentially. Each packet is delayed rx_pkt_pre_processing_delay amount of time before it is passed to the applicable protocol processor to create a new transfer or continue an existing transfer.
6. Once the packet is routed to the appropriate endpoint, based on the endpoint direction, VIP processes the packet further (see Process Received Packets for IN Transfers or Process Received Packets for OUT Transfers).
7. VIP sends the response to the host through the link layer packet output channel.

Detailed Sequences

Receive Phase

1. VIP pulls a USB packet descriptor out of the SS packet input channel (from the link layer), issues *NOTIFY_CB_POST_USB_SS_PACKET_IN_CHAN_GET* before acting on the descriptor.
2. VIP routes the packet.

Process Received Packets for IN Transfers

For non-isochronous transfers, VIP completes the following sequence of operations:

1. VIP receives the packet and modifies the stream state when the device transitions to move_data state. If the received packet has setup_bit =1 and transfer_stage is not SETUP_STAGE, VIP performs the following actions:
   a. Aborts transfer
   b. Routes the packet
   c. Receives in packet
2. VIP prepares the transaction (if it is not yet ready).
3. VIP checks if the received packet has the deferred bit set. If the deferred bit is set, VIP moves to idle stream state. If the deferred bit is not set, VIP issues *NOTIFY_CB_TRANSACTION_ENDED* after the USB Endpoint Manager transaction is complete.

   If the transfer is not completed and the received packet is terminating with an ACK, VIP moves to idle stream state.

For isochronous transfers, VIP completes the following sequence of operations:

1. VIP prepares the transaction.
2. If the received packet is neither a ping or a TP_ACK, VIP terminates the transfer.
3. VIP then follows the process described in Send Packet of IN Transfer.

Process Received Packets for OUT Transfers

For non-isochronous transfers, VIP completes the following sequence of operations:

1. VIP performs a check on the received packet:
   - If transfer_stage = SETUP_STAGE, and the received packet has setup bit set to 1 and deferred bit set to zero and if received packet.payload_presence = payload_present and transfer has setup_with_payload_absent = 1, VIP performs the following actions:
     i. VIP captures the information from the received setup packet and randomizes (or asks for the transfer) to operate on. When a packet is deferred, the payload is striped off. If the setup packet is deferred, the setup bytes are stripped off. So, if the first setup packet, which initiates the transfer is deferred, the device model does not have any valid setup bytes to randomize the transfer. So, when non-deferred setup packets are received after the deferred setup byte, VIP has to capture the correct setup bytes and re-randomize the transfer again.
     ii. VIP generates the basic response information and then modifies the stream state when the device transitions to move_data state.
   - If received packet is DATA_PACKET and setup_bit is 1 and xfer_stage is not SETUP_STAGE in transfer (indicating that the received setup is not a part of the current transfer), VIP performs the following actions:

i. VIP routes the packet.
ii. VIP prepares the transaction.

2. VIP issues *NOTIFY_CB_PRE_RX_PACKET* when a protocol processor thread is ready to start processing a USB packet to be transmitted as part of a USB transfer.
3. VIP issues *NOTIFY_CB_RECEIVED_DATA_PACKET* when it receives an error-free data packet for a transfer. The transfer in progress is passed as the transfer argument, and the data packet is passed as the packet argument. The testbench uses this callback to sample data received for a transaction on-the-fly, regardless of whether the transaction completes at this point or not.
4. VIP calls *do_post_ss_rx_pkt_xfer_update_cb_exec.*

For isochronous transfers, VIP completes the following sequence of operations:

1. VIP prepares the transaction.
2. If the received packet is TP_PING, then VIP calls *send_out_packet*. If the received packet is not a data packet, then VIP terminates the transfer.
3. VIP issues *NOTIFY_CB_TRANSACTION_ENDED* callback immediately after a USB Endpoint Manager transaction is complete.
4. VIP issues *NOTIFY_CB_RECEIVED_DATA_PACKET* when it receives an error-free data packet for a transfer.
5. VIP issues *post_ss_rx_pkt_xfer_update(svt_usb_protocol xactor, svt_usb_transfer transfer, int transaction_ix, int packet_ix)* callback when a protocol processor thread has completed processing a USB packet received and associated with a USB transfer. VIP can optionally extend this callback to collect functional coverage data, or modify the data in the packet so that it is reflected in the transfer.

Send Packet of IN Transfer

For non-isochronous transfers, VIP completes the following sequence of operations:

1. VIP prepares the transaction and the response packet to be send to the host and then updates the variable after the IN endpoint transmits the packet to the host.

For isochronous transfers, VIP completes the following sequence of operations:

1. If the transaction is an SS Ping or Ping response, VIP calls *send_ping_response*.
2. If the transaction is not an SS Ping or Ping response, VIP performs the following operations:
   a. VIP prepares the transaction.
   b. VIP prepares the next generated packet for the specified transaction and then creates packets for USB SS transactions.
   c. If the transaction device response does not time out, VIP calls *send_ss_pkt_to_link* and issues *NOTIFY_CB_TRANSACTION_ENDED* when a USB Endpoint Manager transaction is complete.

3.1.2.3 Protocol SuperSpeed Endpoint Manager Flow

The SuperSpeed endpoint manager flow works parallel to the send and receive phase of host and device. This controls streamable and non-streamable endpoint states before during and after move_data states. For more information, see section 8.12 of the USB specification.

There are two different flows:

- Non-stream endpoint manager - This manages non-stream states.
- Stream endpoint manager - This manages stream states.

3.1.2.3.1 Non-stream Endpoint Manager Flow for Host and Device

Sequence of operations:

1. VIP prepares the transfer for the endpoint.
2. VIP randomizes the endpoint manager factory.

3.1.2.3.2 Stream Endpoint Manager Flow

Figures 3-2 and 3-3 illustrate the stream endpoint manager flow for host and device respectively.

Figure 3-2 Host Stream Endpoint Manager Flow

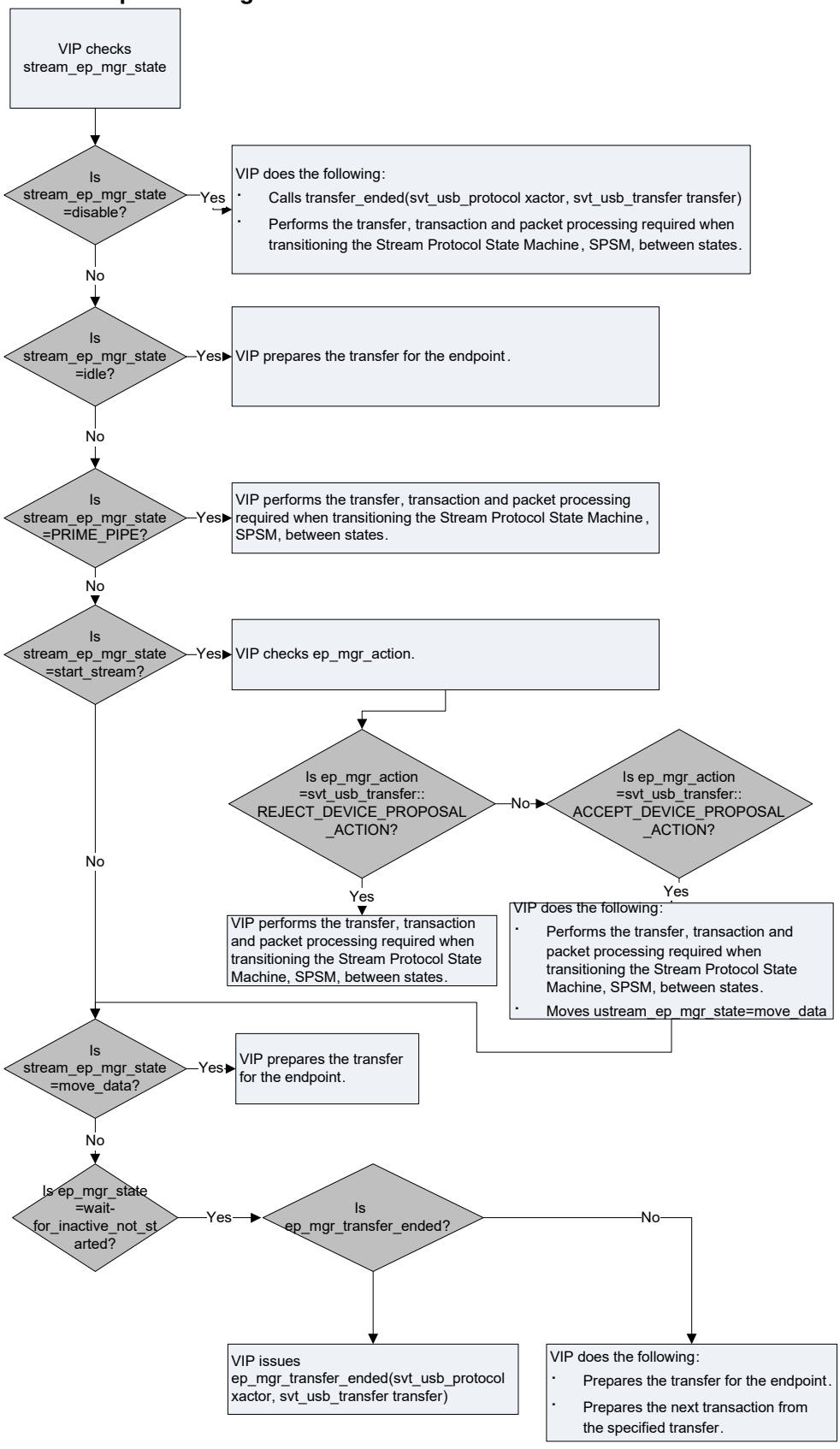
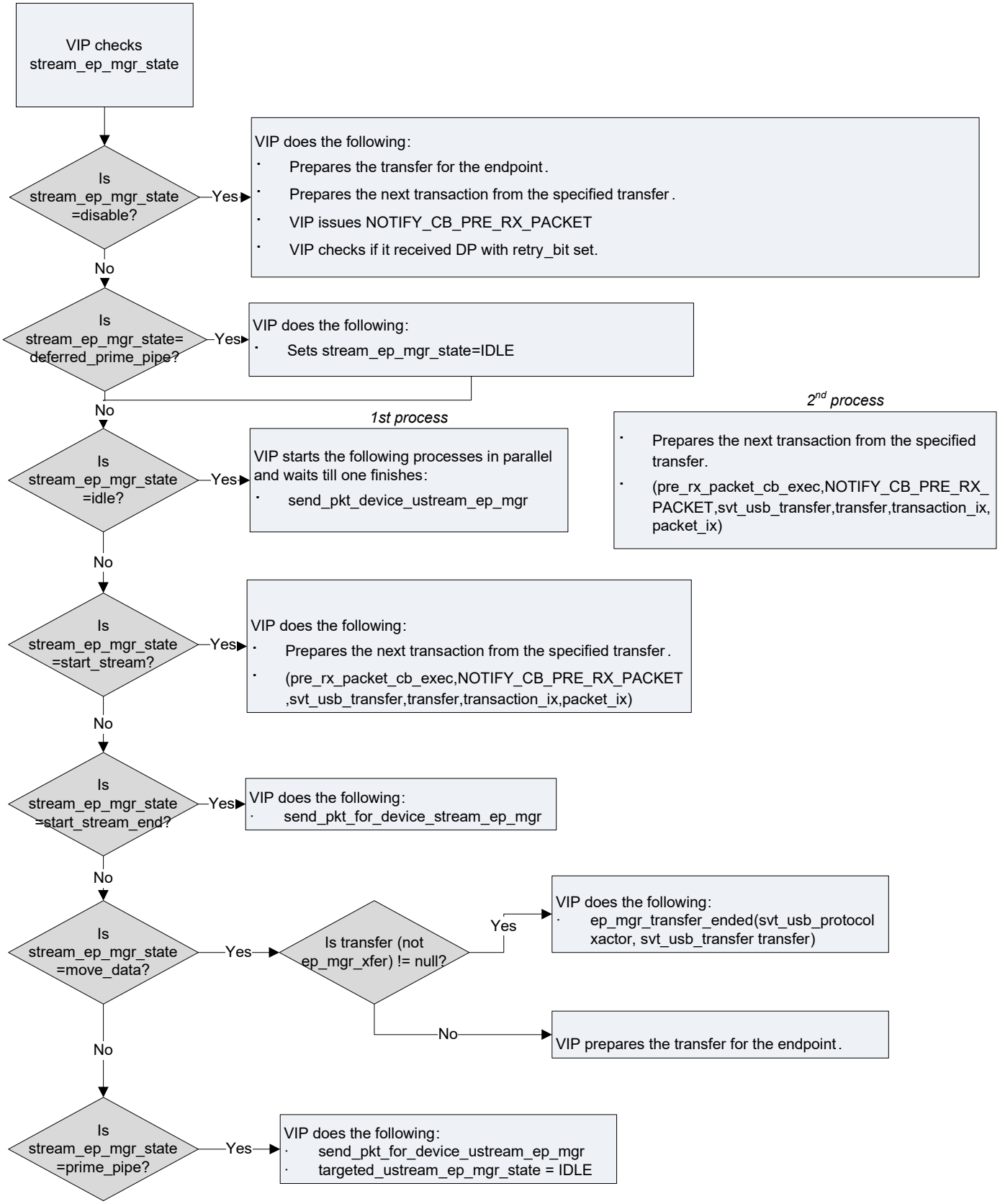


Figure 3-3 Device Stream Endpoint Manager Flow



3.1.3 Protocol Transactor 2.0 Callback and Notification Flows

This section provides detailed information about the sequence and content of callbacks provided by the VIP.

3.1.3.1 Protocol 2.0 Callback and Notification Flow when the VIP is Acting as a Host

3.1.3.1.1 Non-isochronous Transfers

Transfer Start Phase

A Host non-isochronous transfer begins with the Host VIP receiving a transfer object on the transfer input channel.

Sequence of operations:

1. VIP gets transfer from transfer_in input channel.
2. After pulling a USB transfer descriptor out of the input channel, VIP issues *NOTIFY_CB_POST_TRANSFER_IN_CHAN_GET* and then acts on the descriptor.
3. (Optional) This step occurs only if the drop bit from *NOTIFY_CB_POST_TRANSFER_IN_CHAN_GET* returns a false value. VIP issues *NOTIFY_CB_TRANSFER_IN_CHAN_COV* to allow the testbench to collect functional coverage information from a USB transfer received in the transfer_in input channel.

Transfer Processing Phase

The transfer may be completed as a single transaction, or it may require multiple transactions to complete the transfer. A loop is used to process the transfer until all of its transactions are completed (or until the transfer is aborted). The following loops are discussed here:

- Transfer loop for host non-isochronous
- Transaction loop for host non-isochronous (this is a subset of the entire transfer loop)

Transfer Loop for Host non-isochronous

Sequence of operations:

1. Create and prepare new transactions.

   Each time the Transfer Loop is executed, a new transaction object is created that represents the next transaction needed to implement the transfer. For more information on the detailed callback and notification sequences, see "Create and Prepare New Transaction".
2. Process the transaction.

   The transaction is processed according to the transaction loop described in Transaction Loop for Host non-isochronous.
3. (Optional) Repeat the transfer loop if more transactions are required.

   This step depends on the specific transfer. If there is more payload to send and the transfer is not aborted, then continue with the transfer loop.
4. If the transfer loop does not have to be repeated, VIP issues *NOTIFY_CB_TRANSFER_ENDED*.

Transaction Loop for Host non-isochronous

The Transaction Loop occurs as a step within the Transfer Loop. The transaction may require sending or receiving multiple packets. The actual send or receive direction and the number of packets depends on the specific transaction type and traffic conditions. The description of the transaction loop is generic and does

not attempt to detail the specific send and receive order, but rather the order of callbacks associated with a packet send or receive.

Sequence of operations:

1. Prepare and send packet.

   The Host sends a TOKEN packet. For information on the callbacks, see "Prepare and Send Packet". This callback flow applies even if the specific transaction requires the Host to send a DATA or a HANDSHAKE packet.

2. Get the received packet.

   If the transaction requires the Host to receive a DATA or HANDSHAKE packet from the device, then the VIP uses the Get Received Packet callback flow.

3. Host responds to the transaction based on the type of transaction.

If the transaction is a device IN DATA transaction
If the device DATA is a legal protocol response and has no packet exception errors, VIP does the following: 1. VIP appends the payload to the data array. 2. VIP issues *NOTIFY_CB_RECEIVED_DATA_PACKET* when it receives an error free data packet that is receiving data for a transfer.
If (not INTR-CSPLIT) - VIP calls *transfer.payload.append_payload(packet.payload))*
If (last INTR-CSPLIT) - VIP calls *transfer.payload.append_payload(transaction.payload))*
If (last CSPLIT) - VIP calls the following: *1. transaction.status = svt_transaction::ACCEPT* *2. transaction.notify.indicate* *3. svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*
If the transaction is non-split, then VIP does the following: 1. If *(transaction.host_response == HOST_NORMAL_RESPONSE),* complete the flow listed in Prepare and Send Packet. 2. If the host response is ACK, with no injection errors, then VIP does the following: - *transaction.status = svt_transaction::ACCEPT* - *transaction.notify.indicate* - *svt_usb_protocol::notify.indicate(protocol_block.protocol.NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*
If the transaction is SPLIT IN and device responds with an ACK, VIP does the following: • *transaction.status = svt_transaction::ACCEPT* • *transaction.notify.indicate* • *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*
If the transaction exceeds the maximum allowed number of errors, VIP aborts the transfer.

Host Response to Device Handshake for OUT transactions
If the device HANDSHAKE is an ACK, does not have any packet exception errors, and if the transaction is either a Complete SPLIT or not a SPLIT, complete the following steps: 1. Subtract the *transaction.payload.byte_count* value from the *transfer.payload_bytes_remaining* attribute. 2. Mark the transaction status as ACCEPT using *transaction.status = svt_transaction::ACCEPT.* 3. Trigger the transaction notify that the transaction has ENDED using *transaction.notify.indicate.* 4. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy()).*
If the device HANDSHAKE is a NYET, does not have any packet exception errors, and if the transaction is not a SPLIT, complete the following steps: 1. Subtract the *transaction.payload.byte_count* value from the *transfer.payload_bytes_remaining* attribute. 2. Mark the transaction status as ACCEPT using *transaction.status = svt_transaction::ACCEPT.* 3. Trigger the transaction notify that the transaction has ENDED using *transaction.notify.indicate.* 4. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy()).*
If the transaction exceeds the maximum allowed number of errors, VIP aborts the transfer.
Host response to device handshake for SETUP transactions
If the device HANDSHAKE is an ACK, does not have any packet exception errors, and if the transaction is either a Complete SPLIT or not a SPLIT, complete the following steps: 1. Subtract the *transaction.payload.byte_count* value from the *transfer.payload_bytes_remaining* attribute. 2. Mark the transaction status as ACCEPT using *transaction.status = svt_transaction::ACCEPT.* 3. Trigger the transaction notify that the transaction has ENDED using *transaction.notify.indicate.* 4. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy()).*
If the transaction exceeds the maximum allowed number of errors, VIP aborts the transfer.

4. To repeat the transaction loop, go back to the flow at the top of the transaction loop. Otherwise, the transaction has already been marked ENDED and VIP issues *NOTIFY_CB_TRANSACTION_ENDED.*

3.1.3.1.2 Isochronous Transfers

Transfer Start Phase

A host isochronous transfer begins when the Host VIP receives a transfer object on the transfer input channel.

Sequence of operations:

1. VIP gets transfer from transfer_in input channel.
2. After pulling a USB transfer descriptor out of the input channel, VIP issues *NOTIFY_CB_POST_TRANSFER_IN_CHAN_GET* and then acts on the descriptor.

3. (Optional) This step occurs only if the drop bit from *NOTIFY_CB_POST_TRANSFER_IN_CHAN_GET* returns a false value. VIP issues *NOTIFY_CB_TRANSFER_IN_CHAN_COV* to allow the testbench to collect functional coverage information from a USB transfer received in the transfer_in input channel.

Transfer Processing Phase

The transfer may be completed as a single transaction, or it may require multiple transactions to complete the transfer. A loop is used to process the transfer until all of its transactions are completed (or until the transfer is aborted). The following loops are discussed here:

- Transfer loop for host isochronous
- Transaction loop for host isochronous (this is a subset of the entire transfer loop)

Transfer Loop for Host Isochronous

Sequence of operations:

1. Create and prepare new transactions.

   Each time the Transfer Loop is executed, a new transaction object is created that represents the next transaction needed to implement the transfer. For more information on the detailed callback and notification sequences, see Create and Prepare New Transaction.

2. Process the transaction.

   The transaction is processed according to the transaction loop described in Transaction Loop for Host Isochronous.

3. (Optional) Repeat the transfer loop if more transactions are required.

   This step depends on the specific transfer. If there is more payload to send and the transfer is not aborted, then continue with the transfer loop.

4. If the transfer loop does not have to be repeated, VIP issues *transfer.notify.indicate,* and then issues *NOTIFY_CB_TRANSFER_ENDED*.

Transaction Loop for Host Isochronous

The Transaction Loop occurs as a step within the Transfer Loop. The transaction may require sending or receiving multiple packets. The actual send or receive direction and the number of packets depends on the specific transaction type and traffic conditions. The description of the transaction loop is generic and does not attempt to detail the specific send and receive order, but rather the order of callbacks associated with a packet send or receive.

Sequence of operations:

1. Prepare and send packet.

   The Host sends a TOKEN packet. For information on the callbacks, see Prepare and Send Packet. This callback flow applies even if the specific transaction requires the Host to send a DATA or a HANDSHAKE packet.

2. Get the received packet.

   If the transaction requires the Host to receive a DATA or HANDSHAKE packet from the device, then the VIP uses the Get Received Packet callback flow.

3. Host responds to the transaction based on the type of transaction.

Host response to device isochronous IN DATA

If the device DATA is a legal protocol response and has no packet exception errors, VIP does the following:

1. VIP calls *transaction.payload.append_payload(packet.payload)*.
2. VIP issues *NOTIFY_CB_RECEIVED_DATA_PACKET* when it receives an error free data packet that is receiving data for a transfer.
3. VIP calls *transfer.payload.append_payload(packet.payload))*.

If the transaction is non-SPLIT or last CSPLIT, VIP does the following:

- VIP calls *transaction.status = svt_transaction::ACCEPT*
- VIP calls transaction.notify.indicate
- VIP calls svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())

If the transaction exceeds the maximum allowed number of errors, VIP aborts the transfer.

Host response to isochronous OUT transactions

If the transaction is not a SPLIT or is the last Start SPLIT, then VIP does the following:

1. Call *transaction.status = svt_transaction::ACCEPT*.
2. Trigger the notification that the transaction has ENDED using *transaction.notify.indicate.*
3. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*.

If the device handshake is a NYET and the transaction is not a SPLIT, then VIP does the following:

1. Subtract the transaction.payload.byte_count value from the transfer.payload_bytes_remaining attribute.
2. Mark the transaction status as ACCEPT using *transaction.status = svt_transaction::ACCEPT.*
3. Trigger the transaction notify that the transaction has ENDED using *transaction.notify.indicate.*
4. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy()).*

4. To repeat the transaction loop, go back to the flow at the top of the transaction loop. Otherwise, the transaction has already been marked ENDED and VIP issues *NOTIFY_CB_TRANSACTION_ENDED.*

3.1.3.2 Protocol 2.0 Callback Flow when the VIP is Acting as a Device

3.1.3.2.1 Non-isochronous Transfers

Transfer Start Phase

A device non-isochronous transfer begins when the Device VIP receives a TOKEN packet object on the packet input channel.

Sequence of operations:

1. VIP receives the TOKEN packet from the *svt_usb_protocol::packet_in_chan* channel. For more information about the callbacks and notifications, see Get Received Packet.
2. VIP creates the new device VIP transfer object and allows you to modify or replace it. For more information about the callbacks and notifications, see Create and Prepare New Device Transfer.

Transfer Processing Phase

The transfer may be completed as a single transaction, or it may require multiple transactions to complete the transfer. A loop is used to process the transfer until all of its transactions are completed (or until the transfer is aborted).

The following loops are discussed here:

- Transfer loop for device non-isochronous
- Transaction loop for device non-isochronous (this is a subset of the entire transfer loop)

Transfer Loop for Device Non-isochronous

Sequence of operations:

1. Create and prepare new transactions.

   Each time the Transfer Loop is executed, a new transaction object is created that represents the next transaction needed to implement the transfer. For more information on the detailed callback and notification sequences, see Create and Prepare New Transaction.
2. Process the transaction.
3. The transaction is processed according to the transaction loop described in Transaction Loop for Device Non-Isochronous.
4. (Optional) Repeat the transfer loop if more transactions are required.

   This step depends on the specific transfer. If there is more payload to send and the transfer is not aborted, then continue with the transfer loop.
5. If the transfer loop does not have to be repeated, VIP issues *transfer.notify.indicate,* and then issues *NOTIFY_CB_TRANSFER_ENDED*.

Transaction Loop for Device Non-Isochronous

Sequence of operations:

1. If this is the first transaction of the transfer, mark the transfer as STARTED using *transfer.notify.indicate*.
2. If the transaction requires the Device to receive a TOKEN, DATA or HANDSHAKE packet from the Host, use the Get Received Packet callback flow.

   If the transaction requires the Device to send either a DATA packet or a HANDSHAKE packet, use the Prepare and Send Packet callback flow.
3. Device responds to the transaction based on the type of transaction.

Device response if the transaction is not a SPLIT OUT or a SETUP transaction
If you need to send a packet, use the Prepare and Send Packet callback flow.

If the device response is legal protocol and has no error exceptions, complete the following steps:

1. If the transaction is not a SETUP transaction, VIP calls *transaction.payload = received_packet.payload.copy().*
2. VIP then calls the following:
   - *svt_usb_protocol_callbacks::received_data_packet(svt_usb_protocol xactor, svt_usb_transfer transfer, svt_usb_packet packet)*
   - transaction.status = svt_transaction::ACCEPT
   - transaction.notify.indicate
   - svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())
3. If the transaction is not a SETUP transaction, VIP calls *transfer.payload.append_payload(transaction.payload))*

If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.

Device response if the transaction is a START SPLIT transaction

If the transaction is not an Interrupt, complete the following steps:

1. If you need to send a packet, use the Prepare and Send Packet callback flow.
2. If the device response is legal protocol and has no error exceptions, move to Complete SPLIT

If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.

Device response if the transaction is a Complete SPLIT OUT or SETUP transaction

If you need to send a packet, use the Prepare and Send Packet callback flow.

If the device response is legal protocol and has no error exceptions, complete the following steps:

1. If the transaction is not a SETUP transaction, VIP calls the following:
   - *transaction.payload = received_packet.payload.copy()*.
   - svt_usb_protocol_callbacks::received_data_packet(svt_usb_protocol xactor, svt_usb_transfer transfer, svt_usb_packet packet)
   - transfer.payload.append_payload(transaction.payload))
   - If the transaction is a SETUP transaction, VIP calls the following:
   - *svt_usb_protocol_callbacks::received_data_packet(svt_usb_protocol xactor, svt_usb_transfer transfer, svt_usb_packet packet)*
2. VIP then calls the following:
   - *transaction.status = svt_transaction::ACCEPT*
   - *transaction.notify.indicate*
   - *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*
3. If the transaction is not a SETUP transaction, VIP calls *transfer.payload.append_payload(transaction.payload)).*

If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.

Device response if the transaction is a non-SPLIT IN transaction

If you need to send a packet, use the Prepare and Send Packet callback flow.

If you need to receive a packet, use the Get Received Packet callback flow.

If the host handshake packet is legal protocol and has no error exceptions, VIP does the following:

1. *transaction.status = svt_transaction::ACCEPT*
2. *transaction.notify.indicate*
3. *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*

If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.

Device response if the transaction is a Complete SPLIT IN (INTERRUPT) transaction

If you need to send a packet, use the Prepare and Send Packet callback flow.

If this is the last CSPLIT and the Device DATA packet is legal protocol and error free, VIP does the following:

1. *transaction.status = svt_transaction::ACCEPT*
2. *transaction.notify.indicate*
3. *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*

If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.

Device response if the transaction is a Complete SPLIT IN (BULK or CONTROL) transaction

If you need to send a packet, use the Prepare and Send Packet callback flow.

If this is the last CSPLIT and the Device DATA packet is legal protocol and error free, VIP does the following:

1. *transaction.status = svt_transaction::ACCEPT*
2. *transaction.notify.indicate*
3. *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*

If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.

4. To repeat the transaction loop, go back to the flow at the top of the transaction loop. Otherwise, the transaction has already been marked ENDED and VIP issues *NOTIFY_CB_TRANSACTION_ENDED*.

3.1.3.2.2 Isochronous Transfers

Transfer Start Phase

A device isochronous transfer begins when the Device VIP receives a TOKEN packet object on the packet input channel.

Sequence of operations:

1. VIP receives the TOKEN packet from the *svt_usb_protocol::packet_in_chan* channel. For more information about the callbacks and notifications, see Get Received Packet.
2. VIP creates the new device VIP transfer object and allows you to modify or replace it. For more information about the callbacks and notifications, see Create and Prepare New Device Transfer.

Transfer Processing Phase

The transfer may be completed as a single transaction, or it may require multiple transactions to complete the transfer. A loop is used to process the transfer until all of its transactions are completed (or until the transfer is aborted).

The following loops are discussed here:

- Transfer loop for device isochronous
- Transaction loop for device isochronous (this is a subset of the entire transfer loop)

Transfer Loop for Device Isochronous

Sequence of operations:

1. Create and prepare new transactions.

   Each time the Transfer Loop is executed, a new transaction object is created that represents the next transaction needed to implement the transfer. For more information on the detailed callback and notification sequences, see Create and Prepare New Transaction.
2. Process the transaction.
3. The transaction is processed according to the transaction loop described in Transaction Loop for Device Non-Isochronous.
4. (Optional) Repeat the transfer loop if more transactions are required.

   This step depends on the specific transfer. If there is more payload to send and the transfer is not aborted, then continue with the transfer loop.
5. If the transfer loop does not have to be repeated, VIP issues *transfer.notify.indicate,* and then issues *NOTIFY_CB_TRANSFER_ENDED*.

Transaction Loop for Device Isochronous

Sequence of operations:

1. If this is the first transaction of the transfer, mark the transfer as STARTED using *transfer.notify.indicate*.
2. If the transaction requires the Device to receive a TOKEN, DATA or HANDSHAKE packet from the Host, use the Get Received Packet callback flow.
3. Device responds to the transaction based on the type of transaction.

Device response if the transaction is a HOST ISOC IN transaction
If the device response is legal protocol and has no error exceptions, subtract the transaction.payload.byte_count from the transfer.payload_bytes_remaining attribute.
If the transaction is either not a SPLIT or is the last complete SPLIT transaction, complete the following steps: 1. Mark the transaction status as ACCEPT using *transaction.status = svt_transaction::ACCEPT.* 2. Trigger the transaction notify that the transaction has ENDED using *transaction.notify.indicate.* 3. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy()).*
If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.

Device response if the transaction is a HOST ISOC OUT DATA transaction
If the Host DATA is a legal protocol and has no packet exception errors, VIP calls the following: 1. *transaction.payload.append_payload(packet.payload)* 2. *svt_usb_protocol_callbacks::received_data_packet(svt_usb_protocol xactor, svt_usb_transfer transfer, svt_usb_packet packet)*
If the transaction is non-SPLIT or last start SPLIT, VIP calls the following: 1. *transaction.status = svt_transaction::ACCEPT* 2. *transaction.notify.indicate* 3. *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())* 4. *transfer.payload.append_payload(transaction.payload))*
If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.

4. To repeat the transaction loop, go back to the flow at the top of the transaction loop. Otherwise, the transaction has already been marked ENDED and VIP issues *NOTIFY_CB_TRANSACTION_ENDED*.

