

VC Verification IP USB HUB UVM User Guide

Version K-2015.09-1, September 2015



Copyright Notice and Proprietary Information

© 2015 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

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/Pages/Trademarks.aspx>.

All other product or company names may be trademarks of their respective owners.

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.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

| | |
|--|----|
| Preface | 7 |
| Manual Organization | 7 |
| Web Resources | 7 |
| Customer Support | 8 |
| Chapter 1 | |
| Introduction | 9 |
| 1.1 Overview | 9 |
| 1.2 Product Overview | 10 |
| 1.3 USB HUB Feature Support | 10 |
| 1.4 UVM Support Provided by USB VIP | 11 |
| 1.4.1 Data Flow Support | 12 |
| 1.4.2 Interface Options | 16 |
| 1.4.3 Interface File Features | 16 |
| Chapter 2 | |
| General Concepts | 17 |
| 2.1 Introduction to UVM | 17 |
| 2.2 USB VIP in a UVM Environment | 18 |
| 2.2.1 Base Classes | 18 |
| 2.2.2 UVM Components | 19 |
| 2.2.3 USB VIP Objects | 20 |
| 2.2.4 Interfaces and Modports | 26 |
| 2.2.5 Constraints | 27 |
| 2.2.6 Factories | 29 |
| 2.2.7 Messages | 30 |
| Chapter 3 | |
| The USB Agent Overview | 31 |
| 3.1 UVM Hub Agent Stack | 31 |
| 3.2 Stimulus Objects | 32 |
| 3.3 Reporting and Tracking Objects | 32 |
| 3.3.1 Checks | 32 |
| 3.3.2 Reporting | 33 |
| 3.4 UVM Events | 33 |
| Chapter 4 | |
| USB Host/Device Verification | 35 |
| 4.1 Overview | 35 |
| 4.2 Protocol Component | 35 |
| 4.2.1 Protocol Layer Feature Support | 35 |

| | | |
|-------------------------------|---|-----|
| 4.2.2 | Protocol Component Ports | 36 |
| 4.2.3 | Data Objects | 36 |
| 4.2.4 | Protocol Component Modes | 37 |
| 4.2.5 | Data Transformation Objects | 37 |
| 4.2.6 | Protocol Component Status | 42 |
| 4.2.7 | Protocol Component SuperSpeed Link Callback, Factory, and Event Flows | 42 |
| 4.2.8 | Protocol Component 2.0 Link Callback, Factory, and UVM Event Flows | 55 |
| 4.3 | Link Component | 65 |
| 4.3.1 | Link Layer Feature Support | 65 |
| 4.3.2 | Link Component Ports | 65 |
| 4.3.3 | Data Objects | 66 |
| 4.3.4 | Data Transformation Objects | 67 |
| 4.3.5 | Link Component SuperSpeed Link Callback, Factory, and UVM Event Flows | 69 |
| 4.3.6 | SuperSpeed Packet Chronology | 71 |
| 4.3.7 | Related Topics About Link Component | 89 |
| 4.4 | Physical Component | 90 |
| 4.4.1 | Physical Layer Feature Support | 90 |
| 4.4.2 | Data Flow Support | 91 |
| 4.4.3 | Interface Options | 94 |
| 4.4.4 | Interface File Features | 94 |
| 4.4.5 | Information Transformation Objects | 94 |
| 4.4.6 | Exception Support | 95 |
| 4.4.7 | UVM Event Support | 96 |
| 4.4.8 | Physical Component Callbacks | 97 |
| 4.4.9 | Related Topics About Physical Component | 97 |
| Chapter 5 | | |
| Using the USB Verification IP | | 99 |
| 5.1 | Introduction | 99 |
| 5.2 | SystemVerilog UVM Example Testbenches | 100 |
| 5.2.1 | For example (Speed = SS), | 100 |
| 5.2.2 | Configuring Endpoints: | 101 |
| 5.3 | Creating Transactions Using UVM Sequencers | 103 |
| 5.4 | SuperSpeed Low Power Entry Support | 103 |
| 5.4.1 | Overview | 103 |
| 5.4.2 | Automatic Low-Power Entry Attempts | 104 |
| 5.4.3 | Automatic Low-Power Entry for Upstream Ports | 105 |
| 5.4.4 | Testbench-Initiated Low-Power Entry Attempts | 105 |
| 5.5 | Implementing Functional Coverage | 106 |
| 5.5.1 | Default Functional Coverage | 106 |
| 5.5.2 | Covergroup Organization | 107 |
| 5.5.3 | Range Bins | 108 |
| 5.5.4 | Default Functional Coverage Class Hierarchy | 108 |
| 5.5.5 | Coverage Callback Classes | 109 |
| 5.5.6 | Using Functional Coverage | 110 |
| 5.5.7 | Using the High-Level Verification Plans | 111 |
| 5.6 | Executing Aligned Transfers | 111 |
| 5.6.1 | VIP Acting as a Host | 111 |
| 5.6.2 | VIP Acting as a Device | 112 |
| 5.7 | UTMI+ Support | 114 |

| | | |
|---------------------------------|---|-----|
| 5.7.1 | Port Interface | 114 |
| 5.7.2 | Configuring the UTMI+ Interface | 114 |
| 5.7.3 | Error Injection | 115 |
| 5.7.4 | Attach / Detach | 115 |
| 5.7.5 | L1, L2 (Suspend), Resume, Remote wake-up | 115 |
| 5.7.6 | UTMI+ Messages | 115 |
| 5.8 | USB 2.0 OTG Support | 117 |
| 5.8.1 | OTG Interface Signals | 117 |
| 5.8.2 | Session Request Protocol | 119 |
| 5.8.3 | Role Swapping Using the HNP Protocol | 123 |
| 5.8.4 | Attach Detection Protocol | 133 |
| 5.9 | HSIC Overview | 138 |
| 5.9.1 | Supported HSIC Features | 138 |
| 5.9.2 | Unsupported HSIC Features | 138 |
| 5.9.3 | Configuration Parameters | 138 |
| 5.9.4 | Transactions | 139 |
| 5.9.5 | Exceptions | 140 |
| 5.9.6 | HSIC Interface | 140 |
| 5.9.7 | HSIC Signal Interface | 141 |
| 5.9.8 | Callbacks | 141 |
| 5.9.9 | Notifications | 142 |
| 5.9.10 | Factories | 142 |
| 5.9.11 | Shared Status | 142 |
| 5.9.12 | Usage | 142 |
| 5.10 | Using the Passive Monitor | 155 |
| 5.10.1 | VIP as Host Passive Monitor | 155 |
| 5.10.2 | VIP as Device Passive Monitor | 156 |
| 5.11 | Using Test Mode | 157 |
| 5.11.1 | Entering Test Mode | 157 |
| 5.11.2 | Verifying TEST_PACKET | 157 |
| 5.11.3 | Verifying TEST_J | 159 |
| 5.11.4 | Verifying TEST_K | 159 |
| 5.11.5 | Verifying TEST_SE0_NAK | 160 |
| 5.11.6 | Exiting test_mode on Downstream Facing Ports | 161 |
| 5.11.7 | Test Mode Notifications | 161 |
| 5.11.8 | Test Mode Configuration Members | 161 |
| 5.12 | SuperSpeed InterChip Physical (SSIC) Usage (Beta Feature) | 162 |
| 5.12.1 | Configuring the VIP for a DUT Without a SSIC Physical Layer | 162 |
| 5.12.2 | Configuring the VIP for a DUT With a SSIC Physical Layer | 163 |
| 5.13 | SystemVerilog UVM Example Testbenches | 164 |
| Chapter 6 | | |
| Verification Topologies | | 165 |
| 6.1 | USB VIP Downstream and DUT Device Controller | 165 |
| 6.2 | USB VIP Device and DUT Host | 171 |
| Chapter 7 | | |
| Integration Steps for HUB Model | | 177 |
| 7.1 | USB HUB Model Integration | 177 |

| | |
|--|-----|
| Chapter 8 | |
| VIP Tools | 183 |
| 8.1 Using Native Protocol Analyzer for Debugging | 183 |
| 8.1.1 Introduction | 183 |
| 8.1.2 Prerequisites | 183 |
| 8.1.3 Invoking Protocol Analyzer | 184 |
| 8.1.4 Documentation | 184 |
| 8.1.5 Limitations | 184 |
| Chapter 9 | |
| Troubleshooting | 185 |
| 9.1 Using Trace Files for Debugging | 185 |
| 9.2 Enabling Tracing | 187 |
| 9.3 Setting Verbosity Levels | 188 |
| 9.3.1 Setting Verbosity in the Testbench | 188 |
| 9.3.2 Setting Verbosity During Run Time | 189 |
| 9.4 Disabling Specific In-line Checking | 191 |
| Appendix A | |
| Reporting Problems | 193 |
| A.1 Introduction | 193 |
| A.2 Initial Customer Information | 193 |
| A.2.1 Describing Your Issue | 194 |
| A.2.2 Describing Your Verification Environment Description | 194 |
| A.2.3 Creating a Waveform File | 194 |
| A.2.4 Creating a Log File | 194 |
| A.2.5 Identifying an Instance | 195 |
| A.2.6 Impact of Generating Verbose Logs | 196 |
| A.3 Debug Automation | 196 |
| A.3.1 Enabling and Specifying Debug Automation Features | 197 |
| A.3.2 Debug Automation Outputs | 198 |
| A.3.3 Sending Debug Information to Synopsys | 198 |

Preface

About This Manual

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

For information related to installing the VIP, see the VC VIP for USB Install Guide.

Manual Organization

The chapters of this databook are organized as follows:

- ❖ Chapter 1, “[Introduction](#)”, introduces the VC VIP for USB and its features.
- ❖ Chapter 2, “[General Concepts](#)”, introduces the USB VIP within a UVM environment and describes the data objects and components that comprise the VIP.
- ❖ Chapter 3, “[The USB Agent Overview](#)”, describes the USB VIP agent within a UVM environment and describes the data objects and components that comprise the VIP.
- ❖ Chapter 4, “[USB Host/Device Verification](#)”, describes USB HUB VIP components.
- ❖ Chapter 5, “[Using the USB Verification IP](#)”, describes the USB HUB VIP and how to use it.
- ❖ Chapter 6, “[Verification Topologies](#)”, presents agent – testbench configurations that you can use as templates.
- ❖ Chapter 8, “[VIP Tools](#)”, describes the VIP tool support in USB VIP.
- ❖ Chapter 9, “[Troubleshooting](#)” explains how to generate trace files and customize logging
- ❖ Appendix A, “[Reporting Problems](#)”, outlines the process for working through and reporting USB HUB VIP component issues.

Web Resources

- ❖ Documentation through SolvNet: <https://solvnet.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:

- ❖ Enter a call through SolvNet.
 - ◆ Go to <http://solvnet.synopsys.com/EnterACall> and provide the requested information, including:
 - ❖ Product: **USB HUB Verification IP**
 - ❖ Sub Product: **usb_hub_svt**
 - ❖ Tool Version: **K-2015.09-1**
 - ❖ Fill in the remaining fields according to your environment and your issue.
 - ◆ If applicable, provide the information noted in Appendix A, “Reporting Problems” on page 193.
- ❖ Send an e-mail message to support_center@synopsys.com.
 - ◆ Include the Product name, Sub Product name, and Tool Version (as noted above) in your e-mail so it can be routed correctly.
 - ◆ If applicable, provide the information noted in Appendix A, “Reporting Problems” on page 193.
- ❖ 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:
<http://www.synopsys.com/Support/GlobalSupportCenters>

1

Introduction

1.1 Overview

The VC VIP for USB HUB 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 testbenches that comply with the Universal Verification Methodology (UVM). This approach leverages advanced verification technologies and tools that provide:

- ❖ Protocol functionality and abstraction
- ❖ Constrained random verification
- ❖ Functional coverage
- ❖ Rapid creation of complex tests
- ❖ Proven testbench architecture that provides maximum reuse, scalability and modularity
- ❖ Proven verification approach and methodology
- ❖ Transaction-level, self-checking tests
- ❖ Object oriented interface that allows OOP techniques

This document assumes that you are familiar with USB, object oriented programming, SystemVerilog, and UVM.

See also:

- ❖ [Universal Serial Bus 3.0 Specification, Revision 1.0, November 12, 2008](#)
- ❖ [PHY Interface for the PCI Express™ and USB Architectures, Version 3.00, Intel Corp.](#)

For the VC VIP for USB class reference, see:

[\\$DESIGNWARE_HOME/vip/svt/usb_svt/latest/doc/usb_svt_public.uvm/html/index.html](#)

For a walk-through example that demonstrates basic UVM concepts, see the QuickStart (an HTML-based walk-through of the basic example that is included with the USB HUB. After the USB HUB is installed, you can see the QuickStart HTML at:

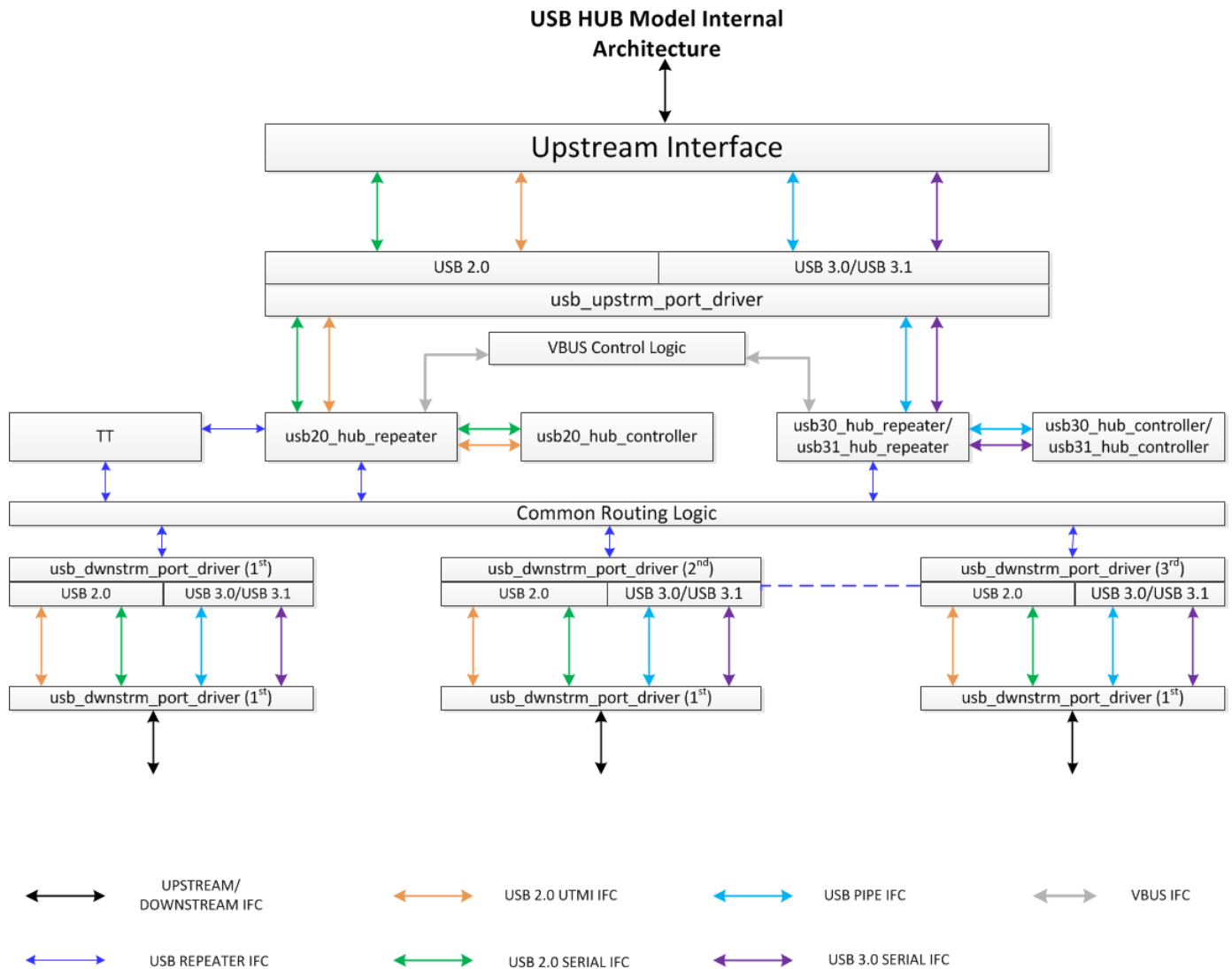
[\\$DESIGNWARE_HOME/vip/svt/usb_svt/latest/examples/sverilog/index.html](#)

For a complete list of examples included with the USB HUB, see “[SystemVerilog UVM Example Testbenches](#)” on page 160.

1.2 Product Overview

The USB HUB VIP is a suite of UVM-based verification components that are compatible for use with SystemVerilog-Compliant testbenches.

Figure 1-1 USB HUB Architecture



1.3 USB HUB Feature Support

The USB HUB VIP supports the following features:

- ❖ Configurable
 - ◆ Currently supported 1 tier.
 - ◆ 'N' number of Downstream Ports for each HUB. (Max Ports 16 per HUB).
 - ◆ 'N' number of device instances.

- ◆ Upstream and Downstream Bus operational in all Interfaces (UTMI/Serial/Pipe) & speeds(SS+/SS/HS).
- ◆ Can be connected as MAC or PHY in UTMI/PIPE.
- ❖ Mode of operation supported are as follows:
 - ◆ USB2.0 HS only
 - ◆ USB3.0 SS only
 - ◆ Basic transfer
 - ◆ Auto enumeration in downstream without hub specific control transfer
 - ◆ Upstream and Downstream configuration to set for hub similar to device and host configuration respectively
 - ◆ Auto-reconfiguration support to do re-configure once set address done

1.4 UVM Support Provided by USB VIP

The following is a summary of the supported UVM features:

- ❖ Top level USB agent that implements the following:
 - ◆ Configurable USB component stack that connects protocol, link, and physical components.
 - ◆ Support automatic end-of-test determination
- ❖ Callback support in all layers of the component stack
- ❖ Sequencers
- ❖ Random stimulus generators
- ❖ Factories
- ❖ Functional coverage for Link and Protocol layer
- ❖ Trace support in all layers of the component stack
- ❖ Digital simulation of analog signaling required for attachment and detachment detection
- ❖ Configuration object support
- ❖ Input through sequence item pull ports. The analysis ports are for 'output' to the testbench for use in coverage, scoreboarding, etc.
 - ◆ USB transfers: BULK, CONTROL, ISOC, and Interrupt
 - ◆ Packets (USB 2.0): TOKEN, DATA, HANDSHAKE, SPECIAL, NO_PID
 - ◆ Packets (SS): TP and DP
 - ◆ PHY requests: Drive Reset, Drive Resume/Suspend, Drive USB states (J, K, SE0, and SE1), High speed disconnect, Attach/detach (serial)
 - ◆ Support for post port get actions using callbacks
 - ◆ Support for post randomization callbacks after transfer, transaction, or packet randomization by the component
 - ◆ Support for input transaction coverage through callbacks
 - ◆ Service input ports:
 - ❖ Available on all components for receiving commands

- ◆ Error injection
 - ◇ Comprehensive built in errors, with constraints to control injection
 - ◇ Support for injecting multiple errors
 - ◇ Support for user override of errors and error constraints
 - ◇ Support for user provided error injection objects
- ❖ Output via TLMs
 - ◆ Performed in response to internal events
 - ◆ Generates transactions
 - ◆ Support for pre-port put actions via callback registered with the component
 - ◆ Output transaction coverage via callbacks
 - ◆ User provided objects supported by the `uvm_object_registry` and the `set_type_override` and `set_inst_override` methods.
 - ◆ An analysis port, for LPM, LMP, and ITP indication.
- ❖ Testbench scoreboarding, done using callbacks.