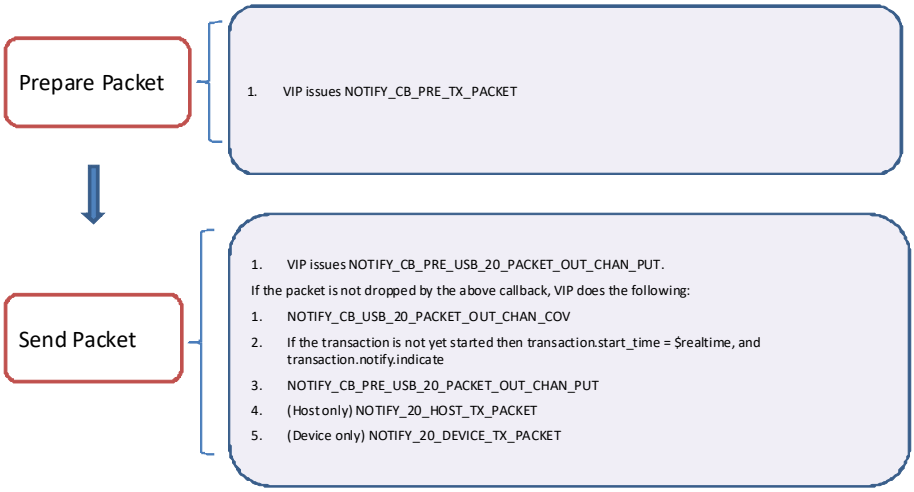
3.1.3.3 Common Protocol 2.0 Callback and Notification Flows

This section documents common sequencing processes and callbacks used by the Host and Device VIP. Some of these callback flows are common to all types of transfers. These sequences are building blocks that are referred to in the Host and Device callback flows.

Prepare and Send Packet

Figure 3-4 illustrates the callback and notification flow.

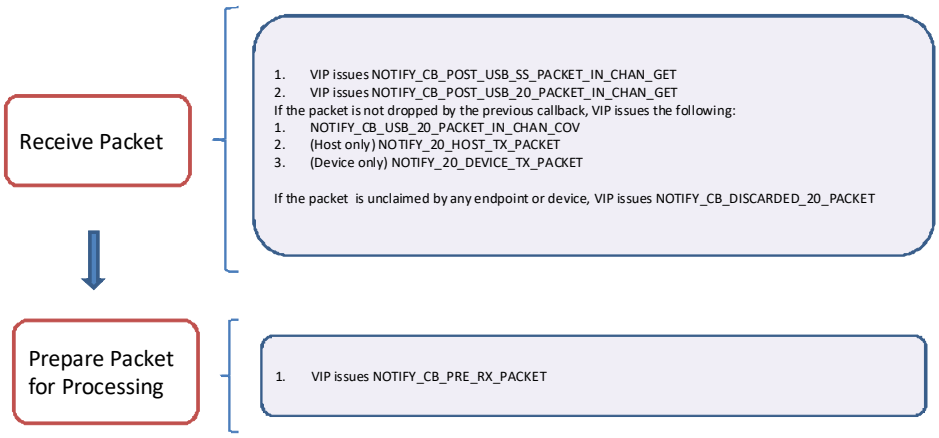
Figure 3-4 Sequence of Callback and Notifications when Preparing and Sending a Packet



Get Received Packet

Figure 3-5 illustrates the callback and notification flow.

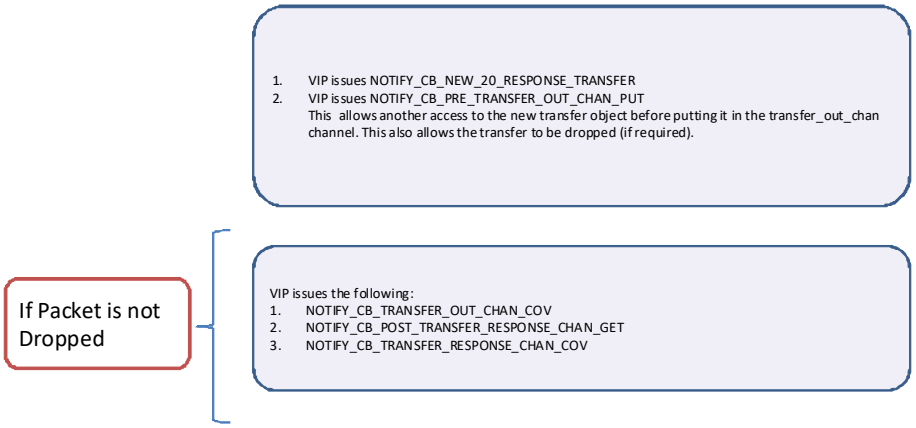
Figure 3-5 Sequence of Callback and Notifications when Receiving and Preparing a Packet for Processing



Create and Prepare New Device Transfer

Figure 3-6 illustrates the callback and notification flow.

Figure 3-6 Callback and Notification Flow when Creating and Preparing New Device Transfer



Create and Prepare New Transaction

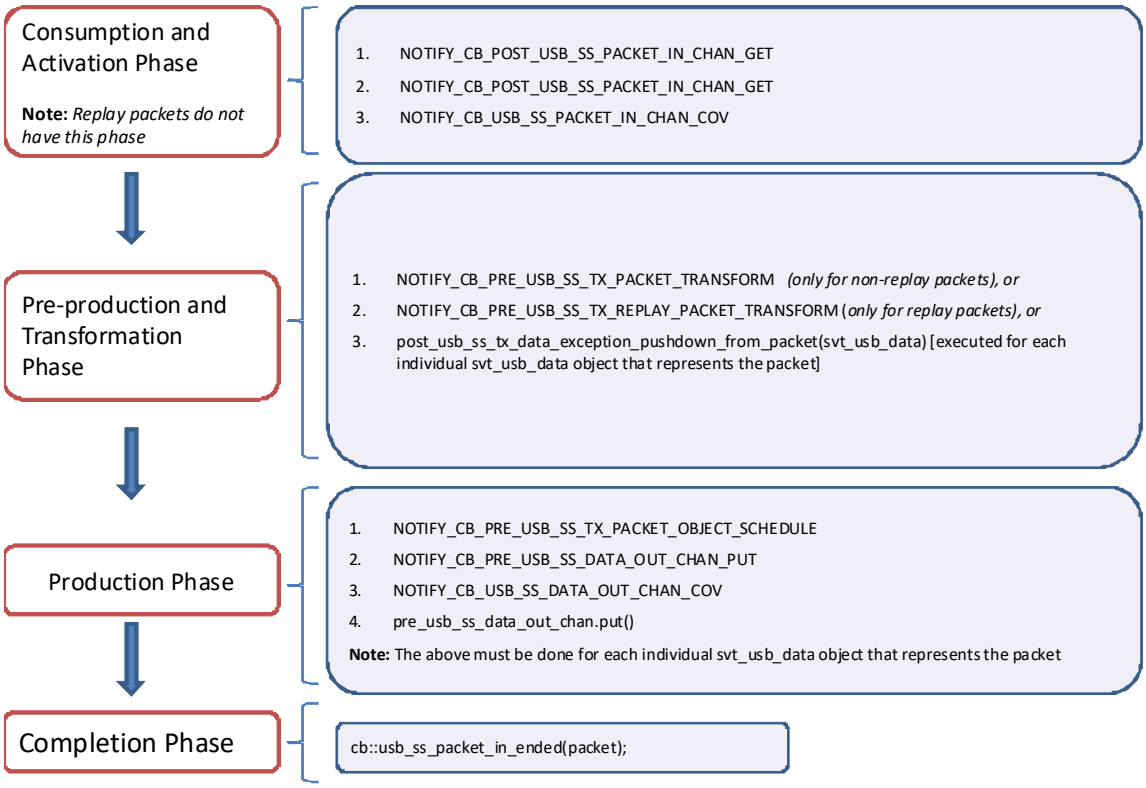
VIP issues NOTIFY_CB_PRE_TRANSACTION.

3.1.4 Link Transactor SuperSpeed Link Notification Flows

This section explains the various link-layer flows through the use of flowcharts. Figures 3-7 to 3-17 illustrate the various SS link transactor flows.

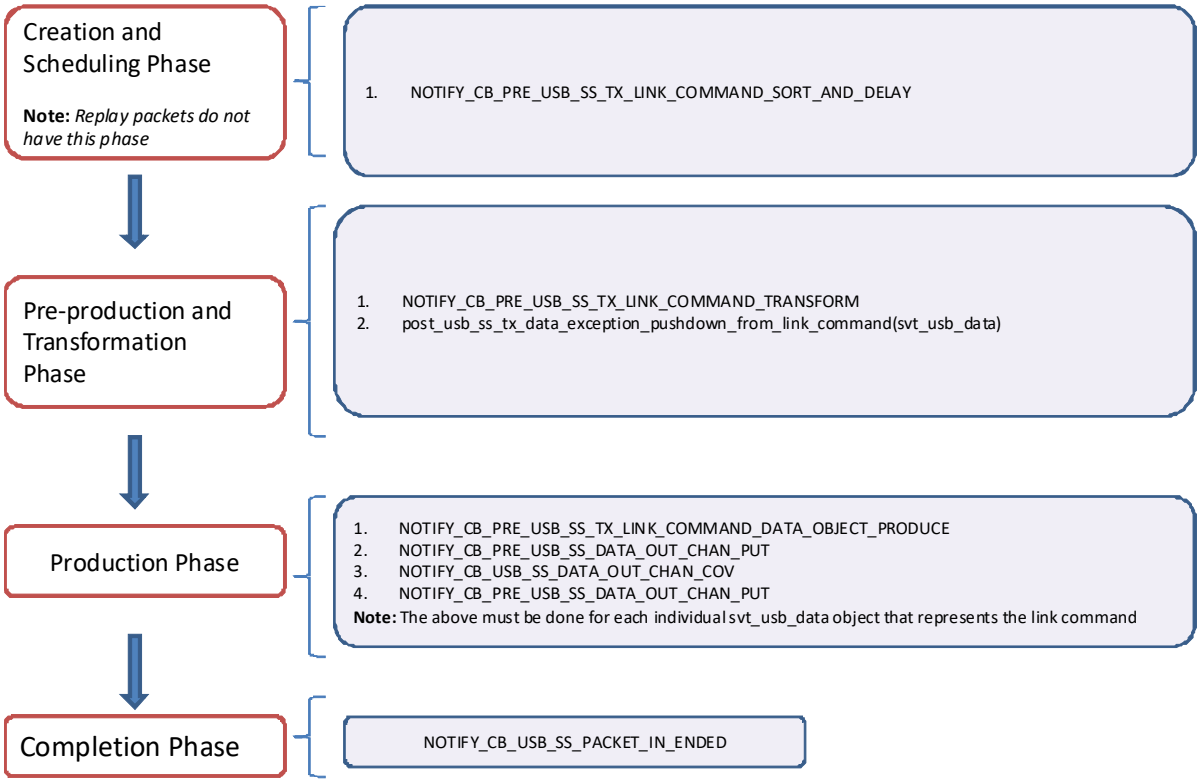
SuperSpeed Packet: Transmit Flow

Figure 3-7 Super Speed Transmit Packet Flow



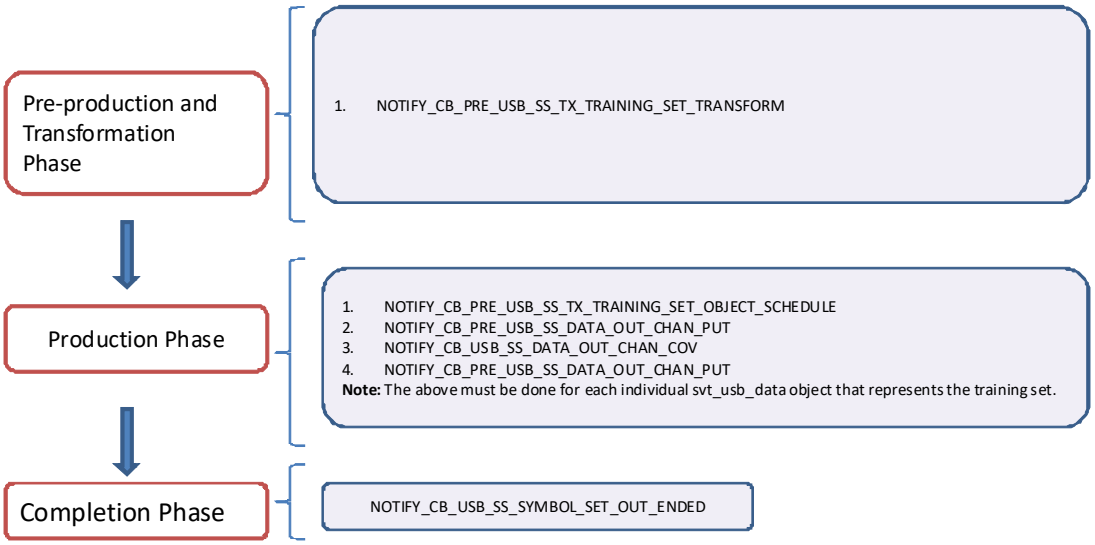
SuperSpeed Packet: Transmit Link Command Flow

Figure 3-8 SuperSpeed Transmit Link Command Flow



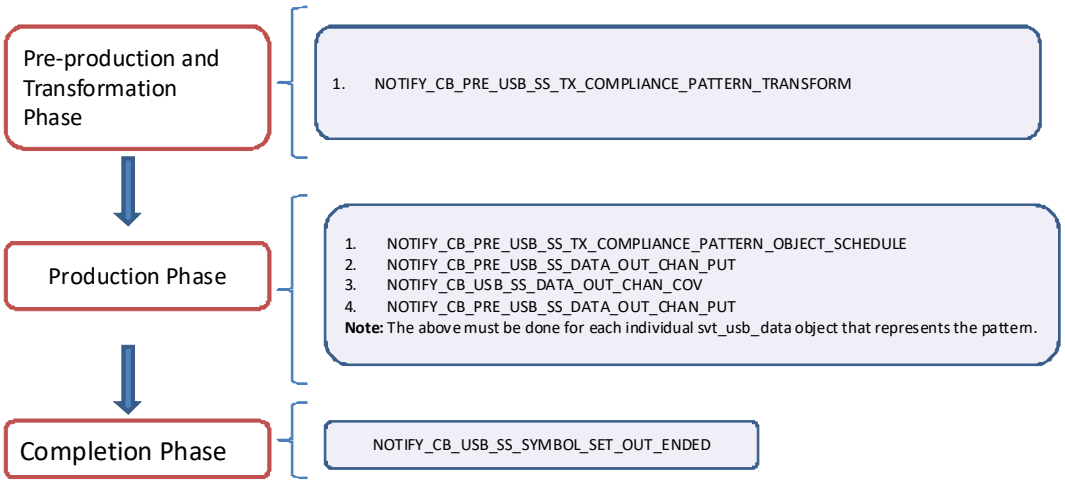
SuperSpeed Packet: Transmit Training Set Flow

Figure 3-9 Super-speed Transmit Training-Set Flow



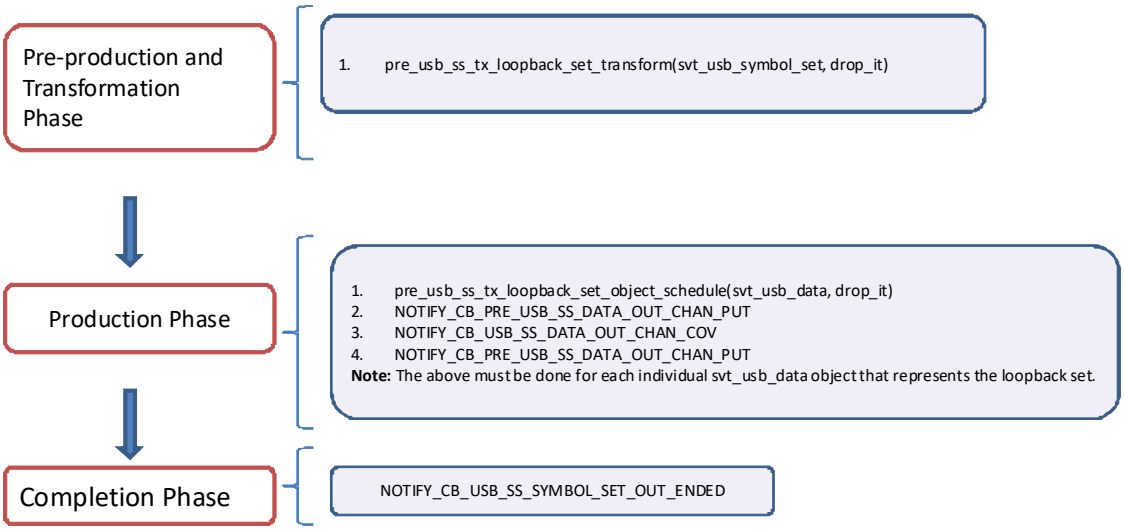
SuperSpeed Packet: Transmit Compliance Pattern Flow

Figure 3-10 SuperSpeed Transmit and Compliance Pattern Flow



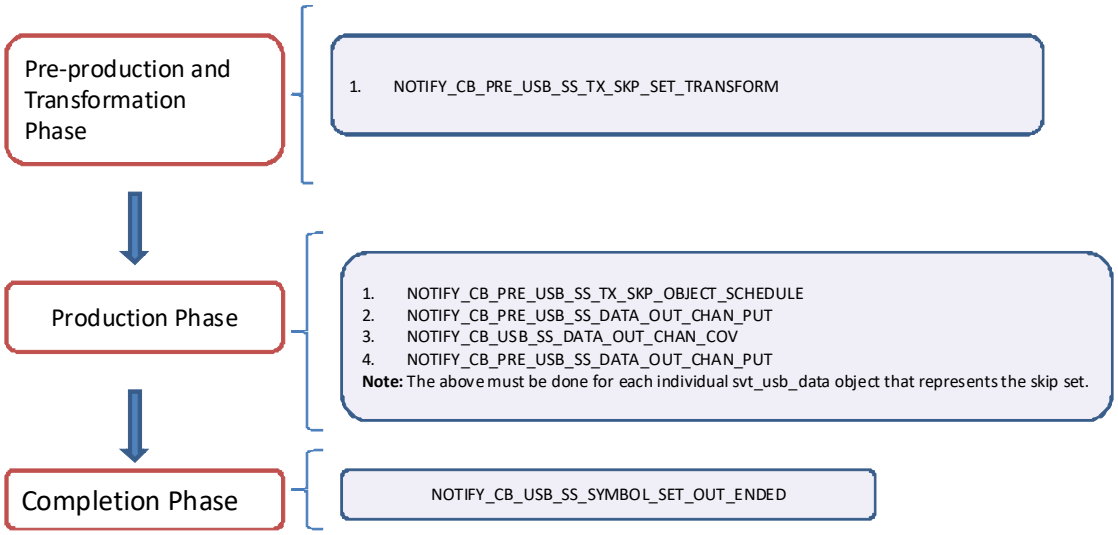
SuperSpeed Packet: Transmit Loopback Set Flow

Figure 3-11 Super-speed Transmit Loopback Set Flow



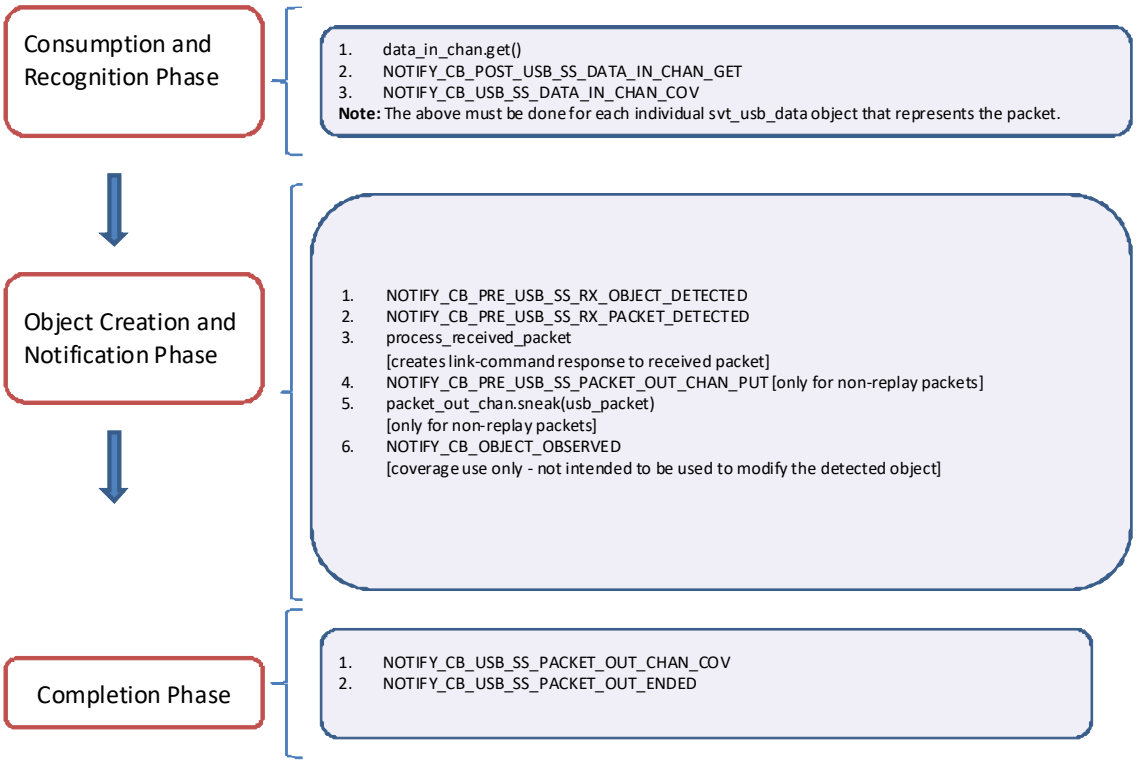
SuperSpeed Packet: Skip-Ordered Set Flow

Figure 3-12 SuperSpeed Transmit Skip Ordered Set Flow



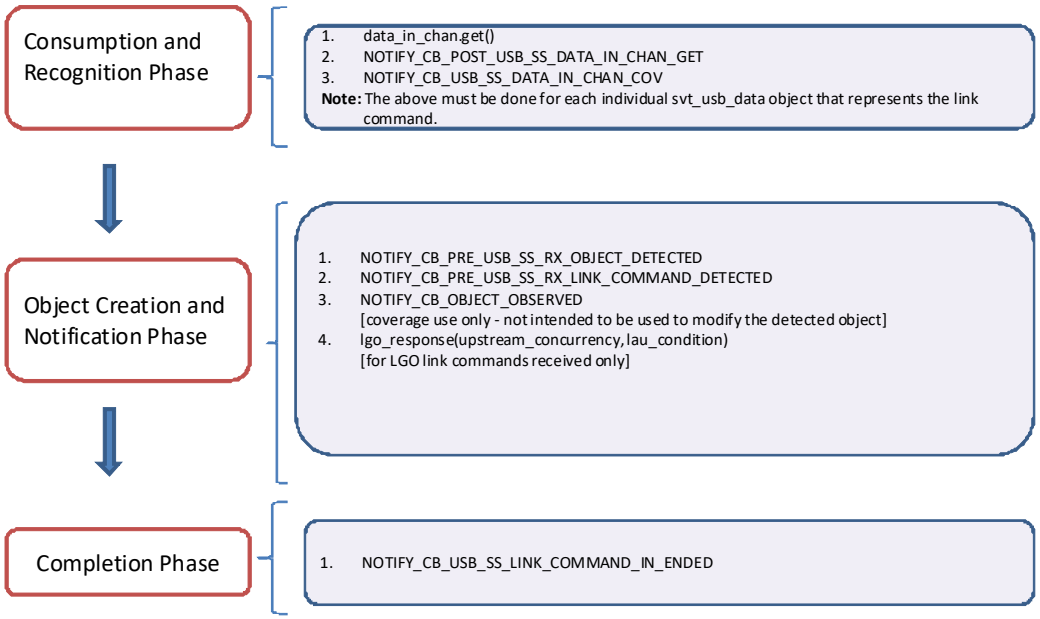
SuperSpeed Packet: Receive Flow

Figure 3-13 SuperSpeed Receive Packet Flow



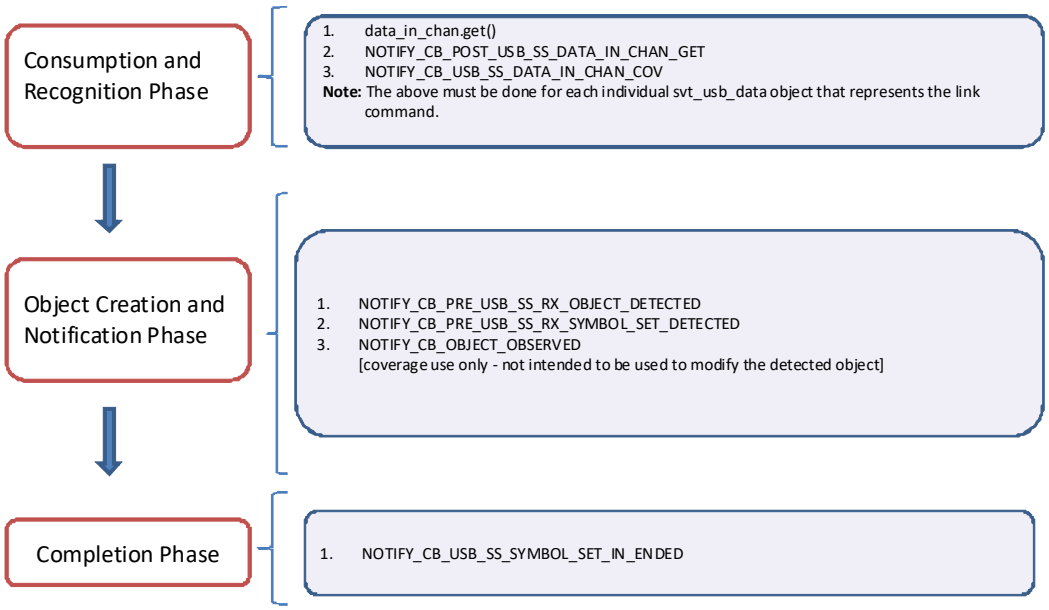
SuperSpeed Packet Receive Link Command Flow

Figure 3-14 SuperSpeed Receive Link Command Flow



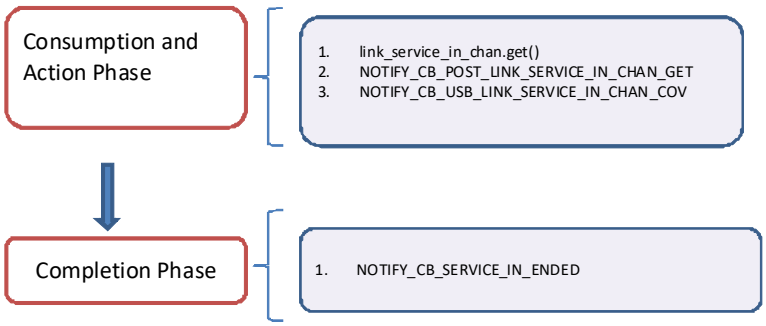
SuperSpeed Packet: Receive Symbol Set Flow

Figure 3-15 SuperSpeed Receive Symbol Set Flow



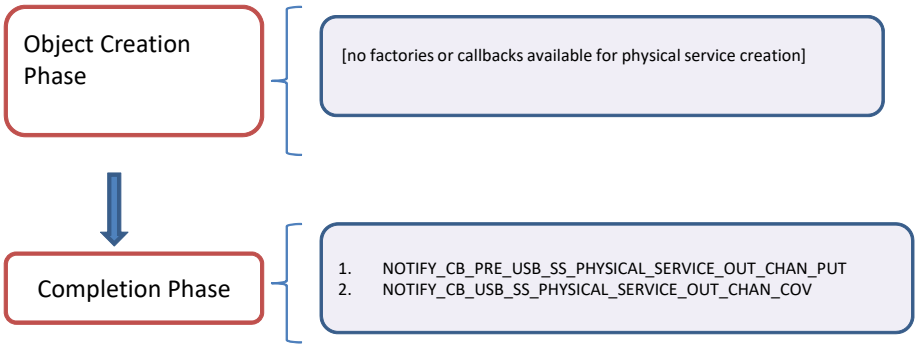
SuperSpeed Packet: Link Service Request Flow (Incoming)

Figure 3-16 Link Service Request Flow (Incoming)



SuperSpeed Packet: Physical Request Flow (Outgoing)

Figure 3-17 SuperSpeed Physical Request Flow (Outgoing)

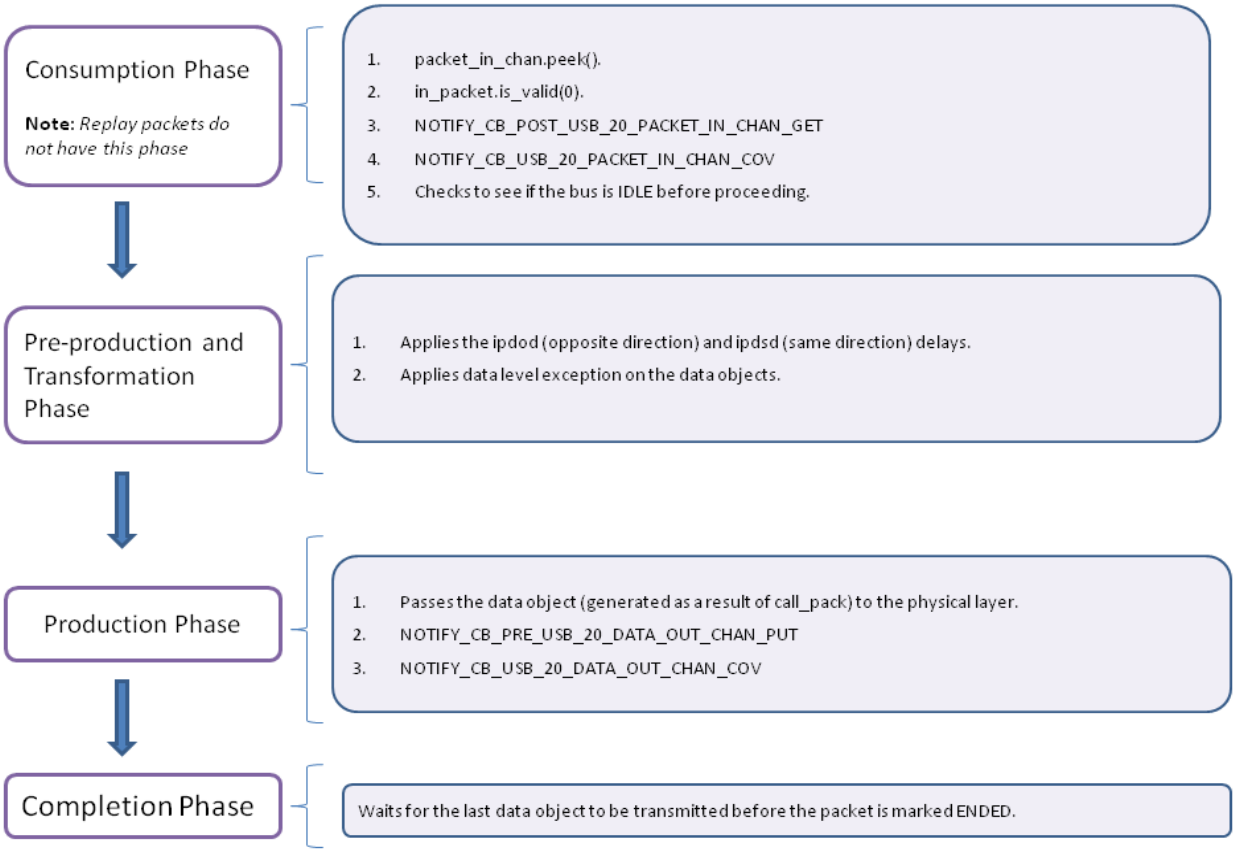


3.1.5 Link Transactor 2.0 Notification Flows

This section explains the various link-layer flows through the use of flowcharts. Figures 3-18 and 3-19 illustrate the various 2.0 link transactor flows.

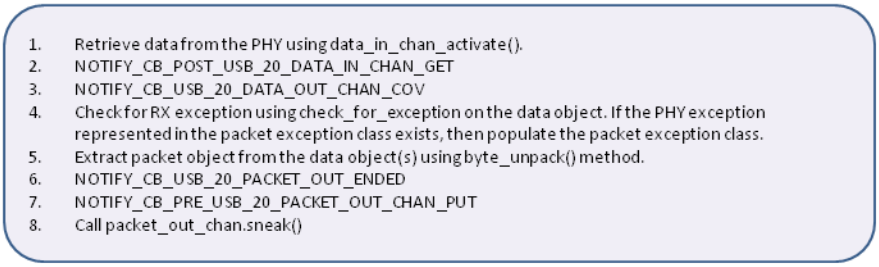
USB 2.0 Link Transmit Packet Flow

Figure 3-18 USB 2.0 Link Transmit Packet Flow



USB 2.0 Receive Packet Flow

Figure 3-19 USB 2.0 Receive Packet Flow



USB 2.0 Service Command Handling

Table 3-1 lists the various USB 2.0 link service commands that are available.

Table 3-1 USB 2.0 Service Command Usage

Command	Host or Device	Usage
SVT_USB_20_PORT_RESET	Host only command	Causes the VIP to start driving protocol reset. Note: The VIP must be in a state other than POWERED_OFF or DISCONNECTED.
SVT_USB_20_SET_PORT_SUSPEND	Host and Device	Allows the VIP to move to the Suspend state (provided the command is issued when the VIP is in Idle state).
SVT_USB_20_CLEAR_PORT_SUSPEND	Host and Device	Allows the VIP to initiate Resume (provided the command is issued when the VIP is in Suspend state).
SVT_USB_20_PORT_START_LPM	Host and Device	Allows the VIP to transition to the L1 Suspend/Resume state machine, instead of the normal Suspend/Resume.
SVT_USB_20_PORT_INITIATE_SRP (NYI)	Host and Device	Allows the user to create SRP manually instead of the VIP doing it automatically based on timers.
SVT_USB_PACKET_ABORT	Host and Device	Kills the packet that is currently being processed. Note: The current implementation does not append an EOP to the packet that is being aborted.

3.1.6 Module Types

This section lists and describes the modules that you should instantiate in the testbench.

The USB VIP defines seven modules:

- svt_usb_subenv_pipe3_plp_hdl.sv
- svt_usb_subenv_20_serial_pipe3_plp_hdl.sv

- svt_usb_subenv_ss_serial_plp_hdl.sv
- svt_usb_subenv_20_serial_plp_hdl.sv
- svt_usb_subenv_20_serial_ss_serial_plp_hdl.sv
- svt_usb_subenv_20_serial_pipe3_plpr_hdl.sv
- svt_usb_subenv_pipe3_plpr_hdl.sv

The following sections describe the available modules

3.1.6.1 PLP Modules

PLP modules are capable of providing three-transactor stacks – protocol, link, and physical layers. The Link and Protocol transactors are optional in each module – the top transactor of the stack is configured through a top_xactor property.

Port interface options differentiate the five provided plp modules.

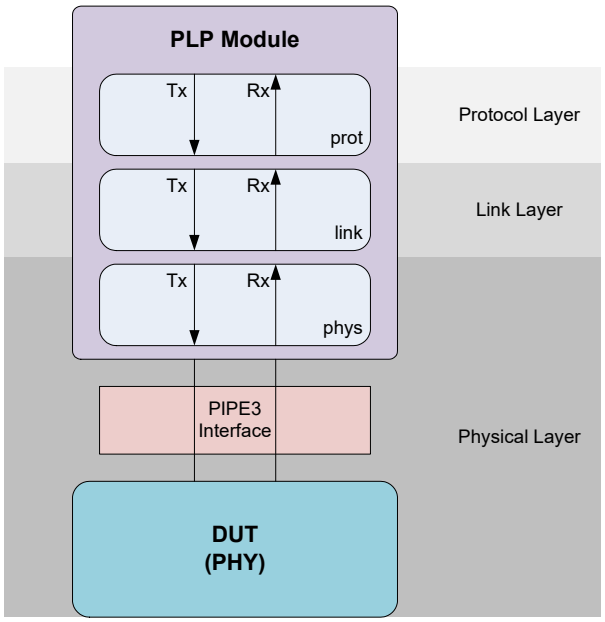
svt_usb_subenv_pipe3_plp_hdl.sv

Module supporting the SuperSpeed Pipe3 and OTG signal interface, conveyed through the local PHY within a potentially 3 element protocol stack:

- Protocol layer (optional),
- Link layer (optional), and
- Physical layer (required)

This module provides a PIPE3 interface from the physical transactor for connecting to a PHY that is local to the VIP. Figure 3-20 displays the module structure.

Figure 3-20 Module Structure – svt_usb_subenv_pipe3_plp_hdl



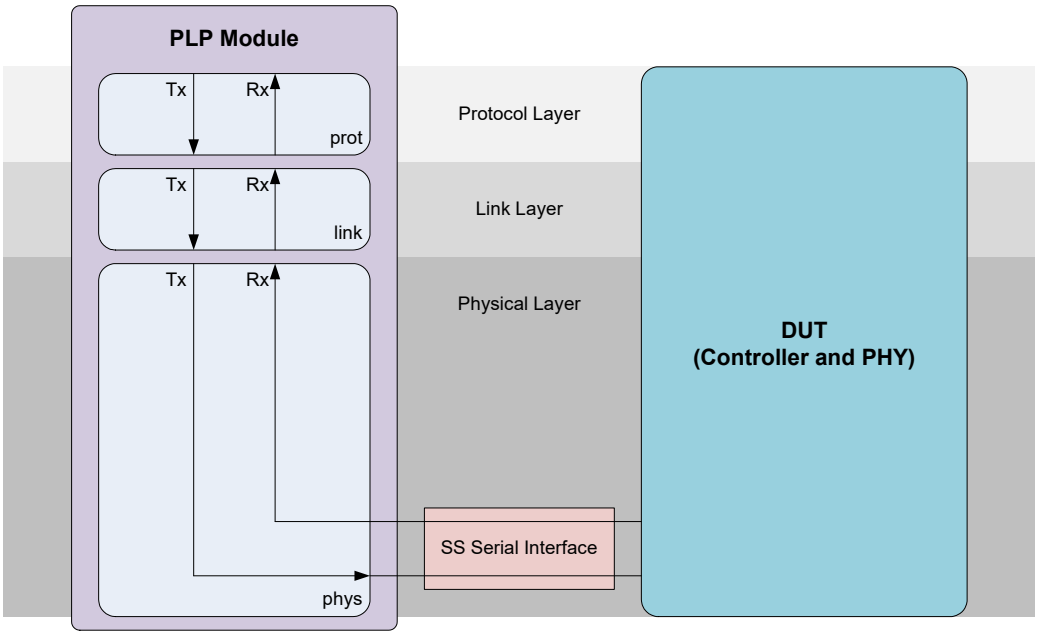
svt_usb_subenv_ss_serial_plp_hdl.sv

Module supporting the SuperSpeed Serial and OTG signal interfaces, conveyed through the local PHY (SuperSpeed Serial) Pipe3) within a 3 element protocol stack:

- Protocol layer (optional),
- Link layer (optional),
- Physical layer (required)

This module provides an SS serial interface from the physical transactor for connecting to a remote object through its PHY. Figure 3-21 displays the module structure.

Figure 3-21 Module Structure – svt_usb_subenv_ss_serial_plp_hdl



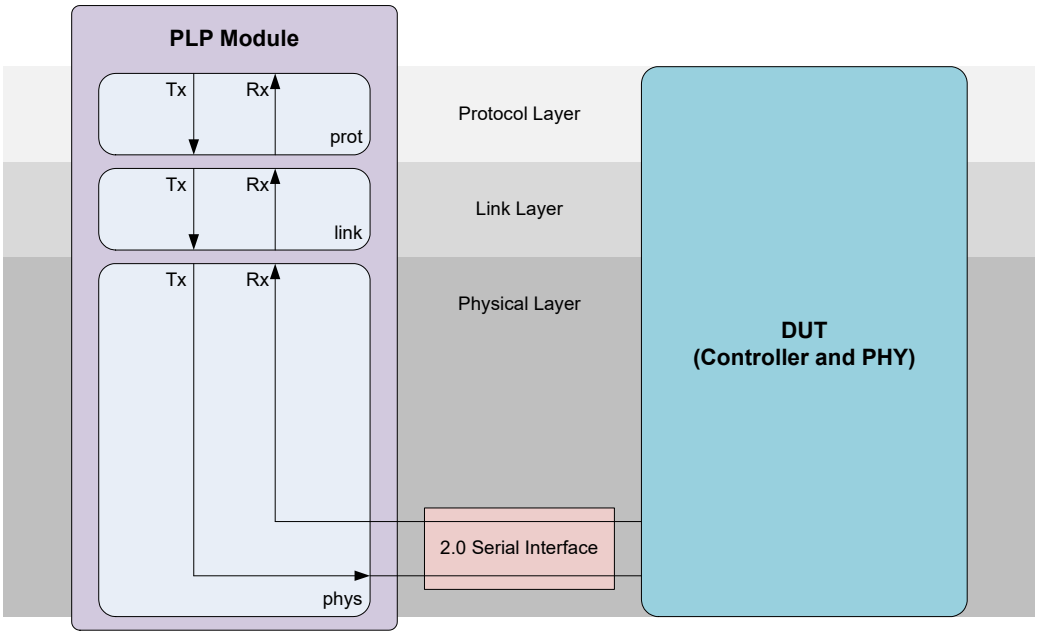
svt_usb_subenv_20_serial_plp_hdl.sv

Module supporting the USB 2.0 Serial and OTG signal interfaces, conveyed through the local PHY within a potentially 3 element protocol stack:

- Protocol layer (optional),
- Link layer (optional), and
- Physical layer (required)

This module provides a 2.0 serial interface from the physical transactor for connecting to a remote object through its PHY. Figure 3-22 displays the module structure.

Figure 3-22 Module Structure – svt_usb_subenv_20_serial_plp_hdl



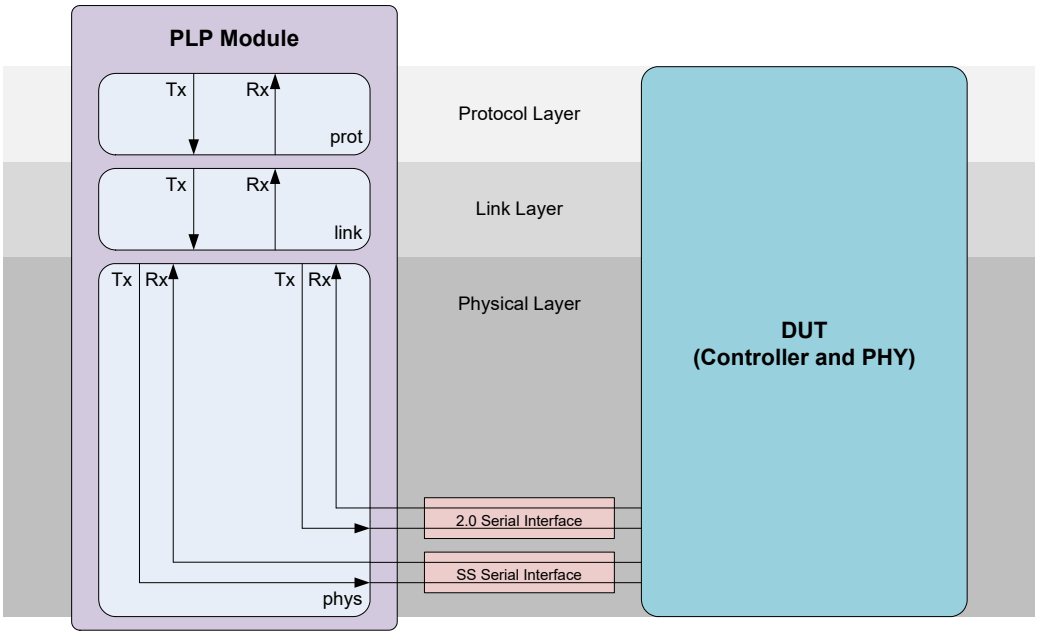
svt_usb_subenv_20_serial_ss_serial_plp_hdl.sv

Module supporting the USB 2.0 Serial, SuperSpeed serial and OTG signal interfaces, conveyed via the local PHY within a potentially 3 element protocol stack:

- Protocol layer (optional),
- Link layer (optional), and
- Physical layer (required)

This module provides a 2.0 serial interface and an SS serial interface from the physical transactor for connecting to a remote object through its PHY. Figure 3-23 displays the module structure.

Figure 3-23 Module Structure – svt_usb_subenv_20_serial_ss_serial_plp_hdl



svt_usb_subenv_20_serial_pipe3_plp_hdl.sv

Module supporting the USB 2.0 Serial, SuperSpeed Pipe3 and OTG signal interfaces, conveyed through the local PHY (USB 2.0 Serial and SuperSpeed Pipe3) within a potentially 3 element protocol stack:

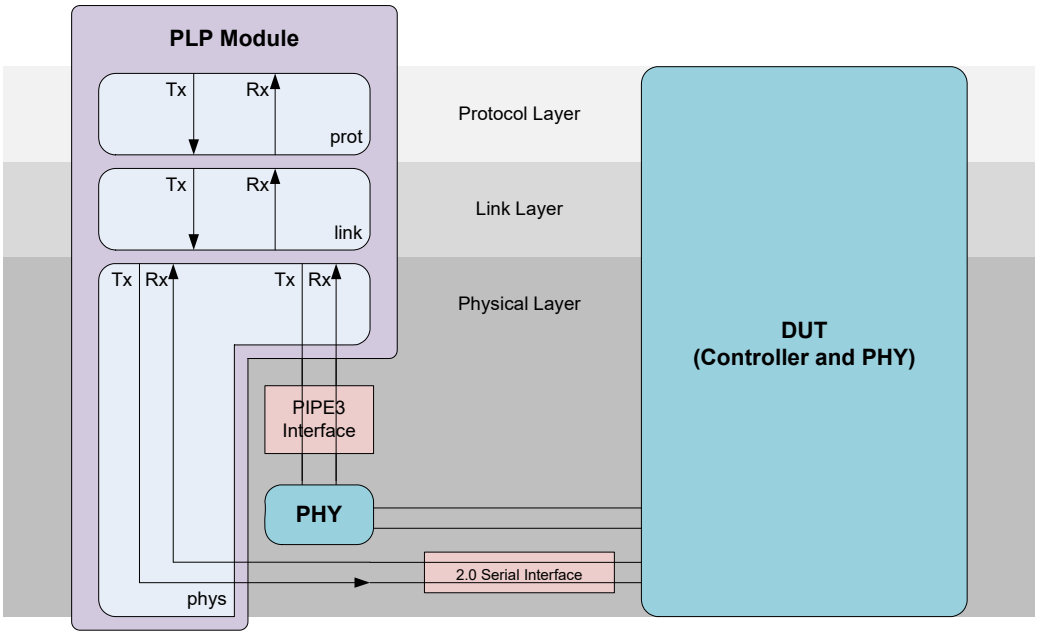
- Protocol layer (optional),
- Link layer (optional), and
- Physical layer (required)

This module provides the following interfaces:

- a PIPE3 interface from the physical transactor for connecting to a PHY that is local to the VIP
- a 2.0 serial interface from the physical transactor for connecting to a remote object through its PHY.

Figure 3-24 displays the module structure.

Figure 3-24 Module Structure – svt_usb_subenv_20_serial_pipe3_plp_hdl



3.1.6.2 PLPR Modules

PLP modules are capable of providing four-transactor stacks – protocol, link, physical, and remote-physical transactors. The Link and Protocol transactors are optional in each module – the top transactor of the stack is configured through a top_xactor property.

Port interface options differentiate the two provided plpr modules.

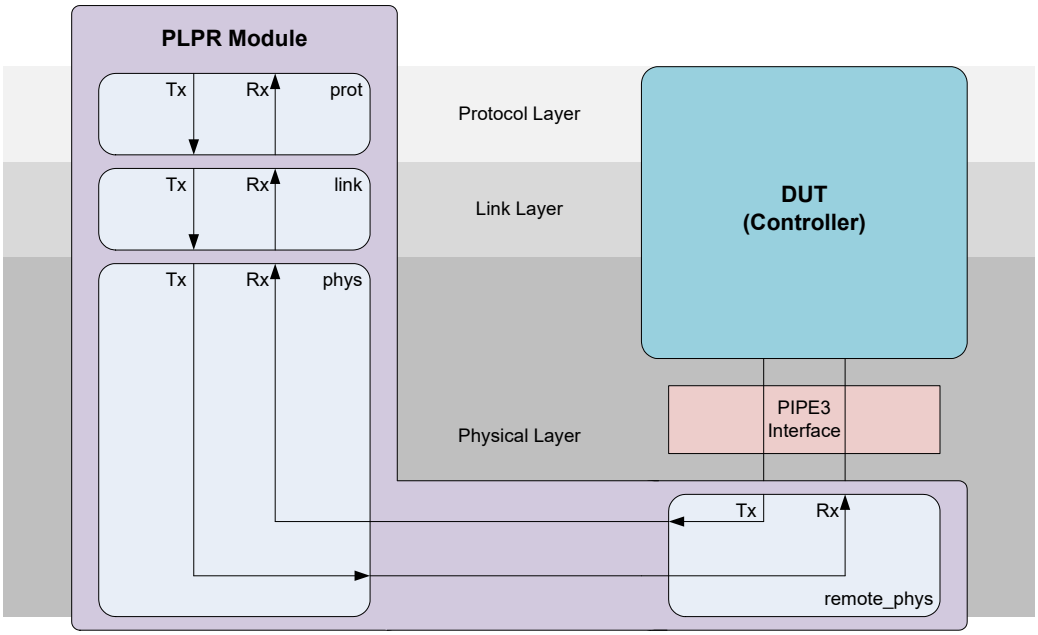
svt_usb_subenv_pipe3_plpr_hdl.sv

Module supporting the SuperSpeed Pipe3 and OTG signal interfaces, conveyed through the remote PHY within a potentially 4 element protocol stack:

- Protocol layer (optional),
- Link layer (optional),
- Physical layer (required), and
- Remote Physical layer (required)

This module provides a PIPE3 interface from the remote physical transactor for connecting to a USB controller. Figure 3-25 displays the module structure.

Figure 3-25 Module Structure – svt_usb_subenv_pipe3_plpr_hdl



svt_usb_subenv_20_serial_pipe3_plpr_hdl.sv

Module supporting the USB 2.0 Serial, SuperSpeed Pipe3 and OTG signal interfaces, conveyed through the local PHY (USB 2.0 Serial) and remote PHY (.e., SuperSpeed Pipe3) within a potentially 4 element protocol stack:

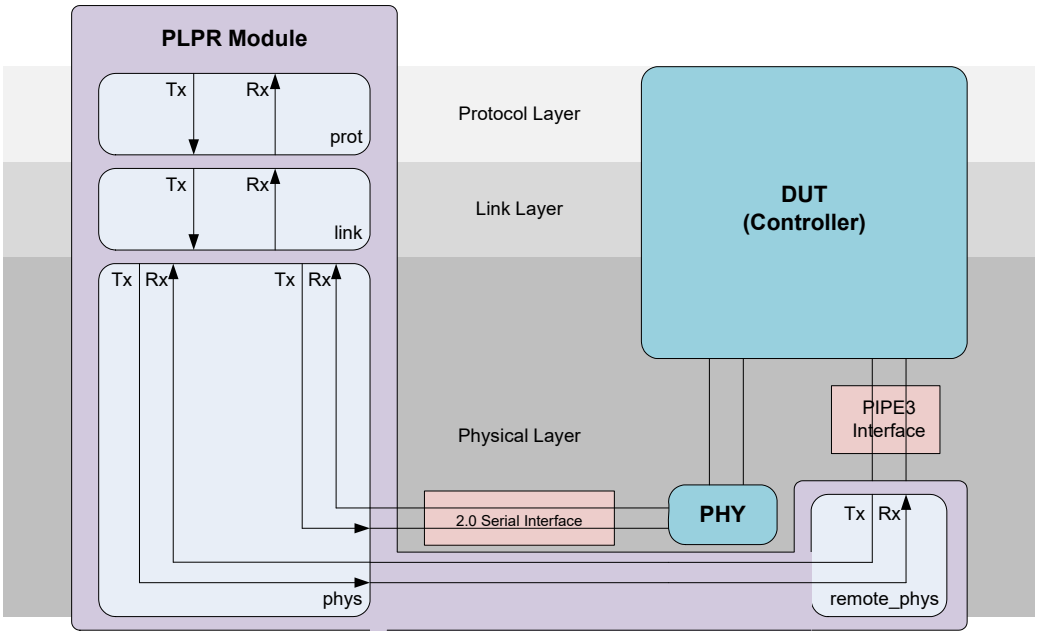
- Protocol layer (optional),
- Link layer (optional),
- Physical layer (required), and
- Remote Physical layer (required)

This module provides the following interfaces:

- a PIPE3 interface from the remote physical transactor for connecting to a USB controller.
- a 2.0 serial interface from the physical transactor for connecting to a remote PHY.

Figure 3-26 displays the module structure.

Figure 3-26 Module Structure – svt_usb_subenv_20_serial_pipe3_plpr_hdl



4 Using the VC VIP for USB

After installing the VIP, you can configure it using coreConsultant, and then instantiate and simulate the VIP.

Attention Commands are described in the USB HDL HTML Reference located at:

`$DESIGNWARE_HOME/vip/usb_svt/latest/doc/usb_svt_hdl/html/index.html`

4.1 Configuring VIP Using coreConsultant

coreConsultant is a software tool that Synopsys provides that you can use to simplify VIP configuration. The coreConsultant tool provides a graphical user interface (GUI) that guides you through the configurations tasks.

Requirements

To use coreConsultant to configure VIP, you must have the following:

- Latest version of coreConsultant installed.

  To download coreConsultant, go to the SolvNetPlus Download Center, at:

  https://solvnetplus.synopsys.com/DownloadCenter/dc/product.jsp
- For built-in validation, set VCS_home to the supported VCS installation.
- Shipped coreKit file located at:

  $DESIGNWARE_HOME/vip/svt/usb_svt/<version>/coreKit/USB3Config.coreKit

Configuring VIP Using coreConsultant

To configure VIP using coreConsultant, complete the following steps:

1. Open coreConsultant.
2. Select **File > Install coreKit** and provide the path to the USB3Config.coreKit file.

3. In the **Specify Configuration tab** in the "Activity List", you can change the default configuration values.

Note If you select **Validate Configuration File**, you must set VCS_HOME and provide the path to DESIGNWARE_HOME. Make sure you have appropriate licenses to run the test.

4. In the **Edit > Tool Installation Roots > VCS** textbox, add VCS_HOME.
5. Click **Apply** to generate the configuration file.
6. Click **Results File** to see the output file and validation log.

The <configuration_name>.cfg is the file generated that can be loaded into VIP. The <configuration_name>.log shows the validation result. You load the file using the set_data_prop comand. The property to be set is "filename", and the name of the file is the prop value.

```
task set_data_prop(output bit is_valid, input int handle, string prop_name, bit [1023:0]
    prop_val, int array_ix)
```

4.2 Preparing the VIP for Simulation

After running dw_vip_setup to set up the VIP, use the following steps to prepare for your simulation:

1. Include in your testbench the following files from your design_dir/include directory:

   Verilog – Use the `include directives in the testbench for the following file:

   - svt_usb_cmd_defines.svi

2. Instantiate one of the Verilog modules for the USB VIP. The choice is based on the desired signal interface. The module files for VCS reside in the following location:

   ```
   $DESIGNWARE_HOME/vip/svt/usb_svt/latest/usb_subenv_svt/sverilog/src/vcs
   ```

   Table 4-1 describes the available modules. Note the following general facts about all of them:

   - Any module can represent either host/device/hub
   - The names indicate which interfaces they support, and what kind of 'stack' they are based on
     - **20_serial** indicates USB 2.0 with a serial connection
     - **ss_serial** indicates USB SS with a serial connection
     - **pipe3** indicates USB SS with a Pipe3 signal connection
     - **plp** indicates a 3 stack (**p**rotocol/**l**ink/**p**hysical) on one side of the physical USB bus. The stack is "above" the signal interface (towards the link/protocol).
     - **plpr** indicates a 4 stack (**p**rotocol/**l**ink/**p**hysical/**r**emote_physical), which spans the physical USB bus. The physical/link/protocol layers make up the complete stack on one side and the remote_physical constitutes the PHY below the signal interface (between the signal interface and the physical USB bus).

Table 4-1 Verilog Modules for Testbench Instantiation

Module	Notes
• svt_usb_subenv_20_serial_pipe3_plp_hdl.sv	Connect these via usb_20_serial for USB 2.0 support, and pipe3mac (3 stacks) for USB support

Table 4-1 Verilog Modules for Testbench Instantiation

Module	Notes
• svt_usb_subenv_20_serial_pipe3_plpr_hdl.sv	Connect these via usb_20_serial for USB 2.0 support, and pipe3phy (4 stacks) for USB support
• svt_usb_subenv_pipe3_plpr_hdl.sv	Connect these via pipe3mac for USB;no USB 2.0 connection.
• svt_usb_subenv_pipe3_plp_hdl.sv	Connect these via pipe3phy for USB;no USB 2.0 connection
• svt_usb_subenv_ss_serial_plp_hdl.sv	Connect via SuperSpeed serial for USB; no USB 2.0 connection
• svt_usb_subenv_20_serial_plp_hdl.sv	Connect via USB 2.0 serial; no USB connection
• svt_usb_subenv_20_serial_ss_serial_plp_hdl.sv	Connect via usb_20_serial for USB 2.0 serial, and SuperSpeed serial for USB

❖ For more information on the different modules (such as signals, parameters, and so on), refer t the USB HDL HTML Reference located at:

```
$DESIGNWARE_HOME/vip/usb_svt/latest/doc/usb_svt_hdl_class_reference/html/index.html
```

3. Write testbench code to drive the VIPs.

   For more information on the different modules (such as signals, parameters, and so on), refer t the USB HDL HTML Reference located at:

```
$DESIGNWARE_HOME/vip/usb_svt/latest/doc/usb_svt_hdl_class_reference/html/index.html
```

4. Build the simulator executable.

```
vcs -l ./logs/compile.log -q -Mdir=./output/csrc -sverilog +plusarg_save -I -
notice +define+SYNOPSYS_SV -ntb_opts use_sigprop -ntb_opts dtm -ntb_opts rvm
+incdir+<design_dir>/src/sverilog/vcs+incdir+<design_dir>/include/sverilog -o
./output/simvcssvlog +warn=noBCNACMBP -f <hdl_files>
```

   Where:

```
<design_dir> = design directory where VIPs are installed
<hdl_files>  = file containing the top.v and other dependencies
```

5. Run the simulator executable.

```
./output/simvcssvlog
```

4.3 Module Instantiation

To create a module instance, creates an instance of one of a module.

Example: The following command creates a PLP sub-environment instance named usb_dev. This sub-environment has a PIPE3 port interface, as shown in Figure 3-19.

```
svt_usb_subenv_pipe3_plp_hdl usb_dev(
    .CLK (CLK),
    .TxData (TxData),
    .TxDataK (TxDataK),
    .RxData (RxData),
```

```
    .RxDataK (RxDataK),
    .PhyMode (PhyMode),
    .ElasticityBufferMode (ElasticityBufferMode),
    .TxDetectRxLoopback (TxDetectRxLoopback),
    .TxElecIdle (TxElecIdle),
    .TxCompliance (TxCompliance),
    .TxOnesZeros (TxOnesZeros),
    .RxPolarity (RxPolarity),
    .RxEqTraining (RxEqTraining),
    .Reset_n (Reset_n),
    .PowerDown (PowerDown),
    .Rate (Rate),
    .TxDeemph (TxDeemph),
    .TxMargin (TxMargin),
    .TxSwing (TxSwing),
    .RxTermination (RxTermination),
    .RxValid (RxValid),
    .PhyStatus (PhyStatus),
    .RxElecIdle (RxElecIdle),
    .RxStatus (RxStatus),
    .PowerPresent (PowerPresent),
    .PCLK (PCLK)
);
```

4.3.1 Testbench Connections

USB VIP modules are instantiated in an HDL testbench, just like any block or module, with a module name, an instance name, and port connections. Signal values are accessed through the ports defined for the module and through the simulator.

4.3.1.1 Ports

The available ports depend on the installed module, as described in Preparing the VIP for Simulation. Table 4-2 lists the ports for each module.

Table 4-2 Available Ports for USB Modules

Module	Ports
svt_usb_subenv_pipe3_plp_hdl	PIPE3 Ports
svt_usb_subenv_ss_serial_plp_hdl	USB SS Ports
svt_usb_subenv_20_serial_plp_hdl	USB 2.0 Ports
svt_usb_subenv_20_serial_ss_serial_plp_hdl	USB 2.0 Ports USB SS Ports
svt_usb_subenv_20_serial_pipe3_plp_hdl	PIPE3 Ports USB 2.0 Ports
svt_usb_subenv_pipe3_plpr_hdl	PIPE3 Ports
svt_usb_subenv_20_serial_pipe3_plpr_hdl	PIPE3 Ports USB 2.0 Ports
svt_usb_subenv_ssic_plps_hdl	SSIC RMMI MAC port
svt_usb_subenv_ssic_plpsr_hdl	SSIC RMMI PHY port

Table 4-2 Available Ports for USB Modules

Module	Ports
svt_usb_subenv_ssic_serial_1_lane_plps_hdl	SSIC serial single lane

Table 4-3 lists the PIPE3 ports.

Table 4-3 PIPE3 Ports

Port	Direction	Size	Description
CLK	input	1	
TxData	output	SVT_USB_MAX_PIPE3_DATA_WIDTH (Valid data widths are 32, 16, and 8)	
TxDataK	output	SVT_USB_MAX_PIPE3_DATAK_WIDTH	
RxData	input	SVT_USB_MAX_PIPE3_DATA_WIDTH (Valid data widths are 32, 16, and 8)	
RxDataK	input	SVT_USB_MAX_PIPE3_DATAK_WIDTH	
PhyMode	output	SVT_USB_PIPE3_PHYMODE_WIDTH	
ElasticityBufferMode	output	1	
TxDetectRxLoopback	output	1	
TxElecIdle	output	1	
TxCompliance	output	1	
RxPolarity	output	1	
RxEqTraining	output	1	
Reset_n	output	1	
PowerDown	output	SVT_USB_PIPE3_POWERDOWN_WIDTH	
Rate	output	1	
TxDeemph	output	SVT_USB_PIPE3_TXDEEMPH_WIDTH	
TxMargin	output	SVT_USB_PIPE3_TXMARGIN_WIDTH	
TxSwing	output	1	
RxTermination	output	1	
RxValid	input	1	
PhyStatus	input	1	
RxElecIdle	input	1	
RxStatus	input	SVT_USB_PIPE3_RXSTATUS_WIDTH	

Table 4-3 PIPE3 Ports

Port	Direction	Size	Description
PowerPresent	input	1	
PCLK	input	1	

Table 4-4 lists the USB 2.0 Serial ports.

Table 4-4 USB 2.0 Serial Ports

Port	Direction	Size	Description
clk	input	1	
dp	inout	1	
dm	inout	1	
vbus	inout	1	

Table 4-5 lists the USB SS Serial ports.

Table 4-5 USB SS Serial Ports

Port	Direction	Size	Description
ssclk	input	1	
ssrxp	inout	1	
ssrxm	inout	1	
sstxp	inout	1	
sstxm	inout	1	
vbus	inout	1	

4.3.2 Instantiation in Verilog Testbenches

If you have installed the example testbench as detailed in Preparing the VIP for Simulation, you can see an example of Verilog instantiation.

4.4 Module Configuration

Configuration objects define the attributes that, when assigned, programs the behavior of sub-environment instances. Configuration objects are ordered hierarchically – lower level objects must be referenced through a handle specified in a higher level object. The following top layer objects are initially accessed by retrieving a handle:

- svt_usb_configuration
- svt_usb_subenv_configuration

The following configuration objects are accessed only after acquiring a handle for the object through a get_data_prop command that accesses an svt_usb_subenv_configuration object:

- svt_usb_device_configuration
- svt_usb_host_configuration

The following configuration object are accessible only after acquiring a handle for the object through a get_data_prop command that accesses an svt_usb_device_configuration:

- svt_usb_endpoint_configuration

The following configuration object are accessible only after acquiring a handle for the object through a get_data_prop command that accesses an svt_usb_endpoint_configuration:

- svt_usb_ustream_resource_configuration

4.4.1 Configuring a Subenv Instance

To access top level configuration attributes, acquire an integer handle that references a temporary copy of the configuration object.

Example: The following command acquires an integer handle, named cfg_handle, for the usb_dev module.

```
usb_dev.get_data_prop(is_valid, `SVT_CMD_NULL_HANDLE, "cfg", cfg_handle, 0)
```

To assign configuration settings to the temporary handle, perform a set_data_prop command.

Example: The following command configures the Protocol transactor as the top transactor in the stack of the usb_dev module:

```
usb_dev.set_data_prop(is_valid, cfg_handle, "top_xactor", `SVT_USB_PROTOCOL_LAYER, 0)
```

To access lower level configuration attributes, create a single device acquire an integer handle to the desired configuration object.