1.4.1 Data Flow Support

1.4.1.1 Physical Component Ports

UVM Ports are the mechanism through which the Physical Layer component connects to other components in the USB VIP sub-environment (`svt_usb_agent`) and/or to the testbench. The Physical component contains various types of ports:

- ❖ Transfer Input ports. These are used by the sequencer to send sequences into the physical layer. You cannot connect to them. They are only used by the sequencer. Consult UVM documentation on how to use sequence related classes.
- ❖ Observed ports (analysis ports). You take data from these ports and use them for either generating inputs into response (output ports), or for creating scoreboards and coverage checks.
- ❖ Response or Transfer Out ports. You use these ports to place data into the physical layer.

UVM Ports utilized for the USB SS data path are:

- ❖ **usb_ss_link_data_in_port**. Input port for USB SS data transactions to be sent by this component.
- ❖ **usb_ss_link_data_observed_port**. Analysis port for USB SS data transactions received by this component.
- ❖ **usb_ss_physical_data_in_port**. Input port for USB SS data transactions received by this component.
- ❖ **usb_ss_physical_data_out_port**. Output port for USB SS data transactions to be sent by this component.
- ❖ **usb_ss_physical_service_in_port**. Input port for USB SS service requests to this component.

UVM Ports utilized for the USB 2.0 data path are:

- ❖ **usb_20_link_data_in_port**. Input port for USB 2.0 data transactions to be sent by this component.
- ❖ **usb_20_link_data_observed_port**. Analysis port for USB 2.0 data transactions received by this component.

- ❖ **usb_20_physical_data_in_port.** Input port for USB 2.0 data transactions received by this component.
- ❖ **usb_20_physical_data_out_port.** Output port for USB 2.0 data transactions to be sent by this component.
- ❖ **usb_20_physical_service_in_port.** Input port for USB 2.0 service requests to this component.

1.4.1.2 Signal Interfaces

The HTML reference documentation contains a complete listing of pins within the signal interfaces listed below.

1.4.1.2.1 PIPE3 Interfaces

This SystemVerilog 'interface' definition declares all of the signals (scoped within an instance of this interface) that are specified by the PIPE3 Specification as being included in a USB PIPE3 connection

The interface declares 'clocking blocks' that define the clock synchronization and directionality of interface signals used by the SVT USB Physical component, and declares 'modports' (which reference the clocking blocks) that are to be used as logical port connections for the SVT USB Physical port `#svt_usb_physical_port_pipe3`.

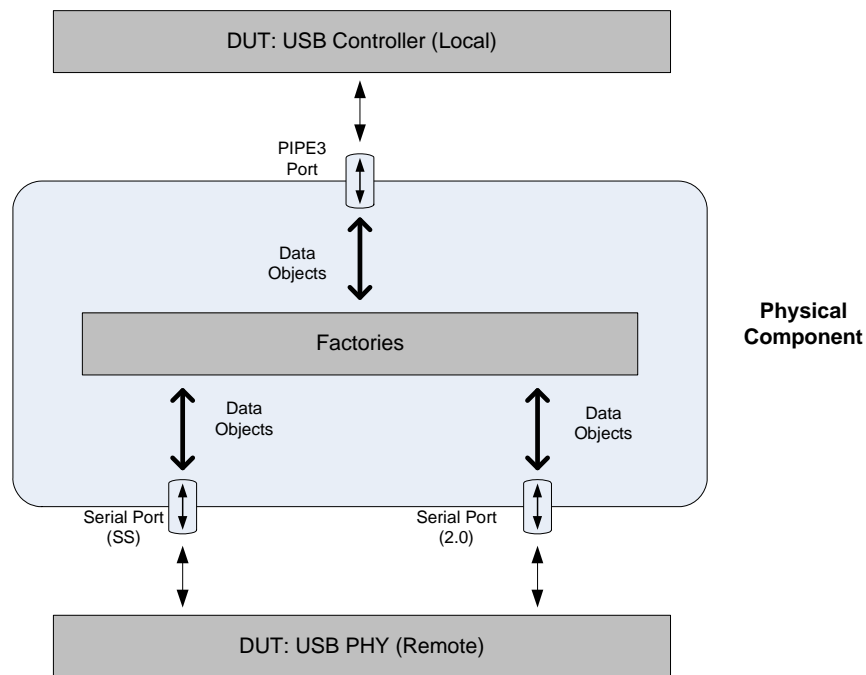
The following lists the PIPE3 signal interfaces into the model:

- ❖ `interface svt_usb_pipe3_dut_mac_if()`.
- ❖ `interface svt_usb_pipe3_dut_phy_if()`.

1.4.1.2.2 Serial Interfaces

The following are the serial interfaces supported by the model:

- ❖ `svt_usb_20_hsic_if ()`. This interface defines the USB 2.0 HSIC signal interface used by the SVT USB component to communicate with a DUT.
- ❖ `svt_usb_20_serial_if ()`. This interface defines the USB 2.0 serial signal interface used by the SVT USB component to communicate with a DUT.
- ❖ `svt_usb_if ()`. This interface defines a top-level interface used by the SVT USB component to hold an instance of each supported signal interface that can be used for communication with a DUT.
- ❖ `svt_usb_otg_if ()`. This interface defines a signal interface used by the SVT USB component to support simulation of OTG related behavior within a testbench.
- ❖ `svt_usb_ss_serial_if ()`. This interface defines the USB SS serial signal interface used by the SVT USB component to communicate with a DUT.

Figure 1-2 Physical Component data flow through Port Objects

Signal interfaces use 4-state logic for communicating across physical signals with the following exception:

- ❖ **USB 2.0 serial signal:** This interface models the serial bus using 9-state logic in combination with wired-OR outputs.

This matches the Synopsys nanoPHY specification. Refer to the DesignWare Cores USB 2.0 nanoPHY One-Port Databook.

USB Physical components do not support the simultaneous configuration of a signal interface on link and physical interfaces. Signal interfaces are only supported on one component side at a time.

Refer to [“Interface Options”](#) on page 16 for a description of valid Physical transaction configurations that utilize signal interfaces.

1.4.1.3 Data Objects

The following is a list of objects that represent information the Physical component receives, sends, or processes.

[“Sequence Item Data Objects”](#) on page 22 describes USB data objects

- ❖ **Data Objects:** These objects that represent USB data transfer that flow between the USB Protocol layer and the entity accessing the USB bus.
- ❖ **Physical Service Transaction Objects:** These objects represent service requests that initiate physical layer events or control physical layer operations.

1.4.1.4 Transaction Support

USB Physical components support using transactions to model data flow – byte and non-byte.

- ❖ **Byte data** – Transmission and reception of data
- ❖ **Non-byte data** – Driving and detecting linestate during idle

Physical components support data flow transactions by providing data transformations associated with data flow in the physical layer:

- ❖ USB Transmit Transformations
 - ◆ Scrambles the byte value
 - ◆ 8b10b encodes the scrambled byte value
- ❖ USB Receive Transformations
 - ◆ Addition or removal of SKP sets
 - ◆ 8b10b decodes the encoded byte value
 - ◆ Unscrambles the decoded byte value
- ❖ USB 2.0 Transmit Transformations
 - ◆ Bit stuffs the byte value
 - ◆ NRZI encodes the bit stuffed byte value
 - ◆ Inserts SYNC/EOP at packet start/end
- ❖ USB 2.0 Receive Transformations
 - ◆ Removes SYNC/EOP at packet start/end
 - ◆ Addition or removal of symbols
 - ◆ NRZI decodes the encoded byte value
 - ◆ Bit un-stuffs the decoded byte value

1.4.2 Interface Options

The VIP Physical Layer component supports several verification topologies. The following sections describe these topologies.

1.4.2.1 Connecting the Physical component to a Link component and a remote Physical component

This Physical component configuration is used within the local USB protocol stack when verifying a USB core that is connected to the PHY interface of the remote Physical component; the core is connected to the other side of the remote Physical component, as described in [“Connecting the Physical Component to a Link Component and a Remote DUT”](#) on page 16. When connecting Link layer and a remote Physical Layer components, ports typically exchange information between the components.

1.4.2.2 Connecting the Physical Component to a Link Component and a Remote DUT PHY

This Physical component configuration is used when connecting the USB protocol stack to the testbench or a DUT. The Physical component exchanges information with the testbench through serial bus ports.

1.4.2.3 Connecting the Physical Component to a Link Component and a Remote DUT

This Physical component configuration is typically used to verify a USB controller independently of its PHY. The Physical component exchanges information with the testbench through PIPE3 bus ports.

1.4.2.4 Connecting the Physical Component to a Link Layer Component and a local PHY (DUT)

This Physical component configuration is typically used to verify a USB PHY. The Physical component exchanges information with the testbench through PIPE3 ports.

1.4.3 Interface File Features

Interface files implement features that UVM environments cannot directly model, such as clock generation. Features best suited for modeling in Verilog, such as clock recovery, are also implemented in interface files.

The following interface related features are implemented directly within the appropriate interface files:

- ❖ PIPE3 PCLK generation – component modeling a PHY
- ❖ Serial clock recovery – receive data/clock recovery
- ❖ USB 2.0 serial signal modeling – (9-state signaling)

The following features use state variables in interface files to maintain a compatible usage model with Synopsys PHY IIP simulation models:

- ❖ SS Receiver Detection
- ❖ HS Disconnect Detection

2

General Concepts

2.1 Introduction to UVM

UVM is an object-oriented approach. It provides a blueprint for building testbenches using a constrained random verification. The resulting structure also supports directed testing.

This chapter describes the data objects that support the higher structures that comprise the USB VIP. Refer to the [Class Reference HTML](#) for a description of attributes and properties of the objects mentioned in this chapter.

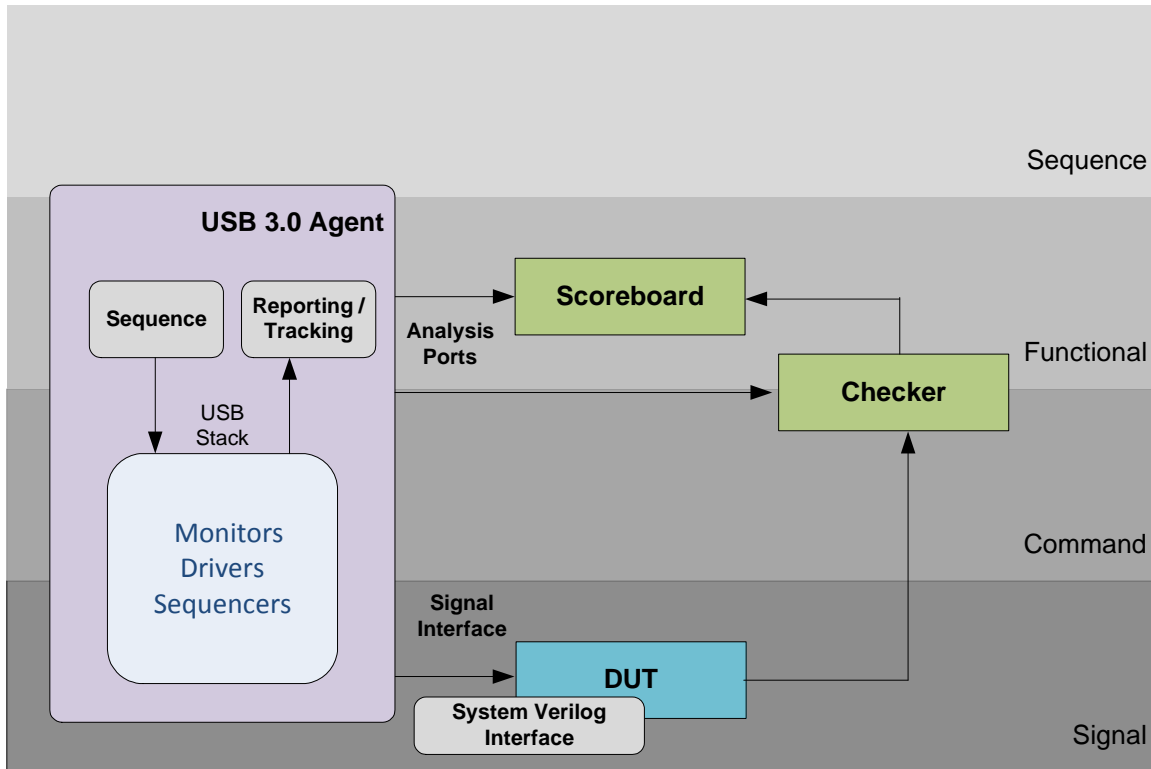
The USB VIP for UVM described in this document provides class libraries that verify features of the USB protocol. The VIP provides a USB UVM agent that contains stacked components that are compatible for use with SystemVerilog-Compliant UVM based testbenches. Agent supports all functionality normally associated with active UVM components, including the creation of transactions by built-in sequencers, checking and reporting the protocol correctness, transaction logging, Symbol logging and functional coverage.

After instantiating the agents, you can select and combine these agents with different mode setups to create an environment that verifies USB features in the DUT.

2.2 USB VIP in a UVM Environment

Figure 2-1 shows where the USB VIP fits into the UVM methodology. In the layered approach that is typical for UVM, the VIP (purple) fits into the lower levels, which allows you to focus on higher levels of abstraction.

Figure 2-1 USB VIP in Layered Testbench Architecture



2.2.1 Base Classes

In an object-oriented programming environment, a set of base classes form the foundation for the entire system. Base classes provide common functionality and structure. The SystemVerilog base classes are specifically designed for the UVM approach to verification. They provide common functionality and structure needed for simulation (such as reporting) and they support any sort of verification function.

The USB VIP classes are extended from these base classes, providing an actual implementation and demonstrating that UVM is not simply a set of guidelines and recommendations. So, instead of writing your own reporting routine, you can reuse the uvm report class. Inheritance, extension, and polymorphism facilitate customization opportunities.

Important UVM base classes used by the USB VIP include:

- ❖ uvm_agent
- ❖ uvm_component
- ❖ uvm_object
- ❖ uvm_callback
- ❖ uvm_sequencer

2.2.2 UVM Components

Objects derived from `uvm_component` are objects in a UVM compliant verification environment. The testbench and stack layers exchange transactions through `uvm_component` in three distinct ways:

❖ Sequencer Ports

- ◆ The sequencers and drivers are connected through TLM ports. Sequencers place transactions that they create in input ports.
- ◆ The sequencer includes an 'implementation' of a pull port which it 'exports' for connection so that `sequence_item_pull_ports` can connect to it. The USB components and drivers have `sequence_item_pull_ports` which are connected to the sequencer provided exports.

❖ Callbacks

- ◆ Before a component 'gets' something from a data source it does a 'get' callback. On the opposite side, before a component 'puts' something to a data sink, it does a 'put' callback.

For example, a 'get' operation can be:

- ❖ get data from a sequence item pull port
- ❖ get recognition of data off the bus
- ❖ get or peek from a lower level component

For example, a 'put' operation can be:

- ❖ get/peek export to a higher level component
- ❖ sequence execution that results in sequence items going to lower level components
- ◆ Callbacks are defined in a callback facade class (associated with each sequencer), and accessed by registering (with the associated component) an instance of a class extended from that facade class.
- ◆ Each `uvm_component` supports additional callbacks to access to data at internal dataflow points. Refer to the HTML documentation for a complete callback list.

❖ UVM Events

- ◆ Events are based on the `uvm_event` class. Some `uvm_component` signal "significant events" through UVM events and include transactions as `uvm_data` objects with these events. Testbenches can be configured to wait for `uvm_event` instances, making a call to the `uvm_component`'s UVM event service instance. Then you can retrieve data by making a call to the `get_trigger_data()` method on the UVM event.

UVM components associate coverage (cov) callbacks with ports, in addition to 'get' and 'put' callbacks. These callbacks connect functional coverage to these ports. Coverage callbacks are called after corresponding get and put methods if none of the methods set 'drop_it' to 1.

2.2.3 USB VIP Objects

The USB VIP defines several classes designed for a UVM environment. This section introduces the major USB VIP objects.

As mentioned earlier, the USB VIP classes extend base classes to handle specific needs of the protocol and provide predefined constraints. The predefined constraints can be used “as is” to produce a wide range of stimulus, or extended to create specific test conditions. For information about constraints, see [“Constraints”](#) on page 27.

An object and its constraints are referred to as a factory object, or factory when used to control the production of, or randomization of a transaction data object. Sequencers which create sequences use factories to create streams of randomized objects. Sequencers are typically responsible for creating sequence items based on factories.

The remainder of this section describes the following USB VIP objects:

- ❖ [“Configuration Objects”](#)
- ❖ [“Sequence Item Data Objects”](#)
- ❖ [“Status Objects”](#)
- ❖ [“Exception and Exception List Objects”](#)
- ❖ [“Callbacks”](#)

2.2.3.1 Configuration Objects

The configuration objects specify attributes and support testbench capabilities, such as randomization and constraints. Configuration objects apply to all appropriate `uvm_components` and `uvm_objects` in the stack.

The configuration data objects are extended from the `svt_configuration` class, which is extended from the `uvm_sequence_item` base class. These objects implement all of the methods specified by UVM for the `uvm_sequence_item` class. The testbench can retrieve the current configuration of the USB agent through the `get_cfg()` method.

Configuration data objects convey the agent’s configuration. This includes:

- ❖ Static information that defines physical design parameters, such as bus width
Static settings are changed only when the system is in a stopped state. Data is sent to the `uvm_components` anytime prior to the start – either between the constructor call and the start or between a hard reset and the start.
- ❖ Dynamic information that defines testbench parameters, such as timeout values
Dynamic information can be changed at any time. The components use the updated values for all new activity. The impact on activity that has already been initiated, however, is unspecified. In some cases new values may result in immediate use, whereas in other cases existing activity (such as a currently active watchdog timer) may be allowed to complete before the new value is recognized.

The user passes the configuration object to the agent. The agent then disburses it to the components. If the user wants to get the `cfg` object, they should access it through the agent. For anything that is “stack wide”, user actions should be aimed at the agent.

Configuration objects are controlled by direct access to their data properties or through randomization. Data properties that control monitoring levels, such as timeouts and visibility of messages, are normally not randomizable and must be set manually. Default randomization allows for a complete randomization of the configuration, including static as well as dynamic information.

Many situations require multiple tests where the static design configuration is frozen while dynamic configuration parameters are randomized. You can use the method “function int `svt_usb_configuration::static_rand_mode(bit on_off)`” to turn static configuration parameter randomization on/off as a block.

In UVM the user sets up the configuration, and then provide it to the `config_db` for later access by the agent. This is done in the test or env 'build_phase' method, prior to the construction of the agent. The main agent construction occurs during the build_phase when the agent retrieves the configuration and any other pertinent support information from the `config_db`.

After constructing the component, you can change the configuration through the `reconfigure()` method, which takes one parameter: the configuration object.

The USB VIP defines the following configuration classes:

- ❖ **Agent configuration** (`svt_usb_agent_configuration`): This class provides settings for the basic testbench capabilities: for example, whether tracing is enabled, or whether exceptions are enabled. These basic testbench capabilities are not randomized, and are controlled via basic 'enable' flags and more complex enumerated choices. The agent configuration is also extended from the basic usb configuration (`svt_usb_configuration`) which includes the basic configuration information for the protocol. It also contains the host, device, etc., configurations.
- ❖ **Device configuration** (`svt_usb_device_configuration`): This 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.
- ❖ **Host configuration** (`svt_usb_host_configuration`): The 'prot' component uses the information to support its breaking of transfers into individual transactions.
- ❖ **Endpoint configuration** (`svt_usb_endpoint_configuration`): This `uvm_data` class provides endpoint information for an individual USB endpoint. The information that this class provides can categorized as being made up of endpoint descriptors and dynamic information
- ❖ **Ustream configuration** (`svt_usb_ustream_resource_configuration`): This `uvm_data` class contains information regarding a 'USB SS stream resource'.

Note the following:

- ❖ Configuration data objects are extended from the `svt_configuration` class, which is extended from the `uvm_sequence_item` base class. These objects implement all of the methods specified for the `uvm_sequence_item` class.
- ❖ The testbench can retrieve the current configuration via the `get_cfg()` method.

Refer to the [Class Reference HTML](#) for more information.

2.2.3.2 Sequence Item Data Objects

The transaction data objects are extended from the `svt_sequence_item` class, which is extended from the `uvm_sequence_item` base class. These objects implement all of the methods specified by UVM for the `uvm_sequence_item` class.

Pre-defined sequence and sequencer classes exist for different UVM sequence item objects. For example `svt_usb_data_sequencer` is a valid object type available to the testbench designer relative to the `svt_usb_data` class.

`uvm_sequence_item` data objects define a unit of bus protocol information that is passed across the bus. The attributes of data objects are public and are accessed directly for setting and getting values. Most sequence item attributes can be randomized. The sequence item data object can represent the desired activity to be simulated on the bus, or the actual bus activity that was monitored. A protocol may have several types of sequence item data objects, such as for different protocol layers.

`uvm_sequence_time` data objects store data content and protocol execution information for USB connection transactions in terms of bit-level and timing details of the transactions. USB transaction data objects are used to:

- ❖ Generate random scenario stimulus
- ❖ Report observed transactions from receiver `uvm_components`
- ❖ Generate random responses to stack requests
- ❖ Collect functional coverage statistics
- ❖ Support error injection

Class properties are public and accessed directly to set and read values. Sequence item data objects support randomization for varying stimulus and to provide valid ranges and reasonable constraints.

- ❖ *valid_ranges* constraints limit generated values to those acceptable to the components. These constraints ensure basic VIP operation and should never be disabled.
- ❖ *reasonable_** constraints, which can be disabled individually or as a block, limit the simulation by:
 - ◆ enforcing the protocol. These constraints are typically enabled unless errors are being injected into the simulation.
 - ◆ setting simulation boundaries. Disabling these constraints may slow the simulation and introduce system memory issues.

The VIP supports extending sequence item data classes for customizing randomization constraints. This allows you to disable some *reasonable_** constraints and replace them with constraints appropriate to your system. Individual *reasonable_** constraints map to independent fields, each of which can be disabled. The class provides the **`reasonable_constraint_mode()`** method to enable or disable blocks of *reasonable_** constraints.

The USB VIP defines the following 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.
- ❖ **Detected Object** (*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 `uvm_event`.
- ❖ **Link Command** (*svt_usb_link_command*): This class represents a USB link command

- ❖ **Link Service** (*svt_usb_link_service*): These objects represent USB link service commands requested by the link service commands that are put into the link component.
- ❖ **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.
- ❖ **Physical Service** (*svt_usb_physical_service*): These objects represent USB physical service commands.
- ❖ **Protocol Service** (*svt_usb_protocol_service*): These objects 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 component 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.

- ❖ **Symbol Set** (*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*).
- ❖ **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 testbench sends transfer objects to the VIP USB Protocol through the Transfer In port. The Protocol *uvm_component* controls USB bus activity using the list of transactions. Alternatively, the testbench can leave the list of data objects empty and the protocol *uvm_component* will create the transactions needed to implement the transfer.

The same transfer data object is used by the VIP USB Protocol layer to receive inbound transactions from packet input ports. The testbench receives these transactions through callbacks and *uvm_events* issued by the VIP USB Protocol. Remotely initiated transfers are also provided to the testbench through the transfer output ports

- ❖ **Transfer** (*svt_usb_transfer*): These objects represent USB transfer data units that flow between the USB Protocol layer and the testbench.

2.2.3.3 Status Objects

Status classes define status data descriptor objects. The configuration status data objects are extended from the *svt_sequence_item* class, which is extended from the *uvm_sequence_item* base class. These objects implement all of the methods specified by UVM for the *uvm_sequence_item* class. The testbench can retrieve the current status as the *shared_status* object is a public member of the agent. As a result, your testbench can access it directly.

The USB VIP defines the following status classes:

- ❖ **Component** (*svt_usb_status*): This class provides a common location for status information coming from the different components in the USB component stack. This status information comes in two forms – data and events (status objects use UVM events). The data members represent the current

status as defined by the component implementing and updating the status information. The notifies are used to indicate a change in status, as defined by the component responsible for indicating the `uvm_event`.

- ❖ **Device** (*svt_usb_device_status*): This class contains status information regarding a USB device either being modeled by or communicating with the USB VIP.
- ❖ **Host** (*svt_usb_host_status*): This class contains status information regarding a USB host either being modeled by or communicating with the USB VIP.
- ❖ **Endpoint** (*svt_usb_endpoint_status*): This class contains information regarding a USB endpoint. In the USB protocol component there will be on 'sub-component' for each endpoint defined in the `svt_usb_configuration` object supplied to the USB VIP. This class makes available the endpoint specific status information.
- ❖ **Ustream Resource** (*svt_usb_ustream_resource_status*): This class contains information regarding a USB SS stream resource. In the USB protocol component there will be one sub-component for each instance of `svt_usb_ustream_resource_status` if the endpoint supports streams. That sub-component indicates the USB stream IDs it is assigned to support.

2.2.3.4 Exception and Exception List Objects

Exception objects represent injected errors or protocol variations. Each sequence data object has an exception list, which is an object that serves as an array of exception objects that may apply.

The USB VIP defines the following exception classes:

- ❖ **Data** (*svt_usb_data_exception, svt_usb_data_exception_list*): This class is the foundation exception descriptor for USB data transactions. 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** (*svt_usb_link_command_exception, svt_usb_link_command_exception_list*): This class is the foundation exception descriptor for USB link command transaction. The exceptions are errors that may be introduced into transaction, for the purpose of testing how the connected port of a DUT responds.
- ❖ **Packet** (*svt_usb_packet_exception, svt_usb_packet_exception_list*): This class is the foundation exception descriptor for USB packet transaction. 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** (*svt_usb_symbol_set_exception, svt_usb_symbol_set_exception_list*): This class is the foundation exception descriptor for USB symbol_set transaction. 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** (*svt_usb_transaction_exception, svt_usb_transaction_exception_list*): This class is the foundation exception descriptor for the USB transaction class. The exceptions are errors that may be introduced into transaction, for the purpose of testing how the DUT responds.
- ❖ **Transfer** (*svt_usb_transfer_exception, svt_usb_transfer_exception_list*): This class is the foundation exception descriptor for USB transaction class. The exceptions are errors that may be introduced into transaction, for the purpose of testing how the DUT responds.

2.2.3.5 Callbacks

Callbacks are an access mechanism that enable the insertion of user-defined code and allow access to objects for scoreboarding and functional coverage. Each uvm_component is associated with a class that contains a set of callback methods. These methods are called as part of the normal flow of procedural code. There are a few differences between callbacks and other methods that set them apart.

- ❖ The `svt_usb_protocol_callback` class is extended from the `svt_xactor_callbacks` base class, which is extended from the `uvm_callback` class. These objects implement all of the methods specified by UVM for the `uvm_callback` class. Callbacks are virtual methods with no code initially so they do not provide any functionality unless they are extended. The exception to this rule is that some of the callback methods for functional coverage already contain a default implementation of a coverage model.
- ❖ The callback class is accessible to users so the class can be extended and user code inserted, potentially including testbench-specific extensions of the default callback methods, and testbench-specific variables and/or methods used to control whatever behavior the testbench is using the callbacks to support.
- ❖ Callbacks are called within the sequential flow at places where external access would be useful. In addition, the arguments to the methods include references to relevant data objects. For example, just before a sequencer puts data object into an output port is a good place to sample for functional coverage since the object reflects the activity that just happened on the pins. A callback at this point with an argument referencing the sequence data object allows this exact scenario.
- ❖ There is no need to invoke callback methods for callbacks that are not extended. To avoid a loss of performance, callbacks are not executed by default. You add a callback in the following manner.

```
uvm_callbacks#(svt_usb_protocol)::add(my_agent.prot, my_20_dev_resp_cb);
```

USB VIP uses callbacks in four main applications:

- ❖ Access for functional coverage
- ❖ Access for scoreboarding
- ❖ Insertion of user-defined code
- ❖ Message processing

The VIP defines the following types of callbacks:

- ❖ **post-port get callbacks:** called after a transaction is pulled from the input port; provided with a handle to the transaction gotten from the port
- ❖ **pre-port put callbacks:** called prior to putting a transaction out on output port, provided with a handle to the transaction being put
- ❖ **traffic or dataflow event callbacks:** called in response to critical traffic or dataflow events, providing a mechanism for responding to the event or introducing errors into the event processing.

The following are the callback classes defined by the VIP:

- ❖ `svt_usb_link_callbacks`
- ❖ `svt_usb_physical_callbacks`
- ❖ `svt_usb_protocol_callbacks`

Refer to the [Class Reference HTML](#) for more information.

2.2.4 Interfaces and Modports

SystemVerilog models signal connections using interfaces and modports. Interfaces define the set of signals which make up a port connection. Modports define logical connections supported by the port.

USB VIP Physical components communicate with USB ports through modports. Modports provide a logical connection between components and the testbench. This connection is bound in after the interface is instantiated and other components are connected to its other modports.

USB VIP Physical components accept the necessary modports through their constructors. Components use modports for connecting to USB ports. Optional debug modports provide diagnostic information.

The following are the interfaces that the USB VIP includes:

- ❖ **PIPE3 Interface:** This interface (*svt_usb_pipe3_if*) declares signals included in a USB Pipe3 connection as defined by the PIPE3 Specification. The PIPE3 interface declares clocking blocks that define clock synchronization and directionality of interface signals used by the USB VIP's Physical component, and declares modports that define logical port connections.
- ❖ **USB SS Serial Interface:** This interface (*svt_usb_ss_serial_if*) declares signals included in a USB SS Serial connection, as defined by the USB Specification. The USB SS Serial Interface declares clocking blocks that define clock synchronization and directionality of interface signals used by the USB VIP's Physical component, and declares modports that define logical port connections.
- ❖ **USB 2.0 Serial Interface:** This interface (*svt_usb_20_serial_if*) declares signals included in a USB 2.0 serial connection that support HS, FS, and LS communication, as defined by the USB Specification. The USB 2.0 Serial Interface declares 'clocking blocks' that define clock synchronization and directionality of interface signals used by the USB VIP's Physical component, and declares modports that define logical port connections.
- ❖ **USB Interface:** This interface (*svt_usb_if*) declares all of the signals that by the USB Specifications define as being included in a USB Tx/Rx connection. The USB Interface consists of the PIPE3 Interface, USB SS Serial Interface, and USB 2.0 Serial Interface as sub-interfaces.



Attention

In UVM the *svt_usb_if* interface comes from the *svt_usb_if.uvm.svi* (as opposed to *svt_usb_if.svi*) file. This ensures that you obtain the correct interface: one which does not have parameters.

Note that this is only for the top level interface.

- ❖ **USB On-The-Go (OTG) Interface:** This interface (*svt_usb_otg_if*) includes signals that support the modeling of the OTG functionality. It includes modport definitions needed to provide the logical connection for various interface connections (such as the MAC to PHY connection).

See the [USB SVT - Interfaces Reference page in the HTML Class Reference](#) for more information.

["Interface Options"](#) describes USB interface options.

2.2.5 Constraints

2.2.5.1 Description

USB VIP uses objects with constraints for transactions, configurations, and exceptions. Tests in a UVM flow are primarily defined by constraints. The constraints define the range of randomized values that are used to create each object during the simulation.

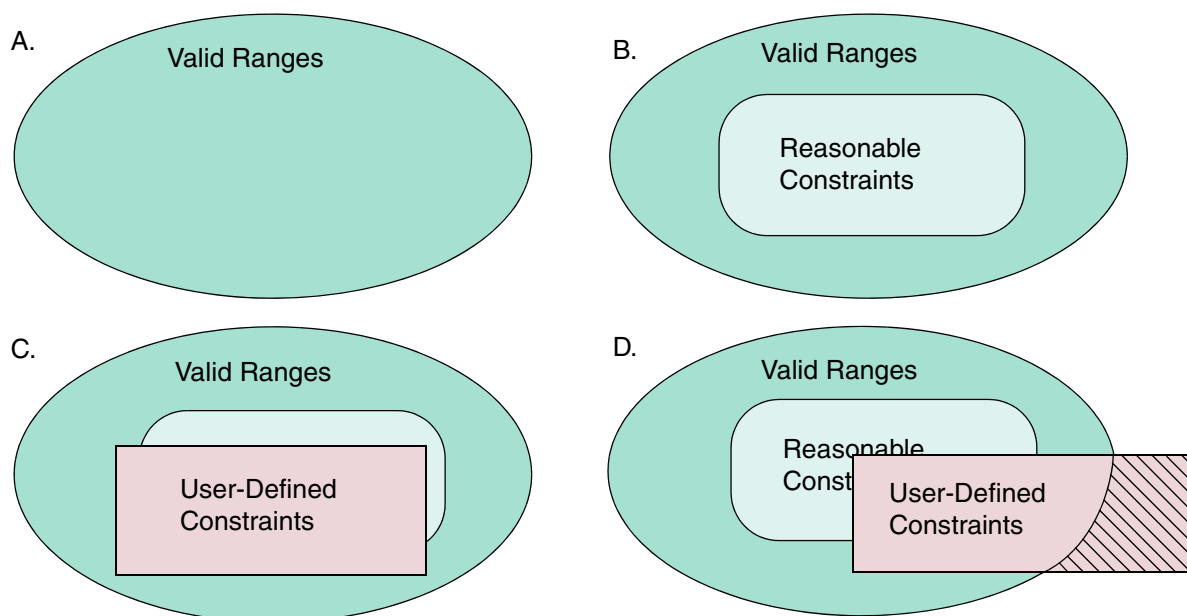
Classes that provide random attributes allow you to constrain the contents of the resulting object. When you call the `randomize()` method, which is a built-in method, all random attributes are randomized using all constraints that are enabled.

Constraint randomization is sometimes misunderstood and seen as a process whereby the simulation engine takes the control of class members away from the user. In fact, the opposite is true. Randomization is an additional way for the user to assign class members and there are several ways to control the process. The following techniques apply when working with randomization:

- ❖ Randomization only occurs when an object's `randomize()` method is called, and it is completely up to the test code when, or even if, this occurs.
- ❖ Constraints form a rule set to follow when randomization is performed. By controlling the constraints, the testbench has influence over the outcome. Direct control can be exerted by constraining a member to a single value. Constraints can also be enabled or disabled.
- ❖ Each rand member has a rand mode that can be turned ON or OFF, giving individual control of what is randomized.
- ❖ A user can assign a member to a value at any time. Randomization does not affect the other methods of assigning class members.

The following diagram shows the scope of the constraints that are part of all USB VIP.

Figure 2-2 Constraints: Valid Ranges, Reasonable, and User-Defined



- ❖ Valid range constraints:
 - ◆ Provided with USB VIP

- ◆ Keep values within a range that the components can handle
- ◆ Are not tied to protocol limits
- ◆ On by default, and should not be turned off or modified
- ❖ **Reasonable constraints:**
 - ◆ Provided with USB VIP
 - ◆ Keep values within protocol limits (typically) to generate worthwhile traffic
 - ◆ In some cases, keep simulations to a reasonable length and size
 - ◆ Defined to be “reasonable” by Synopsys (user can override)
 - ◆ May result in conditions that are a subset of the protocol
 - ◆ On by default and can be turned off or modified (user should review these constraints)
- ❖ User-defined constraints:
 - ◆ Provide a way for you to define specific tests
 - ◆ Constraints that lie outside of the valid ranges are not included during randomization

All constraints that are enabled are included in the simulation. The constraint solver resolves any conflicts.

2.2.5.2 Implementation

The following two methods implement constraints:

- ❖ Add an extension of a pre-defined external constraint block.

Most VIP data classes include pre-defined, but empty, external constraint blocks. This mechanism allows a user to add constraints to any and all instances or uses of a class by adding an implementation to this constraint block anywhere in the test code outside of a structure.

Example:

```
svt_usb_transfer::test_constraints1 { payload_intended_byte_count <= 4096; }
```

The constraint ensures that no transfers larger than 4K bytes are produced by any transfer randomization in the testbench.

If adding constraints to any and all instances/ usages of a class meets the needs of the test, this approach is very quick and easy to implement.

- ❖ Declare a class that extends the VIP's data class, add new constraint blocks to the extended class, and replace a specific VIP factory instance with an instance of the extended class (created by the testbench or testcase).

Example (this code might be in a testcase)

```
class my_transfer extends svt_usb_transfer;
    constraint my_constraint { payload_intended_byte_count <= 4096; }
endclass;
my_class my_randomized_transfer_response = new();

dev_agent.prot.randomized_usb_ss_transfer_response =
my_randomized_transfer_response;
```

A constraint is added to a class extension that is used only to replace the VIP USB Device agent's protocol layer transfer response factory. This constraint causes the VIP Device to always create transfers with 4K data bytes or less. Other instances/ usages of the svt_usb_transfer data class will be unaffected by this constraint.

2.2.6 Factories

The object that is provided to a sequencer is referred to as a factory, or factory object. It is a blueprint for randomization and serves as the template for the generated objects. A sequencer uses the factory to create streams of randomized transactions. Also, USB VIP components create factory objects for output ports so user-defined extensions to a transaction class can be handled for scoreboarding.