Example: The following commands accesses configuration commands that create a local device configuration and acquires an integer handle to the resulting configuration:

```
usb_dev.set_data_prop(is_valid, cfg_handle, "local_device_cfg_size", 1, 0)
usb_dev.get_data_prop(is_valid, cfg_handle, "local_device_cfg", dev_cfg_handle, 0)
```

As a result of these commands, the handle that accesses local device configuration attributes is dev_cfg_handle.

To assign configuration settings from a lower level configuration object, perform a set_data_prop command, using the acquired handle.

Example: The following configures the number of devices, not including the Default Control Pipe:

```
usb_dev.set_data_prop(is_valid, dev_cfg_handle, "local_device_cfg_size", 3, 0)
```

To copy the configuration property data from the temporary handle into the internal configuration of the subenv instance, use an apply data command.

Example: The following commands loads the configuration data to the subenv instance:

```
usb_dev.apply_data(is_valid, cfg_handle, 1)
usb_dev.apply_data(is_valid, dev_cfg_handle, 1)
```

4.4.2 Configuration Objects

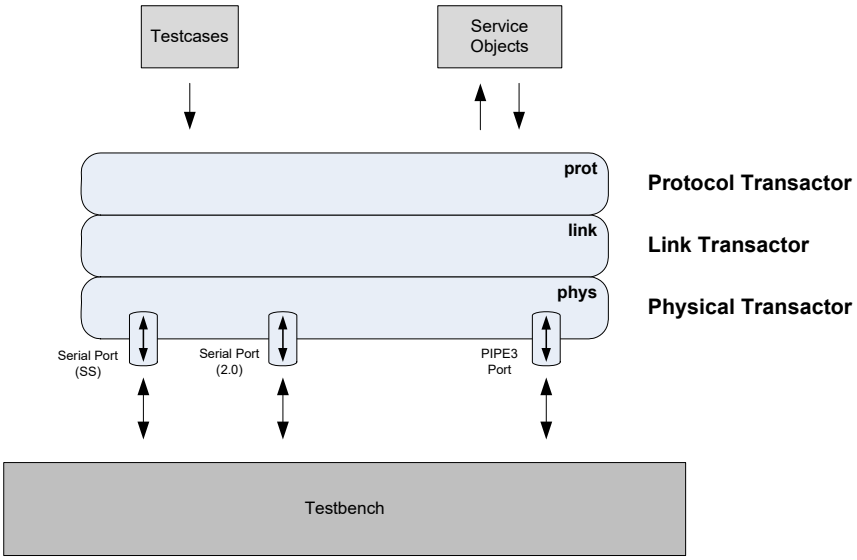
4.4.2.1 Host Configuration

When configured as a USB Host, the Protocol transactor transfers data to and from USB device endpoints. Figure 4-1 displays the structure of the USB VIP in Host mode.

Host VIP operation includes a scheduling mechanism for determining traffic in a frame/bus interval based on endpoint type and priority. Additional scheduler capabilities include specifying the frame/bus interval when a transfer starts, preventing the interleaving of OUT transfers to multiple endpoints, preventing the interleaving of IN transfers, and allocating bus bandwidth to endpoints based on endpoint type. All additional capabilities are controlled by the testbench.

The Protocol uses the information to support its breaking of transfers into individual transactions.

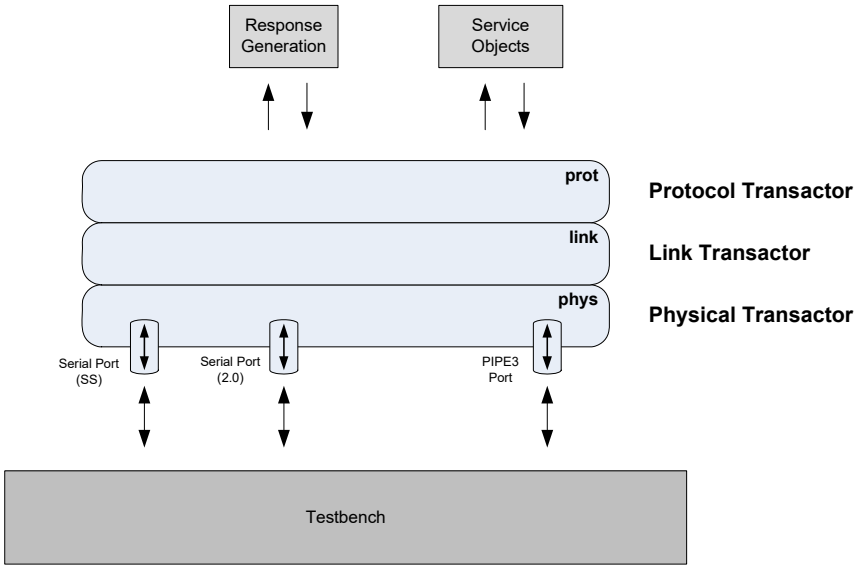
Figure 4-1 USB VIP – Host Mode



4.4.2.2 Device Configuration

When configured as a USB Device, the Protocol transactor receives data and responds to data requests from USB Hosts. The transactor constructs transfer objects from inbound traffic received from USB interfaces, then sends these objects to the testbench. The testbench responds to transfer objects sent by the Protocol transactor. Figure 4-2 displays the structure of the USB VIP in Device mode.

Figure 4-2 USB VIP – Device Mode



The configuration class provides device information for an individual USB device. This object contains information normally conveyed in the USB specified Device, Configuration, and Interface descriptors, collapsed into a single object for use by the protocol layer and the external testbench. This assumes the loading of a specific configuration and interface into the device.

This class also includes descriptions of all of the endpoints for the current configuration and interface. The individual endpoint objects contain pertinent information normally conveyed in the USB specified Endpoint descriptors.

As this information is normally interpreted by the software stack, this information is normally provided by the testbench, and not by the protocol transactor.

The object must be provided as part of the configuration information for a USB device or as part of the remote state information provided to the host communicating with such a device.

4.4.2.3 Endpoint Configuration

This class provides endpoint information for an individual USB endpoint. The information that this class provides can categorized as being made up of:

- Endpoint descriptors – This object contains information that is normally conveyed in the USB endpoint descriptors. As this information is normally interpreted by the software stack, it is normally provided by the testbench, and not by the protocol transactor. The object must be provided as part of the configuration information for a USB device or as part of the remote state information provided to the host communicating with such a device.
- Dynamic info – There are certain variable in the object that are updated by the VIP dynamically. These dynamic changing variables provide the testbench with the current state of the DUT. These properties are to be updated by the protocol xactor ONLY.

4.4.2.4 Ustream Resource Configuration

This class contains information regarding a 'USB SS stream resource'. In the USB protocol transactor there will be on 'sub-transactor' for each instance of svt_usb_ustream_resource_configuration if the endpoint supports streams. That sub-transactor needs to know which USB stream ids it is to support. This class provides that information.

4.5 Data Objects

4.5.1 USB Transfer Procedure

The USB VIP module's primary function is to move USB information that is represented by transaction objects. Data movement is specified in the USB specification.

The data object that is created depends on the top transactor layer of the stack:

- Transfer data objects are placed in Protocol transactor.
- Packet data objects are place in Link transactor.
- Data objects are placed in Physical transactors.

The following describes the process of executing USB data exchange:

1. Create a new data object using new_data

   If an existing data object is similar to a new one you want to build, you may use the copy_data command to make a copy.
2. Fill in (or change) the properties representing host-driven signals using set_data_prop
3. Queue the transfer for sending using apply_data for temporary data objects to maintain access to the data object over time.
4. Watch the transfer progress via the callback supported by cmd_callback_wait_for.

To create a new transfer object and return a handle for that object, use the new_data command:

Example: The following commands creates a transfer object and increments the object counter. The command returns the handle named *transfer_handle*.

```
usb_host.new_data(is_valid, transfer_handle, "svt_usb_transfer")

xfer_cnt = xfer_cnt + 1
```

To assign properties to the transfer object, use the set_data_prop command.

Example: The following commands assigns properties to the previously created transfer object. Properties assigned to the object includes a sequence number (xfer_cnt) and payload).

```
usb_host.set_data_prop(is_valid, transfer_handle,"data_id", xfer_cnt, 1)
usb_host.set_data_prop(is_valid, transfer_handle,"device_address", 0, 0)
usb_host.set_data_prop(is_valid, transfer_handle,"endpoint_number", 1, 0)
usb_host.set_data_prop(is_valid, transfer_handle,"xfer_type",
`SVT_USB_BULK_OUT_TRANSFER, 0)
usb_host.set_data_prop(is_valid, transfer_handle,"payload_bytes_remaining", 2048, 0)
usb_host.set_data_prop(is_valid, transfer_handle,"payload_intended_byte_count", 2048, 0)
usb_host.set_data_prop(is_valid, transfer_handle,"payload_start_ix", 0, 0)
usb_host.set_data_prop(is_valid, transfer_handle,"payload_end_ix", 2048, 0)
usb_host.set_data_prop(is_valid, transfer_handle,"payload.data_generation_algorithm",
`SVT_USB_TWO_SEED_BASED_ALGORITHM, 0)
usb_host.set_data_prop(is_valid, transfer_handle,"payload.seed_0", 0, 0)
usb_host.set_data_prop(is_valid, transfer_handle,"payload.seed_1", 1, 0)
usb_host.set_data_prop(is_valid, transfer_handle,"payload.byte_count", 2048, 0)
```

To execute the transfer, use the new_data command.

Example: The following command executes the transfer:

```
usb_host.apply_data(is_valid, transfer_handle,"data_id", 1)
```

4.5.2 Transaction Data Objects

Transaction classes define data descriptor objects that represent USB information. The USB VIP defines the following Transaction data classes:

- **Data** (*svt_usb_data*): These objects represent the information required to send one USB data byte. This class includes support for physical layer transformations.
- **Packet** (*svt_usb_packet*): These objects represent USB packet data units that flow between the USB Protocol layer and the USB Link layer. Objects represent either USB SS or USB 2.0 packets.
- **Transaction** (*svt_usb_transaction*): These objects represent USB transaction data units that the Protocol layer processes.

  The testbench creates, randomizes, or sets transfer object attributes to define USB transfers. The Protocol transactor controls USB bus activity using the list of transactions. Alternatively, the testbench can leave the list of transaction objects empty and the protocol transactor will create the transactions needed to implement the transfer.

  The same transfer data object is used by the VIP USB Protocol to receive inbound transactions. The testbench receives these transactions through callbacks and notifications issued by the VIP USB Protocol. Remotely initiated transfers are also provided to the testbench.
- **Transfer** (*svt_usb_transfer*): These objects represent USB transfer data units that flow between the USB Protocol layer and the testbench.

4.5.3 Service Data Objects

Service classes define data descriptor objects that represent USB service commands. The USB VIP defines the following Service classes:

4.5.3.1 Physical Service Objects

These objects (*svt_usb_physical_service)* represent USB physical service commands.

The Physical Transactor supports the following Physical Service commands.

- Reset – Instructs the super-speed physical layer to perform a reset.
- VBUS – Instructs the super-speed physical layer to enable or disable VBUS.
- Scrambling – Instructs the super-speed physical layer to enable or disable scrambling.
- Termination – Instructs the super-speed physical layer to enable or disable termination.
- Electrical Idle – Instructs the super-speed physical layer to enable or disable transmit of electrical idle.
- Emphasis – Instructs the super-speed physical layer to enable or disable transmitter de-emphasis (used by compliance pattern CP5 and CP7).
- Ones-Zeros – Instructs the super-speed physical layer to enable or disable transmitter compliance patterns CP5 and CP7.
- Low Frequency Periodic Signal Transmission – Instructs the super-speed physical layer to send LFPS, likely as a result of hot or warm reset.
- Receiver Detect – Instructs the super-speed physical layer enter into receiver detect mode.
- Loopback – Instructs the super-speed physical layer to enter or exit loopback mode.

- Polarity Inversion – Instructs the super-speed physical layer to invert the polarity of input signals. This command is issued when the super-speed receiver expects a D10.2 data object and receives a D21.5.
- Power State Change – Instructs the physical transactor to change the current power state according to protocol and the current LTSSM state.

4.5.3.2 Link Service Objects

These objects (*svt_usb_link_service*) represent USB link service commands requested by the Protocol transactor of the Link transactor.

The Link Service object supports the following commands:

- Super-speed link commands
  - POWER_ON_RESET – Initiates a super-speed LTSSM Power-On Reset from the link service channel.
  - HOT_RESET – Initiates a super-speed LTSSM Hot Reset from the link service channel.
  - WARM_RESET – Initiates a super-speed LTSSM Warm Reset from the link service channel.
  - STATE_CHANGE – Triggers an immediate LTSSM state change.

    Requires setting the power_state_request attribute. Setting power_sub_state_request attribute is optional.
  - LOOPBACK – Places the LTSSM in Loopback mode from the link service channel.

    Requires setting "is_loopback_slave" attribute.
  - U2_TIMEOUT – Updates the super-speed U2 Inactivity time-out value in shared status from the link service channel.

    Requires setting the eight-bit "u2_inactivity_timeout" attribute.
  - FORCE_LINKPM – Updates force_linkpm_accept_status value in shared status from the link service channel.

    Requires setting the bit "force_linkpm_accept_status" attribute.
- USB 2.0 commands
  - PORT RESET – Instructs the host model to start driving SE0 on the bus.
  - PORT_POWER_OFF – Sends the host state machine into port power off state emulating a condition such as termination of source power (e.g., external power removed)
  - PORT_DISCONNECT – Causes the host to mimic a high speed disconnect.
  - STOP_PACKET_IN_CHAN_PROCESSING – Sends the transactor into suspend state.

4.5.3.3 Protocol Service Objects

These objects (*svt_usb_protocol_service*) represent protocol service commands that flow between the Protocol layer and the testbench. Commands that Protocol Service objects support include:

- LMP Transactions – Link Management Packet (LMP) transactions manage USB links.
- LPM Transactions – Link Power Management (LPM) transactions manage the USB 2.0 link power state. The Host Protocol transactor receives Protocol Service object containing power management transaction properties from the testbench.

- ***SOF Commands*** – Start Of Frame (SOF) commands allow the testbench to control the automatic production of SOF packets by the host. Commands include turning SOF packets on and off, and setting and getting the current SOF frame number.

The following commands enables SOF transmission:

```
`GET_DATA_PROP_W_CHECK("usb_host", `SVT_CMD_NULL_HANDLE,
"shared_status.link_usb_20_state", link_usb_20_state_val, 0, 1, `FATAL_SEV)
      while (link_usb_20_state_val !== `SVT_USB_ENABLED)
begin
        test_top.usb_host.notify_wait_for(is_valid,
"shared_status.NOTIFY_LINK20_STATE_CHANGE");
        check_for_1(is_valid, "host shared_status.NOTIFY_LINK20_STATE_CHANGE",
`ERROR_SEV);
       `GET_DATA_PROP_W_CHECK("usb_host", `SVT_CMD_NULL_HANDLE,
"shared_status.link_usb_20_state", link_usb_20_state_val, 0, 1, `FATAL_SEV)
end
```

//Create a Host VIP protocol service to start SOF

```
      test_top.usb_host.new_data(is_valid, service_handle, "svt_usb_protocol_service");
      check_for_1(is_valid, "host new_data svt_usb_protocol_service", `FATAL_SEV);
```

// This 'if' is just for completeness. If the `new_data()` call was bad, it

// would cause us to bypass the property settings

```
      if (is_valid)
begin
```

// Display the (default) new transfer

```
$display("  TB @%0d %m - new_data() returned 'service_handle' = %0d", $time,
service_handle);
```

// Set the properties to start the ITPs

```
$display("  TB @%0d %m - Setting service properties...", $time);
         `SET_DATA_PROP_W_CHECK("usb_host", service_handle, "service_type",
`SVT_USB_PROTOCOL_SERVICE_SOF, 0, 1, `FATAL_SEV)
         `SET_DATA_PROP_W_CHECK("usb_host", service_handle, "protocol_20_command_type",
`SVT_USB_20_SOF_ON, 0, 1, `FATAL_SEV)
```

```
    //usb_host.display_data(is_valid, service_handle, " Service 1: ");
```

// Program the model to execute the service request

// (`apply_data` is called only once for a service request)

```
test_top.usb_host.apply_data(is_valid, service_handle);
check_for_1(is_valid, "host apply_data service 1", `FATAL_SEV);
end //end is_valid
```

4.5.4 Other Data Descriptor Objects

This section lists and describes additional data descriptor objects. These objects cannot be submitted; they are read-only.

4.5.4.1 svt_usb_detected_object

This class represents objects detected by the class svt_usb_object_detect. When the svt_usb_object_detect class detects an object, it constructs the appropriate object and generates a corresponding notify with the new object as the data associated with the notification.

4.5.4.2 svt_usb_link_command

This class represents a USB link command

4.5.4.3 svt_usb_symbol_set

This class represents objects detected by svt_usb_object_detect that do not have their own object (such as packet and link command) and are represented by an array of symbols (svt_usb_data).

4.6 VIP Elements

Follow are various VIP Elements:

- Notifications. Notifications are messages that are used strictly for testbench notification. These messages do not print anything out, however they provide full WatchPoint support, including data arguments.
- Data Exceptions. The exceptions are errors that may be introduced into transactions, for the purpose of testing how the connected port of a DUT responds.
- Link Command Exceptions. The exceptions are errors that may be introduced into transaction, for the purpose of testing how the connected port of a DUT response.
- Packet Exceptions. For a packet to be transmitted the class represents errors that are introduced into transactions, for the purpose of testing how a DUT responds. For a packet that is received the class represents the ERROR seen on the bus.
- Symbol Set Exceptions. For a symbol_set to be transmitted the class represents errors that are introduced into transactions, for the purpose of testing how a DUT responds.
- Transaction Exceptions. The exceptions are errors that may be introduced into transaction, for the purpose of testing how the Transfer Exceptions
- Transfer Exceptions. The exceptions are errors that may be introduced into transaction, for the purpose of testing how the DUT responds.

4.6.1 Status Data

The following parameters provide status on data descriptor objects.

4.6.1.1 Subenvironment Objects

The subenvironment uses the following objects:

- **USB Status and Report Objects**
  - ***remote_shared_status*** *(svt_usb_status)*: Shared notify object used to convey remote events and states between transactors
  - ***shared_status*** *(svt_usb_status)*: Shared notify object used to convey events and states between transactors

4.6.1.2 USB Status

This class provides a common location for status information coming from the different transactors in the USB transactor stack. The data members represent the current status as defined by the transactor implementing and updating the status information. Each transactor is responsible for a subset of these data members. Transactors keep the data members they are responsible up to date.

Use of this class facilitates status sharing between the transactors implementing the USB protocol stack.

4.6.1.3 Device Status

This lass contains status information regarding a 'USB device' either being modeled by or communicating with the USB VIP.

4.6.1.4 Host Status

This class contains status information regarding a 'USB device' either being modeled by or communicating with the USB VIP.

4.6.1.5 Endpoint Status

This vmm_data class contains information regarding a 'USB endpoint'. In the USB protocol transactor there will be on 'sub-transactor' for each endpoint defined in the svt_usb_configuration object supplied to the USB VIP. This class makes available the endpoint specific status information.

4.6.1.6 Ustream Resource Status

This vmm_data class contains information regarding a 'USB SS stream resource'. In the USB protocol transactor there will be on 'sub-transactor' for each instance of svt_usb_ustream_resource_status if the endpoint supports streams. That sub-transactor needs to know which USB stream ids it is to support. This class provides that information.

4.7 Log Output

The USB VIP modules produce log (transcript) messages that may be useful for monitoring or debugging the progress of simulations. The modules also may produce error and warning messages.

The *format* of all messages (including errors and warnings) produced by the module, is controlled by the module and is not user-modifiable. Messages use a 2-line format, as follows:

- The first line indicates the simulation timestamp, the type and severity of message, the module type (possibly an internal sub-module or data object) that produced the message, and (if applicable) the instance name of the module.
- The second line is the actual message. Its contents depend on the type of message and the reason it was generated.

In addition to messages, the modules can display data object content if the level of verbosity is increased by the user's testbench.

Log messages as implemented by the USB VIP have three primary attributes: type, severity, and handling. These attributes, and the values relevant to them are described below:

type - Specifies the type of log message. Legal values are:

FAILURE_TYP	Used for operations intended to affect Warning, Error, or Fatal Error messages.
NOTE_TYP	Used for operations intended to affect general, non-filtered log messages.
DEBUG_TYP	Used for operations intended to affect verbosity-filtered log messages; for debug usage.
DEFAULT_TYP	Each message severity (as described below, is associated with a default message type). This value represents that type in a context-dependant situation (such as when severity has been user-specified in an operation).
ALL_TYPES	Used for operations intended to affect all message types.

severity - Specifies the severity or verbosity level of log messages. Legal values are:

FATAL_SEV	Used to represent messages that are associated with fatal errors.
ERROR_SEV	Used to represent messages that are associated with non-fatal errors.
WARNING_SEV	Used to represent messages that are associated with warnings.
NORMAL_SEV	Used to represent messages that are always logged (not filtered) for information purposes only.
TRACE_SEV	Used to represent messages that contain basic tracking information, for coarse debugging.
DEBUG_SEV	Used to represent messages that contain detailed information, for fine debugging.
VERBOSE_SEV	Used to represent messages that contain deeply detailed information to support debugging of the USB VIP.
DEFAULT_SEV	Each message type has an associated default severity. This value is used in a context-dependant situation to represent that severity (such as when the type has been user-specified in an operation).
ALL_SEVERITIES	Used in operations intended to affect messages of all severity values.

handling - Specifies how a message should be handled by the simulator (such as what the end effect associated with the message being displayed is). Legal values are:

ABORT	Indicates that the simulator should abort the simulation (and exit) after the message is logged.
COUNT_AS_ERROR	Indicates that the overall error count (as stored by the VIP) should be incremented when the message is logged.
STOP	Indicates that the simulator should halt, but not exit, when the message is logged.
DEBUGGER	Indicates that the simulator should halt and start its interactive debugger (if possible) when the message is logged.
DUMP	Indicates that the simulator should dump simulation values.
CONTINUE	Indicates that the simulator should continue without taking further action when the message is logged.

The attribute values described above are not available directly as symbolic constants. However, the log_msg_val command allows you to get an integer value that is equivalent to a specified message attribute/value pair.

4.7.1 Errors and Warnings

4.7.1.1 Fatal Errors

A fatal error is reported by the module if it encounters a situation that will not allow it to continue to operate. In such a case, the error is reported and the simulation is terminated.

A good example of a fatal error is the case in which a user has attempted to apply an invalid configuration to the module. In such a case, the resulting error report would appear as follows:

```
@0 ns *FATAL* [Failure] on usb_svt (mstr) :
usb_svt - Config provided to constructor is not valid!
```

Because a fatal error terminates the simulation abruptly, some operations that are ongoing at the time of the error may not complete correctly, resulting in invalid messages and/or behavior. The user should always check and correct fatal errors first, before attempting to debug unexpected behavior in this circumstance.

4.7.1.2 Errors

This type of error message indicates that some incorrect data or behavior was encountered by the module.

Unlike fatal errors, standard errors do not automatically cause the simulation to terminate (although they may, if enough of them occur). An error may or may not allow the module to continue operating normally, depending on the circumstances.

The user should investigate the root cause of all error messages (starting with the first one that occurs in the transcript). Error messages produced by the VIP modules may be located by searching the transcript for *ERROR*.

4.7.1.3 Warnings

Warnings are produced to alert the user to situations encountered by the module that are unexpected but are not necessarily incorrect. Warning messages do not cause the simulation to terminate.

The user should asses the cause and implications of warning messages on a case-by-case basis. Warning messages produced by the USB VIP modules may be located by searching the transcript for *WARNING*.

4.7.2 Tracking/Debugging Messages

The USB VIP modules may be set to output (to log / transcript) several levels of tracking and debugging messages. By default the modules do not produce any log output, except for error and warning messages.

The levels of message verbosity that may be chosen for logging are designated as *severity values*. Use the log_get_verbosity and log_set_verbosity commands to find out and modify the levels of messages that are being logged by a model module.

4.8 Implementing Functional Coverage

USB3.0 Verification IP coverage uses following mechanisms

- Pattern sequences between Initiator (USB Host) and Responder (USB Device) entities.
  - Super speed protocol layer, link layer and USB20 protocol layer coverage is based on pattern sequences
- Signaling on the bus between USB Host and device.
  - USB2.0 link layer coverage based on signaling on the bus.

4.8.1 Default Functional Coverage

The VC VIP for USB supports protocol and link layer functional coverage for Super Speed (SS) and USB20 modes of operation.

Table 4-6 lists the default functional coverage features provided with VC VIP for USB.

Table 4-6 Default Functional Coverage Features

Layer/Coverage	SuperSpeed	USB20
Protocol	• Bulk transfers • Interrupt transfers • Control transfers • ISOC transfers	• Bulk Transfers • Interrupt transfers • Control transfers • ISOC transfers • Split transfers
Link	Link Management and Flow control • Header packet • Link Command • Low Power Management • DPP • Link Initialization Link Training and Status State Machine	• Connect/Disconnect • Reset • Suspend/Resume • Link Power Management

The functional coverage data collected is instance based. This means that each instance of the VIP gathers and maintains its own functional coverage information.

4.8.2 Enabling Functional Coverage

Attention Functional coverage is supported with VCS and MTI simulators only.

For VCS and MTI

The default functional coverage can be enabled by setting the attributes in the host or device sub-environment configuration. The attributes are:

- enable_prot_cov - enables all protocol layer cover groups
- enable_link_cov - enables all link layer cover groups

To enable the coverage flags, edit the example test file (for example, ts.usb_ss_serial.v) that is associated with the example to include the following code:

```
//Enabling Coverage
`SET_DATA_PROP_W_CHECK("usb_host",cfg_handle,"enable_prot_cov",1, 0, 1, `FATAL_SEV)
`SET_DATA_PROP_W_CHECK("usb_host",cfg_handle,"enable_link_cov",1, 0, 1, `FATAL_SEV)

//Enabling Coverage
`SET_DATA_PROP_W_CHECK("usb_dev",cfg_handle,"enable_prot_cov",1, 0, 1, `FATAL_SEV)
`SET_DATA_PROP_W_CHECK("usb_dev",cfg_handle,"enable_link_cov",1, 0, 1, `FATAL_SEV)
```

Once the coverage is enabled, all coverage information is sent to the coverage database.

If you are using MTI simulators, you also need to set coverage_db_name as described in the following section.

For MTI

After enabling the default functional coverage (as mentioned in the previous section), you must set coverage_db_name in the top.v file that is associated with the example.

For example, to generate a coverage database for ts.basic_ss_serial.v example, edit the corresponding top file (that is top.usb_ss_serial.v) as follows:

```
initial begin
  $set_coverage_db_name("usb_ss_coverage.vcdb");
end
```

4.8.3 Using the High-Level Verification Plans

High-level verification plans (HVPs) are provided for typical USB verification topologies. The top-level verification plans can be found after installation at $DESIGNWARE_HOME/vip/svt/usb_svt/<version>/doc/VerificationPlans directory. These plans have the following naming convention:

```
svt_<suite>_<operation_mode>_dut_<protocol_mode>_toplevel_fc_plan
```

In addition, there are several sub-plans. Each sub-plan has the following naming convention:

```
svt_<suite>_<vip_layer>_<protocol_mode>_<transfer_type>
```

For information on back-annotating the HVP with VMMPlanner, see the README provided with the verification plans in the VerificationPlans directory.

4.9 Executing Aligned Transfers

The USB specification defines transfers (IN or OUT) as 'aligned', when their total payload is equal to an integral multiple of the maximum packet size as configured in the particular endpoint configuration. The specification also allows two options for aligned transfers to end:

- End with a zero-length data packet (default behavior), or
- End without a zero-length when both host and device have such an expectation for particular transfers to certain endpoints

The following VIP attributes are key to executing an aligned transfer:

- Endpoint configuration may or may not allow aligned transfers without zero length DP
- Transfer class object attribute to execute a particular transfer with or without (assuming targeted endpoint configuration allows) zero length data packet

The usage of VIP with different combinations of the above attributes are described in the following sections.

4.9.1 VIP Acting as a Host

Table 4-7 describes the VIP behavior when it is acting as a host.

Table 4-7 Behavior of VIP Acting as a Host for Different Endpoint Configurations and Transfers

Transfer	Endpoint Configuration allow_aligned_transfers_without_zero_length=0	Endpoint Configuration allow_aligned_transfers_without_zero_length=1
aligned_transfer_with_zero_length=0	VIP performs a check, and if txfer::payload_intended_count is aligned • OUT: VIP reports an error that endpoint configuration and transfer attributes are inconsistent. VIP then issues one or more OUT transactions to transmit payload_count bytes, and does not end with a 0-length data packet. • IN: VIP reports an error that endpoint configuration and transfer attributes are inconsistent. VIP then continues to issue IN transactions until it receives all payload_intended_count or <max_packet_size> length payload is received (whichever occurs first), hence does not require a 0-length data packet.	VIP performs a check, and if txfer::payload_intended_count is aligned • OUT: VIP issues one or more OUT transactions to transmit payload_count bytes, and does not end with a 0-length data packet. • IN: VIP continues to issue IN transactions until it receives all payload_intended_count or <max_packet_size> length payload is received (whichever occurs first), hence does not require a 0-length data packet.
	VIP performs a check, and if txfer::payload_intended_count is not aligned, it reports the following constraint solver error: `txfer:: aligned_transfer_ends_with_zero_length must be set to 1 when payload is not aligned.`	
aligned_transfer_with_zero_length=1	• OUT: VIP issues one or more OUT transactions to transmit payload_intended_byte_count number of bytes, and always ends with a <max_packet_size> (0 or short) length data packet. • IN: VIP continues to issue IN transactions until a <max_packet_size> (0 or short) length payload is received.	

4.9.2 VIP Acting as a Device

Table 4-8 describes the VIP behavior when it is acting as a host

Table 4-8 Behavior of VIP Acting as a Device for Different Endpoint Configurations and Transfers

Transfer	Endpoint Configuration allow_aligned_transfers_without_zero_length=0	Endpoint Configuration allow_aligned_transfers_without_zero_length=1
aligned_transfer_with_zero_length=0	VIP performs a check, and if txfer::payload_intended_count is aligned • OUT: VIP reports an error that endpoint configuration and transfer attributes are inconsistent. VIP then continues to receive data packets until it receives all payload_intended_count or <max_packet_size> length payload is received (whichever occurs first), hence VIP does not require a 0-length data packet to end transfer. • IN: VIP reports an error that endpoint configuration and transfer attributes are inconsistent. VIP then continues to provide data packets to IN transactions until it transmits all of payload_intended_count, and does not transmit a 0-length data packet.	VIP performs a check, and if txfer::payload_intended_count is aligned • OUT: VIP continues to receive data packets until it receives all payload_intended_count or <max_packet_size> length payload is received (whichever occurs first), hence VIP does not require a 0-length data packet to end transfer. • IN: VIP continues to provide data packets to IN transactions until it transmits all of payload_intended_count, and does not transmit a 0-length data packet.
	VIP performs a check, and if txfer::payload_intended_count is not aligned, it reports the following constraint solver error: `txfer:: aligned_transfer_ends_with_zero_length must be set to 1 when payload is not aligned.`	
aligned_transfer_with_zero_length=1	• OUT: VIP continues to receive data packets until a data packet with <max_packet_size> (0 or short) length payload is received. • IN: VIP continues to provide data packets to IN transactions until it transmits all of payload_intended_count and always ends with a < max_packet_size> (0 or short) length data packet.	

4.10 SuperSpeed Serial LTSSM Flow (SS.Disabled to U0)

This section describes VIP's usage for starting with SS.disabled and progressing to U0 link state, with SuperSpeed serial interface. VIP's timers that are directly relevant in this usage, are stated in 'italics'.

Whenever VIP's operation or usage is different for down-stream facing port or up-stream facing port configuration, VIP's operating mode is explicitly mentioned. Otherwise, VIP operation is the same for either facing port.

LTSSM state is SS.Disabled

VIP's initial LTSSM state is SS.Disabled (based on VIP's configuration parameter, usb_ss_initial_ltssm_state = svt_usb_types::SS_DISABLED)

If VIP's interface is a down-stream facing port (when VIP is configured as Host), VIP does the following:

1. Drives vbus output signal to 1,
2. Drives vip_rx_termination output to 0, and
3. Waits for a Link Service command (Directed or PowerOn Reset) to change state to Rx.Detect.Reset

If VIP's interface is an up-stream facing port (when VIP is configured as Device), VIP does the following:

1. After detecting vbus input=1, VIP drives vip_rx_termination output to 1, and
2. Transitions to Rx.Detect.Reset state.

LTSSM state is Rx.Detect.Reset

If it is not a warm reset, then VIP immediately proceeds to Rx.Detect.Active state.