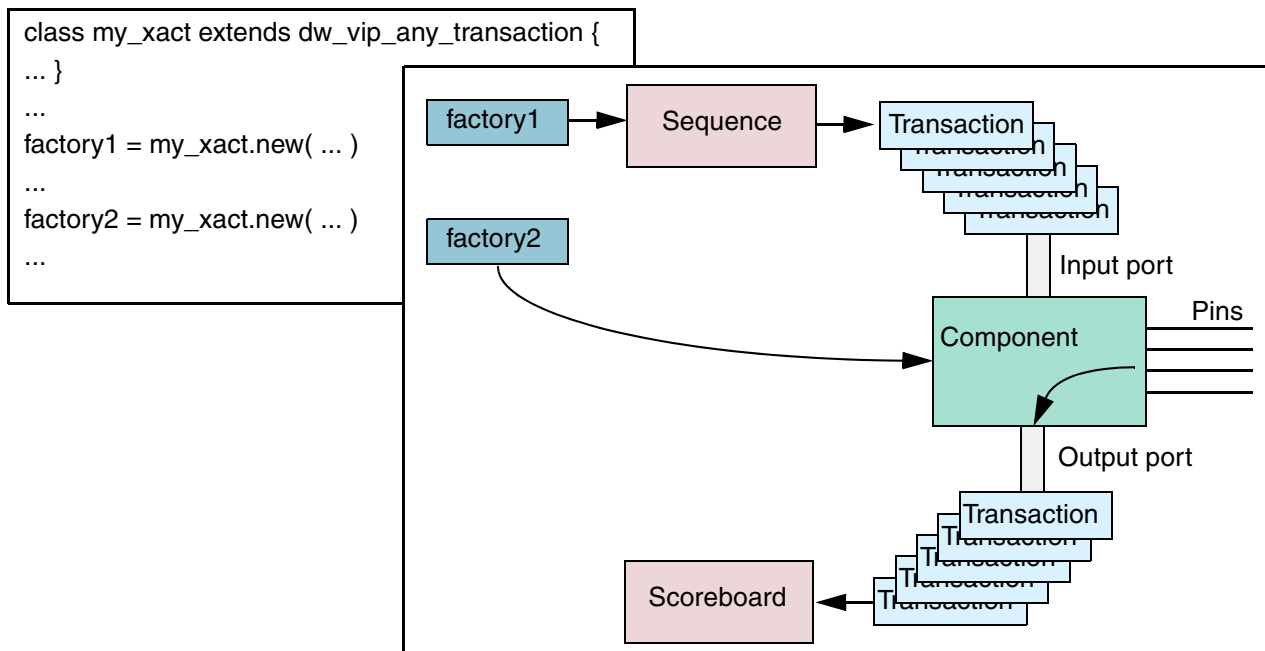
In UVM factories for the user sequencers as well as for VIP components should all go through the `uvm_object_registry`. Use `set_type_override` and `set_inst_override` to establish factories. The sequences/sequencers/components will then 'create' objects based on these registered factories.

There are two situations that require factories:

- ❖ Provide a mechanism for creating a template containing user constraints. During randomization, to give the user control over constraints and other objects
The nomenclature for this is 'randomized object'; these variables are prefixed with the 'randomized_' string.
- ❖ Constructing an transaction that is loaded and handed to the user without any randomization.
These objects are designated by the term 'factory' as a suffix – `usb_ss_packet_out_port_factory`.

Figure 2-3 illustrates how a factory object works with a sequence and a USB VIP component.

Figure 2-3 Factories with USB VIP



When using USB VIP, the factory object is typically a sequence. In Figure 2-3, the code excerpt extends a USB VIP sequence item class and then establishes two instances to use as factory objects—one for the sequence and one for the component. Typically, extensions to a sequence class are user constraints that scope the randomization to the desired test conditions. Based on the factory object and the extended constraints, the sequence creates transactions and puts them into the input port of the component. The component generates the protocol activity, handles any response, and optionally passes scoreboard information through the output port to the scoreboard.

When a component creates an object to be output on an activity or output port, the `allocate()` method is used to ensure that the resulting object is of the extended type (the factory type) and not of the base type. Note that, for this type of object, the extended members are only initialized because the VIP does not process the functionality of the extra members. Handling any added members must be provided by the testbench.

In UVM, constructors come in entirely through `uvm_config_db` (that is, all 'randomized_' field)s and/or the factory override subsystem (i.e., all `randomized_` fields if not found via `config_db`, as well as all `_factory` fields in all situations).

2.2.7 Messages

Messages can be controlled individually or in groups. This section describes messages and how to use them.

The messages originate in two scopes:

- ❖ Methodology messages, which report base class conditions and errors
- ❖ Protocol-specific messages that report protocol conditions, events, and errors

Messages can have a number of attributes, such as type, severity, ID, and text. Here are some qualities of these attributes:

- ❖ **Type:** Messages are categorized into types.
- ❖ **Severity:** Severity is similar to the urgency of the message or how serious it is.
- ❖ **Text:** This is the text of the message. UVM supports and promotes identifying messages by string matching against a regular expression.

3

The USB Agent Overview

UVM system-level verification environments is constructed with an UVM HUB Agent with the following UVM agents:

- ❖ Hub Upstream
- ❖ Hub Downstream Agent

3.1 UVM Hub Agent Stack

USB Hub `uvm_agents` are component objects in a UVM-compliant environment that implement a specific layer of the USB protocol. The USB HUB VIP defines the following `uvm_components`, for the corresponding streams from the following `uvm_component` class:

- ❖ Upstream `uvm_component`
- ❖ Downstream `uvm_component`

In addition, there is another `uvm_component` for the corresponding component layer that models the physical layer of the remote USB stack.

The set of components within a stack are continuous. The agent class defines four component attributes that correspond to the four components configurable within the agent.

The following lists the four component attributes, listed in order from top to bottom:

- ❖ **prot:** An object of type `svt_usb_protocol` that models the protocol layer of the upstream USB stack.
- ❖ **link:** An object of type `svt_usb_link` that models the link layer of the local USB stack.
- ❖ **phys:** An object of type `svt_usb_physical` that models the physical layer of the local USB stack.
- ❖ **remote_phys:** An object of type `svt_usb_physical` that models the physical layer of the remote USB stack.

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

The agent configuration specifies that top and bottom components in the agent using the `uvm_components`. The top layer of the component stack receives stimulus objects that drives the verification through a port interface. While the most typical stacks install the Protocol component at the top of the stack, the link or physical component may also be defined as the top component. Stimulus data that is input into the stack must be consistent with that accepted by the top layer of the defined stack. For more information, see the

Data Objects section of the respective component descriptions in USB Verification UVM IP Stack Components.

Note

All components have the required ports.

The agent sets up sequencers for all of the supported data streams. For example, if SS is the only stream, then it only creates sequencers for the SS communication.

Use the model provided sequencers, and it is recommended to provide sequences for the Synopsys provided sequencers.

The bottom layer is a Physical `uvm_component` that connects to the DUT through a signal interface driver.

The USB VIP defines serial and PIPE3 interfaces, allowing the Physical `uvm_component` to connect with a USB controller, a USB PHY through a PIPE3 interface, or a USB PHY through a serial interface. The VIP supplies SS and 2.0 serial interfaces.

The following examples display `uvm_component` stack formations in common agent configurations. For information on additional examples, see the Verification Topologies on page 181.

3.2 Stimulus Objects

The model uses sequences and sequencers to drive transactions into the USB protocol layers. You can utilize the virtual sequencer in the agent (of type `svt_usb_virtual_sequencer`, with sub-sequencers for transfers, packets, data, service requests, etc.).

3.3 Reporting and Tracking Objects

The agent supports the following reporting and tracking objects:

3.3.1 Checks

VIP components use an object-based mechanism for defining and encapsulating checks dynamically performed by the components. Such an object oriented approach is useful for controlling checks and for functional coverage of check execution and outcome. In that context, the classes discussed in this section are the 'container' classes (known to their associated components) within which these structured checks are defined and controlled.

The test environment can selectively enable and disable check levels, as defined and supported by the components. The USB VIP provides protocol checks defined in following classes:

- ❖ `svt_usb_link_20_sc`
- ❖ `svt_usb_link_ss_lcm_sc`
- ❖ `svt_usb_link_ss_rx_sc`
- ❖ `svt_usb_object_detect_sc`
- ❖ `svt_usb_physical_20_sc`
- ❖ `svt_usb_physical_sc`
- ❖ `svt_usb_physical_ss_sc`
- ❖ `svt_usb_protocol_sc`
- ❖ `svt_usb_protocol_ep_sc`

Refer to the [Class Reference HTML](#) for a list of checks provided by these classes.

Checks are enabled by default. The following agent configuration attributes enable checks:

- ❖ enable_prot_chk
- ❖ enable_link_chk
- ❖ enable_phys_chk

3.3.2 Reporting

The test environment can selectively enable and disable transaction reporting. Enabling this results in displaying all top level component input transactions in the report object. The following agent configuration attributes enable reporting:

- ❖ enable_link_reporting
- ❖ enable_phys_reporting
- ❖ enable_prot_reporting

The agent also permits the generation of trace files by each component. The following agent configuration attributes enables trace generation:

- ❖ enable_link_tracing
- ❖ enable_phys_tracing
- ❖ enable_prot_tracing

3.4 UVM Events

Agents denote significant actions through UVM events (uvm_event instances), and include transactions as data objects with these UVM events. Testbenches can be configured to wait for UVM events and receive transactions as part of a UVM event.

The following are the uvm_event instances provided by the agent.

- ❖ NOTIFY_GENERATED_LINK_SVC_ENDED
- ❖ NOTIFY_GENERATED_PROT_SVC_ENDED
- ❖ NOTIFY_GENERATED_USB_20_DATA_XACT_ENDED
- ❖ NOTIFY_GENERATED_USB_20_PHYS_SVC_ENDED
- ❖ NOTIFY_GENERATED_USB_20_PKT_ENDED
- ❖ NOTIFY_GENERATED_USB_SS_DATA_XACT_ENDED
- ❖ NOTIFY_GENERATED_USB_SS_PHYS_SVC_ENDED
- ❖ NOTIFY_GENERATED_USB_SS_PKT_ENDED
- ❖ NOTIFY_GENERATED_XFER_ENDED

Refer to the [Class Reference HTML](#) for more information.



4

USB Host/Device Verification

4.1 Overview

This chapter describes the `uvm_component` objects that the USB VIP supports. Refer to the [Class Reference HTML](#) for a description of objects, classes, and attributes mentioned in this chapter.

4.2 Protocol Component

USB Protocol components are component objects in a UVM-compliant environment that implement level 3 of the USB protocols, processing and exchanging transfers with the testbench and communicating with level 2 components, such as USB Link components. Protocol components operate as USB Hosts or Devices, transmitting, receiving, and processing USB 3 and USB 2.0 data streams.

The USB Protocol component object extends from the `uvm_component` class.

The [Class Reference HTML](#) describes Protocol component functions and attributes.

4.2.1 Protocol Layer Feature Support

“[Protocol Layer Features](#)” on page 25 lists the protocol layer features supported by the USB VIP. The following is a list of supported protocol layer verification features:

- ❖ Port support
 - ◆ Transfer In, Out and Response
 - ◆ Service In, Out (ITP, SOF, LMP, LPM)
- ❖ Configurable Stimulus to input port
- ❖ Error injection
 - ◆ USB Transfer
 - ◆ USB Transaction
 - ◆ USB Packets
- ❖ Testbench visibility and control through callbacks
- ❖ Randomization factories
 - ◆ USB Transfer
 - ◆ USB Transaction
 - ◆ USB Packet

- ◆ Exception lists

4.2.2 Protocol Component Ports

UVM Ports are the mechanism through which the Protocol Layer component connects to other components in the USB VIP sub-environment (`svt_usb_agent`) and/or to the testbench. The Protocol component contains various types of ports:

- ❖ **Transfer Input ports.** These are used by the sequencer to send sequences into the protocol layer. You cannot connect to them. They are only used by the sequencer. Consult UVM documentation on how to use sequence related classes.
- ❖ **Observed ports (analysis ports).** You take data from these ports and use them for either generating inputs into response (output ports), or for creating scoreboards and coverage checks.
- ❖ **Response or Transfer Out ports.** You use these ports to place data into the protocol layer.

The Protocol Layer component has the following UVM Port interfaces:

- ❖ **transfer_in_port.** Transfer input port used to supply stimulus transfers for the VIP as USB Host to execute. Used only by sequencers. You cannot connect to them.
- ❖ **transfer_out_port.** Transfer blocking_peek port used to give testbench access to transfers created by VIP as USB Device.
- ❖ **transfer_observed_port.** Transfer analysis port used to give testbench access to completed transfers.
- ❖ **transfer_response_port.** Transfer response input port used to allow testbench to supply transfer response details and control to VIP as USB Device.
- ❖ **usb_ss_packet_in_port.** Rx Packet data objects coming from the USB SS link layer arrive through this get_peek port.
- ❖ **usb_20_packet_in_port.** Rx Packet data objects coming from the USB 2.0 link layer arrive through this get_peek port.
- ❖ **protocol_service_in_port.** Protocol Layer service request data objects (`svt_usb_protocol_service`) to be acted upon by the Protocol Layer component are sent in through this port.
- ❖ **protocol_service_observed_port.** Protocol Layer service request (`svt_usb_protocol_service`) analysis port used to give testbench access to ITP (SS), LMP (SS), and LPM (2.0) service requests consumed (received).

4.2.3 Data Objects

The following is a list of objects that represent information the Protocol component receives, sends, or processes. 4.2.3 displays the flow of information objects within the Protocol component.

“Sequence Item Data Objects” on page 22 describes USB data objects

- ❖ **Transfer Objects:** These objects represent USB transfer data units that flow between the USB Protocol layer and the testbench. In this context a 'transfer' is a sequence of USB transactions.
- ❖ **Transaction Objects:** These objects represent USB transaction data units that the Protocol layer processes. USB transactions consist of sequences of packets exchanged between Host and Device. The USB Transaction level of abstraction is used internally by the VIP protocol component.
- ❖ **Packet Objects:** 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 a USB 2.0 packets.

- ❖ **Protocol Service Objects:** These objects represent protocol service commands that flow between the Protocol layer and the testbench.
- ❖ **Link Service Objects:** These objects represent USB link service commands requested by the Protocol layer.

4.2.4 Protocol Component Modes

4.2.4.1 Host Mode

When configured as a USB Host, the Protocol component transfers data to and from USB device endpoints. Host model 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 or must end, preventing the interleaving of OUT transfers to multiple endpoints, preventing the interleaving of IN transfers, interleaving of IN and OUT transfers, and allocating bus bandwidth to endpoints based on endpoint type. All additional capabilities are controlled by the testbench.

Host Mode requires one Transfer Input port and does not use Transfer Out or Transfer Response ports.

4.2.4.2 Device Mode

When configured as a USB Device, the Protocol component receives data and responds to data requests from USB Hosts. The Protocol component constructs transfer objects from inbound traffic received from USB interfaces. The transfer out port is a 'blocking_peek' port, which means the testbench can use a 'peek' to wait for and retrieve transfers as they arrive on the device side. You can do this within a 'response' sequence, which sends responses to the transfer response port.

The testbench responds to transfer objects sent by the Protocol component through the Transfer Response port. If the testbench does not provide any response transfer object, the Protocol component randomizes a transfer response.

Transfer Out and Transfer Response ports are optional in Device mode. If these ports are not present, then the testbench can send response transfer objects through callback methods. Device mode does not use the Transfer In port.

4.2.5 Data Transformation Objects

The following list describes Protocol component objects that manipulate data objects. [Figure 4-1](#) displays the objects that affect Protocol component data flow objects.

4.2.5.1 Transfer Processing

The component accepts transfers from the Transfer In port until the port is empty. Transfers are separated into groups of USB SS and USB 2.0 transfers, then further categorized on the basis of addressing (device address, endpoint number, direction, and USB stream ID) and configuration attributes. The component processes transfers in each group similarly, as quickly as possible, and independent of the other group's progress. Transfers for same address are executed in the order supplied.

When a transfer is operated on, transactions are randomly implemented one transaction at a time as the protocol processes the transfer. The scheduler then schedules individual transactions on the basis of endpoint direction and priority. Transfers for the same endpoint/stream are executed in the order supplied to the component by the testbench. Transfers for different endpoints or streams are mixed as specified by the scheduling technique defined in the component.

The component assumes that received transfers can fit in the framing interval for the simulated bus speed. When receiving more transaction requests than a frame/microFrame/bus interval can handle, the component defers those transactions to the next interval. Instead of rejecting endpoints/devices, the VIP attempts to send all requests and deals appropriately with allocation violations.

4.2.5.2 Factory Objects

The Protocol component supports the following factories:

- ❖ **Transfer factory:** These factories create transfer objects.
Protocol component attribute name: *randomized_usb_20_transfer_response*, *randomized_usb_ss_ep_mgr_transfer*, *randomized_usb_ss_transfer_response*.
- ❖ **Transaction factory:** These factories create transaction objects.
Protocol component attribute name: *randomized_usb_20_transaction*, *randomized_usb_ss_transaction*.
- ❖ **Packet factory:** These factories create packets for transmission to the Link component through the Packet Output ports.
Protocol component attribute names: *randomized_usb_20_packet*, *randomized_usb_ss_packet*.
- ❖ **Protocol service factory:** These factories create Protocol Service transfers.
Protocol component attribute name: *randomized_usb_protocol_service_response*.

These randomized objects support the following types of setting actions:

- ❖ Setting via `config_db` in UVM
- ❖ Setting via the 'inst_override' and 'type_override' capabilities in UVM

Using `config_db` conveys an actual object from the test environment to the VIP. For example, if you create the object in the testbench, you set it using `config_db`. The VIP pulls it out of `config_db`. It pulls the same object, with all the weight, etc., values that were placed in it by the test environment.

The `inst_override` and `type_override` methods provide a means to create a test specified object. The test registers a specific class (e.g., `my_xfer_class`) as an 'override' for a VIP class (e.g., `svt_usb_transfer`). When the VIP needs to create a new instance of the overridden class (e.g., `svt_usb_transfer`), the UVM interface helps it to create it as an instance of the override class (e.g., `my_xfer_class`).

In UVM you would do the following:

1. Create an `svt_usb_transaction` instance.
2. Set its weights.
3. Submit this transaction to the `config_db` using 'prot.usb_ss_transaction_factory'.

The model shares the same object the user put into the `config_db`. Note the following:

- ❖ The model uses `config_db` to get access to the same object provided by the testbench
- ❖ The model fills the object with "current data" values
- ❖ The model randomizes "in place", in the same shared object. That is, the model creates a new instance, by copying the randomized object. There is a copy at this point.

The randomization of the object takes place inside the testbench supplied object -- not in a "copied" object. In UVM, a 'copy' will be accomplished if the object is specified via `config_db`. An 'create' will be accomplished if the object is specified via one of the override methods.

Figure 4-1 displays a functional representation of the Protocol component factory objects

4.2.5.3 Exception List Factories

USB Protocol components support exception list factories associated with each type of object that it processes. Exception List factories are null by default; null factories are not randomized. [Figure 4-1](#) displays a functional representation of the Protocol component exception list factories.

The Protocol component supports the following exception list factories. Note that these factories do not support the 'override' methodology. They support the 'config_db' methodology. For example, you can set them via the config_db (or directly).

- ❖ **Packet exception list factory:** These randomization factories create exceptions that are injected into USB packets:

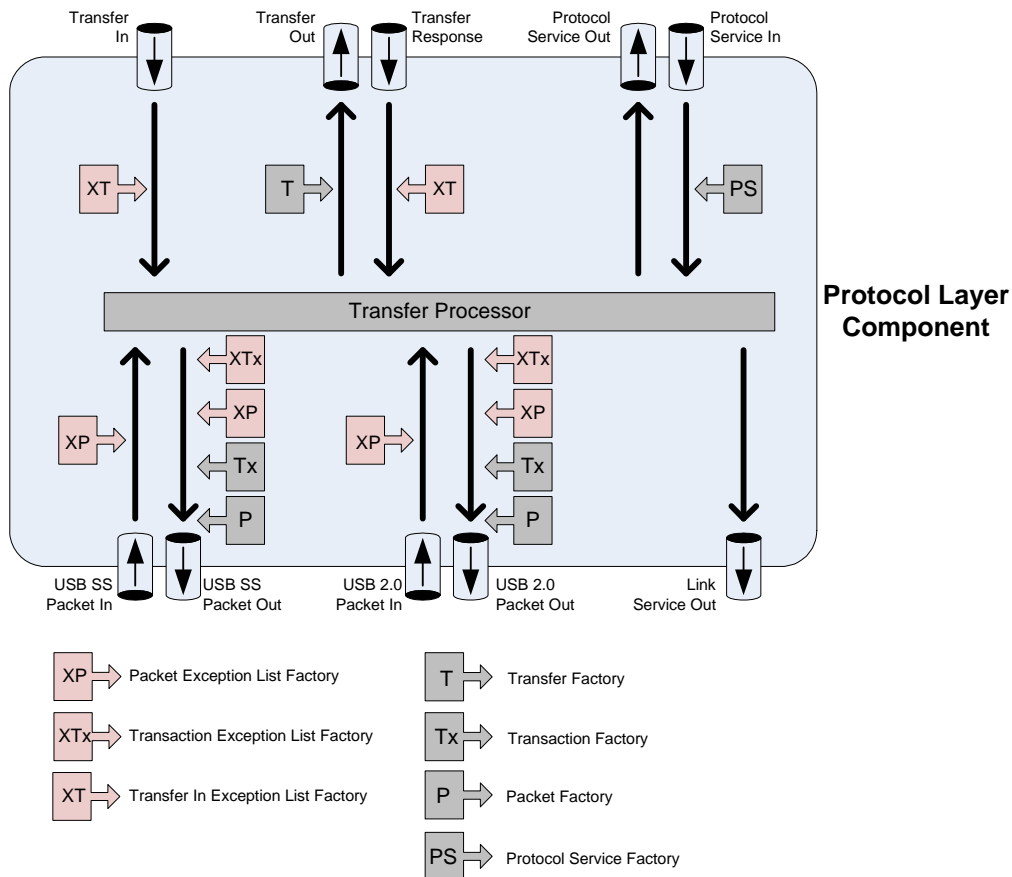
Protocol component attribute name: *randomized_usb_20_rx_packet_exception_list*,
randomized_usb_20_tx_packet_exception_list, *randomized_usb_ss_rx_packet_exception_list*,
randomized_usb_ss_tx_packet_exception_list.

- ❖ **Transaction exception list factory:** These randomization factories create exceptions that are injected into USB transactions.

Protocol component attribute name: *randomized_usb_20_transaction_exception_list*,
randomized_usb_ss_transaction_exception_list.

- ❖ **Transfer exception list factory:** These randomization factories create exceptions that are injected into USB transfers.

Protocol component attribute name: *randomized_transfer_in_exception_list*,
randomized_transfer_response_exception_list.

Figure 4-1 Protocol Component Factories – Objects and Exception List

4.2.5.4 Callbacks

The VIP supports more than 30 Protocol component callbacks for controlling randomization, viewing process flow data points, and covering data and activities. To create unique implementation of Protocol component callbacks, extend the `svt_usb_protocol_callbacks` class.

The Protocol component supports the following callbacks:

- ❖ **Packet object callbacks:** These callbacks are triggered by or report status on packets.
Protocol component callback method names: `discarded_20_packet`, `discarded_ss_packet`, `invalid_packet`, `new_20_response_transfer`, `new_ss_response_transfer`, `post_usb_20_packet_in_port_get`, `post_usb_ss_packet_in_port_get`, `pre_rx_packet`, `pre_usb_20_packet_out_port_put`, `pre_usb_ss_packet_out_port_put`, `randomized_packet`, `received_data_packet`, `usb_20_packet_in_port_cov`, `usb_ss_packet_in_port_cov`, `usb_20_packet_out_port_cov`, `usb_ss_packet_out_port_cov`.
- ❖ **Transaction object callbacks:** These callbacks are triggered by or report status on transactions.
Protocol component callback method names: `invalid_transaction`, `link_service_out_port_cov`, `pre_transaction`, `protocol_service_in_port_cov`, `protocol_service_out_port_cov`, `randomized_transaction`, `transaction_ended`.
- ❖ **Transfer object callbacks:** These callbacks are triggered by or report status on transfers.

Protocol component callback method names: *invalid_transfer*, *new_ss_ep_mgr_transfer*, *post_transfer_in_port_get*, *post_transfer_response_port_get*, *pre_transfer_out_port_put*, *protocol_service_ended*, *randomized_ep_mgr_transfer*, *randomized_transfer_basic_response*, *randomized_transfer_complete_response*, *transfer_ended*, *transfer_in_port_cov*, *transfer_out_port_cov*, *transfer_response_port_cov*

- ❖ **Service object callbacks:** These callbacks are triggered by or report status on service objects.

Protocol component callback method names: *invalid_protocol_service*, *post_protocol_service_in_port_get*, *pre_link_service_out_port_put*, *pre_protocol_service_out_port_put*, *randomized_protocol_service_response*

4.2.5.5 UVM Protocol Events

The USB Protocol component supports UVM and USB VIP specific UVM events. UVM event instances for which the current transaction is pertinent include a handle to the current data entity (transfer, transaction, or packet) with the *uvm_event*, which is accessible via the '*uvm_event::get_trigger_data()*' method. Events cannot cause an immediate change to the component behavior because they are non-blocking. Callbacks are available to support test bench modification of behavior.

The Protocol component supports the following UVM-compliant events:

- ❖ **Packet object events:** These notifications are triggered by or report status on packets.

Protocol component attribute names: *NOTIFY_20_DEVICE_RX_PACKET*, *NOTIFY_20_DEVICE_TX_PACKET*, *NOTIFY_20_HOST_RX_PACKET*, *NOTIFY_20_HOST_TX_PACKET*, *NOTIFY_SS_DEVICE_RX_PACKET*, *NOTIFY_SS_DEVICE_TX_PACKET*, *NOTIFY_SS_HOST_RX_PACKET*, *NOTIFY_SS_HOST_TX_PACKET*

- ❖ **Transaction object events:** These UVM events are triggered by or report status on transactions.

Protocol component attribute names: *NOTIFY_USB_TRANSACTION_ENDED*

- ❖ **Transfer object events:** These events are triggered by or report status on transfers.

Protocol component attribute names: *NOTIFY_ABORT_20_STARTED_TRANSFERS*, *NOTIFY_ABORT_SS_STARTED_TRANSFERS*, *NOTIFY_ALLOW_20_START_NEW_TRANSFERS*, *NOTIFY_ALLOW_SS_START_NEW_TRANSFERS*, *NOTIFY_DEVICE_TIMEOUT*, *NOTIFY_ERDY_RECEIVED_IN_INACTIVE_USTREAM_STATE*, *NOTIFY_TRANSFER_RESPONSE_RECEIVED*, *NOTIFY_USB_SCHEDULE_REQUESTED*, *NOTIFY_USB_TRANSFER_ENDED*

- ❖ **Service object events:** These events are triggered by or report status on service objects.

Protocol component attribute names: *NOTIFY_LINK_ENTERED_RECOVERY*, *NOTIFY_LINK_ENTERED_U0*, *NOTIFY_PORT_CONFIG_LMP_REQUIRED*

The *svt_usb_protocol_callbacks* class defines Protocol component callbacks.

4.2.6 Protocol Component Status

The USB VIP Protocol component provides the following status information

- ❖ **Bus Interval/frame/μframe:** The protocol component updates a set of shared status objects to provide:
 - ◆ The number of bus intervals since the component started or reset.
 - ◆ The current bus interval start time (realtime).
 - ◆ The number of μframes since the component started or reset.
 - ◆ The current μframe start time (realtime).
- ❖ **Bus speed:** The protocol component retrieves bus speed out from the shared status object. The component retrieves SS and 2.0 speeds if both are active.
- ❖ **Link state:** The protocol component retrieves the link state from stored in the shared status object.

4.2.7 Protocol Component SuperSpeed Link Callback, Factory, and Event Flows

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

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

4.2.7.1.1 Non-isochronous Transfers

Sequence of operations:

1. VIP gets transfer from `transfer_in` input port.
2. VIP uses the `svt_usb_protocol::randomized_transfer_in_exception_list` randomization factory to assist in creating exceptions to be injected into USB transfers.
3. After pulling a USB transfer descriptor out of the input port, VIP calls `svt_usb_protocol_callbacks::post_transfer_in_port_get(svt_usb_protocol component, int port_id, svt_usb_transfer transfer, ref bit drop)`; and then acts on the descriptor.
4. VIP calls `svt_usb_protocol_callbacks::transfer_in_port_cov(svt_usb_protocol component, int port_id, svt_usb_transfer transfer)`; callback to allow the testbench to collect functional coverage information from a USB transfer received in the `transfer_in` input port.

After performing the previous steps, the component can modify and process the object as required.

Send Phase

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

1. VIP uses the `svt_usb_protocol::randomized_usb_ss_transfer_response` factory to create new USB transactions.
2. VIP can use the `svt_usb_protocol_callbacks::randomized_transaction(this, transfer, transaction_ix, rand_point)`; callback to modify the transaction created in the first step.
3. VIP uses the `svt_usb_protocol::randomized_usb_ss_transaction_exception_list` randomization factory to create exceptions to be injected into USB SS transactions that need to be sent.
4. VIP issues the `svt_usb_protocol_callbacks::pre_transaction((svt_usb_transfer transfer, int transaction_ix)` callback 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.

5. VIP uses *randomized_usb_ss_transfer_response* to create new USB SS packets.
6. VIP uses *randomized_usb_ss_packet* for creating packets for USB SuperSpeed transactions.
7. If *rand_point* = *PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY*, then VIP issues *randomized_packet_rx_pkt_pre_processing_delay(svt_usb_protocol component, svt_usb_packet packet)*; callback when a protocol processor thread randomizes a received SS packet in order to create a randomized *svt_usb_packet::rx_pkt_pre_processing_delay* value.
If *rand_point* != *PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY*, then VIP issues *svt_usb_protocol_callbacks::randomized_packet(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix, int packet_ix, svt_usb_types::protocol_randomization_point_enum rand_point)* callback when a protocol processor thread creates a new USB packet by randomizing the packet factory.
8. VIP uses the *randomized_usb_ss_tx_packet_exception_list* to create exceptions to be injected into outgoing USB SS packets.
9. VIP may extend the *svt_usb_protocol_callbacks::pre_tx_packet(transfer, transaction_ix, packet_ix)* callback to collect functional coverage data, or to check the packet against a scoreboard.
10. VIP calls *svt_usb_protocol_callbacks::pre_usb_ss_packet_out_port_put(my_pkt, curr_xact, curr_xact.get_packet_index(tx_pkt), curr_xfer, curr_xfer.tx_xact_ix, drop)* callback before putting a USB packet descriptor into the SS packet output port.
11. VIP issues *svt_usb_protocol_callbacks::usb_ss_packet_out_port_cov(svt_usb_packet)* callback 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 port.

Receive Phase

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

1. VIP uses the *svt_usb_protocol::randomized_transfer_in_exception_list* randomization factory to create exceptions to be injected into USB transfers.
2. The VIP pulls a USB packet descriptor out of the USB SuperSpeed input port (link layer), calls *post_usb_ss_packet_in_port_get(svt_usb_protocol component, int port_id, svt_usb_packet packet, ref bit drop)*, and then acts on the descriptor.
3. 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 component, 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 port (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 *discarded_ss_packet(svt_usb_protocol component, svt_usb_packet packet)*; and then discards the USB packet descriptor that came through the USB SuperSpeed Packet input port (from the link layer). This is called if the packet cannot be handled by the applicable protocol block.

4. VIP uses *svt_usb_protocol::randomized_usb_ss_transfer_response* to create new USB SuperSpeed transactions. The transaction argument is a handle to the transaction whose variables are to be randomized. The *rand_point* argument defines the context of this randomization, and implies a specific set of variables that will be randomized.
5. After the protocol processor thread creates a new USB transaction (by randomizing the transaction factory), VIP issues *svt_usb_protocol_callbacks::randomized_transaction(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix, svt_usb_types::protocol_randomization_point_enum rand_point)*.
6. VIP uses the *svt_usb_protocol::randomized_usb_ss_transaction_exception_list* randomization factory to create exceptions to be injected into the USB SuperSpeed transactions that are to be sent.
7. VIP issues *pre_transaction(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix)* callback 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.
8. VIP then performs actions based on one of the following conditions:
 - ◆ The received packet is not TP_ERDY
 - ◆ The received packet is for IN endpoint processor and *retry_pending* IN transfer is set to zero
 - ◆ The received packet is for IN endpoint processor

If the received packet is not TP_ERDY

VIP uses *svt_usb_protocol::randomized_usb_ss_rx_packet_exception_list* randomization factory to create exceptions to be added to incoming USB SuperSpeed packets.

VIP issues *svt_usb_protocol_callbacks::pre_rx_packet(svt_usb_transfer transfer, int transaction_ix, int packet_ix)* callback when a protocol processor thread is ready to begin processing a USB Packet received as part of a USB transfer. This callback can be extended to collect functional coverage data, or to check the packet against a scoreboard, or other such operations.

If the received packet is for IN endpoint processor and *retry_pending* in transfer is set to zero

VIP issues *svt_usb_protocol_callbacks::transaction_ended(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix)*; callback when a USB transaction is completed. This callback can be extended to collect functional coverage data, or to check the transaction against a scoreboard, or other such operations.

If the received packet is for IN endpoint processor

VIP issues *received_data_packet(svt_usb_protocol component, svt_usb_transfer transfer, svt_usb_packet packet)*; callback when an error free data packet that is receiving data for a transfer has just been received. The transfer in progress is passed as the transfer argument, and the data packet (DP) is passed as the packet argument. This callback is intended to be used by a testbench to sample and/or check data received for a transaction on-the-fly, regardless of whether the transaction completes at this point or not. For example, this callback can still be made if an extra DP is received for a transaction that has already received a flow-control response.

VIP issues *svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol component, svt_usb_transfer transfer)* when a USB transfer is complete (when the *transfer.end_tr()* event is sent for that USB transfer). This callback can be extended to collect functional coverage data, or to check the transaction against a scoreboard, or other such operations.



For all other packets

VIP issues `svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol component, svt_usb_transfer transfer)` when a USB transfer is complete (when the `transfer.end_tr()` event is sent for that USB transfer). This callback can be extended to collect functional coverage data, or to check the transaction against a scoreboard, or other such operations.

4.2.7.1.2 Ping Operation for OUT Transfers

Sequence of operations:

1. VIP gets the USB transfer descriptor from the `transfer_in` input port.
2. VIP uses `svt_usb_protocol::randomized_transfer_in_exception_list` randomization factory to create exceptions that will be injected into USB transfers.
3. After pulling the USB transfer descriptor from the `transfer_in` input port, VIP calls `svt_usb_protocol_callbacks::post_transfer_in_port_get(svt_usb_protocol component, int port_id, svt_usb_transfer transfer, ref bit drop);`
4. VIP issues `svt_usb_protocol_callbacks::transfer_in_port_cov(svt_usb_protocol component, int port_id, svt_usb_transfer transfer);` callback to allow the testbench to collect functional coverage information from a USB transfer received from the `transfer_in` input port.
5. VIP uses `svt_usb_protocol::usb_ss_transaction_factory` factory to create new USB SuperSpeed transactions.
6. VIP can use `svt_usb_protocol_callbacks::randomized_transaction(this, transfer, transaction_ix, rand_point)` to modify the transaction created in the previous step.
7. VIP creates exceptions using `svt_usb_protocol::randomized_usb_ss_transaction_exception_list` randomization factory. These exceptions are injected into USB SuperSpeed transactions that will be sent.

If `rand_point = PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY`, then VIP issues `randomized_packet_rx_pkt_pre_processing_delay(svt_usb_protocol component, svt_usb_packet packet);` callback when a protocol processor thread randomizes a received SS packet in order to create a randomized `svt_usb_packet::rx_pkt_pre_processing_delay` value.

If `rand_point != PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY`, then VIP issues `svt_usb_protocol_callbacks::randomized_packet(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix, int packet_ix, svt_usb_types::protocol_randomization_point_enum rand_point)` callback when a protocol processor thread creates a new USB packet by randomizing the packet factory.

8. VIP issues `svt_usb_protocol_callbacks::pre_transaction((svt_usb_transfer transfer, int transaction_ix)` callback 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.
9. VIP uses the `randomized_usb_ss_packet` randomization factory to create packets for a USB SuperSpeed transaction.
10. After the protocol processor thread randomizes a received SuperSpeed packet (in order to create a randomized `svt_usb_packet::rx_pkt_pre_processing_delay` value), VIP issues `randomized_packet(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix, int packet_ix, svt_usb_types::protocol_randomization_point_enum rand_point)` callback.

11. After a protocol processor thread creates a new USB packet (by randomizing the packet factory), VIP issues *randomized_packet* (*svt_usb_protocol* component, *svt_usb_transfer* transfer, *int* transaction_ix, *int* packet_ix, *svt_usb_types::protocol_randomization_point_enum* rand_point) callback.
12. VIP then creates exceptions using the *randomized_usb_ss_tx_packet_exception_list* randomization factory. These exceptions are injected into outgoing SS packets.
13. VIP can optionally extend the *svt_usb_protocol_callbacks::pre_tx_packet*(transfer, transaction_ix, packet_ix) callback to collect functional coverage data, check the packet against a scoreboard, or other such operations.
14. VIP calls *svt_usb_protocol_callbacks::pre_usb_ss_packet_out_port_put*(my_pkt, curr_xact, curr_xact.get_packet_index(tx_pkt), curr_xfer, curr_xfer.tx_xact_ix, drop) before putting a USB packet descriptor into the SS output port.
15. VIP then issues *svt_usb_protocol_callbacks::usb_ss_packet_out_port_cov*(*svt_usb_packet* packet) 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 port.