LTSSM state is Rx.Detect.Active

VIP initiates detection of far-end receiver termination by driving sstxp=1 and sstxm=0, and VIP keeps track of number of attempts to detect receiver termination.

After *receiver_detect_time* duration, VIP stops driving sstxp or sstxm and checks the value of the dut_ss_termination input signal. VIP then checks for the following conditions and then performs the corresponding actions.

IF...	THEN...
dut_ss_termination == 1	VIP reports that far-end termination is present and proceeds to Polling.LFPS.
(number of attempts == rx_detect_termination_detect_count)	VIP transitions to SS.disabled

If none of the above conditions are true, VIP transitions to Rx.Detect.Quiet, and waits for rx_detect_quiet_timeout duration and then reverts to initiating detection of far-end receiver termination.

- VIP's interface is constructed such that a sampled value of 1 is seen on the dut_ss_termination signal by default, unless the signal is specifically driven to 0.
- If VBUS input signal of VIP (configured as upstream facing port) is 0, far-end receiver detection is not successful.

LTSSM state is Polling.LFPS

After *p2_to_p0_transition_time* duration, VIP's PHY transitions from P2 to P0 power state. VIP starts LFPS transmission based on following parameters:

- polling_lfps_burst_time
- polling_lfps_repeat_time
- tx_lfps_duty_cycle
- tx_lfps_period

VIP expects to receive LFPS pulses that are in following range:

- Received LFPS burst time must be >=polling_lfps_burst_min and <=polling_lfps_burst_max
- Received LFPS repeat time must be >=polling_lfps_repeat_min and <=polling_lfps_repeat_max
- Received LFPS period must be >= tperiod_min and <= tperiod_max

As valid LFPS pulses are received, VIP keeps track of number of valid Polling.LFPS pulses received.

When received count == *polling_lfps_received_count*, VIP transmits *polling_lfps_sent_after_received_count* more pulses, and proceeds to Polling.RxEQ state.

Until received count is not equal to *polling_lfps_received_count*, VIP continues to drive LFPS pulses until the *polling_lfps_timeout* expires, and then transitions to SS.disabled state.

LTSSM state is Polling.RxEQ

If *ltssm_skip_polling_rxeq* ==1 then VIP immediately proceeds to Polling.Active state. If *ltssm_skip_polling_rxeq*!=1, then VIP drives *polling_rxeq_tseq_count* number of TSEQ symbols and proceeds to Polling.Active state.

LTSSM state is Polling.Active

VIP transitions to Polling.Configuration state after receiving *polling_active_received_ts_count* number of consecutive and identical TS1 or TS2 ordered sets. VIP transmits TS1 symbols until either VIP receives *polling_active_received_ts_count* number of consecutive and identical TS1 or TS2 symbols or *polling_active_timeout* timer expires, whichever happens first.

If expected number of TS1s or TS2s are not received before *polling_active_timeout* expires, VIP transitions to SS.disabled state.

LTSSM state is Polling.Configuration

VIP transitions to Polling.Idle state when both the following conditions are met:

- *polling_configuration_received_ts2_count* number of consecutive and identical TS2 ordered sets are received.
- *polling_configuration_sent_ts2_count* number of TS2 ordered sets are sent after receiving the first of the *polling_configuration_received_ts2_count* consecutive and identical TS2 ordered sets

If the expected number of TS2s are not received before *polling_configuration_timeout* expires, VIP transitions to SS.disabled state.

LTSSM state is Polling.Idle

VIP transitions to U0 when both the following conditions are met:

- *polling_idle_received_idle_count* consecutive Idle Symbols are received.
- *polling_idle_sent_idle_count* Idle Symbols are sent after receiving one Idle Symbol

If expected number of IDLEs are not received before *polling_idle_timeout* expires, VIP transitions to SS.disabled state.

4.11 USB 2.0 OTG Support

Attention This is a Beta feature and is not LCA ready.

This section provides information about how the USB VIP supports different functionalities necessary when testing an On-The-Go (OTG) DUT.

This section discusses the following topics:

- "OTG Signals"
- SRP Signals
- Role Swapping Using the HNP Protocol

- HNP Polling
- Attach Detection Protocol

4.11.1 OTG Signals

To support OTG 2.0, all the VIP modules include the following different types of signals:

- "Link Layer Signals"
- Serial Interface Signals

4.11.1.1 Link Layer Signals

The VIP uses the OTG signals defined by the UTMI+ specification as the basis for modeling OTG behavior across a link-level interface. These signals support device configuration, ID, and modeling of SRP signaling.

When modeling a USB 2.0 or SS serial-level signal interface, these signals reflect the link-level status within the VIP. When modeling a USB 2.0 link-level signal interface, these signals are used as the actual OTG signal interface.

Note Because the UTMI+ specification is based on OTG 1.3, not all the signals described below may be active when modeling OTG 2.0 behavior (for example, VBUS pulsing is no longer supported under OTG 2.0, therefore the chrgvbus and dischrgvbus signals will be inactive).

Table 4-9 lists and describes the link layer signals.

Table 4-9 OTG Link Layer Signals

Signal	Direction	Polarity	Default	Size	Description and Values
VbusValid	Output	High	Tri-state	1	VBUS Valid signal • 1'b0: VBUS voltage < VBUS Valid threshold • 1'b1: VBUS voltage >= VBUS Valid threshold
AValid	Output	High	Tri-state	1	Session for A peripheral is Valid signal (DUT is A-device and device VIP is B-device) • 1'b0: VBUS voltage < VBUS A-device Session Valid threshold • 1'b1: VBUS voltage >= VBUS A-device Session Valid threshold
BValid	Output	High	Tri-state	1	Session for B peripheral is Valid signal (DUT is B-device and device VIP is A-device) • 1'b0: VBUS voltage < VBUS B-device Session Valid threshold • 1'b1: VBUS voltage >= VBUS B-device Session Valid threshold
SessEnd	Output	High	Tri-state	1	Session End signal • 1'b0: VBUS voltage >= VBUS Session End threshold • 1'b1: VBUS voltage is not above VBUS Session End threshold
DrvVbus	Input	High	-	1	VBUS Drive signal • 1'b0: VBUS is not driven to 5V • 1'b1: VBUS is driven to 5V

Table 4-9 OTG Link Layer Signals

Signal	Direction	Polarity	Default	Size	Description and Values
ChrgVbus	Input	High	-	1	VBUS Charging signal • 1'b0: VBUS is not charged up to A session valid voltage • 1'b1: VBUS is charged up to A session valid voltage
DischrgVbus	Input	High	-	1	VBUS Discharging signal • 1'b0: B-device is not discharging the VBUS • 1'b1: B-device is discharging the VBUS
IdDig	Output	-	1	1	ID Status signal (drives the ID of the DUT); VIP asserts iddig when testbench issues start command • 1'b0: Mini-A plug is connected • 1'b1: Mini-B plug is connected
HostDisconnect	Output	High	1	1	Indicates if a peripheral is connected. Valid only if dppulldown and dmpulldown are 1'b1 • 1'b1: - No peripheral is connected • 1'b0: - A peripheral is connected
IdPullup	Input	-	-	1	Enables pull-up resistor on the ID line and sampling of the signal level • 1'b0: Disables sampling of the ID line • 1'b1: Enables sampling of the ID line
DpPulldown	Input	-	-	1	Enables pull-down resistor on the DP line • 1'b0: Pull-down resistor is not connected to DP • 1'b1: Pull-down resistor is connected to DP
DmPulldown	Input	-	-	1	Enables pull-down resistor on the DM line • 1'b0: Pull-down resistor is not connected to DM • 1'b1: Pull-down resistor is connected to DM

4.11.1.2 Serial Interface Signals

VIP-defined signals support modeling OTG behavior across a serial-level interface. These signals provide the ability to model OTG functionality not clearly defined under existing specifications and/or analog functionality outside the normal scope of digital simulation (such as ADP probing and sensing).

When modeling any USB 2.0 or SS signal interface, these signals model and/or reflect the analog state of a serial interface that exists between the VIP and the DUT, regardless of whether or not that interface is real or virtual.

4.11.1.2.1 ADP Signals

Table 4-10 lists and describes the signals that support the modeling of ADP behavior.

These VIP-defined abstract signals model the following fundamental ADP behavior:

- Whether or not a device is attached to the bus, and

- Whether or not that device is performing ADP probing.

Table 4-10 OTG ADP Serial Interface Signals

Signal	Direction	Polarity	Default	Size	Description and Values
vip_attached	Output	High	1	1	VIP is attached • 1'b0: VIP is not attached • 1'b1: VIP is attached
vip_adp_prb	Output	High	0	1	VIP ADP probe is active • 1'b0: VIP is not performing an ADP probe • 1'b1: VIP is performing an ADP probe
dut_attached	Input	High	–	1	DUT is attached • 1'b0: DUT is not attached • 1'b1: DUT is attached
dut_adp_prb	Input	High	–	1	DUT ADP probe is active • 1'b0: DUT is not performing an ADP probe • 1'b1: DUT is performing an ADP probe

4.11.1.2.2 SRP Signals

Table 4-11 lists and describes the signals that support the modeling of ADP behavior. These VIP-defined signals model whether or not a device is detecting a VBUS voltage above the valid OTG session level (VOTG_SESS_VLD).

Table 4-11 SRP Signals

Signal	Direction	Polarity	Default	Size	Description and Values
vip_sess_vld	Output	High	0	1	VIP VBUS level detector - VOTG_SESS_VLD • 1'b0: VIP detected/driven VBUS <= VOTG_SESS_VLD • 1'b1: VIP detected/driven VBUS > VOTG_SESS_VLD
dut_sess_vld	Input	High	0	1	DUT VBUS level detector - VOTG_SESS_VLD • 1'b0: DUT detected/driven VBUS <= VOTG_SESS_VLD • 1'b1: DUT detected/driven VBUS > VOTG_SESS_VLD

4.11.2 Session Request Protocol

Session Request Protocol (SRP) is a mechanism used to converse power. SRP allows an A-device to turn off VBUS when the bus is not in use. It also allows a B-device to request the A-device turn VBUS back on and start a session. A session, defined as the period of time VBUS is powered, ends once VBUS is turned off.

At the link-level, there are two parts to the SRP flow:

- Generating and responding to the SRP request, and
- Controlling the SRP response

The OTG Host and OTG peripheral VIP generate and respond to an SRP request if the srp_supported attribute is set in the svt_usb_subenv_configuration object.

4.11.2.1 SRP Protocol Flow When the VIP is Acting as an A-Device

Figure 4-3 explains the SRP process flow when the VIP is acting as an A-device.

Figure 4-3 SRP Protocol Flow when the VIP is Acting as an A-Device

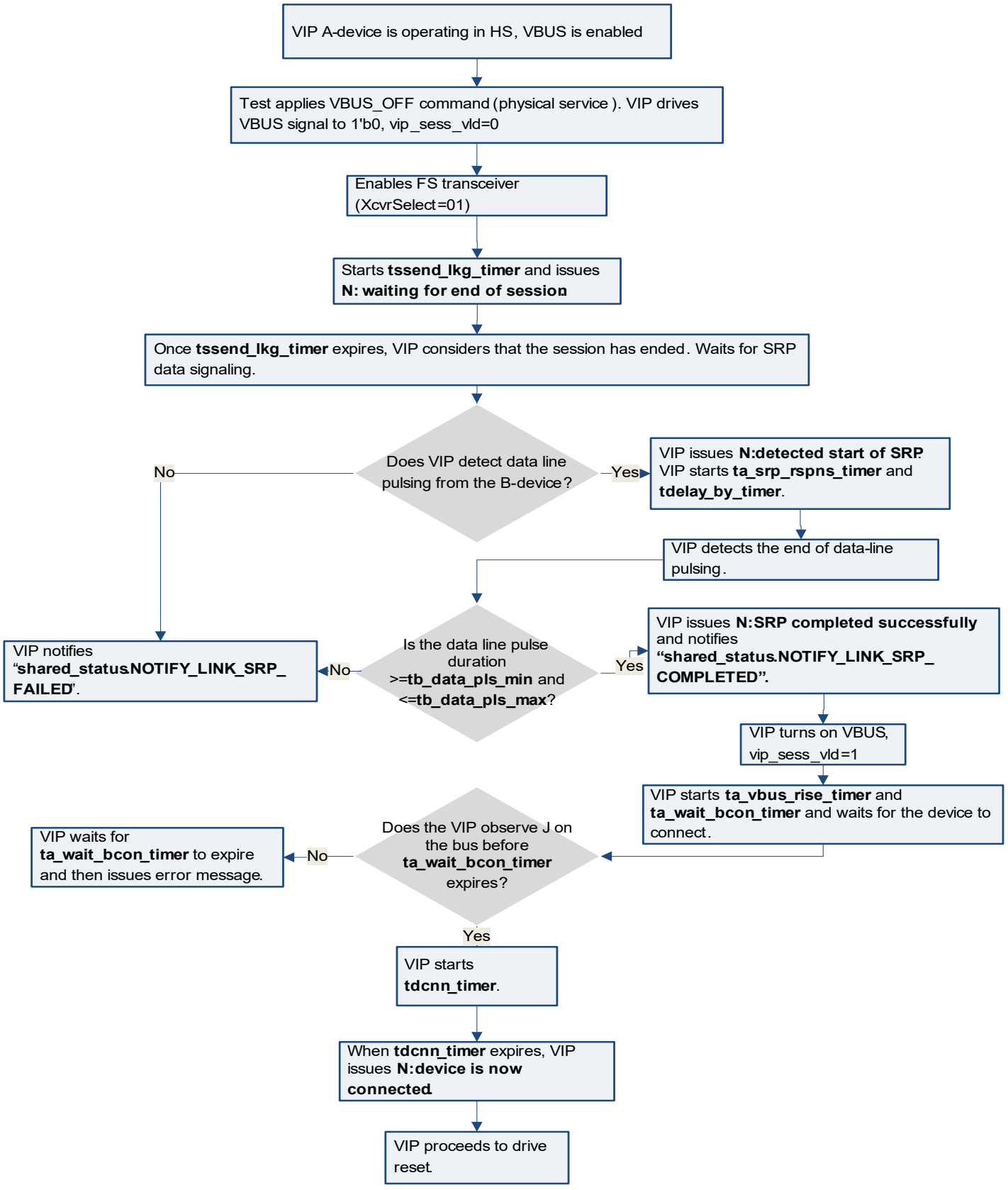


Table 4-12 expands on the abbreviated messages that are used in Figures 4-3 and 4-4.

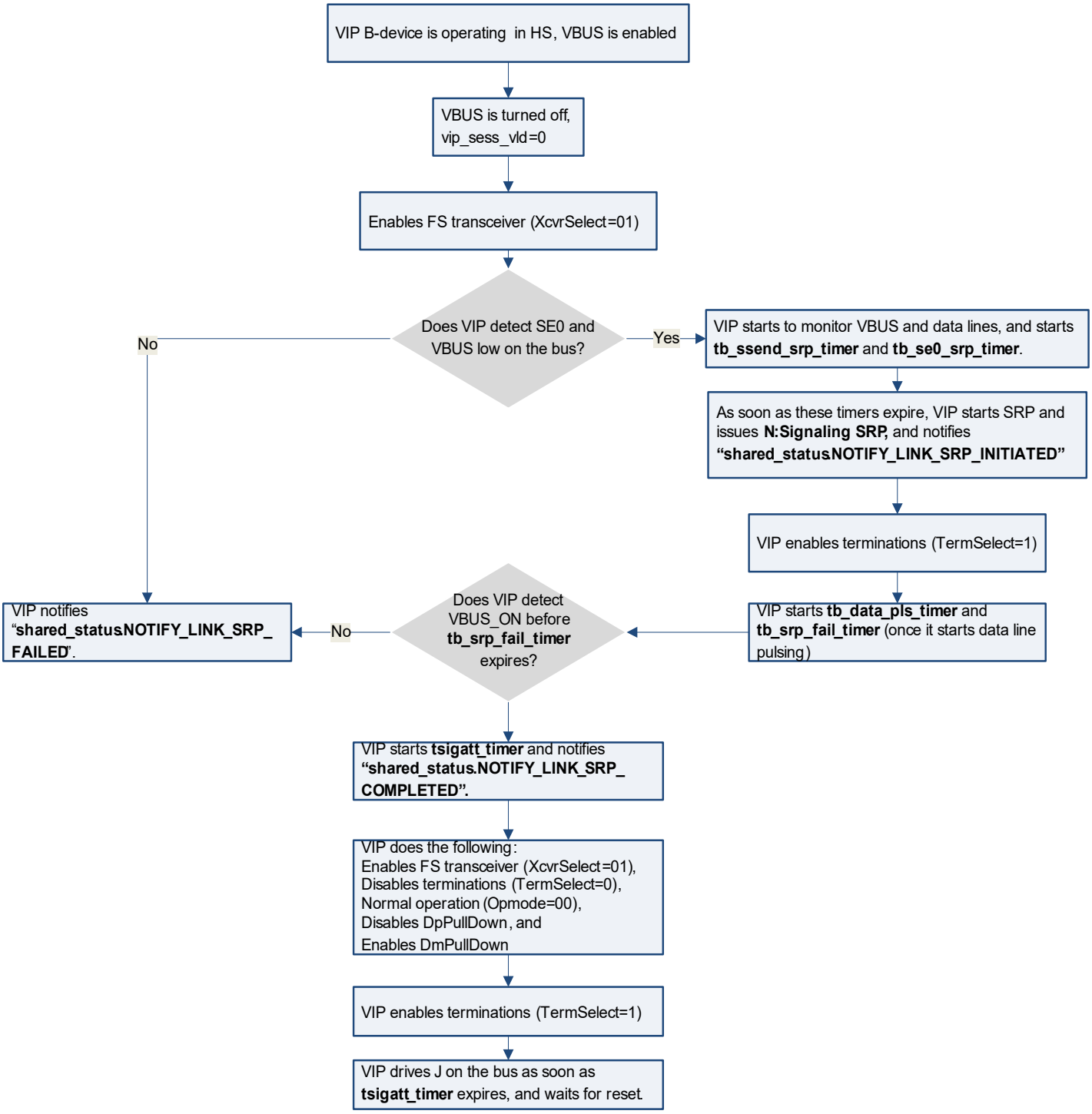
Table 4-12 Legend Explaining the Abbreviated Messages

Abbreviated Messages Used in Figure 4-3	Actual Messages Issued by VIP
N: waiting for end of session	Waiting for the end of a session by monitoring VBUS and data lines.
N: detected start of SRP	A-device detected start of SRP since D+ went high
N: SRP completed successfully	SRP completed successfully since D+ pull-up resistor remained ON for a period within the range specified by TB_DATA_PLS
N: device is now connected	tdcnn timer expired. Device is now connected.
Abbreviated Message Used in Figure 4-4	**Actual Message Issued by VIP**
N: signaling SRP	B-device signaling SRP by generating data line pulsing.

4.11.2.2 SRP Protocol Flow When the VIP is Acting as a B-Device

Figure 4-4 explains the SRP process flow when the VIP is acting as a B-device. The abbreviated message is expanded in detail in Table 4-12.

Figure 4-4 SRP Protocol Flow when the VIP is Acting as a B-Device



4.11.3 Role Swapping Using the HNP Protocol

Host Negotiation Protocol (HNP) is the protocol by which an OTG Host relinquishes the role of USB Host to an OTG Peripheral that is requesting the role. To do this the OTG Host suspends the bus, and then signaling during suspend and resume determines which device has the role of USB Host when the bus comes out of suspend.

At the start of a session, the A-device defaults to having the role of host. During a session, the role of host can be transferred back and forth between the A-device and the B-device any number of times, using HNP.

The acting USB Host may suspend the bus at any time when there is no traffic. HNP is applicable only if an acting OTG Host has detected an acting OTG Peripheral's request to assume the role of USB Host prior to the suspend. The initiation of the role swap first initiated by the OTG device.

Note Role swapping is possible only when an OTG Host is directly connected to an OTG device and only if hnp_capable and hnp_supported attributes are set in their respective configurations.

4.11.3.1 Initializing and Storing Configurations for OTG Roles

To successfully support role swapping, the VIP sub-environment must store the configuration data objects for the initial OTG role as well as the swapped OTG role.

The initial configuration is first stored when the VIP sub-environment is constructed with the handle for the initial configuration (for example, initial_cfg_handle). The swapped configuration is stored with the handle for the swapped configuration (for example, swapped_cfg_handle). In the testbench (if hnp_enable is set to 1), a role swap is then initiated by the testbench through the attempt_otg_role_swap command. The sub-environment also notifies the testbench whether the role swap was successful or not through one of the following notifications, which are a part of the status object:

- NOTIFY_OTG_ROLE_SWAP_SUCCEEDED or
- NOTIFY_OTG_ROLE_SWAP_FAILED

After a successful role swap, the VIP sub-environment and the transactor stack is reconfigured based on the newly swapped configuration. This means that the initial and swapped configurations are retained across role swaps. Changes to the active configuration directly by the user (testbench) are not retained by the subenv in the initial_cfg or swapped_cfg configurations. If a role swap occurs, the configuration used for the new role is the most recent configuration applied for that role.

If the testbench wishes to cause the VIP to start operating in a different configuration when the role is swapped, it must get the handle for the initial configuration using the get_data_prop and set_data_prop methods.

Note All interactions with the configurations (initial_cfg, and swapped_cfg) must be done using the get_data_prop and set_data_prop methods.

4.11.3.2 Role Swapping Process Overview

This section provides a brief overview of the sequence of events:

1. Testbench creates two svt_usb_subenv_configuration instances, one each representing the configuration for the initial OTG role (initial_cfg) and the swapped OTG role (swapped_cfg).
2. Testbench constructs the VIP subenv using the initial_cfg configuration data object that represents its initial role (based on module_type value).
3. Testbench stores the swapped_cfg configuration data object.
4. If the VIP sub-environment initially has the OTG Peripheral role, the testbench requests a role swap by calling the attempt_otg_role_swap command.
   - If the role swap succeeds, the sub-environment reconfigures the transactor stack (and itself) using the swapped_cfg configuration data object (which represents the configuration of the VIP

while in the OTG Host role), and notifies the Testbench through the NOTIFY_OTG_ROLE_SWAP_SUCCEEDED notification.

- ✦ If the role swap fails the sub-environment does not change anything, but notifies the testbench through the NOTIFY_OTG_ROLE_SWAP_FAILED notification.

5. If VIP sub-environment initially has the OTG Host role, and the DUT OTG Device initiates a role swap attempt, the sub-environment notifies the testbench that a role swap attempt has started.
   - ✦ If the role swap succeeds, the sub-environment reconfigures the transactor stack (and itself) using the swapped_cfg configuration data object (which represents the configuration of the VIP while in the OTG Peripheral role), and notifies the testbench through the NOTIFY_OTG_ROLE_SWAP_SUCCEEDED notification.
   - ✦ If the role swap fails the sub-environment does not change anything, but notifies the testbench through the NOTIFY_OTG_ROLE_SWAP_FAILED notification.
6. When a successful role swap back to the initial OTG role occurs, (after step 4 or 5), the sub-environment reconfigures the transactor stack (and itself) using the initial_cfg configuration data object.

4.11.3.3 HNP Polling

HNP polling is a mechanism that allows the OTG device currently acting as a host to determine when the other attached OTG device wants to take the host role. When an OTG host is connected to an OTG device, it polls the device regularly to determine whether it requires a role-swap.

When doing HNP polling, the VIP acting as an A-device executes the GetStatus() control transfers directed at a control endpoint in the OTG device. However, this kind of polling is not really required because the VIP does not modify its behavior based on the information content of the transfers.

Alternate methods can be used to model the possible outcomes of polling. If required, the testbench can implement or detect such polling and synchronize the usage of the alternate mechanisms accordingly.

4.11.3.3.1 HNP Polling When the VIP is Configured as an OTG Host

When the VIP is acting as OTG Host, the testbench must create and send appropriate control transfers to enable and accomplish HNP Polling. These include the control transfers that implement the GetDescriptor(), SetFeature(), and GetStatus() tasks used in HNP polling.

- ❖ When GetDescriptor() is used to access the DUT's OTG descriptor, the value returned for the "HNP Support" attribute (bmAttributes bit D1), should be saved, and the VIP should be dynamically reconfigured to set the equivalent hnp_supported bit in the remote_device_cfg that represents the DUT's configuration to the VIP as Host.
- ❖ When SetFeature() is used to set the b_hnp_enable bit successfully, the testbench can then call the subenv attempt_otg_role_swap() method, which will enable the HNP procedure.
- ❖ When GetStatus() is used to interrogate the value of the DUT's "Host request flag" status bit, if the flag is set a protocol service data object should be created by the testbench and sent to the VIP to initiate a role swap attempt.
- ❖ It is also up to the testbench to control the timing of the GetStatus() control transfers to adhere to the timing requirements specified for HNP Polling.

4.11.3.3.2 HNP Polling When the VIP is Configured as an OTG Device

When the VIP is acting as OTG Peripheral, the testbench must interpret and respond to the following control transfers involved in HNP Polling:

- The value of the b_hnp_supported bit in the VIP's device configuration does not automatically result in the proper bit being set in the values returned for the GetDescriptor() control transfer. It is up to the testbench to interpret the control transfer, and control the return value based on the value of the b_hnp_supported bit in the VIP's device configuration.
- When a SetFeature() control transfer is received who's intent is to set the b_hnp_enable bit in the VIP, the corresponding bit in the VIP's device status is not automatically set. It is up to the testbench to interpret the control transfer and then call the subenv's attempt_otg_role_swap() method to enable the HNP process.
- If the VIP is configured with b_hnp_supported = 1 and an attempt_otg_role_swap() method call is made, it will cause (in the case of a 2.0 connection) the host_request_flag in the VIP's shared device status to become set. However, this does not automatically result in the proper bit being set in the value returned for the GetStatus() control transfer. It is the responsibility of the testbench to interpret the control transfer, and control the return value based on the value of the host_request_flag in the VIP's shared device status.

4.11.3.4 HNP Sequence of Events When VIP is Configured as an A-Device

Figure 4-5 explains the HNP flow when the VIP is configured as an A-device. Figure 4-6 explains the role reversal sequence of events when the A-device reverts back to acting as a host.

Figure 4-5 HNP Flow When VIP is Configured as an A-Device

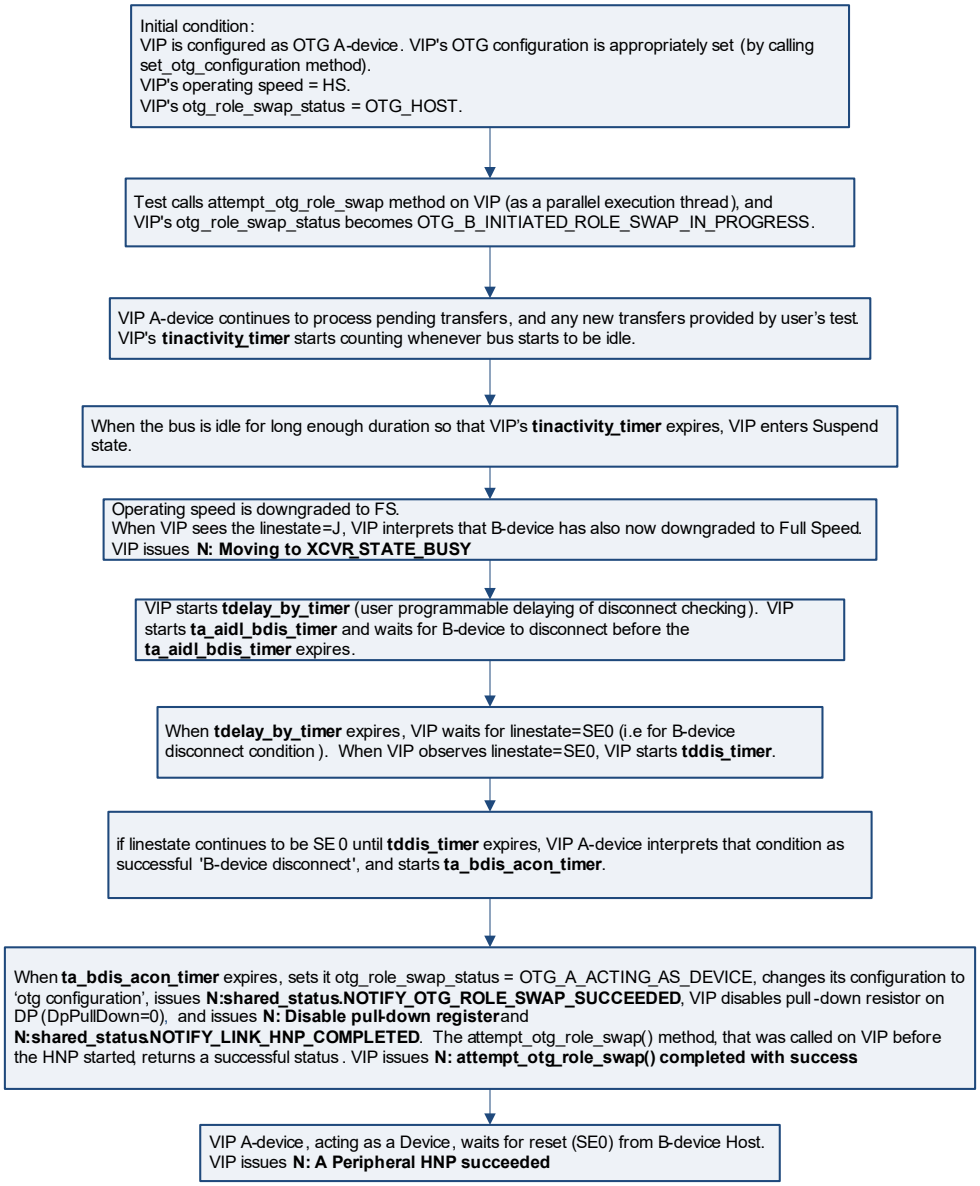


Figure 4-6 Reverse HNP Flow when the VIP A-Device Reverts Back to Acting as a Host

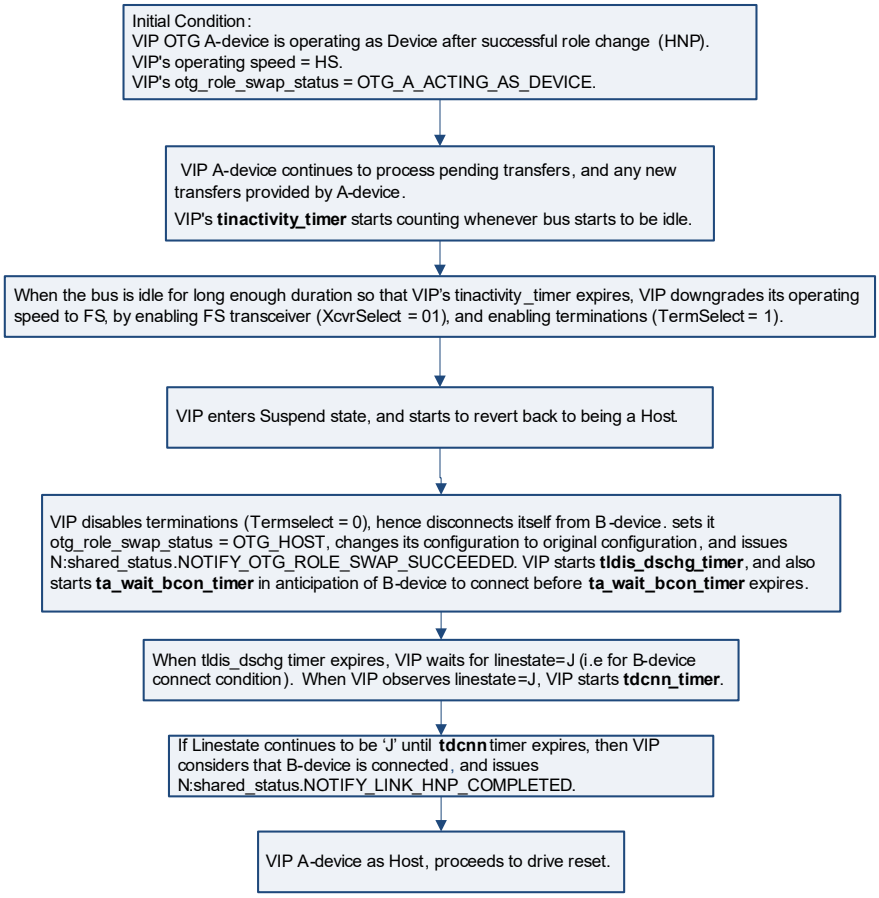


Table 4-13 expands on the abbreviated messages that are used in Figures 4-5 and 4-6.