4.2.7.1.3 Ping Response Operation for OUT Transactions

Sequence of operations:

1. VIP uses *svt_usb_protocol::usb_ss_transaction_factory* to create new USB SS transactions.
2. VIP modifies the transaction created in the previous step using *svt_usb_protocol_callbacks::randomized_transaction*(*svt_usb_protocol* component, *svt_usb_transfer* transfer, *int* transaction_ix, *svt_usb_types::protocol_randomization_point_enum* rand_point) callback.
3. VIP creates a randomization factory using *svt_usb_protocol::randomized_usb_ss_transaction_exception_list*, which is then injected into outgoing USB SS packets.
4. VIP can then optionally extend the *svt_usb_protocol_callbacks::pre_rx_packet*(transfer, transaction_ix, packet_ix) callback to collect functional coverage data, check the packet against a scoreboard, or other such operations.
5. VIP then issues *svt_usb_protocol_callbacks::transaction_ended* (*svt_usb_protocol* component, *svt_usb_transfer* transfer, *int* transaction_ix) callback when a USB transaction is complete when the transfer.end_tr() event is sent for a particular USB transaction, the VIP can optionally extend this callback to collect functional coverage data, check the packet against a scoreboard, or other such operations.

4.2.7.1.4 Isochronous Transfer Operation

Sequence of operations:

1. VIP gets transfer from transfer_in input port.
2. VIP uses the *svt_usb_protocol::randomized_transfer_in_exception_list* randomization factory to assist in creating exceptions to be injected into USB transfers.
3. After pulling a USB transfer descriptor out of the input port, VIP calls *svt_usb_protocol_callbacks::post_transfer_in_port_get*(*svt_usb_protocol* component, *int* port_id, *svt_usb_transfer* transfer, *ref* bit drop); and then acts on the descriptor.
4. VIP calls *svt_usb_protocol_callbacks::transfer_in_port_cov*(*svt_usb_protocol* component, *int* port_id, *svt_usb_transfer* transfer); callback to allow the testbench to collect functional coverage information from a USB transfer received in the transfer_in input port.

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

Send Phase

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

1. VIP uses the *svt_usb_protocol::usb_ss_transaction_factory* factory to create new USB transactions.
2. VIP can use the *svt_usb_protocol_callbacks::randomized_transaction(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix, svt_usb_types::protocol_randomization_point_enum rand_point);* callback to modify the transaction created in the first step.
3. VIP uses the *svt_usb_protocol::randomized_usb_ss_transaction_exception_list* randomization factory to create exceptions to be injected into USB SS transactions that need to be sent.
4. VIP issues the *svt_usb_protocol_callbacks::pre_transaction((svt_usb_transfer transfer, int transaction_ix)* callback 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.
5. VIP uses *randomized_usb_ss_packet* to assist in creating packets for USB SuperSpeed transactions.

Receive Phase

Sequence of operations:

1. If *rand_point* = *PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY*, then VIP issues *randomized_packet_rx_pkt_pre_processing_delay(svt_usb_protocol component, svt_usb_packet packet);* callback when a protocol processor thread randomizes a received SS packet in order to create a randomized *svt_usb_packet::rx_pkt_pre_processing_delay* value.
If *rand_point* != *PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY*, then VIP issues *svt_usb_protocol_callbacks::randomized_packet(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix, int packet_ix, svt_usb_types::protocol_randomization_point_enum rand_point)* callback when a protocol processor thread creates a new USB packet by randomizing the packet factory.
2. VIP uses *randomized_usb_ss_tx_packet_exception_list* randomization factory to create exceptions to be injected into outgoing USB SS packets.
3. VIP can then optionally extend the *svt_usb_protocol_callbacks::pre_tx_packet(transfer, transaction_ix, packet_ix)* callback to collect functional coverage data, check the packet against a scoreboard, or other such operations.
4. VIP calls *svt_usb_protocol_callbacks::pre_usb_ss_packet_out_port_put(svt_usb_protocol component, int port_id, svt_usb_packet packet, svt_usb_transaction transaction, int packet_ix, svt_usb_transfer transfer, int transaction_ix, ref bit drop)* before it puts a USB packet descriptor into the SS packet output port.
5. VIP issues *svt_usb_protocol_callbacks::usb_ss_packet_out_port_cov(svt_usb_packet packet)* callback 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 port.
6. VIP issues *svt_usb_protocol_callbacks::transaction_ended(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix);* callback when a USB transaction is complete (when the *transfer.end_tr()* event is sent for a particular USB transaction).

4.2.7.2 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_port.
2. VIP retrieves packets from the link layer through the appropriate packet input port (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. Randomizes the downstream packet using the PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY randomization point randomizing both rx_pkt_pre_processing_delay and deferred_pkt_reflection_delay.
 - d. Uses downstream_pkt as factory to create a copy assigning to upstream_pkt.
 - e. 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.
 - f. Places downstream_pkt in rx_pkt_ppd_queue.
 - g. 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. Randomizes the downstream packet using the PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY randomization point randomizing both rx_pkt_pre_processing_delay and deferred_pkt_reflection_delay.
 - c. 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 port.

Detailed Sequences

Receive Phase

1. VIP pulls a USB packet descriptor out of the SS packet input port (from the link layer) and calls *post_usb_ss_packet_in_port_get(svt_usb_protocol component, int port_id, svt_usb_packet packet, ref bit drop)*; before acting on the descriptor.
2. VIP calls *route_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. Calls *route_packet*
 - c. Exits *receive_in_packet*
2. VIP prepares the transaction (if it is not yet ready).
3. VIP uses *randomized_usb_ss_rx_packet_exception_list* randomization factory to create fake exceptions to be added to incoming USB SS packets.
4. 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 *transaction_ended(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix)* after the USB Endpoint Manager transaction is complete. VIP can optionally extend this callback to collect functional coverage data, check the transaction against a scoreboard, and other such operations.

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. VIP uses *randomized_usb_ss_rx_packet_exception_list* randomization factory to create fake exceptions to be added to incoming USB SS packets.
3. If the received packet is neither a ping or a TP_ACK, VIP terminates the transfer.
4. If the received packet is a ping or a TP_ACK, VIP issues *post_ss_rx_pkt_xfer_update(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix, int packet_ix)* callback immediately after a protocol processor thread completes processing a USB packet received and associated with a USB transfer. VIP can optionally extend this callback to collect functional coverage data, check the transaction against a scoreboard, and other such operations.
5. 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 calls `route_packet`.
 - ii. VIP prepares the transaction.
- 2. VIP uses `randomized_usb_ss_rx_packet_exception_list` randomization factory to create fake exceptions to be added to incoming SS packets.
- 3. VIP issues `pre_rx_packet(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix, int packet_ix)` callback when a protocol processor thread is ready to start processing a USB packet to be transmitted as part of a USB transfer. VIP can optionally extend this callback to collect functional coverage data, check the packet against a scoreboard, or other such operations.
- 4. VIP uses `randomize_transaction` to determine if `EARLY_NAK_RESPONSE` or `EARLY_WAIT_FOR_ENDED_RESPONSE` is the response. VIP uses the randomization keys of `xact.early_response` to control the response types that are allowed.
- 5. VIP issues `received_data_packet(svt_usb_protocol component, svt_usb_transfer transfer, svt_usb_packet packet)` callback 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.
- 6. 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. VIP uses `randomized_usb_ss_rx_packet_exception_list` randomization factory to create fake exceptions to be added to incoming SS packets.
3. 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.
4. VIP issues `transaction_ended(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix)` callback immediately after a USB Endpoint Manager transaction is complete. VIP can optionally extend this callback to collect functional coverage data, check the packet against a scoreboard, or other such operations.
5. VIP issues `received_data_packet(svt_usb_protocol component, svt_usb_transfer transfer, svt_usb_packet packet)` callback 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.

6. VIP issues *post_ss_rx_pkt_xfer_update*(*svt_usb_protocol* component, *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.
2. VIP uses *randomized_usb_ss_transaction* randomization factory to create transactions for a USB SS transfer.
3. VIP issues *randomized_transaction*(*svt_usb_protocol* component, *svt_usb_transfer* transfer, *int transaction_ix*, *svt_usb_types::protocol_randomization_point_enum* rand_point) callback immediately after a protocol processor thread creates a new USB transaction by randomizing the transaction factory (*randomized_usb_transaction* object or array).
4. VIP prepares 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 uses *randomized_usb_ss_transaction* randomization factory to create transactions for a USB SS transfer.
 - c. VIP issues *randomized_transaction*(*svt_usb_protocol* component, *svt_usb_transfer* transfer, *int transaction_ix*, *svt_usb_types::protocol_randomization_point_enum* rand_point) callback immediately after a protocol processor thread creates a new USB transaction by randomizing the transaction factory (*randomized_usb_transaction* object or array).
 - d. VIP prepares the next generated packet for the specified transaction and then creates packets for USB SS transactions.
 - e. If the packet is randomized with *PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY*, VIP issues *randomized_packet_rx_pkt_pre_processing_delay*(*svt_usb_protocol* component, *svt_usb_packet* packet) callback immediately after a protocol processor thread randomizes a received USB SS packet in order to create a randomized *svt_usb_packet::rx_pkt_pre_processing_delay* value.
 - f. VIP issues *randomized_packet*(*svt_usb_protocol* component, *svt_usb_transfer* transfer, *int transaction_ix*, *int packet_ix*, *svt_usb_types::protocol_randomization_point_enum* rand_point) callback immediately after the protocol processor thread creates a new USB transaction by randomizing the packet factory (*randomized_usb_packet* object or array).
 - g. If the transaction device response does not time out, VIP calls *send_ss_pkt_to_link* and issues *transaction_ended*(*svt_usb_protocol* component, *svt_usb_transfer* transfer, *int transaction_ix*) callback when a USB Endpoint Manager transaction is complete. VIP can optionally extend this callback

to collect functional coverage data, check the transaction against a scoreboard, or other such operations.

Send Packet of OUT Transfer

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

1. VIP uses *randomized_usb_ss_transaction* randomization factory to create transactions for a USB SS transfer.
2. VIP issues *randomized_transaction(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix, svt_usb_types::protocol_randomization_point_enum rand_point)* callback immediately after a protocol processor thread creates a new USB transaction by randomizing the transaction factory (*randomized_usb_transaction* object or array).
3. VIP prepares the response packet to be sent to the host.
4. If the transfer completes successfully or if received packet *pp_bit=0* and *retyr_bit=0*, VIP moves to idle stream state.

For isochronous transfers, VIP checks if the transfer is an *isoc_ping* transfer, and then it sends a ping response.

4.2.7.3 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.

4.2.7.3.1 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.

4.2.7.3.2 Stream Endpoint Manager Flow

Figures 4-2 and 4-3 illustrate the stream endpoint manager flow for host and device respectively.

Figure 4-2 Host Stream Endpoint Manager Flow

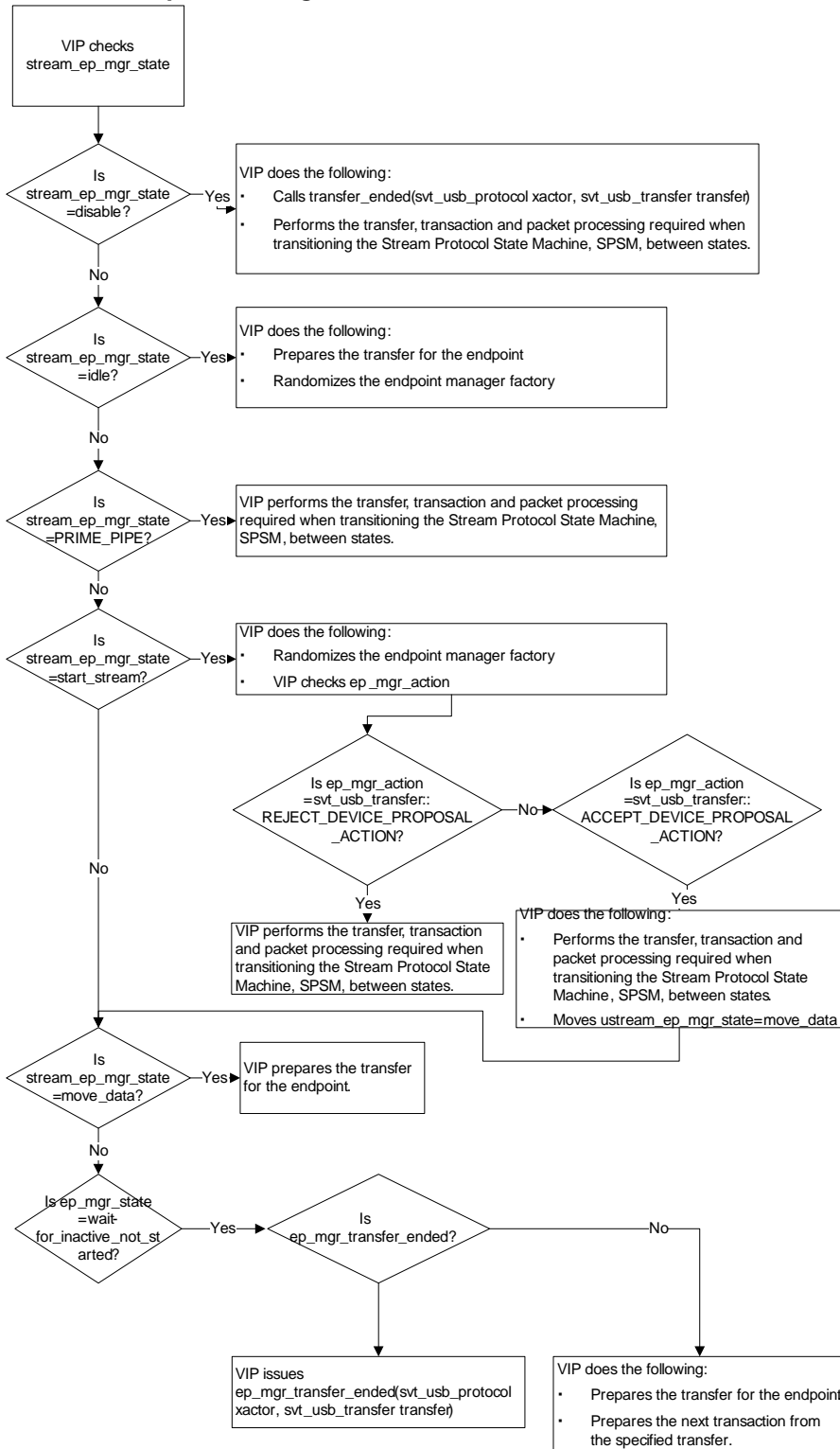
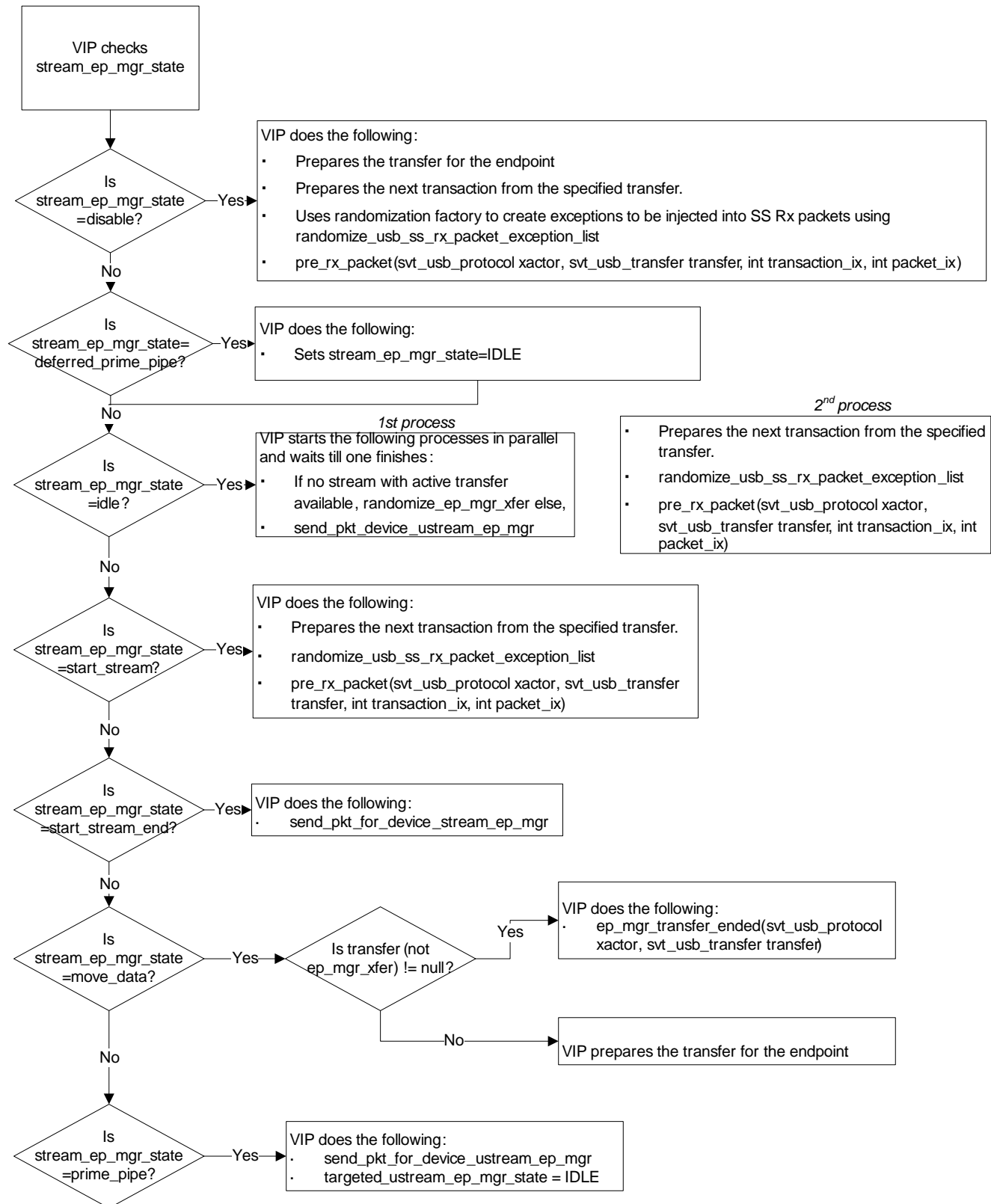


Figure 4-3 Device Stream Endpoint Manager Flow

4.2.8 Protocol Component 2.0 Link Callback, Factory, and UVM Event Flows

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

4.2.8.1 Protocol 2.0 Link, Callback, and UVM Event Flow when the VIP is Acting as a Host

4.2.8.1.1 Non-isochronous Transfers

Transfer Start Phase

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

Sequence of operations:

1. VIP gets transfer from `transfer_in` input port.
2. VIP uses the `svt_usb_protocol::randomized_transfer_in_exception_list` randomization factory to assist in creating exceptions to be injected into USB transfers. The `randomized_transfer_in_exception_list` is then randomized and added to the transfer's `exception_list` attribute. A new `exception_list` is created for the transfer if one did not exist.
3. After pulling a USB transfer descriptor out of the input port, VIP calls `svt_usb_protocol_callbacks::post_transfer_in_port_get(svt_usb_protocol component, int port_id, svt_usb_transfer transfer, ref bit drop)`; and then acts on the descriptor.
4. (Optional) This step occurs only if the drop bit from `post_transfer_in_port_get()` returns a false value. VIP calls `svt_usb_protocol_callbacks::transfer_in_port_cov(svt_usb_protocol component, int port_id, svt_usb_transfer transfer)`; to allow the testbench to collect functional coverage information from a USB transfer received in the `transfer_in` input port.