Table 4-13 Legend Explaining the Abbreviated Messages Used in Figures 4-5 and 4-6

Abbreviated Messages Used in Figure 4-5	Actual Messages Issued by VIP
N: Moving to XCVR_STATE_BUSY	Active low PHY suspend (SuspendM = 0). Moving to state XCVR_STATE_BUSY.
N: attempt_otg_role_swap() completed with success	attempt_otg_role_swap() completed with usb_otg_b_hnp_success 1
N: Disable pull-down register	Disable pull-down resistor on DP (DpPullDown = 0)
N: A Peripheral HNP succeeded	A-PERIPHERAL HNP succeeded
Additional Abbreviated Messages Used in Figure 4-6	**Actual Messages Issued by VIP**
N: Enable FS transceiver	Enable FS transceiver (XcvrSelect = 01)
N: Enable terminations	Enable terminations (TermSelect = 1)

Table 4-13 Legend Explaining the Abbreviated Messages Used in Figures 4-5 and 4-6

Abbreviated Messages Used in Figure 4-5	Actual Messages Issued by VIP
N: Disable terminations	Disable terminations (TermSelect = 0)
N: Reverse HNP Complete	Reverse HNP Complete on A-Device

4.11.3.5 HNP Sequence of Events When VIP is Configured as a B-Device

Figure 4-7 explains the HNP flow when the VIP is configured as a B-device.

Figure 4-7 HNP Flow When VIP is Configured as a B-Device

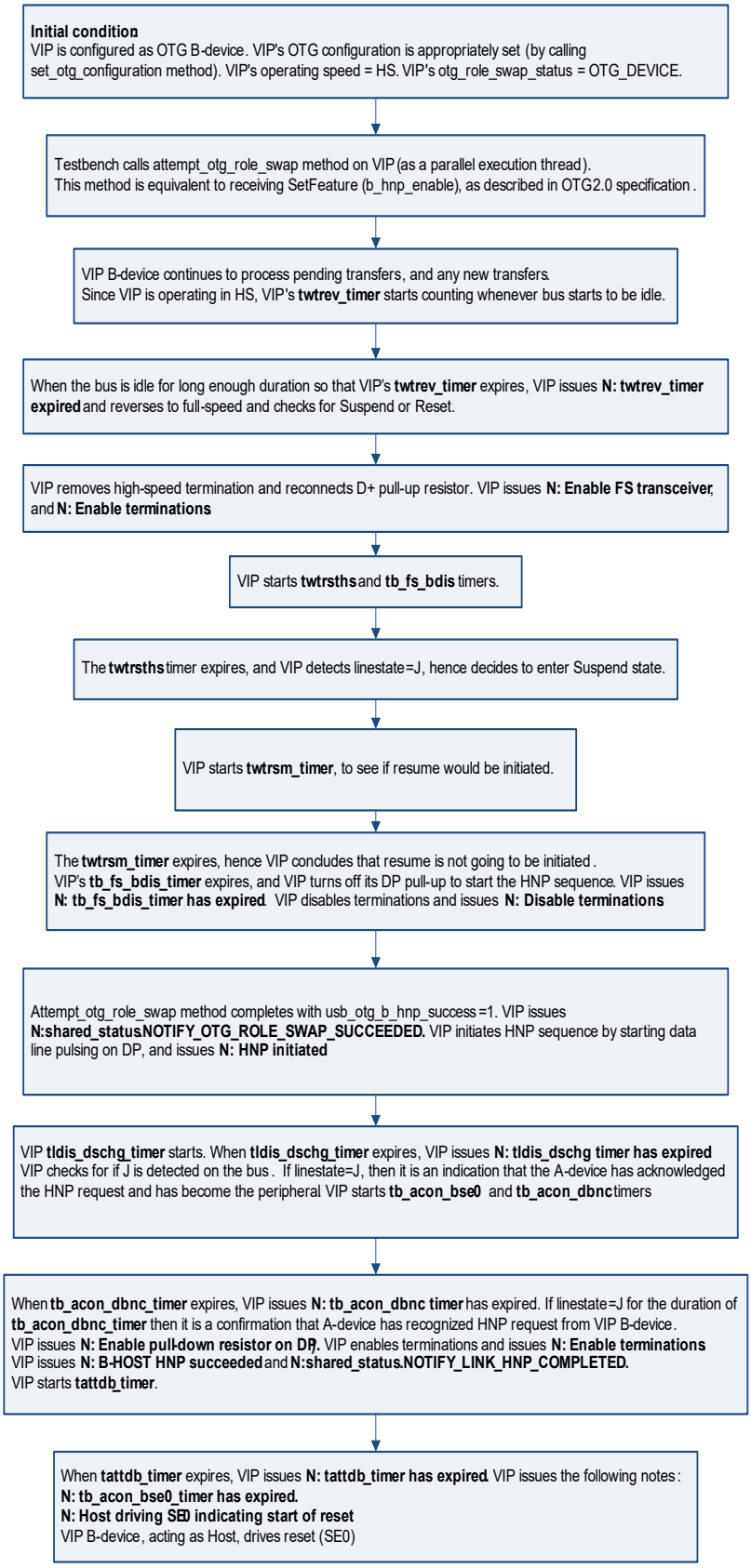


Figure 4-8 Reverse HNP Flow When the VIP B-Device Reverts Back to Acting as a Device

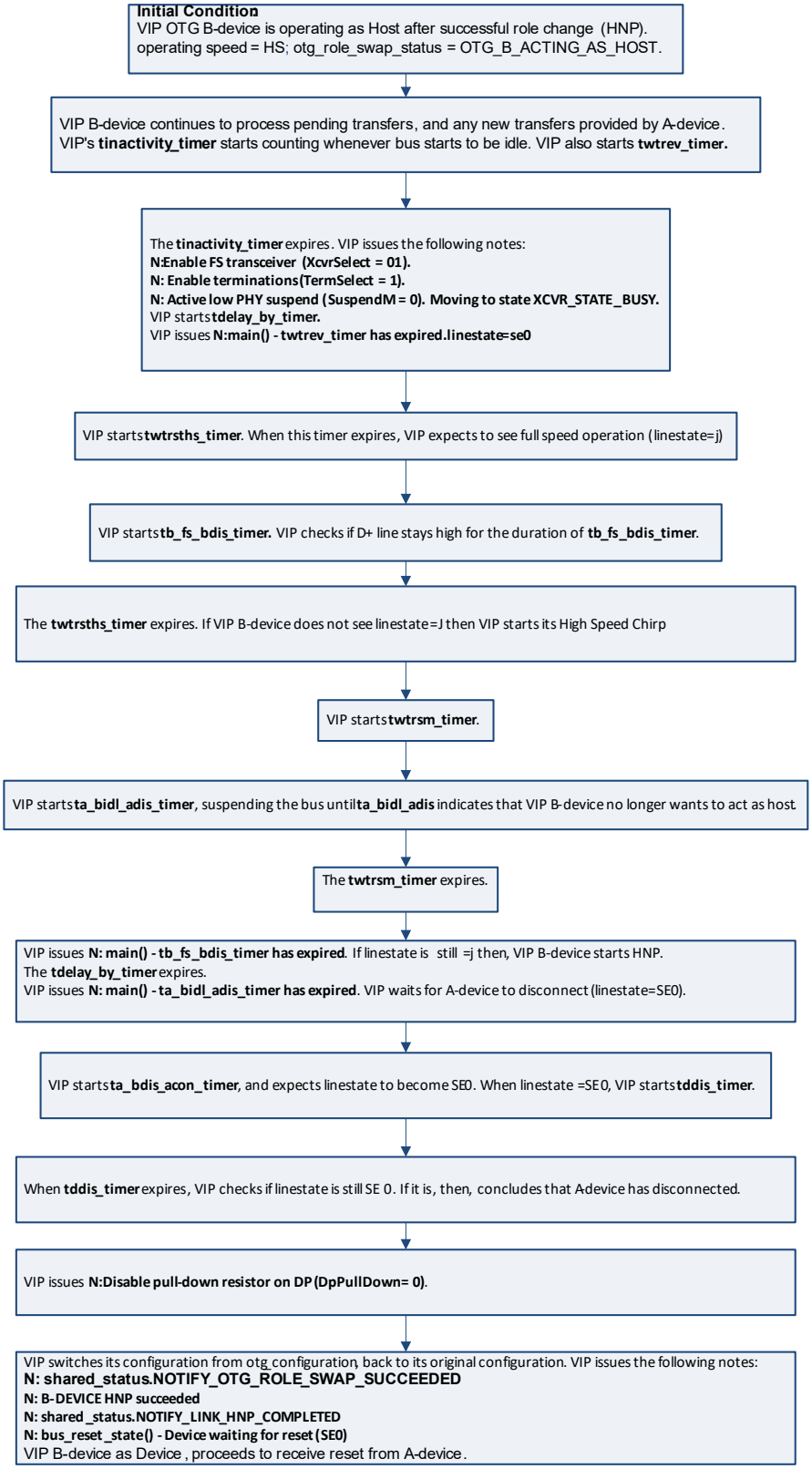


Table 4-14 expands on the abbreviated messages that are used in Figures 4-7 and 4-8.

Table 4-14 Legend Explaining the Abbreviated Messages Used in Figures 4-7 and 4-6

Abbreviated Messages Used in Figure 4-7	Actual Messages Issued by VIP
N: twtrev_timer expired	N: twtrev_timer expired
N: Enable FS transceiver	N: Enable FS transceiver (XcvrSelect = 01)
N: Enable terminations	N: Enable terminations (TermSelect = 1)
N: tb_fs_bdis_timer has expired	N: tb_fs_bdis_timer has expired
N: Disable terminations	N: Disable terminations (TermSelect = 0)
N: HNP initiated	N: HNP initiated
N: Enable pull-down resistor on DP	N: Enable pull-down resistor on DP (DpPullDown =1)
N: B-HOST HNP succeeded	N: B-HOST HNP succeeded
Additional Abbreviated Messages Used in Figure 4-8	**Actual Messages Issued by VIP**
N: Moving to XCVR_STATE_BUSY	N: Active low PHY suspend (SuspendM = 0). Moving to state XCVR_STATE_BUSY
N: twtrev_timer has expired	N: main() - twtrev_timer has expired.linestate=se0
N: tb_fs_bdis_timer has expired	N: main() - tb_fs_bdis_timer has expired
N: ta_bidl_adis_timer has expired	N: main() - ta_bidl_adis_timer has expired
N: B-DEVICE HNP succeeded	N: B-DEVICE HNP succeeded

4.11.4 Attach Detection Protocol

Attach Detection Protocol (ADP) is the mechanism used to determine whether or not a remote device has been attached or detached when VBUS is not present.

ADP involves the following primary functions:

- ADP probing (performed by either an A-device or B-device). It is used to detect a change in the attachment state of a remote device.
- ADP sensing (performed only by B-device). It is used to detect whether or not a remote A-device is performing ADP probing

4.11.4.1 ADP Probing

The VIP performs ADP probing when the following conditions are met:

- The adp_supported configuration property is set
- VBUS is below the valid OTG level (at simulation startup or following a session end)
- ADP probing has not been disabled (through an ADP_PROBING_OFF physical service request)
- ADP sensing is not active (this only applies to a B-Device)

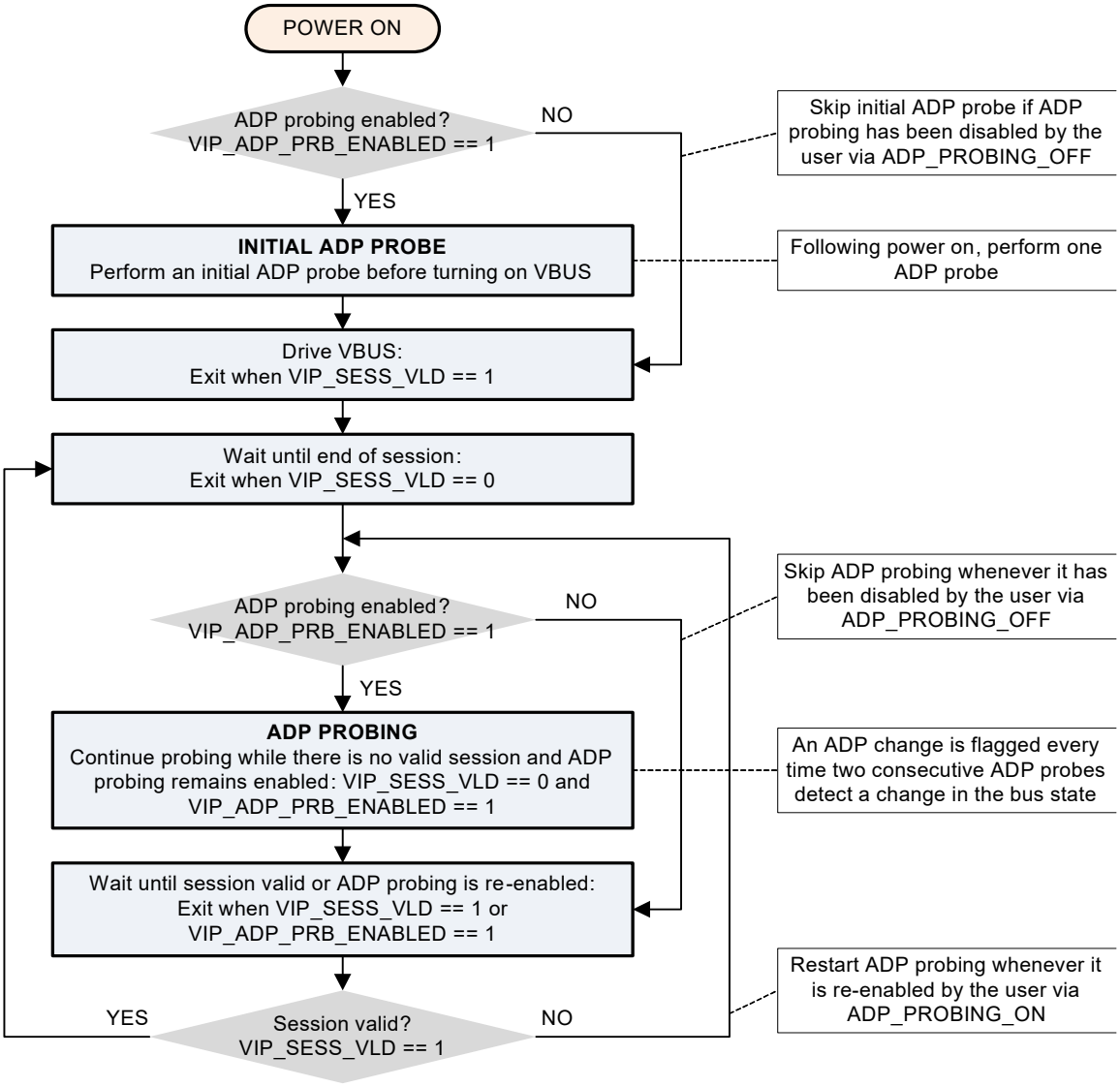
4.11.4.1.1 ADP Probing When VIP is Acting as an A-Device

When VIP is configured as an A-device, it only supports ADP probing. The ADP probing sequence is as follows:

1. At power on, the VIP performs an initial ADP probe if VBUS is not present.
2. After power on, the VIP starts ADP probing when the VBUS falls below the valid OTG session level (for example, when the VBUS is switched off). VIP continues ADP probing until it detects an ADP change, or if the VBUS rises above the valid OTG session level.
3. VIP issues an ADP change whenever it detects an attachment difference between two consecutive ADP probes.

Figure 4-9 illustrates a simple ADP Probing process for VIP as an A-device or an Embedded Host using a waveform.

Figure 4-9 ADP Probing Flow when VIP is Acting as an A-Device or Embedded Host



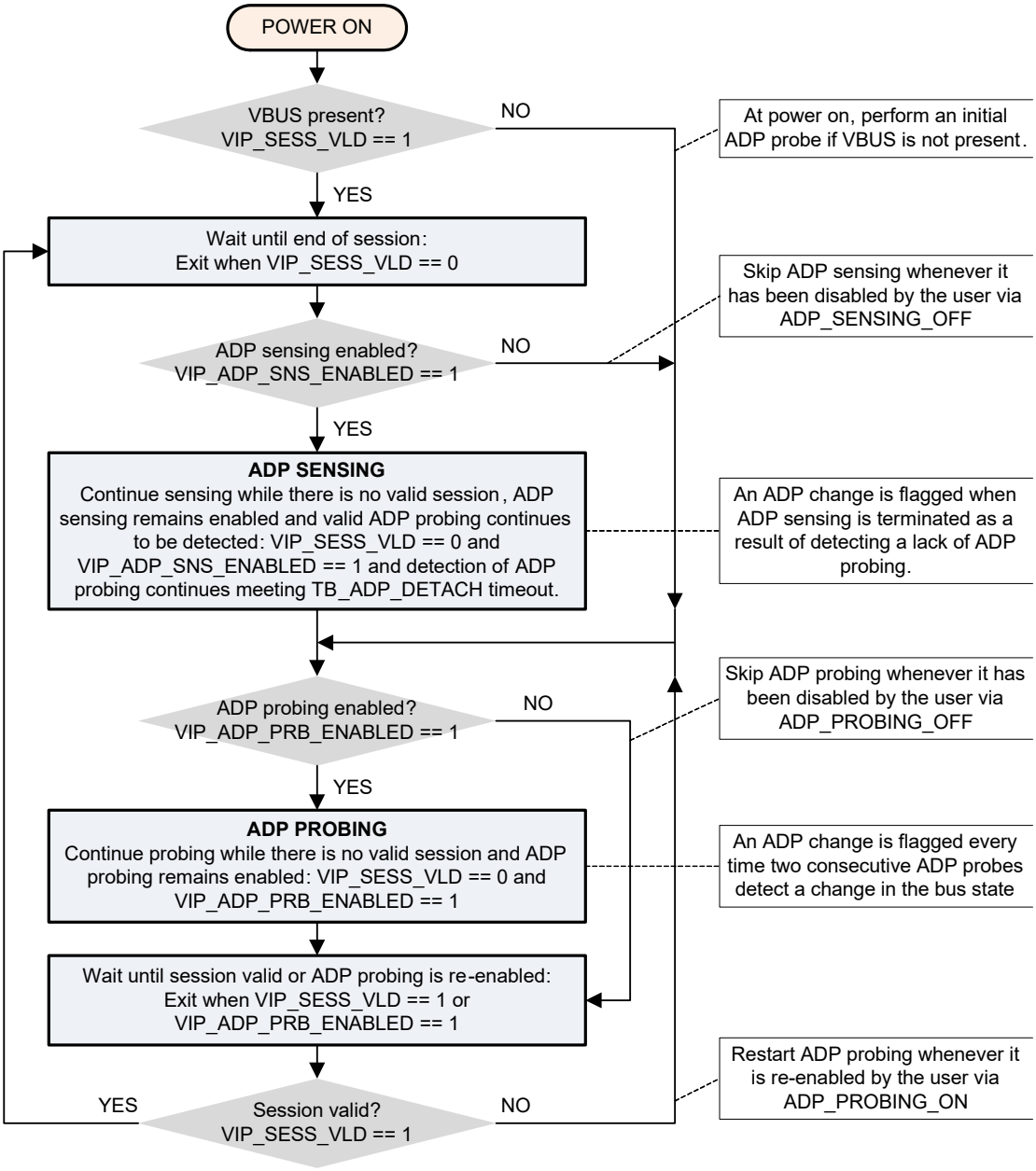
4.11.4.1.2 ADP Probing When VIP is Acting as a B-Device

When VIP is configured as a B-device, the ADP probing sequence is as follows:

1. At power on if VBUS is not present, the VIP performs an initial ADP probe before starting an SRP request.
2. After power on, the VIP starts ADP probing when the VBUS falls below the valid OTG session level (for example, when the VBUS is switched off), and if ADP sensing is not active.VIP continues ADP probing until the VBUS rises above the valid OTG session level.
3. VIP issues an ADP change whenever it detects an attachment difference between two consecutive ADP probes.

Figure 4-10 explains the ADP control when VIP is acting as a B-device.

Figure 4-10 ADP Probing and Sensing Flow when the VIP is Acting as a B-Device



4.11.4.2 ADP Sensing

The VIP performs ADP sensing when the following conditions are met:

- The adp_supported configuration property is set
- The VIP is configured as a peripheral (B-Device)
- VBUS has just fallen below the valid OTG level (following a session end)
- ADP sensing has not been disabled (through an ADP_SENSING_OFF physical service request)
- While the VIP continues to detect ADP probing by the host

4.11.4.2.1 ADP Sensing When VIP is Acting as a B-Device

1. After power on, VIP starts ADP sensing when the VBUS falls below the OTG session valid level.
2. VIP continues ADP sensing until the VBUS rises above the OTG session valid level, or if VIP detects a lack of ADP probing by the A-device. If VIP detects a lack of probing by the A-device, it issues an ADP change.

For more information on the ADP control flow when VIP is acting as a B-device, see Figure 4-10.

4.11.4.3 ADP Notifications

VIP issues the following notifications (which are a part of the status object) to indicate the success or failure of an ADP probing or sensing operation:

- NOTIFY_ADP_PRB_INITIATED - Issued at the start of a VIP ADP probe
- NOTIFY_ADP_PRB_COMPLETED - Issued at the end of a VIP ADP probe
- NOTIFY_ADP_SNS_INITIATED - Issued when VIP ADP sensing starts
- NOTIFY_ADP_SNS_COMPLETED - Issued when VIP ADP sensing ends
- NOTIFY_PHYSICAL_ADP_CHANGE - Issued following detection of an ADP change. Issue during a VIP ADP probing or VIP ADP sensing event.
  - VIP ADP probe – Issued at the end of a VIP ADP probe event if a change in bus state was detected (a change in the physical_adp_probe_state). This notification is issued just before an ADP probe is completed.
  - VIP ADP sensing – Issued during a VIP ADP sensing event whenever DUT ADP probing is not detected within the expected window of time (and is denoted by a change in the physical_adp_change_detected detected value).
- NOTIFY_PHYSICAL_VBUS_CHANGE - General notification issued whenever a change on VBUS (driven or received) is detected.
- NOTIFY_PHYSICAL_USB_20_LINESTATE_CHANGE - General notification issued whenever a change in received linestate is detected.

4.11.4.4 ADP User Control

The VIP provides physical service commands that can be used to control the VIP's modeling of ADP probing and sensing. These commands allow a user to override the automatic default behavior of the VIP.

Use the following physical service commands to enable and disable ADP probing:

- SVT_USB_PHYSICAL_SERVICE_ADP_PROBING_ON - This command enables ADP probing by the VIP. This command re-enables the VIP's default ADP probing behavior.
- SVT_USB_PHYSICAL_SERVICE_ADP_PROBING_OFF - This command disables ADP probing by the VIP. This command disables the VIP's default ADP probing behavior. Once issued, this command permanently disables all VIP ADP probing until it is manually re-enabled (using the ADP_PROBING_ON command).

Use the following commands to enable and disable ADP sensing:

- SVT_USB_PHYSICAL_SERVICE_ADP_SENSING_ON - This command enables ADP sensing by the VIP. This command re-enables the VIP's default ADP sensing behavior.

- SVT_USB_PHYSICAL_SERVICE_ADP_SENSING_OFF - This command disables ADP sensing by the VIP. This command disables the VIP's default ADP sensing behavior. Once issued, this command permanently disables all VIP ADP sensing until it is manually re-enabled (using the ADP_SENSING_ON command).

By default, both ADP probing and sensing are enabled.

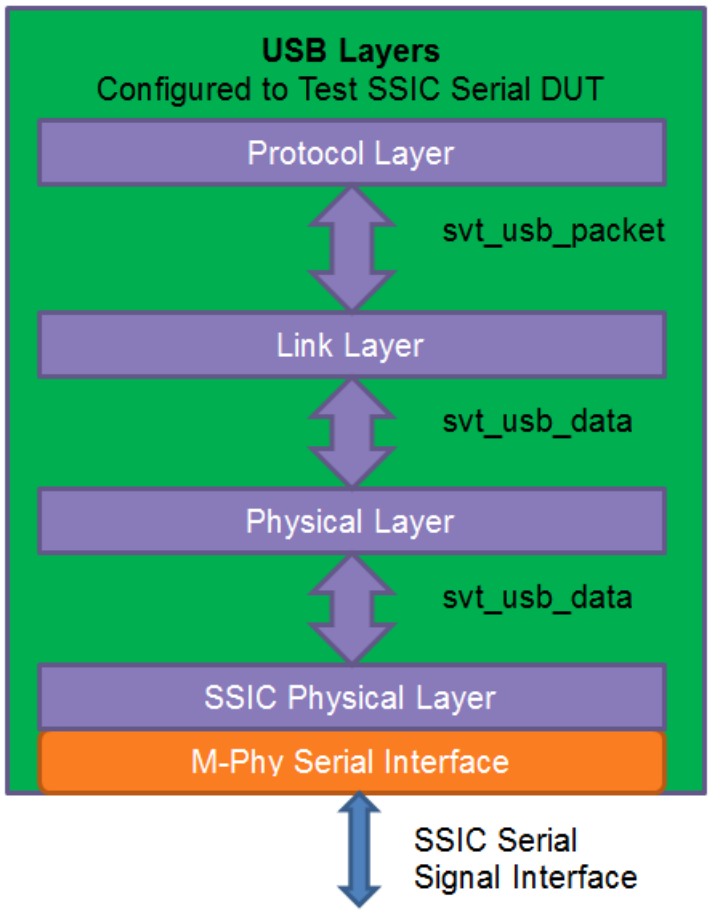
4.12 SSIC Physical Layer

The USB VIP implements the Super Speed Inter-chip (SSIC) interface through an uvm_component called **svt_usb_ssic_physical.** The USB VIP SSIC Physical component object extends from the uvm_component class.

When a SSIC interface is being simulated, the SSIC Physical layer, and not the Physical layer models the interface to a DUT. In this capacity, the USB VIP SSIC Physical component provides the features defined in the USB SSIC specification related to the PHY Adapter (PA) block and instances the necessary M-Phy VIP components.

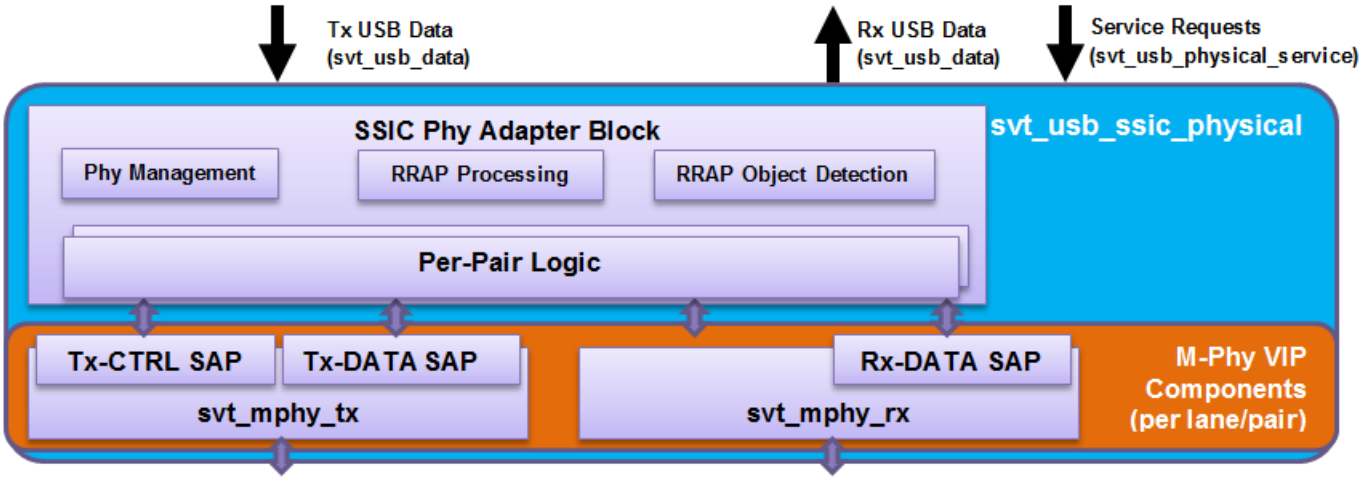
The following figure shows the relationship of the SSIC Physical Layer to the other stacked layers of the VIP.

Figure 4-11 VIP Layered Architecture with SSIC/M-Phy Serial Interface



The following illustration shows the make up of the SSIC Physical Layer.

Figure 4-12 Functionality of SSIC Physical Layer



The VIP may be configured with 1, 2 or 4 M-Phy lanes.

- A 'lane' is an associated pair of M-Phy Tx/Rx interfaces.
- SSIC also supports the notion that a logical
- *pair* may map to a different physical *lane*. For example logical *Pair 0* may have its data sent/received on physical *Lane 2*.

  Supported by the VIP's **svt_usb_configuration:: ssic_pair_to_lane[$]** array, which holds the logical *pair* number for each physical *lane* number.
- The M-Phy lanes are independent of each other, coordinated only by the USB SSIC Physical layer.

Each lane also supports simultaneous CRTL and DATA accesses:

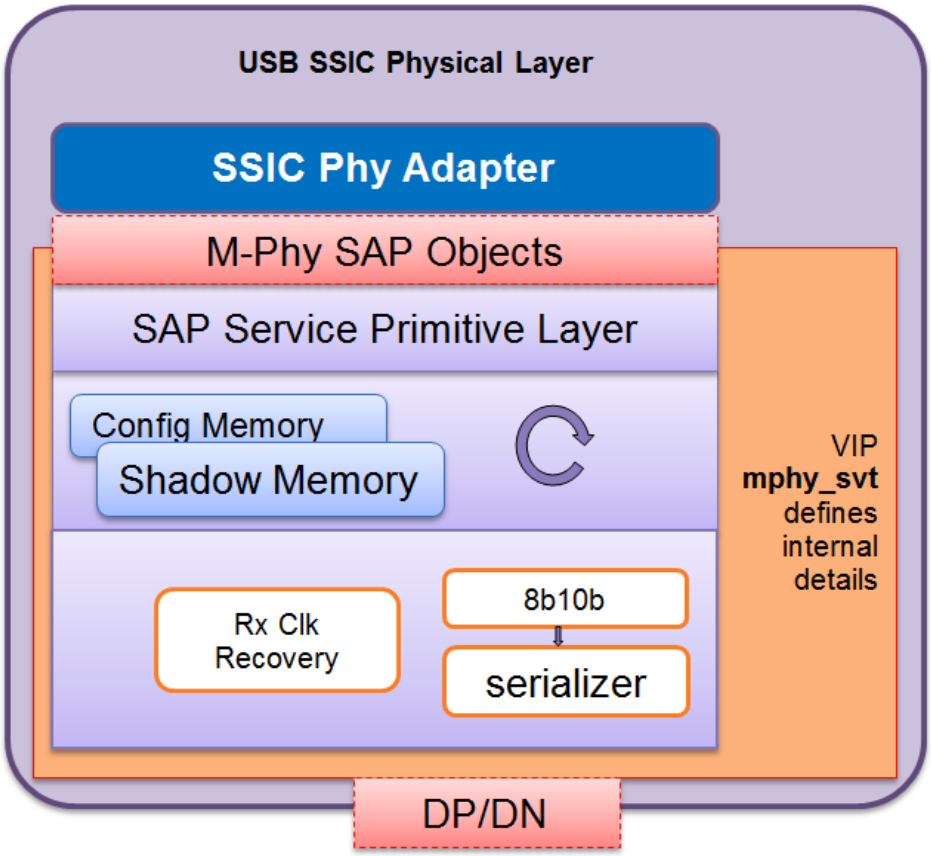
- Connection to/from the SSIC Physical layer, and between Phy Adapter and M-Phy is channel/port based.
  - For Tx, USB Data (**svt_usb_data**) objects produced by the USB Agent's Link Layer are passed to the Agent's Physical Layer and then on to the Agent's SSIC Physical Layer. There they are transformed into CTRL SAP / DATA SAP objects and passed to the M-Phy components.
  - For Rx, activity on the signal interface is detected and interpreted by the M-Phy components to create CTRL SAP / DATA SAP objects, which are passed to the USB Agent's SSIC Physical Layer where they are interpreted and transformed into USB Data (**svt_usb_data**) objects. The USB Data objects are then passed through the Agent's Physical Layer and up to the Agent's Link Layer.
  - CTRL SAP and DATA SAP are both **svt_mphy_transaction** objects.
  - A pair of SAPs (1 CTRL, 1 DATA) is required to Send/Receive each USB Data object.
  - Callbacks allow testbench access to data objects passing through the various channels/ports.
- Service Requests allow the Link Layer and the testbench to trigger the SSIC Physical layer to take specific actions (for example, direct M-Phys to go to the HIBERN8 state).

Refer to Service Tran'sactions for additional information on Service Transactions.

Attention While not shown in any of the figures, the SSIC Physical Layer can also be configured to contain an M-Phy Monitor (**svt_mphy_monitor**) component associated with each M-Phy Tx and M-Phy Rx component. These monitor components may be used for functional coverage and/or protocol checking of the M-Phy interfaces. There are properties (e.g. ssic_enable_mphy_monitor) in the USB Configuration (svt_usb_configuration) object class that may be used to control the behavior of the M-Phy monitors. Please refer to the class reference for more details.

The following illustration provides a high-level of the SSIC Layer functions.

Figure 4-13 Overview of SSIC Physical Layer



4.12.1 Configuration Classes

You use two sets of configuration classes. One for the SSIC Physical Layer. Another for the MPHY lager.

- **svt_usb_configuration**. To support SSIC functionality the **svt_usb_configuration** data class contains sub-object arrays for the M-Phy Tx/Rx components. Each array index corresponds to the M-Phy Lane for which the corresponding M-Phy Tx or Rx component is being configured.

  In addition to the sub-configurations for the M-Phy component there a number of configuration properties specific to SSIC. These all have names with the ssic_ prefix (e.g. ssic_profile). Please refer to the class reference for more details. This is the top level configuration class for SSIC as it includes the following class within the following arrays:

- **usb_mphy_rx_cfg**[$] : an array of type svt_usb_mphy_configuration
- **usb_mphy_tx_cfg**[$] : an array of type svt_usb_mphy_configuration

- **svt_usb_mphy_configuration**. This is a class extended from the **svt_mphy_configuration** class. There are no attributes--only constraints and validity checking methods are present to restrict M-Tx and M-Rx capability attributes to values in Tables 2-2 and 2-3 of the SSIC specification.

  Note: This must have M-Tx and/or M-Rx capability attributes from the [M-Phy] specification Tables 48 and 52.

4.12.2 SSIC Physical Service Channel

The SSIC physical component uses the Physical Layer component to obtain svt_usb_physical_service objects for processing.

- Providing svt_usb_physical_service objects directly to the SSIC Physical component is not supported.

4.12.3 Signal Interfaces

Unlike other USB transactors, the USB SSIC Physical transactors communicate through signal interfaces as well as channels. But the USB SSIC Physical transactor does not use a port instance like the USB Physical transactor does. Instead the USB SSIC Physical instances the number of svt_mphy_tx and svt_mphy_rx transactors required by the total number of LANEs being simulated. The svt_mphy_tx and svt_mphy_rx transactors either communicated via channel connects or directly with the M-Phy interface specified in the svt_mphy_configuration class supplied to the M-Phy components.

4.12.4 Connection Options

The VIP SSIC Physical Layer Transactor supports the following verification topologies:

- SSIC Physical transactor connecting a Physical transactor and a remote DUT.
- SSIC Physical transactor connecting a Physical transactor and a local PHY (DUT).

4.12.5 Data Factory Objects

The following are SSIC's Physical layer data factory objects. These factories allocate new data descriptor instances to represent data placed in data out channel. This factory is always present. Replace this factory with the extended version to cause the SSIC Physical transactor to use an extended data descriptor.

- Attribute Name: usb_data_factory. SSIC Physical out factories: These factories allocate new data descriptor instances to represent data placed in SAP DATA and SAP CONTROL channels. This factory is always present. Replace this factory with the extended version to cause the SSIC Physical transactor to use an extended data descriptor.
- Attribute Names: sap_data_factory & sap_ctrl_factory. SSIC Physical RRAP factories: These factories allocate new data descriptor instances to represent RRAP activity occurring in the model or across the bus.
- Attribute Names: rrap_packet_factory & rrap_transaction_factory

4.12.6 Exception List Factories

The SSIC Physical transactor supplies the following exception list factories. They are used for creating exceptions for the processing of RRAP packets or RRAP transactions:

- randomized_ssic_rrap_packet_tx_exception_list
- randomized_ssic_rrap_transaction_exception_list

4.12.7 Error injection

The SSIC physical layer provides the following error injection methods.

- RRAP transaction. For RRAP target processing can inject invalid response, no response, and incorrect processing. See the HTML documentation for svt_usb_ssic_rrap_transaction::rrap_response_type_enum
- RRAP Packets
  - Parity error
  - Reserved field error
  - Invalid packet type

4.12.8 Error Detection

The SSIC layer provides the following error detection methods.

- RRAP Transaction. For RRAP Master processing can detect invalid response and no response. See the HTML documentation of svt_usb_ssic_rrap_transaction::rrap_response_type_enum.
- RRAP Packet. Invalid packet type errors are captured as RRAP transaction invalid response exceptions.
  - Parity error
  - Reserved field error

4.12.9 Notifications

Following are notifications supported byte SSIC Physical transactor:

- NOTIFY_SSIC_BURSTEND_PADDING
- NOTIFY_SSIC_RECEIVED_INVALID_RRAP_PACKET
- NOTIFY_MPHY_RX_AGGREGATE_STATE_CHANGE

4.12.10 Transactions

The following sections describe data transaction classes that support SSIC functionality in the VIP.

4.12.10.1 Data Transactions

The following data transaction classes support SSIC functionality in the VIP.

- **svt_usb_data –** Transfers data symbols between the Link, Physical, and SSIC Physical layers.
- **svt_mphy_transaction –** Produced and/or consumed by the SSIC Physical Layer to communicate CTRL and DATA SAPs to/from the M-Phy interface components.

- **svt_usb_ssic_rrap_transaction –** Represents an SSIC *Remote Register Access Protocol* command / response pair. Its **implementation** array will consist of 2 **svt_usb_ssic_rrap_packet** data objects.
- **svt_usb_ssic_rrap_packet** - Represents a single SSIC *Remote Register Access Protocol* command or response.

4.12.10.2 Service Tran'sactions

The **svt_usb_physical_service** data class is used to request specific non-dataflow actions from the USB VIP's Physical and/or SSIC Physical layers. The following new values of the **physical_command** enumerated property are used to identify SSIC-related services, as described:

- SSIC_RRAP
  - Requests execution of the SSIC RRAP transaction defined by **rrap_transaction** (USB SSIC only)
- SSIC_MTX_EXIT_HIBERN8
  - Requests SSIC Tx Phy(s) to exit Hibernate state (USB SSIC only)
- SSIC_MTXRX_CFG_FOR_HIBERN8
  - Requests SSIC Tx & Rx Phy(s) to be configured to allow entry to Hibernate state. If issued while in HS_BURST, upon exiting HS_BURST CFGRDY CTRL SAP objects are issued to all local M-Phy instances. (USB SSIC only)
- SSIC_MTX_EXIT_BURST
  - Requests SSIC Tx Phy(s) to exit BURST state. Use of this in LS burst will not cause the SSIC specification defined LS BURST CLOSURE sequence. To follow the defined LS BURST CLOSURE sequence, use a physical service with **physical_command** set to **SSIC_RRAP** and have the **rrap_transaction** setup to write to the BURST_CLOSURE RRAP register. (USB SSIC only)
  - If a **SSIC_MTXRX_CFG_FOR_HIBERN8** occurred while in HS_BURST mode when this command is processed, a CFGRDY SAP CTRL is issued to all M-Phys once the mphy_tx_aggregate_fsm_state and mphy_rx_aggregate_fsm_state in svt_usb_status indicate STALL has been reached. The command is not ENDED until both Phy states indicate HIBERN8 was reached.
- SSIC_MTX_ENTER_BURST
  - Requests SSIC Tx Phy(s) to enter BURST state (USB SSIC only)
- SSIC_MTX_LINE_RESET
  - Requests SSIC Tx Phy(s) to perform LINE Reset (USB SSIC only)
- SSIC_MPHY_RESET
  - Requests SSIC Tx Phy(s) to perform the reset operation selected by mphy_reset_kind (USB SSIC only)
- SSIC_DSP_DISCONNECT
  - Requests a DSP disconnect (USB SSIC only)
- SSIC_TEST_MODE_HS_BURST
  - Used in Test mode. Moves Tx Phy of pair under test from STALL to HS_BURST, Transmit the payload in the object attached to this service request (USB SSIC only)

In conjunction with some of these command types the following members of the **svt_usb_physical_service** class are used to specify the details of the service request:

- mphy_reset_kind
- reg_addr
- reg_data
- rrap_packet
- rrap_transaction
- test_mode_transaction

4.12.11 SSIC Physical Transactor Callbacks

The SSIC Physical transactor contains the following callbacks:

- post_randomize_ssic_rrap_transaction_exception_list_cb_exec. Callback issued when the SSIC Physical layer has randomized the RRAP transaction's exception_list
- post_sap_data_from_mphy_rx_chan_get_cb_exec. Callback issued by transactor post getting a sap_data from the M-phy Rx input channel.
- pre_data_out_chan_put_cb_exec. Callback issued by transactor prior to putting a received transaction descriptor into the USB SS link output channel.
- pre_randomize_rrap_transaction_cb_exec. Callback issued just prior to actually randomizing the rrap_transaction.
- pre_sap_ctrl_to_mphy_rx_chan_put_cb_exec. Callback issued by transactor prior to putting a sap_ctrl into the M-phy Rx output channel.
- pre_sap_ctrl_to_mphy_tx_chan_put_cb_exec.Callback issued by transactor prior to putting a sap_ctrl into the M-phy Tx output channel.
- pre_sap_data_to_mphy_tx_chan_put_cb_exec. Callback issued by transactor prior to putting a sap_data into the M-phy Tx output channel.
- pre_tx_rrap_packet_cb_exec. Callback issued when the SSIC Physical layer is ready transform a Tx RRAP packet into the M-Phy SAP objects transmitted over the bus

4.12.12 Shared Status

The following properties in the **svt_usb_status** class are accessible as shared status information related to SSIC and M-Phy state:

- **mphy_tx_status[$] –** Array of objects (one per configured M-Phy lane) of type **svt_mphy_status**. These instances become the shared status objects associated with each M-Phy Tx component in use in the **svt_usb_ssic_physical** layer component.
- **mphy_rx_status[$] –** Array of objects (one per configured M-Phy lane) of type **svt_mphy_status**. These instances become the shared status objects associated with each M-Phy Rx component in use in the **svt_usb_ssic_physical** layer component.
- **mphy_tx_aggregate_fsm_state –** Represents the effective state of the M-Phy Tx components, as combined from all active lanes.
- **mphy_rx_aggregate_fsm_state –** Represents the effective state of the M-Phy Rx components, as combined from all active lanes.

- **ssic_mphy_tx_aggregate_state_stable –** Bit which is high when all the M-Phy Tx components have the same FSM state, and low otherwise.
- **ssic_mphy_rx_aggregate_state_stable –** Bit which is high when all the M-Phy Rx components have the same FSM state, and low otherwise.

In addition to the above (highlighted here due to their connection to the M-Phy components) there are a number of other notifications and status variables related to SSIC, with names like ssic_* and `NOTIFY_SSIC_*`. Please refer to the class reference for more details on these.

With respect to the mphy_tx_status and mphy_rx_status arrays mentioned above, the svt_mphy_status class (from the mphy_svt VIP) holds current (M-Phy) Config Memory values, and the current (M-Phy) Shadow Memory values for each M-Phy component active in the **usb_svt** VIP agent.

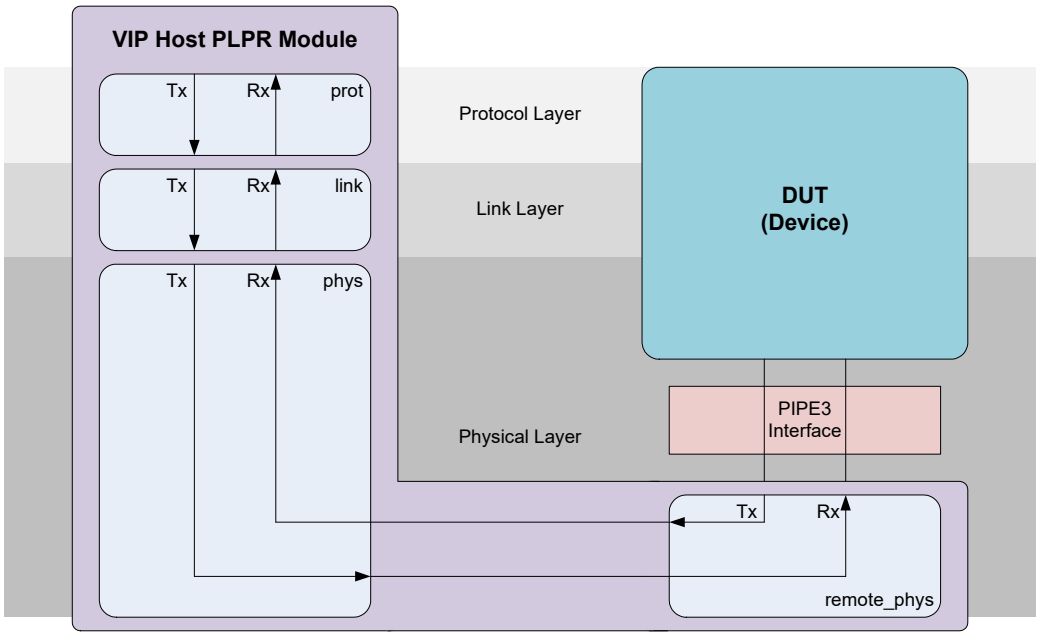
5 Verification Topologies

This chapter presents several testbench topologies. In addition to a brief description of the testbench topologies, each section lists configuration parameters required to instantiate the VIP.

5.1 USB VIP Host and DUT Device Controller

This topology consists of a VIP that tests a USB device controller. The VIP contains local protocol, link, and physical layers that emulates a USB host, along with a remote physical layer that emulates a USB device PHY. The VIP and the DUT connect through a PIPE3 interface.

Figure 5-1 USB VIP Host and DUT Device Controller



The following code implements this topology:

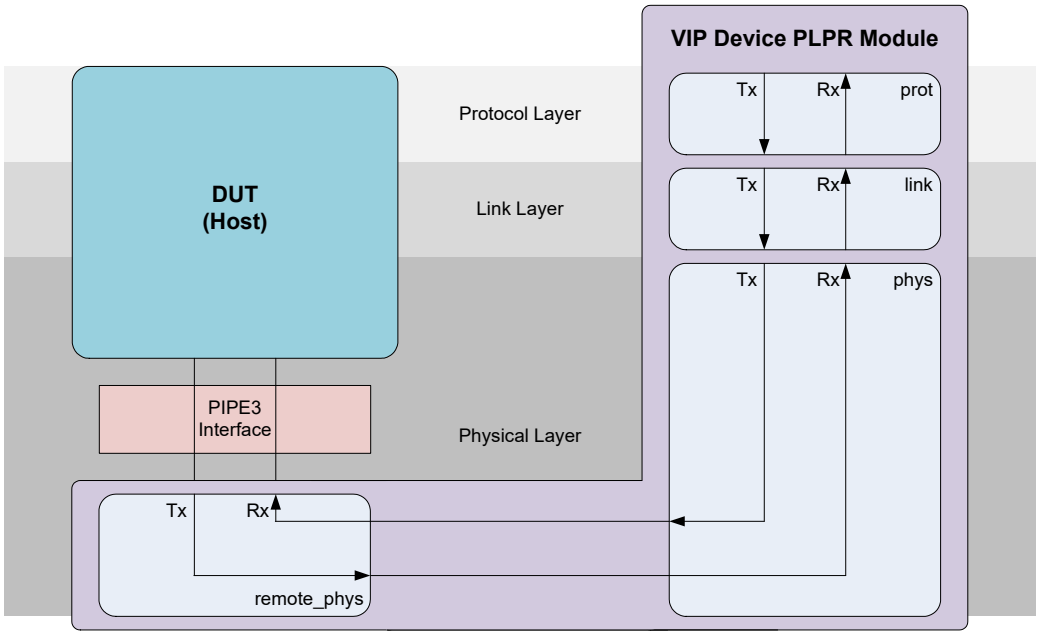
```
svt_usb_subenv_pipe3_plpr_hdl usb_host(....);

/*********************************/
int is_valid, cfg_handle, ep_cfg_handle;
```

```
// Initialize the VIP USB Host module instance.
usb_host.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "instance", "usb_host", 0);

// Set Up the Host's Configuration
// Get an integer handle that references a temporary copy of the Host's (default)
// internal configuration data object. Using this handle, the desired
// configuration settings are applied to the properties of the temporary copy.
usb_host.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "cfg", cfg_handle, 0);

// Setting the VIP as the host
usb_host.set_data_prop(is_valid, cfg_handle, "module_type", `SVT_USB_HOST, 0);

// Setting the speed(SS),capability(SS_ONLY) and interface(PIPE_IF) for the VIP
usb_host.set_data_prop(is_valid, ep_cfg_handle, "speed", 3, 0);
usb_host.set_data_prop(is_valid, ep_cfg_handle, "usb_ss_signal_interface", 0, 0);
usb_host.set_data_prop(is_valid, ep_cfg_handle, "usb_capability", 1, 0);
```

5.2 USB VIP Device and DUT Host Controller

This topology consists of a VIP that tests a USB host controller. The VIP contains local protocol, link, and physical layers that emulates a USB device, along with a remote physical layer that emulates a USB device PHY. The VIP and the DUT connect through a PIPE3 interface.

Figure 5-2 USB VIP Device and DUT Host Controller



The following code implements this topology:

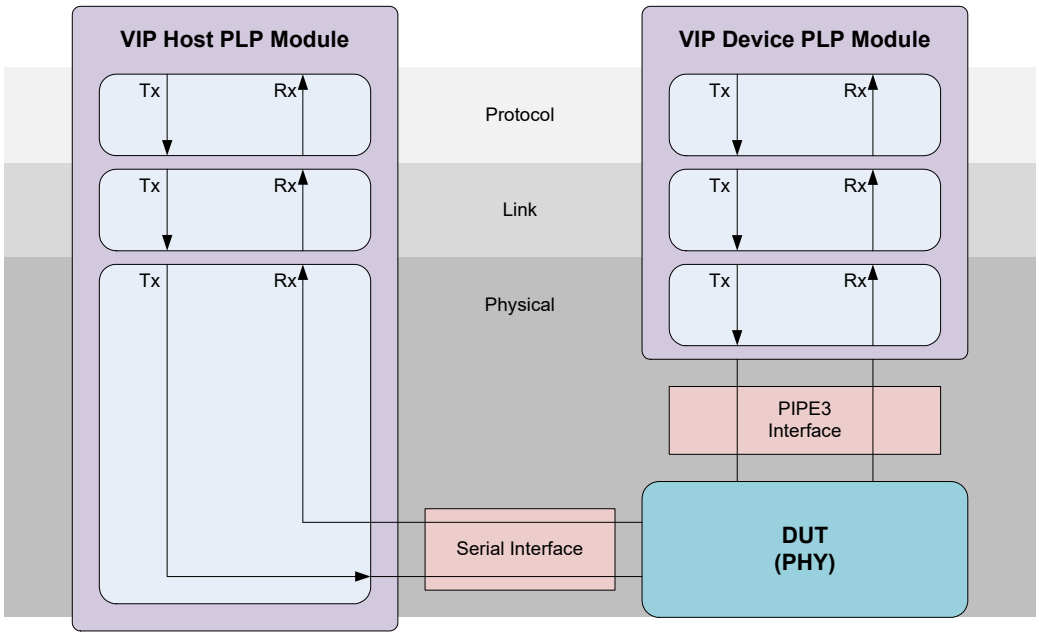
```
svt_usb_subenv_pipe3_plpr_hdl usb_dev(....);

/*********************************/
int is_valid, cfg_handle, ep_cfg_handle;
```

```
// Initialize the VIP USB Device module instance.
usb_dev.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "instance", "usb_dev", 0);

// Set Up the Device's Configuration
//    Get an integer handle that references a temporary copy of the Device's (default)
//    internal configuration data object. Using this handle, the desired configuration
//    settings will be applied to the properties of the temporary copy.
usb_dev.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "cfg", cfg_handle, 0);

// Setting the VIP as the device
usb_dev.set_data_prop(is_valid, cfg_handle, "module_type", `SVT_USB_DEVICE, 0);

// Setting the speed(SS),capability(SS_ONLY) and interface(PIPE_IF) for the VIP
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "speed", 3, 0);
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "usb_ss_signal_interface", 0, 0);
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "usb_capability", 1, 0);
```

5.3 USB VIP Host and DUT Device PHY

This topology consists of two VIP instances and a USB Device PHY. The host VIP contains local protocol, link, and physical layers that connect to the DUT through SS interface. The device VIP contains local, link, and physical layers that connect to the local DUT through a PIPE3 interface.

Figure 5-3 USB3 VIP Host and DUT Device PHY



The following code implements this topology:

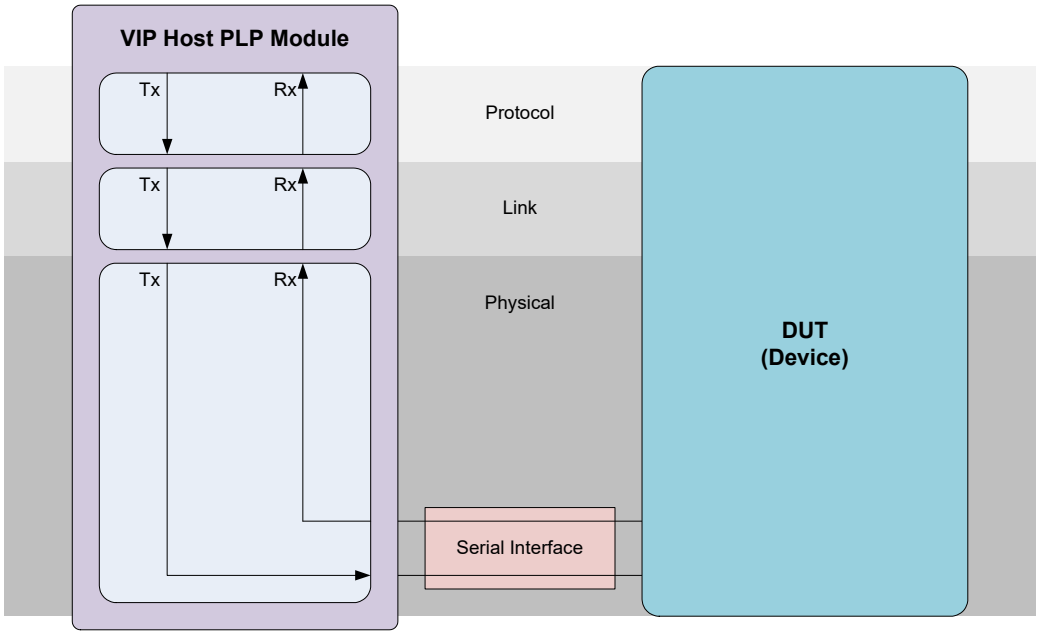
```
svt_usb_subenv_pipe3_plp_hdl usb_host(....);
svt_usb_subenv_pipe3_plp_hdl usb_dev(....);

/*********************************/
int is_valid, cfg_handle, ep_cfg_handle;
```

```
// Initialize the VIP USB Host module instance.
usb_host.set_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "instance", "usb_host", 0);

// Set Up the Host's Configuration
//    Get an integer handle that references a temporary copy of the Host's (default)
//    internal configuration data object. Using this handle, the desired configuration
//    settings will be applied to the properties of the temporary copy.
usb_host.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "cfg", cfg_handle, 0);

// Setting the VIP as the host
usb_host.set_data_prop(is_valid, cfg_handle, "module_type", `SVT_USB_HOST, 0);

// Setting the speed(SS),capability(SS_ONLY) and interface(SERIAL) for the VIP
usb_host.set_data_prop(is_valid, ep_cfg_handle, "speed", 3, 0);
usb_host.set_data_prop(is_valid, ep_cfg_handle, "usb_ss_signal_interface", 2, 0);
usb_host.set_data_prop(is_valid, ep_cfg_handle, "usb_capability", 1, 0);


/*********************************/
// Initialize the VIP USB Device module instance.
usb_dev.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "instance", "usb_dev", 0);

// Set Up the Device's Configuration
//    Get an integer handle that references a temporary copy of the Device's (default)
//    internal configuration data object. Using this handle, the desired configuration
//    settings will be applied to the properties of the temporary copy.
usb_dev.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "cfg", cfg_handle, 0);

// Setting the VIP as the device
usb_dev.set_data_prop(is_valid, cfg_handle, "module_type", `SVT_USB_DEVICE, 0);

// Setting the speed(SS),capability(SS_ONLY) and interface(PIPE_IF) for the VIP
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "speed", 3, 0);
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "usb_ss_signal_interface", 1, 0);
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "usb_capability", 1, 0);
```

5.4 USB VIP Host and DUT Device

This topology consists of a VIP that tests a USB host. The VIP contains local protocol, link, and physical layers that emulates a USB host. The VIP and the DUT connect through a serial interface.

Figure 5-4 USB3 VIP Host and DUT Device



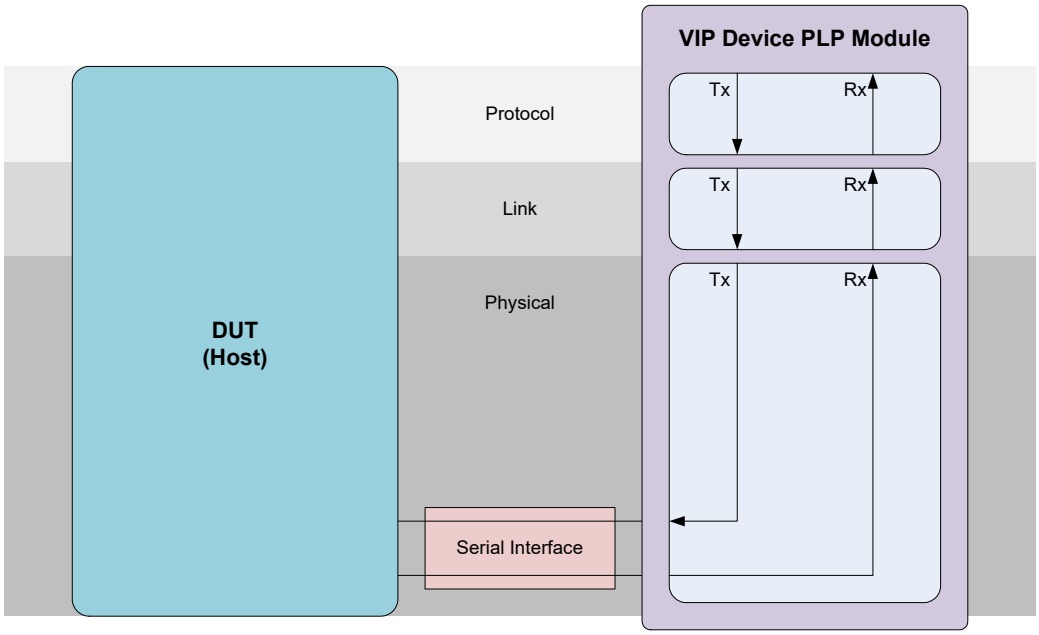
The following code implements this topology:

```
svt_usb_subenv_pipe3_plp_hdl usb_host(....);

/*********************************/
int is_valid, cfg_handle, ep_cfg_handle;

// Initialize the VIP USB Host module instance.
usb_host.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "instance", "usb_host", 0);

// Set Up the Host's Configuration
//    Get an integer handle that references a temporary copy of the Host's (default)
//    internal configuration data object. Using this handle, the desired configuration
//     settings will be applied to the properties of the temporary copy.
usb_host.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "cfg", cfg_handle, 0);

// Setting the VIP as the host
usb_host.set_data_prop(is_valid, cfg_handle, "module_type", `SVT_USB_HOST, 0);

// Setting the speed(SS),capability(SS_ONLY) and interface(SERIAL) for the VIP
usb_host.set_data_prop(is_valid, ep_cfg_handle, "speed", `SVT_USB_SPEED_SS, 0);
usb_host.set_data_prop(is_valid, ep_cfg_handle, "usb_ss_signal_interface",
`SVT_USB_SS_SERIAL_IF, 0);
usb_host.set_data_prop(is_valid, ep_cfg_handle, "usb_capability", `SVT_USB_SS_ONLY, 0);
```

5.5 USB VIP Device and DUT Host

This topology consists of a VIP that tests a USB host. The VIP contains local protocol, link, and physical layers that emulates a USB device. The VIP and the DUT connect through a serial interface.

Figure 5-5 USB3 VIP Device and DUT Host



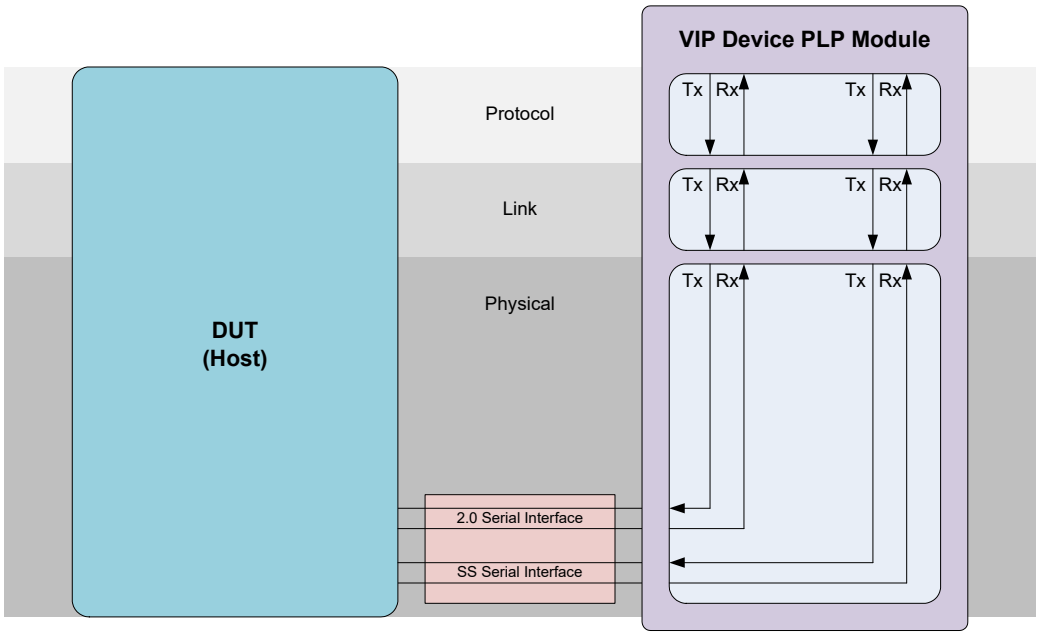
The following code implements this topology:

```
svt_usb_subenv_pipe3_plp_hdl usb_dev(....);

/*********************************/
int is_valid, cfg_handle, ep_cfg_handle;

// Initialize the VIP USB device module instance.
usb_dev.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "instance", "usb_dev", 0);

// Set Up the dev's Configuration
//    Get an integer handle that references a temporary copy of the dev's (default)
//    internal configuration data object. Using this handle, the desired configuration
//    settings will be applied to the properties of the temporary copy.
usb_dev.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "cfg", cfg_handle, 0);

// Setting the VIP as the dev
usb_dev.set_data_prop(is_valid, cfg_handle, "module_type", `SVT_USB_dev, 0);

// Setting the speed(SS),capability(SS_ONLY) and interface(SERIAL) for the VIP
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "speed", `SVT_USB_SPEED_SS, 0);
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "usb_ss_signal_interface",
`SVT_USB_SS_SERIAL_IF, 0);
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "usb_capability", `SVT_USB_SS_ONLY, 0);
```

5.6 USB VIP Device and DUT Host – Concurrent SS and 2.0 Traffic

This topology consists of a VIP that tests a USB host. The VIP contains local protocol, link, and physical layers that emulates a USB device. The VIP and the DUT connect through a serial interface that provides concurrent SS and 2.0 traffic.