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 `uvm_event` sequences, see [“Create and Prepare New Transaction”](#) on page 70.
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 *transfer.end_tr()*, and then issues *svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol component, svt_usb_transfer transfer)*.

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”](#) on page 68. 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 *svt_usb_protocol_callbacks::received_data_packet(svt_usb_protocol component, svt_usb_transfer transfer, svt_usb_packet 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. *transfer.end_tr()*
 3. *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*
-

If the transaction is non-split, then VIP does the following:

1. VIP randomizes the transaction object using *svt_usb_protocol::randomized_usb_20_transaction*. First the current transaction object is copied into the *randomized_usb_20_transaction* object. The *host_response* attribute is then randomized. And finally, the randomized factory object is copied back into the current transaction object.
 2. VIP calls *svt_usb_protocol_callbacks::randomized_transaction(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix, svt_usb_types::protocol_randomization_point_enum rand_point)*.
 3. If (*transaction.host_response == HOST_NORMAL_RESPONSE*), complete the flow listed in [“Prepare and Send Packet”](#) on page 68.
 4. If the host response is ACK, with no injection errors, then VIP does the following:
 - *transaction.status = svt_transaction::ACCEPT*
 - *transfer.end_tr()*
 - *protocol_block.protocol.NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone());*
-

If the transaction is SPLIT IN and device responds with an ACK, VIP does the following:

- *transaction.status = svt_transaction::ACCEPT*
- *transfer.end_tr()*
- *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*

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 *transfer.end_tr()*
4. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*

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 *transfer.end_tr()*.
4. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*

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 *transfer.end_tr()*.
4. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*.

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 the callback to *transaction_ended* is made using *svt_usb_protocol_callbacks::transaction_ended (svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix)*.

4.2.8.1.2 Isochronous Transfers

Transfer Start Phase

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

Sequence of operations:

1. VIP gets transfer from `transfer_in` input port.
2. VIP uses the `svt_usb_protocol::randomized_transfer_in_exception_list` randomization factory to assist in creating exceptions to be injected into USB transfers. The `randomized_transfer_in_exception_list` is then randomized and added to the transfer's `exception_list` attribute. A new `exception_list` is created for the transfer if one did not exist.
3. After pulling a USB transfer descriptor out of the input port, VIP calls `svt_usb_protocol_callbacks::post_transfer_in_port_get(svt_usb_protocol component, int port_id, svt_usb_transfer transfer, ref bit drop)`; and then acts on the descriptor.
4. (Optional) This step occurs only if the drop bit from `post_transfer_in_port_get()` returns a false value. VIP calls `svt_usb_protocol_callbacks::transfer_in_port_cov(svt_usb_protocol component, int port_id, svt_usb_transfer transfer)`; to allow the testbench to collect functional coverage information from a USB transfer received in the `transfer_in` input port.

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 `uvm_event` sequences, see [“Create and Prepare New Transaction”](#) on page 70.
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.end_tr()`, and then issues `svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol component, svt_usb_transfer transfer)`.

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”](#) on page 68. 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 *svt_usb_protocol_callbacks::received_data_packet(svt_usb_protocol component, svt_usb_transfer transfer, svt_usb_packet 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 *transfer.end_tr()*
- VIP calls *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*

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 *uvm_event* that the transaction has ENDED using *transfer.end_tr()*.
3. Provide a copy of the ENDED transaction with the top protocol *uvm_event* that the transaction has ENDED using *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*.

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 *transfer.end_tr()*.
4. Provide a copy of the ENDED transaction with the top protocol *uvm_event* that the transaction has ENDED using *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*

-
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 the callback to *transaction_ended* is now made using *svt_usb_protocol_callbacks::transaction_ended(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix)*.

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

4.2.8.2.1 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 port.

Sequence of operations:

1. VIP receives the TOKEN packet from the `svt_usb_protocol::packet_in_port` port. For more information about the callbacks and `uvm_events`, see [“Get Received Packet”](#) on page 69.
2. VIP creates the new device VIP transfer object and allows you to modify or replace it. For more information about the callbacks and `uvm_events`, see [“Create and Prepare New Device Transfer”](#) on page 69.

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 `uvm_event` sequences, see [“Create and Prepare New Transaction”](#) on page 70.
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.end_tr()`, and then issues `svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol component, svt_usb_transfer transfer)`.

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 `uvm_event begin_event` (`uvm_transaction`'s `begin_event` field)
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](#)” on page 68 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

Randomize the transaction for response using the [Randomize Transaction for Response](#) callback flow.

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 component, svt_usb_transfer transfer, svt_usb_packet packet)*
 - *transaction.status = svt_transaction::ACCEPT*
 - *t transfer.end_tr()*
 - *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*
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. Randomize the transaction for response using the [Randomize Transaction for Response](#) callback flow.
2. If you need to send a packet, use the [Prepare and Send Packet](#) callback flow.
3. 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

Randomize the transaction for response using the [Randomize Transaction for Response](#) callback flow.

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 component, 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 component, svt_usb_transfer transfer, svt_usb_packet packet)*
2. VIP then calls the following:
 - *transaction.status = svt_transaction::ACCEPT*
 - *transfer.end_tr()*
 - *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*
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

Randomize the transaction for response using the [Randomize Transaction for Response](#) callback flow.

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. *transfer.end_tr()*
3. *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*

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

Randomize the transaction for response using the [Randomize Transaction for Response](#) callback flow.

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. *transfer.end_tr()*
3. *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*

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

Randomize the transaction for response using the [Randomize Transaction for Response](#) callback flow.

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. *transfer.end_tr()*
3. *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*

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 the callback to *transaction_ended* is now made using *svt_usb_protocol_callbacks::transaction_ended* (*svt_usb_protocol* component, *svt_usb_transfer* transfer, *int* transaction_ix).

4.2.8.2.2 Isochronous Transfers

Transfer Start Phase

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

Sequence of operations:

1. VIP receives the TOKEN packet from the `svt_usb_protocol::packet_in_port` port. For more information about the callbacks and `uvm_events`, see [“Get Received Packet”](#) on page 69.
2. VIP creates the new device VIP transfer object and allows you to modify or replace it. For more information about the callbacks and `uvm_events`, see [“Create and Prepare New Device Transfer”](#) on page 69.

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 `uvm_event` sequences, see [“Create and Prepare New Transaction”](#) on page 70.
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 `-> uvm_event begin_event` (`uvm_transaction's begin_event` field)
6. , and then issues `svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol component, svt_usb_transfer transfer)`.

Transaction Loop for Device Isochronous

Sequence of operations:

1. If this is the first transaction of the transfer, mark the transfer as STARTED using `uvm_event begin_event` (`uvm_transaction's begin_event` field).
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. Randomize the *transaction.device_response*, and create and send the response packet (if needed) as follows:
 - a. Use *svt_usb_protocol::randomized_usb_20_transaction* to randomize the transaction object. Copy the current transaction object into the *randomized_usb_20_transaction* object. Then *randomized_usb_20_transaction.randomize()* is called. Only the *device_response* attribute is randomized. Finally, the randomized factory object is copied back into the current transaction object.
 - b. VIP calls *svt_usb_protocol_callbacks::randomized_transaction(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix, svt_usb_types::protocol_randomization_point_enum rand_point)* after the transaction is randomized.
 - c. If the specific transaction requires the Device to send either a DATA or a HANDSHAKE packet, use the [Prepare and Send Packet](#) callback flow.
4. 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 *transfer.end_tr()*.
3. Provide a copy of the ENDED transaction with the top protocol uvm_event that the transaction has ENDED using *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*

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 component, 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. *transfer.end_tr()*
3. *svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED.trigger(transaction.clone)*
4. *transfer.payload.append_payload(transaction.payload)*

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

5. 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 the callback to *transaction_ended* is now made using *svt_usb_protocol_callbacks::transaction_ended(svt_usb_protocol component, svt_usb_transfer transfer, int transaction_ix)*.

4.3 Link Component

USB Link component are component objects in a UVM-compliant verification environment. The USB Link component object extends from the *uvm_component* class, which extends from the *uvm_component* base class.

The USB Link component implements the Level 2 (Link) of the USB protocol and communicates directly with level 1 and level 3 components.

The [Class Reference HTML](#) describes Link component functions and attributes.

4.3.1 Link Layer Feature Support

“[Link Layer Features](#)” on page 26 lists the link layer features supported by the USB VIP. The following is a list of supported verification features:

- ❖ Packet input and output ports – SS and 2.0
- ❖ Data input and output ports – SS and 2.0
- ❖ Configurable input port stimulus
 - ◆ Auto connect to protocol layer
 - ◆ Direct port
- ❖ Error injection
 - ◆ USB Packet
 - ◆ USB Link Command
 - ◆ USB Symbol
- ❖ Callbacks providing testbench visibility and control

4.3.2 Link Component Ports

UVM Ports are the mechanism through which the Link Layer component connects to other components in the USB VIP sub-environment (*svt_usb_agent*) and/or to the testbench. The Link component contains various types of ports:

- ❖ Transfer Input ports. These are used by the sequencer to send sequences into the link layer. You cannot connect to them. They are only used by the sequencer. Consult UVM documentation on how to use sequence related classes.
- ❖ Observed ports (analysis ports). You take data from these ports and use them for either generating inputs into response (output ports), or for creating scoreboards and coverage checks.
- ❖ Response or Transfer Out ports. You use these ports to place data into the link layer.

The following list describes objects that move information through the Link component.

- ❖ **usb_ss_packet_in_port**. Super-speed Packet input port used to supply stimulus packets for the VIP to produce (transmit).
- ❖ **usb_ss_packet_out_port**. Super-speed Packet get_peek output port used to give upper-layer access to packets consumed (received).
- ❖ **usb_ss_packet_observed_port**. Super-speed Packet analysis port used to give testbench access to packets consumed (received).
- ❖ **usb_20_packet_in_port**. 2.0 Packet input port used to supply stimulus packets for the VIP to produce (transmit).

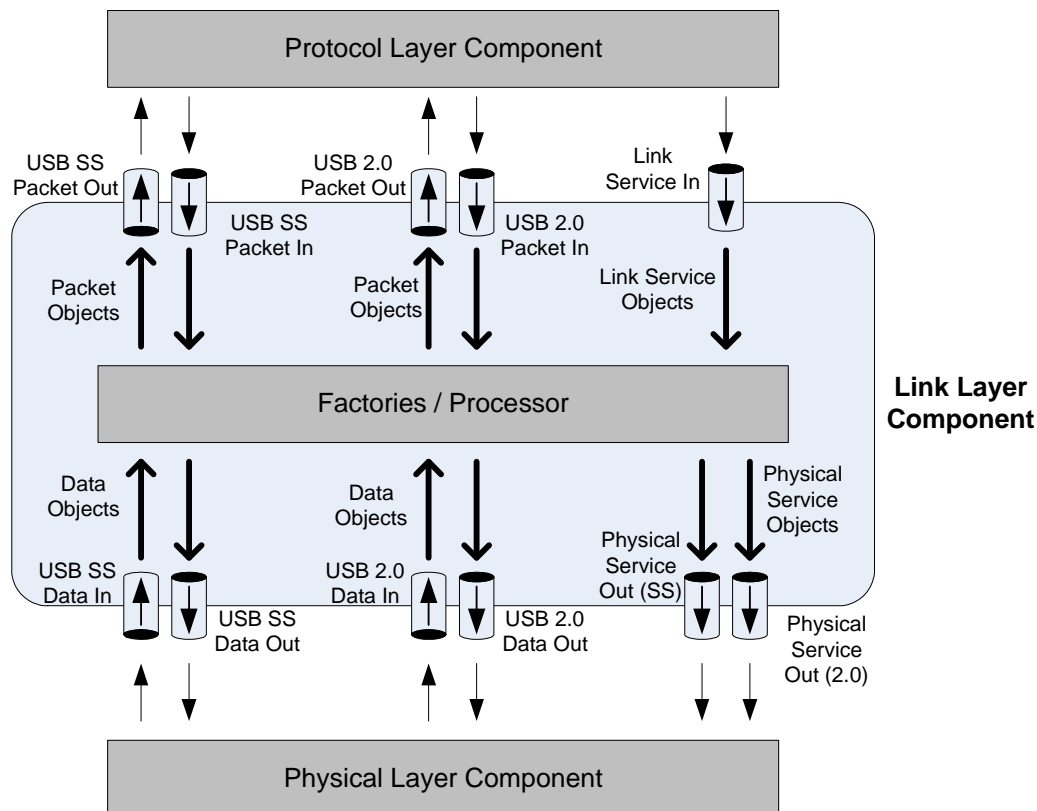
- ❖ **usb_20_packet_out_port** . 2.0 Packet get_peek output port used to give upper-layer access to packets consumed (received).
- ❖ **usb_20_packet_observed_port**. 2.0 Packet analysis port used to give testbench access to packets consumed (received).
- ❖ **usb_ss_data_in_port**. Rx Data objects coming from the USB super-speed physical layer arrive through this port.
- ❖ **usb_20_data_in_port**. Rx Data objects coming from the USB 2.0 physical layer arrive through this port.
- ❖ **link_service_in_port**. Link Layer service request data objects (svt_usb_link_service) to be acted upon by the Link Layer component are sent in through this port.

4.3.3 Data Objects

The following is a list of objects that represent information the Link component receives, sends, or processes. [Figure 4-4](#) displays the flow of information objects within the Link component.

“Sequence Item Data Objects” on page 22 describes USB data objects

Figure 4-4 Link Component Data Flow



- ❖ **Packet Objects:** These objects represent USB packet data units that flow between the USB Protocol layer and the USB Link layer.
- ❖ **Data Objects:** These objects represent the information required to send one USB data byte. This class includes support for physical layer transformations.

- ❖ **Link Service Objects:** These objects represent USB link service commands requested by the Protocol layer.
- ❖ **Physical Service Objects:** These objects represent USB physical service commands.

4.3.4 Data Transformation Objects

The following list describes Link component objects that manipulate data objects.

4.3.4.1 Factory Objects

The link component creates these Factory data objects. To create them, the component uses UVM factory capabilities, in combination with the override capabilities to come up with templates that can be used to create these objects.

The 'overrides' are related to the 'override_by_type' and 'override_by_name' methods in the `uvm_factory` class:

- ❖ `set_inst_override_by_type`
- ❖ `set_inst_override_by_name`
- ❖ etc.

You can use one of these override methods to define a "factory" for objects based on a certain name or type. If its based on 'name', then the factory names (e.g., `usb_20_tx_data_out_factory`) are used to establish those names.

For example; if you want the `host_agent.link` to use the 'my_packet' class when creating `usb_20_rx_packet_out_factory` objects, you would execute the following:

```
set_inst_override_by_type("host_agent.link.usb_20_rx_packet_out_factory",  
    svt_usb_packet::get_type(), my_packet::get_type());
```

The Link component supports the following factories:

- ❖ **Rx object factories:** These factories allocate objects for injecting into an Rx data stream.
Link component attribute names: `usb_20_rx_packet_out_factory`, `usb_ss_rx_detected_object_factory`, `usb_ss_rx_link_command_factory`, `usb_ss_rx_packet_out_factory`, `usb_ss_rx_symbol_set_factory`.
- ❖ **Tx object factories:** These factories allocate objects for injecting into a Tx data stream.
Link component attribute names: `usb_20_tx_data_out_factory`, `usb_ss_tx_data_out_factory`, `usb_ss_tx_link_command_factory`, `usb_ss_tx_symbol_set_factory`.

4.3.4.2 Exception List Factories

USB Link components support exception list factories associated with each of its ports. Exception List factories are null by default; null factories are not randomized.

The Link component supports the following exception list factories:

- ❖ **Rx object factories:** These factories generate exceptions for injecting into an Rx data stream.
Link component attribute names: `randomized_usb_ss_rx_link_command_exception_list`, `randomized_usb_20_rx_packet_exception_list`, `randomized_usb_ss_rx_packet_exception_list`, `randomized_usb_ss_rx_symbol_set_exception_list`.
- ❖ **Tx object factories:** These factories generate exceptions for injecting into a Tx data stream.
Link component attribute names: `randomized_usb_ss_tx_data_exception_list`, `randomized_usb_ss_tx_link_command`, `randomized_usb_ss_tx_link_command_exception_list`,

*randomized_usb_20_tx_packet_exception_list, randomized_usb_ss_tx_packet_exception_list,
randomized_usb_ss_tx_symbol_set_exception_list.*

4.3.4.3 Callbacks

USB Link component Callback objects extend from the *svt_xactor_callbacks* class. USB Link component Callback objects implement all methods specified by UVM.

To create unique implementation of Protocol component callbacks, extend the *svt_usb_link_callbacks* class. Following is an example to add an instance of the callback object with the USB Link component:

```
uvm_callbacks#(svt_usb_link)::add(agent.link, my_cb);
```

The Link component supports the following callbacks

- ❖ **Input port:** These callbacks indicate that the component collected data from an input port:

Link component callback method names: *post_usb_20_data_in_port_get, post_usb_ss_data_in_port_get, usb_20_data_in_port_cov, usb_ss_data_in_port_cov, post_link_service_in_port_get, usb_link_service_in_port_cov, post_usb_20_packet_in_port_get, post_usb_ss_packet_in_port_get, pre_usb_ss_rx_symbol_set_detected, usb_20_packet_in_port_cov, usb_20_packet_in_ended, usb_ss_packet_in_port_cov*

- ❖ **Port output:** These callbacks indicate that the component placed data on an output port:

Link component callback method names: *pre_usb_20_data_out_port_put, pre_usb_ss_data_out_port_put, pre_usb_20_packet_out_port_put, pre_usb_ss_packet_out_port_put, pre_usb_20_physical_service_out_port_put, pre_usb_ss_physical_service_out_port_put, usb_20_data_out_port_cov, usb_20_packet_out_port_cov, usb_20_physical_service_out_port_cov, usb_ss_data_out_port_cov, usb_ss_packet_out_port_cov, usb_ss_physical_service_out_port_cov*

- ❖ **Rx Event:** These callback indicate an event related to data stream received from the Physical component:

Link component callback method names: *usb_ss_packet_out_ended, pre_usb_ss_rx_link_command_detected, usb_ss_link_command_in_ended, pre_usb_ss_rx_object_detected, pre_usb_ss_rx_packet_detected, usb_ss_symbol_set_in_ended, usb_20_packet_out_ended*

- ❖ **Tx Event:** These callbacks indicate an event related to the transmission of a data stream to the Physical component:

Link component callback method names: *pre_usb_ss_tx_compliance_pattern_object_schedule, pre_usb_ss_tx_compliance_pattern_transform, usb_ss_symbol_set_out_ended, pre_usb_ss_tx_link_command_data_object_produce, pre_usb_ss_tx_link_command_sort_and_delay, pre_usb_ss_tx_link_command_transform, pre_usb_ss_tx_logical_idle_object_schedule, pre_usb_ss_tx_loopback_set_object_schedule, pre_usb_ss_tx_loopback_set_transform, pre_usb_ss_tx_skp_object_schedule, pre_usb_ss_tx_skp_set_transform, post_usb_ss_tx_itp_packet_its_update, pre_usb_ss_tx_packet_object_schedule, pre_usb_ss_tx_packet_transform, pre_usb_ss_tx_replay_packet_transform, usb_ss_packet_in_ended, pre_usb_ss_tx_training_set_object_schedule, pre_usb_ss_tx_training_set_transform, randomized_ss_tx_lcmd, usb_ss_link_command_out_ended*

- ❖ **Link layer event:** These callbacks indicate an event related to a link command or a link service command:

Link component callback method names: *object_observed, service_in_ended*

4.3.4.4 UVM Events

USB Link components support general UVM events and USB VIP specific events. For events where the current transaction is pertinent, the event data includes a handle to the current transaction.