Figure 5-6 USB3 VIP Device and DUT Host – Concurrent SS and 2.0 Traffic



The following code implements this topology:

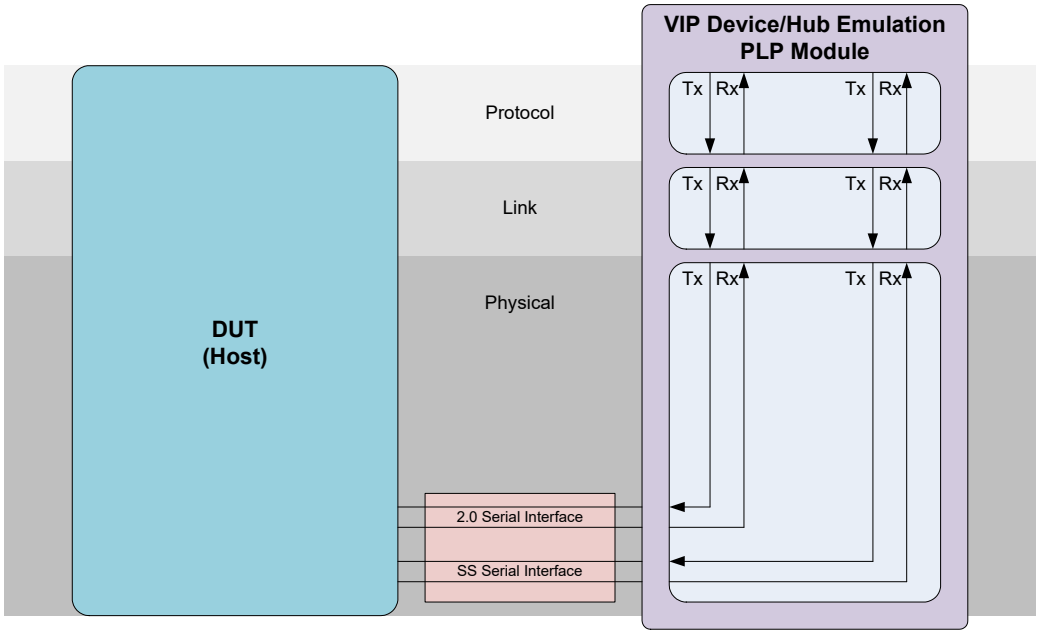
```
svt_usb_subenv_pipe3_plp_hdl usb_dev(....);

/*********************************/
int is_valid, cfg_handle, ep_cfg_handle;

// Initialize the VIP USB device module instance.
usb_dev.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "instance", "usb_dev", 0);

// Set Up the dev's Configuration
//    Get an integer handle that references a temporary copy of the dev's (default)
//    internal configuration data object. Using this handle, the desired configuration
//    settings will be applied to the properties of the temporary copy.
usb_dev.get_data_prop(is_valid,`SVT_CMD_NULL_HANDLE, "cfg", cfg_handle, 0);

// Setting the VIP as the dev
usb_dev.set_data_prop(is_valid, cfg_handle, "module_type", `SVT_USB_dev, 0);

// Setting the speed(SS),capability(SS_ONLY) and interface(SERIAL) for the VIP
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "speed", `SVT_USB_SPEED_SS, 0);
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "usb_ss_signal_interface",
`SVT_USB_SS_SERIAL_IF, 0);
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "usb_20_signal_interface",
`SVT_USB_20_SERIAL_IF, 0);
```

```
usb_dev.set_data_prop(is_valid, ep_cfg_handle, "usb_capability", `SVT_USB_SS_CAPABLE,
0);
```

5.7 USB VIP Device with Hub Emulation and DUT Host – Concurrent Serial Interface Traffic

This topology consists of a VIP that tests a USB host. The VIP contains local protocol, link, and physical layers that emulates a USB device with a hub. The VIP and the DUT connect through a serial interface that provides concurrent SS and 2.0 traffic.

Figure 5-7 USB3 VIP Device with Hub Emulation and DUT Host – Concurrent Serial Interface Traffic



6 VIP Tools

6.1 Using Native Protocol Analyzer for Debugging

This feature enables you to invoke Protocol Analyzer from Verdi GUI. You can synchronize the Verdi wave window, smart log and the source code with the Protocol Analyzer transaction view.

Protocol Analyzer can be enabled in an interactive and post-processing mode. The new features available in Native Protocol Analyzer includes layer based grouping of the transactions, Quick filter, Call stack, horizontal zoom and reverse debug with the interactive support.

6.1.1 Prerequisites

Protocol Analyzer uses transaction-level dump database. You can use the following settings to dump the transaction database:

Compile Time Options

- `-lca`
- `-kdb // dumps the work.lib++ data for source coding view`
- `+define+SVT_FSDB_ENABLE // enables FSDB dumping`
- `-debug_access`

  You can dump the transaction database either by setting the `pa_format_type` configuration variable or by passing the runtime switch.

  For more information on how to set the FSDB dumping libraries, see Appendix B section in Linking Novas Files with Simulators and Enabling FSDB Dumping guide available at:

  $VERDI_HOME/doc/linking_dumping.pdf

  Configuration Variable Setting:

  Set `pa_xml_generation_enable` parameter of USB configuration class `svt_usb_configuration` to enable the generation of PA files.

  For Example:

  ```
  <usb_xmtr_agent_configuration>.usb_cfg.pa_xml_generation_enable = 1
  ```

  Similarly for USB receiver:

  ```
  <usb_rcvr_agent_configuration>.usb_cfg.pa_xml_generation_enable = 1
  ```

6.1.2 Invoking Protocol Analyzer

Perform the following steps to invoke Protocol Analyzer in interactive or post-processing mode:

Post-processing Mode

- Load the transaction dump data and issue the following command to invoke the GUI:

```
verdi -ssf <dump.fsdb> -lib work.lib++
```

- In Verdi, navigate to Tools->Transaction Debug-> Transaction and Protocol Analyzer.

Interactive Mode

- Issue the following command to invoke Protocol Analyzer in an interactive mode:

```
<simv> -gui=verdi
```

You can invoke the Protocol Analyzer as shown above through Verdi. The Protocol Analyzer transaction view gets updated during the simulation.

6.1.3 Documentation

The documentation for Protocol Analyzer is available at the following path:

$VERDI_HOME/doc/Verdi_Transaction_and_Protocol_Debug.pdf

6.1.4 Limitations

Interactive support is available only for VCS.

7 Troubleshooting

This chapter provides some useful information that can help you troubleshoot common issues that you may encounter while using the USB VIP. This chapter discusses the following topics:

- Using Trace Files for Debugging
- Enabling Tracing
- Setting Verbosity Levels
- Elevating or Demoting Messages
- Disabling Specific In-line Checking

7.1 Using Trace Files for Debugging

Trace files contain information about the objects that have been transmitted across a particular channel. There are different types of trace files such as:

- Data trace files - “Data” objects (such as symbols) from 'phys' transactor are available in data trace files.
- Packet trace files - “Packet” objects from 'link' transactor are available in packet trace files.
- Transaction trace files - “Transaction” objects from 'prot' transactor are available in transaction trace files.
- Transfer trace files - “Transfer” objects from 'prot' transactor are available in 'packet, 'transaction', and 'transfer' trace files.

There are separate trace files for transmit (TX) and receive (RX) directions.

In a typical VIP sub-environment configuration, where three transactors (prot, link and phys) are included with SS operation, the following trace files can be generated:

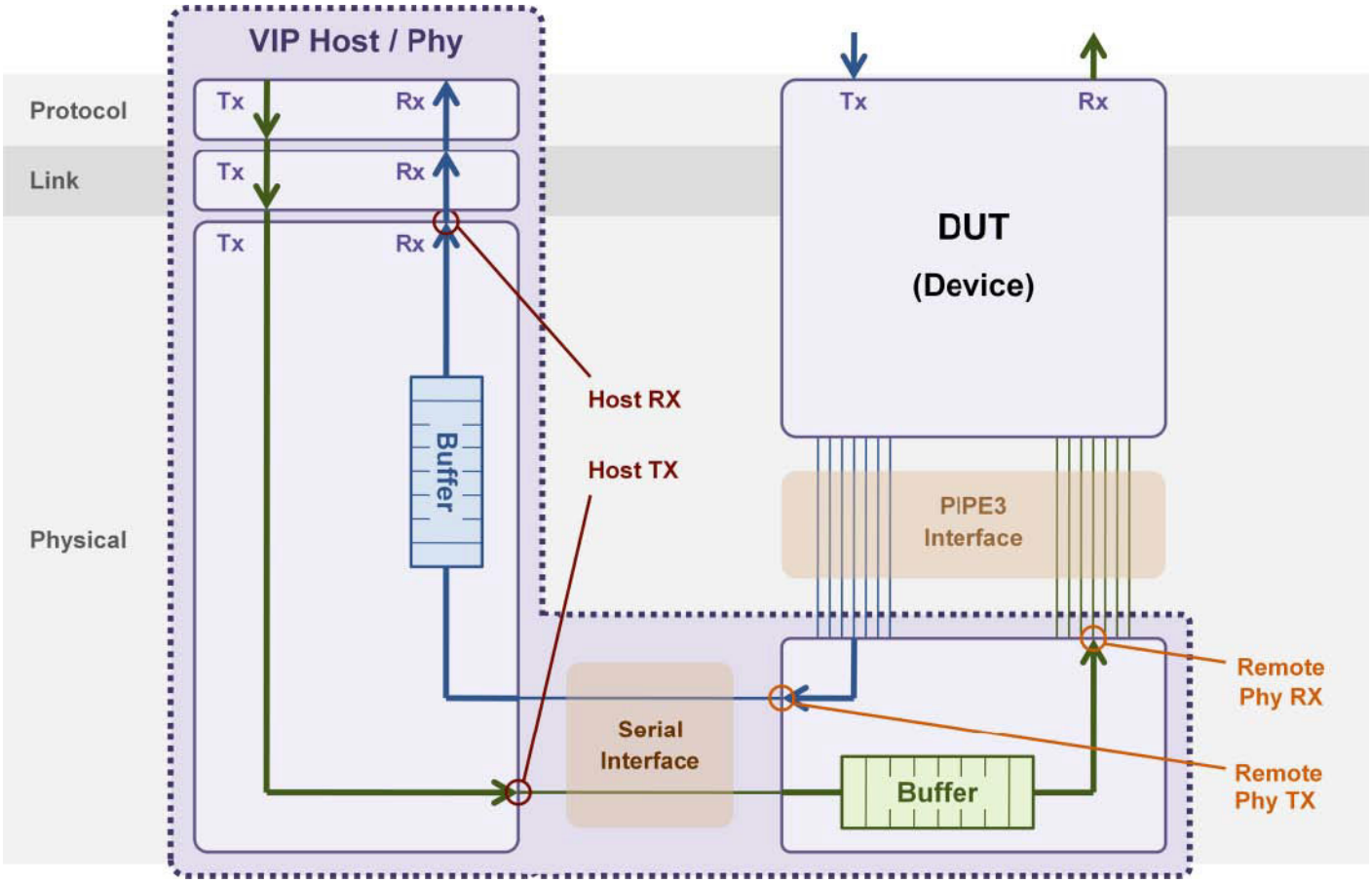
```
vip_subenv.phys.SS.RX.data_trace
vip_subenv.phys.SS.TX.data_trace
vip_subenv.link.SS.RX.packet_trace
vip_subenv.link.SS.TX.packet_trace
vip_subenv.prot.SS.transaction_trace
vip_subenv.prot.SS.transfer_trace
```

In a VIP sub-environment configuration, where a fourth transactor ('phys' representing remote Phy) is included, two additional data trace files can be generated. The example names of data trace files from remote PHY are as follows:

```
vip_subenv.remote_phys.SS.RX.data_trace
vip_subenv.remote_phys.SS.TX.data_trace
```

Figure 7-1 shows the output channels of each transactor, where objects are captured for corresponding trace files content.

Figure 7-1 An Environment with a Device DUT Connected to a Host VIP and Remote PHY



These files are created in the directory in which the simulator was invoked and adhere to the following naming convention:

Data Trace Files

```
<subenv instance name>.<physical layer location>.<speed>.<direction>.data_trace
```

where:

`sub-env instance name:` is taken from the instance name of the corresponding VIP sub-environment.

`physical layer location:` is either “phys” or “remote_phys” depending upon whether the data is associated with the “local” physical layer, or the “remote” physical layer.

`speed:` Indicates the speed. This can be SS, HS, or FS.

`direction:` is either “TX” or “RX” corresponding to “transmit” and “receive” respectively.

Packet Trace Files

```
<subenv instance name>.link.<speed>.<direction>.packet_trace
```

where:

`sub-env instance name:` is taken from the instance name of the corresponding VIP sub-environment.

`speed:` Indicates the speed. This can be SS, HS, or FS.

`direction:` is either “TX” or “RX” corresponding to “transmit” and “receive” respectively.

Transaction Trace Files

```
<subenv instance name>.prot.<speed>.transaction_trace
```

where:

`sub-env instance name:` is taken from the instance name of the corresponding VIP sub-environment.

`speed:` Indicates the speed. This can be SS, HS, or FS.

Transfer Trace Files

```
<subenv instance name>.prot.<speed>.transfer_trace
```

where:

`sub-env instance name:` is taken from the instance name of the corresponding VIP sub-environment.

`speed:` Indicates the speed. This can be SS, HS, or FS.

For example, if the instance name of the VIP sub-environment is “vip_subenv”. then the simulation of this environment produces the files listed in Table 7-1.

Table 7-1 Generated Trace Files

Trace File Type	Trace File Name

Data trace files	`vip_subenv.phys.SS.RX.data_trace`
	`vip_subenv.phys.SS.TX.data_trace`
	`vip_subenv.remote_phys.SS.RX.data_trace`
	`vip_subenv.remote_phys.SS.TX.data_trace`

Packet trace file	`vip_subenv.link.SS.RX.packet_trace` `vip_subenv.link.SS.TX.packet_trace`
Transfer trace file	`vip_subenv.prot.SS.transfer_trace`
Transaction trace file	`vip_subenv.prot.SS.transaction_trace`

7.2 Enabling Tracing

Tracing is enabled or disabled through the “enable_phys_tracing” variable that is defined in the “svt_usb_subenv_configuration” class. The default value of this variable is “0”, which means that tracing is disabled by default.

To enable tracing, set the value of this variable to “1” in the derived class that is used in your VIP sub-environment. You can set additional values (such as 2,3, and 4) depending on the trace requirements.

The next time that the environment is simulated, data trace files are generated according to the configuration. The following code snippets illustrate how tracing has been enabled assuming vip_cfg is of svt_usb_subenv_configuration class.

Code Sample:

```
integer cfg_handle;
// Get a handle to the configuration.
`GET_DATA_PROP_W_CHECK(usb_host, `SVT_CMD_NULL_HANDLE, "cfg", cfg_handle, 0, 1,
`FATAL_SEV)
// Enable generation of nested transfer, transaction and packet files
`SET_DATA_PROP_W_CHECK(usb_host, cfg_handle, "enable_transfer_tracing", 3, 0, 1,
`FATAL_SEV)

// Get a handle to the remote configuration, in case VIP's remote PHY is included.
`GET_DATA_PROP_W_CHECK(usb_host, `SVT_CMD_NULL_HANDLE, "remote_cfg", remote_cfg_handle,
0, 1, `FATAL_SEV)
// Enable generation of Data Trace files for the remote configuration of VIP
`SET_DATA_PROP_W_CHECK(usb_host, remote_cfg_handle, "enable_phys_tracing", 1, 0, 1,
`FATAL_SEV)

// Above mentioned macros are assumed to be defined earlier (as shown below), using
VIP's get_data_prop()/set_data_prop() commands as shown earlier.
// Macros include checks for the returned validity flag, and error response if the
propert/action was not valid.
```

Macro Definitions:

```
`define
GET_DATA_PROP_W_CHECK(transactor,cfghandle,propname,propval,propidx,expisvalid,failacti
on) \
  transactor.get_data_prop(is_valid, cfghandle, propname, propval, propidx); \
  if (is_valid !=expisvalid) begin \
    $display("%m: %s - is_valid is %0b, expected %0b (transactor.get_data_prop(is_valid,
cfghandle, '%s', %0hh, %0d))", \
            (failaction == `FATAL_SEV) ? "FATAL" : (failaction == `ERROR_SEV) ? "ERROR"
: "WARNING", \
             is_valid, expisvalid, propname, propval,propidx); \
    if (failaction == `FATAL_SEV) begin \
      $display("%m: FATAL - Aborting Simulation..."); \
      $finish; \
    end \
  end

`define
SET_DATA_PROP_W_CHECK(transactor,cfghandle,propname,propval,propidx,expisvalid,failacti
on) \
  transactor.set_data_prop(is_valid, cfghandle, propname, propval, propidx); \
  if (is_valid !=expisvalid) begin \
    $display("%m: %s - is_valid is %0b, expected %0b (transactor.set_data_prop(is_valid,
cfghandle, '%s', %0hh, %0d))", \
```

```
            (failaction == `FATAL_SEV) ? "FATAL" : (failaction == `ERROR_SEV) ? "ERROR"
: "WARNING", \
             is_valid, expisvalid, propname, propval,propidx); \
    if (failaction == `FATAL_SEV) begin \
      $display("%m: FATAL - Aborting Simulation..."); \
      $finish; \
    end \
  end
```

Table 7-2 Descriptions of Macro Arguments

Macro Arguments	Description
transactor	User-provided instance names used to instantiate the VIP module
cfghandle	Integer handle that references the configuration or data object
propname	Data object property name such as `speed`, `xfer_type`, or `ustream_id`
propval	if GET, this is the returned property value, if SET it is the value that needs to be set
propidx	Index used if propname refers to an array type property
expisvalid	The expected 'is_valid' return value. Under normal conditions, this value is 1. Under error conditions, this value is 0.
failaction	Determines how to treat the action if the `is_valid` return value is 0. This may be `FATAL_SEV or `WARNING_SEV. `WARNING_SEV causes the test to ignore the failure whereas `FATAL_SEV causes the test to exit the simulation.

Note All the above code samples can be used in `start_cfg()` either before VIP is constructed (in build function) or before VIP's configuration is changed dynamically after VIP is constructed (using `change_xactor_config`).
For additional usage code, see the SV-VMM test examples provided along with the VIP.

7.3 Setting Verbosity Levels

You can set VIP debug verbosity levels either in the testbench or as an option during run-time.

To set the verbosity level in the testbench, use the following logging commands:

- log_get_verbosity
- log_set_verbosity
- log_enable_types
- log_disable_types

To set verbosity levels through the command-line, use the following command:

```
"+vip_verbosity=<vip_subunit_1_name>:<verbosity_level_1>,<vip_subunit_2_name>:<verbosit
y_level_2>…."
```

7.3.1 To enable the specified severity to specific sub-classes of VIP

```
"+vip_verbosity=<vip_subunit_1_name>:<verbosity_level_1>,<vip_subunit_2_name>:
<verbosity_level_2>…."
```

Use Example: VCS

```
vcs <other run time options> -R
+vip_verbosity=svt_usb_link_ss_lcm:verbose,svt_usb_link_ss_ltssm_base:debug,
svt_usb_physical:debug,svt_usb_link_ss_tx:debug
```

Use Example: simv

```
./simv <other run time options>
+vip_verbosity=svt_usb_link_ss_lcm:verbose,svt_usb_link_ss_ltssm_base:debug,
svt_usb_physical:debug,svt_usb_link_ss_tx:debug
```

7.3.1.1 Valid VIP Sub-Units

Table 7-3 lists the valid VIP sub-units that you can use.

Table 7-3 Valid VIP Sub-Units

Layer	Sub-unit or Component
Entire sub-environment	`svt_usb_subenv`
Protocol Layer - SS:	• `svt_usb_protocol` • `svt_usb_protocol_block` • `svt_usb_protocol_device` • `svt_usb_protocol_processor` • `svt_usb_protocol_ss_host` • `svt_usb_protocol_ss_host_non_isoc_ep_processor` • `svt_usb_protocol_ss_host_isoc_ep_processor` • `svt_usb_protocol_ss_device` • `svt_usb_protocol_ss_device_non_isoc_ep_processor` • `svt_usb_protocol_ss_device_isoc_ep_processor` • `svt_usb_protocol_scheduler` • `svt_usb_protocol_host_scheduler` • `svt_usb_protocol_device_scheduler` • `svt_usb_protocol_ss_host_isoc_ep_processor` • `svt_usb_protocol_ss_device_isoc_ep_processor` • `svt_usb_protocol_ss_lmp_processor` • `svt_usb_protocol_ss_itp_processor`

Table 7-3 Valid VIP Sub-Units

Layer	Sub-unit or Component
Protocol Layer - USB 2.0	• svt_usb_protocol_processor • svt_usb_protocol_20_host_non_isoc_ep_processor • svt_usb_protocol_20_host_isoc_ep_processor • svt_usb_protocol_20_device_non_isoc_ep_processor • svt_usb_protocol_20_device_isoc_ep_processor • svt_usb_protocol_block • svt_usb_protocol_20_host • svt_usb_protocol_20_device • svt_usb_protocol_20_lpm_processor • svt_usb_protocol_20_sof_processor
Link layer-SS:	• svt_usb_link_ss_tx • svt_usb_link_ss_rx • svt_usb_link_ss_ltssm_base • svt_usb_link_ss_lcm
Link layer-USB 2.0	• svt_usb_link_20 • svt_usb_link_20_device_a_sm • svt_usb_link_20_device_b_sm • svt_usb_link_20_timer
Physical Layer	svt_usb_physical

7.4 Elevating or Demoting Messages

Sometimes, for example in tests with error injection, you may want to demote or elevate specific VIP messages.

If you want to modify debug or verbose messages, you have to set the appropriate controls in the svt_usb_configuration class as follows:

```
/**
   * Controls the ability to elevate physical level debug and verbose messages
   * using the vmm_log::modify() method. This capability is enabled for physical
   * level messages by setting this field to 1.
   */
  bit phys_log_modify_enable = 0;

  /**
   * Controls the ability to elevate link level debug and verbose messages
   * using the vmm_log::modify() method. This capability is enabled for link
   * level messages by setting this field to 1.
   */
  bit link_log_modify_enable = 0;

  /**
```

```
   * Controls the ability to elevate protocol level debug and verbose messages
   * using the vmm_log::modify() method. This capability is enabled for protocol
   * level messages by setting this field to 1.
   */
  bit prot_log_modify_enable = 0;
```

Message severity can be demoted using the log_modify command as shown in the following example:

```
reg [80*12-1:0] msg_text_reg;
string msg_str;
integer  modify_handle, actual_msg_typ, actual_msg_sev, modified_msg_typ,
modified_msg_sev, msg_handling;
….
msg_text_reg =  "Check if the SS Rx calculates an appropriate SKP to other-symbol
ratio";
msg_str = string'(msg_text_reg);
usb_host.log_msg_val(is_valid, actual_msg_typ, "type", "ALL_TYPS");
usb_host.log_msg_val(is_valid, actual_msg_sev, "severity", "ALL_SEVS");
usb_host.log_msg_val(is_valid, modified_msg_typ, "type", "DEBUG_TYP");
usb_host.log_msg_val(is_valid, modified_msg_sev, "severity", "DEBUG_SEV");
usb_host.log_msg_val(is_valid, msg_handling, "handling", "CONTINUE");
usb_host.log_modify(modify_handle,
                    actual_msg_typ,
                    actual_msg_sev,
                    msg_str,
                    modified_msg_typ,
                    modified_msg_sev,
                    msg_handling);
```

7.5 Disabling Specific In-line Checking

To disable specific in-line checks, you can use the log_modify command to demote the check.

A
Reporting Problems

This chapter outlines the process for working through and reporting VIP transactor issues encountered in the field. It describes the data you must submit when a problem is initially reported to Synopsys. After a review of the initial information, Synopsys may decide to request adjustments to the information being requested, which is the focus of the next section. This section outlines the process for working through and reporting problems. It shows how to use Debug Automation to enable all the debug capabilities of any VIP. In addition, the VIP provides a case submittal tool to help you pack and send all pertinent debug information to Synopsys Support.

A.1 Debug Automation

Every Synopsys model contains a feature called "debug automation". It is enabled through *svt_debug_opts* plusarg. The Debug Automation feature allows you to enable all relevant debug information. The following are critical features of debug automation:

- Enabled by the use of a command line run-time plusarg.
- Can be enabled on individual VIP instances or multiple instances using regular expressions.
- Enables debug or verbose message verbosity:
  - The timing window for message verbosity modification can be controlled by supplying `start_time` and `end_time`.
- Enables at one time any, or all, standard debug features of the VIP:
  - Transaction Trace File generation
  - Transaction Reporting enabled in the transcript
  - PA database generation enabled
  - Debug Port enabled
  - Optionally, generates a file name *svt_model_out.fsdb* when Verdi libraries are available

When the Debug feature is enabled, then all VIP instances that are enabled for debug will have their messages routed to a file named *svt_debug.transcript*.

A.2 Enabling and Specifying Debug Automation Features

Debug Automation is enabled through the use of a run-time plusarg named +*svt_debug_opts. T*his plusarg accepts an optional string-based specification to control various aspects Debug Automation. If this

command control specification is not supplied, then the feature will default to being enabled on all VIP instances with the default options listed as follows:

Note the following about the plusarg:

- The command control string is a comma separated string that is split into the multiple fields.
- All fields are optional and can be supplied in any order.

The command control string uses the following format (white space is disallowed):

```
inst:<inst>,type:<string>,feature:<string>,start_time:<longint>,end_time:<longint>,verbosity:<string>
```

The following table explains each control string:

Table A-1 Control Strings for Debug Automation plusarg

Field	Description
inst	Identifies the VIP instance to apply the debug automation features. Regular expressions can be used to identify multiple VIP instances. If this value is not supplied, and if a type value is not supplied, then the debug automation feature will be enabled on all VIP instances.
type	Identifies a class type to apply the debug automation features. When this value is supplied then debug automation will be enabled for all instances of this class type.
feature	Identifies a sub-feature that can be defined by VIP designers to identify smaller grouping of functionality that is specific to that title. The definition and implementation of this field is left to VIP designers, and by default it has no effect on the debug automation feature. (Specific to VIP titles)
start_time	Identifies when the debug verbosity settings will be applied. The time must be supplied in terms of the timescale that the VIP is compiled. If this value is not supplied, then the verbosity settings will be applied at time zero.
end_time	Identifies when the debug verbosity settings will be removed. The time must be supplied in terms of the timescale that the VIP is compiled. If this value is not supplied, then the debug verbosity remains in effect until the end of the simulation.
verbosity	Message verbosity setting that is applied at the `start_time`. Two values are accepted in all methodologies: DEBUG and VERBOSE. UVM and OVM users can also supply the verbosity that is native to their respective methodologies (UVM_HIGH/UVM_FULL and OVM_HIGH/OVM_FULL). If this value is not supplied then the verbosity defaults to DEBUG/UVM_HIGH/OVM_HIGH. When this feature is enabled, then all VIP instances that are enabled for debug will have their messages routed to a file named `svt_debug.transcript`.

Examples:

Enable on all VIP instances with default options:

```
+svt_debug_opts
```

Enable on all instances:

- containing the string "endpoint" with a verbosity of UVM_HIGH
- starting at time zero (default) until the end of the simulation (default):

```
+svt_debug_opts=inst:/.*endpoint.*/,verbosity:UVM_HIGH
```

Enable on all instances:

- starting at time 1000 until time 1500:

```
+svt_debug_opts=start_time:1000,end_time:1500,verbosity:VERBOSE
```

Enable debug feature on all instances using default options:

- By setting the macro SVT_DEBUG_OPTS to 1 in the command line, the debug feature is enabled on all instances using default options. The macro will enable the XMLs and Trace files.

```
gmake <testname> SVT_DEBUG_OPTS=1 PA=FSDB
```

- The SVT_DEBUG_OPTS option is available through the installed VIP examples, but if required, in customer environments, then a similar feature should be added to their environment.
- The PA=FSDB option is available in public examples and is required to enable Verdi libraries, and that when this option is used, then the Debug Opts file will record VIP activity to a file named `svt_model_log.fsdb`.
- In addition, the SVT Automated Debug feature will enable waveform generation to an FSDB file, if the Verdi libraries are available. When enabled this feature, it should cause the simulator to dump waveform information only for the VIP interfaces.

When this feature is enabled then all VIP instances that have been enabled for debug will have their messages routed to a file named `svt_debug.transcript`.

A.3 Debug Automation Outputs

The Automated Debug feature generates a *svt_debug.out* file. It records important information about the debug feature itself, and data about the environment that the VIPs are operating in. This file records the following information:

- The compiled timeunit for the SVT package
- The compiled timeunit for each SVT VIP package
- Version information for the SVT library
- Version information for each SVT VIP
- Every SVT VIP instance, and whether the VIP instance has been enabled for debug
- For every SVT VIP enabled for debug, a list of configuration properties that have been modified to enable debug will be listed
- A list of all methodology phases will be recorded, along with the start time for each phase

The following are the output files generated:

- *svt_debug.out*: It records important information about the debug feature itself, and data about the environment that the VIPs are operating. One file is optionally created when this feature is enabled, depending on if the Verdi libraries are available.
- *svt_debug.transcript*: Log files generated by the simulation run.
- *transaction_trace*: Log files that records all the different transaction activities generated by VIPs.
- *svt_model_log.fsdb*: Contains PA FSDB information (if the VIP supports this), and which contains other recorded activity. The additional information records signal activity associated with the VIP interface, TLM input (through SIPP ports), other TLM output activity, configurations applied to the VIP, and all callback activity (recorded by before and after callback execution).

A.4 FSDB File Generation

To enable FSDB writing capabilities, the simulator compile-time options and environment must be updated to enable this. The steps to enable this are specific to the simulator being used (the {LINUX/LINUX64} label needs to be replaced based on the platform being used). The ability to write to an FSDB file requires that the user supplies the Verdi dumper libraries when they compile their testbench. If these are not supplied then the VIP will not be enabled to generate the *svt_model_log.fsdb* file.

A.4.1 VCS

The following must be added to the compile-time command:

```
-debug_access
```

You can dump the transaction database either by setting the `pa_format_type` configuration variable or by passing the runtime switch.

For more information on how to set the FSDB dumping libraries, see Appendix B section in Linking Novas Files with Simulators and Enabling FSDB Dumping guide available at:

$VERDI_HOME/doc/linking_dumping.pdf

A.4.2 Questa

The following must be added to the compile-time command:

```
+define+SVT_FSDB_ENABLE -pli novas_fli.so
```

A.4.3 Incisive

The following must be added to the compile-time command:

```
+define+SVT_FSDB_ENABLE -access +r
```

A.5 Initial Customer Information

Follow these steps when you call the Synopsys Support Center:

1. Before you contact technical support, be prepared to provide the following:
   - A description of the issue under investigation.
   - A description of your verification environment.

Enable the Debug Opts feature. For more information, see the Debug Automation.

A.6 Sending Debug Information to Synopsys

To help you debug testing issues, follow the given instructions to pack all pertinent debug information into one file which you can send to Synopsys (or to other users in your company):

1. Create a description of the issue under investigation. Include the simulation time and bus cycle of the failure, as well as any error or warning messages that are part of the failure.
2. Create a description of your verification environment. Assemble information about your simulation environment, making sure to include:
   - OS type and version
   - Testbench language (SystemVerilog or Verilog)
   - Simulator and version

- DUT languages (Verilog)

3. Use the VIP case submittal tool to pack a file with the appropriate debug information. It has the following usage syntax:

   ```
   $DESIGNWARE_HOME/bin/snps_vip_debug [-directory <path>]
   ```

   The tool will generate a “<username>.<uniqid>.svd” file in the current directory. The following files are packed into a single file:

   - FSDB
   - HISTL
   - MISC
   - SLID
   - SVTO
   - SVTX
   - TRACE
   - VCD
   - VPD
   - XML

   If any one of the above files are present, then the files will be saved in the "<username>.<uniqid>.svd" in the current directory. The simulation transcript file will not be part of this and it will be saved separately.

   The -directory switch can be specified to select an alternate source directory.

4. You will be prompted by the case submittal tool with the option to include additional files within the SVD file. The simulation transcript files cannot be automatically identified and it must be provided during this step.
5. The case submittal tool will display options on how to send the file to Synopsys.

A.7 Limitations

Enabling DEBUG or VERBOSE verbosity is an expensive operation, both in terms of runtime and disk space utilization. The following steps can be used to minimize this cost:

- Only enable the VIP instance necessary for debug. By default, the `+svt_debug_opts` command enables Debug Opts on all instances, but the `'inst'` argument can be used to select a specific instance.
- Use the `start_time` and `end_time` arguments to limit the verbosity changes to the specific time window that needs to be debugged.