Because UVM events are non-blocking, they cannot cause an immediate change in the component behavior. Callbacks are required for immediate component behavior changes.

USB Link component supported callbacks are defined in *svt_usb_link_callbacks*.

The Link component supports the following events:

- ❖ **USB Bus Status:** *NOTIFY_BUS_IS_IDLE*, *NOTIFY_BUS_IS_L1SUSPENDED*, *NOTIFY_BUS_IS_SUSPENDED*
- ❖ **Chirp Status:** *NOTIFY_CHIRP_K_DETECTED*, *NOTIFY_CHIRPING_SEQUENCE*, *NOTIFY_CHIRP_K_SIGNALING*, *NOTIFY_DETECTED_THREE_CHIRP_PAIRS*, *NOTIFY_THREE_CHIRP_PAIRS_SENT*
- ❖ **Device Status:** *NOTIFY_DEVICE_DISCONNECTED*, *NOTIFY_FULL_SPEED_DEVICE_DETECTED*, *NOTIFY_HIGH_SPEED_DEVICE_DETECTED*, *NOTIFY_LOW_SPEED_DEVICE_DETECTED*, *NOTIFY_WAIT_DEVICE_ATTACH*, *NOTIFY_WAIT_DEVICE_CONNECTED*
- ❖ **Host Status:** *NOTIFY_HOST_INITIATED_RESUME*
- ❖ **Interval Status:** *NOTIFY_DEBOUNCE_INTERVAL*, *NOTIFY_L1RESIDENCY_TIMER*
- ❖ **LFPS:** *NOTIFY_LFPS_ANY*, *NOTIFY_LFPS_HANDSHAKE_FAILED*, *NOTIFY_LFPS_OFF*, *NOTIFY_LFPS_ON*, *NOTIFY_LFPS_PING*, *NOTIFY_LFPS_POLLING*, *NOTIFY_LFPS_U1_EXIT*, *NOTIFY_LFPS_U2_EXIT*, *NOTIFY_LFPS_U3_WAKEUP*, *NOTIFY_LFPS_UNRECOGNIZED*, *NOTIFY_LFPS_WARM_RESET*
- ❖ **Remote Object Status:** *NOTIFY_REMOTE_WAKEUP*
- ❖ **Protocol Reset:** *NOTIFY_DETECTED_PROTOCOL_RESET*, *NOTIFY_DRIVING_PROTOCOL_RESET*
- ❖ **Packet completion (Rx or Tx):** *NOTIFY_RX_PACKET_ENDED*, *NOTIFY_TX_PACKET_ENDED*

4.3.5 Link Component SuperSpeed Link Callback, Factory, and UVM Event Flows

This section explains the various link-layer flows through the use of flowcharts. [Figures 4-5](#) to [4-15](#) illustrate the various SS link component flows.

**Note**

In the following flow charts, the abbreviation “**cb**” has been used to denote *svt_usb_link_callbacks*.

USB 2.0 Service Command Handling

[Table 4-1](#) lists the various USB 2.0 link service commands that are available.

Table 4-1 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. |

Table 4-1 USB 2.0 Service Command Usage

| Command | Host or Device | Usage |
|------------------------------------|-----------------|---|
| 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. |

4.3.6 SuperSpeed Packet Chronology

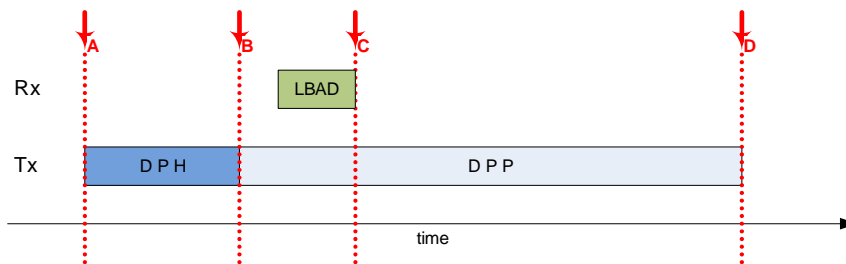
This section describes the timing of packets on the bus. Possible outcomes of low-level packet exchange on the bus are represented by the events and status values of the packet data objects within the VIP.

4.3.6.1 Receiving LBAD

4.3.6.1.1 During DPP Tx

An LBAD link command is received during the transmission of a DPP payload. The super-speed transmitter continues producing the payload to completion.

Figure 4-5 Packet Chronology: Receiving LBAD During DPP Tx



At point “A” the following packet attributes are updated:

- ❖ status attribute is set to ACTIVE
- ❖ header_status is set to ACTIVE
- ❖ notify.indicate STARTED
- ❖ start_time is set to \$realtime
- ❖ packet_start_time is set to \$realtime

At point “B” the following packet attributes are updated:

- ❖ header_status is set to ACCEPT
- ❖ payload_status is set to ACTIVE
- ❖ packet_header_end_time is set to \$realtime

At point “C” the following packet attributes are updated:

- ❖ link_command_response is stored

At point “D” the following packet attributes are updated:

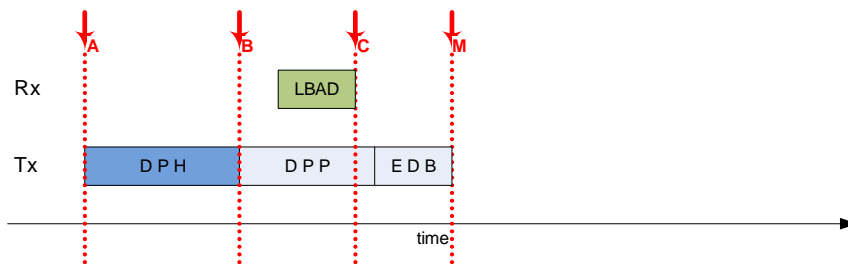
- ❖ status attribute is set to RETRY
- ❖ payload_status attribute is set to ACCEPT
- ❖ packet_end_time is set to \$realtime
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

Table 4-2 Status Attribute Chronology: Receiving LBAD During DPP Tx

| | Prior to Point "A" | at Point "A" | at Point "B" | at Point "C" | at Point "D" |
|------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
| status | INITIAL | ACTIVE | ACTIVE | ACTIVE | RETRY |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT | ACCEPT |
| payload_status | INITIAL | INITIAL | ACTIVE | ACTIVE | ACCEPT |
| payload_presence | PAYLOAD_ PRESENT | PAYLOAD_ PRESENT | PAYLOAD_ PRESENT | PAYLOAD_ PRESENT | PAYLOAD_ PRESENT |

4.3.6.1.2 During DPP Tx, EDB Result

An LBAD link command is received during the transmission of a DPP payload. The super-speed transmitter aborts production of the payload and ends the DPP with appropriate EDB framing.

Figure 4-6 Packet Chronology: Receiving LBAD During DPP Tx, EDB Result

At points "A" through "C" the events are identical to [Figure 4-5](#) at points "A" through "C"

At point "M" the following packet attributes are updated:

- ❖ status attribute is set to RETRY
- ❖ payload_status attribute is set to ACCEPT
- ❖ payload_presence is set to PAYLOAD_PRESENT_BUT_ABORTED
- ❖ packet_end_time is set to \$realtime
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

Table 4-3 Status Attribute Chronology: Receiving LBAD During DPP Tx, EDB Result

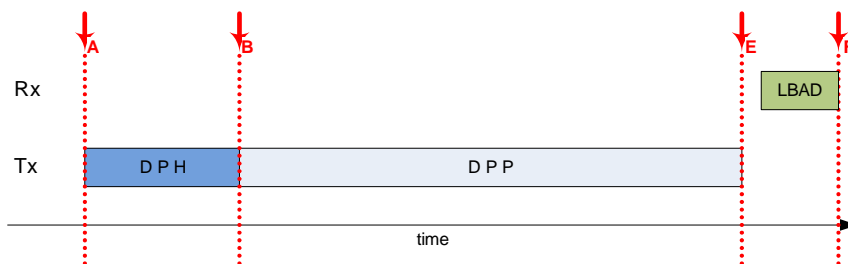
| | Prior to Point "A" | at Point "A" | at Point "B" | at Point "C" | at Point "M" |
|----------------|--------------------|--------------|--------------|--------------|--------------|
| status | INITIAL | ACTIVE | ACTIVE | ACTIVE | RETRY |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT | ACCEPT |
| payload_status | INITIAL | INITIAL | ACTIVE | ACTIVE | ACCEPT |

Table 4-3 Status Attribute Chronology: Receiving LBAD During DPP Tx, EDB Result

| | Prior to Point "A" | at Point "A" | at Point "B" | at Point "C" | at Point "M" |
|------------------|--------------------|-----------------|-----------------|-----------------|-----------------------------|
| payload_presence | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT_BUT_ABORTED |

4.3.6.1.3 After DPP Tx Complete

An LBAD link command is received after the complete transmission of a DPP payload.

Figure 4-7 Packet Chronology: Receiving LBAD After DPP Tx Complete

At points "A" and "B" the events are identical to [Figure 4-5](#) at points "A" and "B"

At point "E" the following packet attributes are updated:

- ❖ status attribute is set to PARTIAL_ACCEPT
- ❖ payload_status is set to ACCEPT
- ❖ packet_end_time is set to \$realtime

At point "F" the following packet attributes are updated:

- ❖ link_command_response is stored
- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

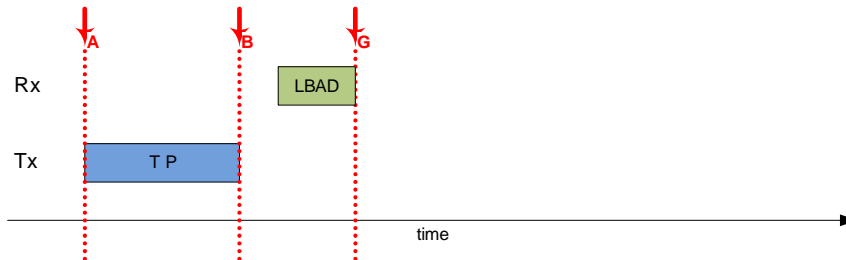
Table 4-4 Status Attribute Chronology: Receiving LBAD After DPP Tx Complete

| | Prior to Point "A" | at Point "A" | at Point "B" | at Point "E" | at Point "F" |
|------------------|--------------------|-----------------|-----------------|-----------------|-----------------|
| status | INITIAL | ACTIVE | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT | ACCEPT |
| payload_status | INITIAL | INITIAL | ACTIVE | ACCEPT | ACCEPT |
| payload_presence | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT |

4.3.6.1.4 After HP, No DPP

An LBAD link command is received after the complete transmission of Header Packet data; no payload is associated to the header (this could be a TP, ISOC, ITP or LMP packet).

Figure 4-8 Packet Chronology: Receiving LBAD After HP, No DPP



At point “A” the events are identical to [Figure 4-5](#) at point “A”

At point “B” the following packet attributes are updated:

- ❖ status is set to PARTIAL_ACCEPT
- ❖ header_status is set to ACCEPT
- ❖ packet_header_end_time is set to \$realtime
- ❖ packet_end_time is set to \$realtime

At point “G” the following packet attributes are updated:

- ❖ link_command_response is stored
- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

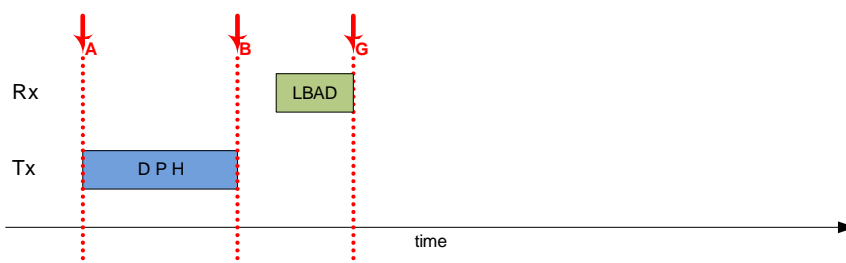
Table 4-5 Status Attribute Chronology: Receiving LBAD After HP, No DPP

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “G” |
|------------------|-------------------------|-------------------------|-------------------------|-------------------------|
| status | INITIAL | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT |
| payload_status | DISABLED | DISABLED | DISABLED | DISABLED |
| payload_presence | PAYLOAD_ NOT_PRESENT | PAYLOAD_ NOT_PRESENT | PAYLOAD_ NOT_PRESENT | PAYLOAD_ NOT_PRESENT |

4.3.6.1.5 After Data Packet Replay

An LBAD link command is received command is after the complete transmission of replayed Data Packet; no payload is associated to the replay packet.

Figure 4-9 Packet Chronology: Receiving LBAD After Data Packet Replay



At point “A” the following packet attributes are updated:

- ❖ status attribute is set to ACTIVE
- ❖ header_status is set to ACTIVE
- ❖ payload_status is set to CANCELLED
- ❖ payload_presence is set to PAYLOAD_NOT_PRESENT
- ❖ packet_start_time is set to \$realtime

At point “B” the following packet attributes are updated:

- ❖ status is set to PARTIAL_ACCEPT
- ❖ header_status is set to ACCEPT
- ❖ packet_header_end_time is set to \$realtime
- ❖ packet_end_time is set to \$realtime

At point “G” the following packet attributes are updated:

- ❖ link_command_response is stored
- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

Table 4-6 Status Attribute Chronology: Receiving LBAD After Data Packet replay

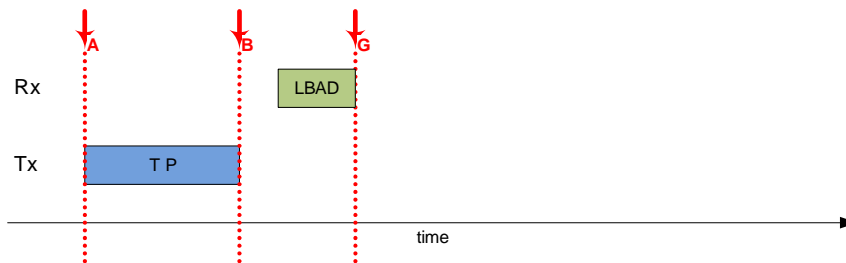
| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “G” |
|----------------|--------------------|--------------|----------------|--------------|
| status | RETRY | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | RETRY | ACTIVE | ACCEPT | ACCEPT |
| payload_status | (various possible) | CANCELLED | CANCELLED | CANCELLED |

Table 4-6 Status Attribute Chronology: Receiving LBAD After Data Packet replay

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “G” |
|------------------|--------------------|-------------------------|-------------------------|-------------------------|
| payload_presence | (various possible) | PAYLOAD_ NOT_PRESENT | PAYLOAD_ NOT_PRESENT | PAYLOAD_ NOT_PRESENT |

4.3.6.1.6 After Non-Data Packet Replay

An LBAD link command is received command is after the complete transmission of replayed Data Packet; no payload is associated to the replay packet.

Figure 4-10 Packet Chronology: Receiving LBAD After Non-Data Packet Replay

At point “A” the following packet attributes are updated:

- ❖ status attribute is set to ACTIVE
- ❖ header_status is set to ACTIVE
- ❖ packet_start_time is set to \$realtime

At point “B” the following packet attributes are updated:

- ❖ status is set to PARTIAL_ACCEPT
- ❖ header_status is set to ACCEPT
- ❖ packet_header_end_time is set to \$realtime
- ❖ packet_end_time is set to \$realtime

At point “G” the following packet attributes are updated:

- ❖ link_command_response is stored
- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

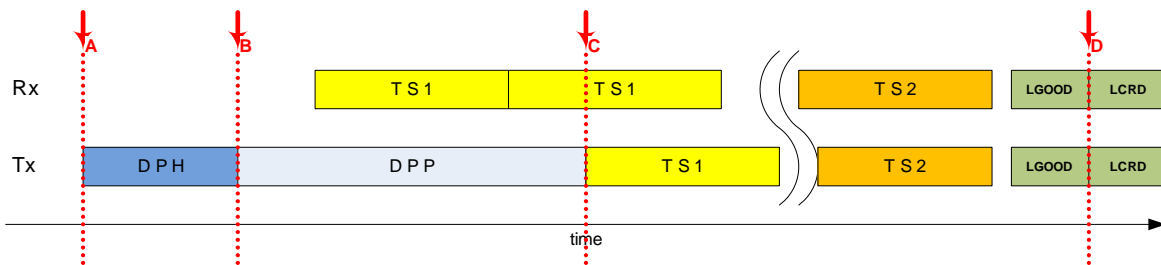
Table 4-7 Status Attribute Chronology: Receiving LBAD After Non-Data Packet Replay

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “G” |
|------------------|---------------------|---------------------|---------------------|---------------------|
| status | RETRY | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | RETRY | ACTIVE | ACCEPT | ACCEPT |
| payload_status | DISABLED | DISABLED | DISABLED | DISABLED |
| payload_presence | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT |

4.3.6.2 Receiving TS1 Ordered Sets

4.3.6.2.1 During DPP Tx

A TS1 ordered set is received during the transmission of a DPP payload (due to the link partner entering recovery state). The super-speed transmitter continues producing the payload to completion. After recovery, the LGOOD advertisement indicates a header sequence number prior to the DPP in question (such that the DPP packet will be replayed).

Figure 4-11 Packet Chronology: Receiving TS1 ordered sets During DPP Tx

At points “A” and “B” the events are identical to [Figure 4-5](#) at points “A” and “B”

At point “C” the following packet attributes are updated:

- ❖ status is set to PARTIAL_ACCEPT
- ❖ payload_status is set to ACCEPT
- ❖ packet_end_time is set to \$realtime

At point “D” the following packet attributes are updated:

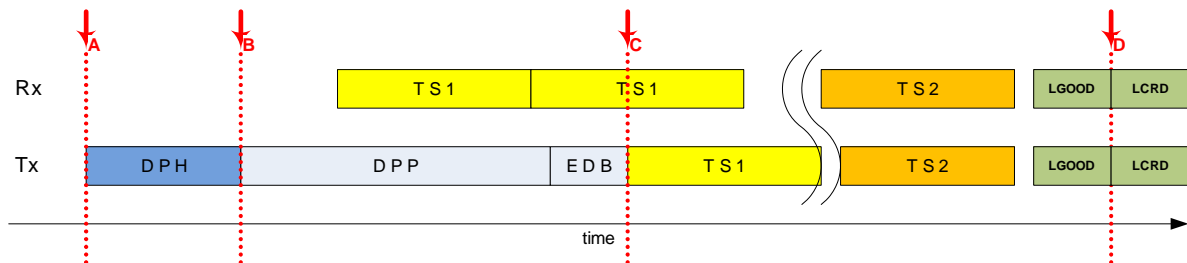
- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

Table 4-8 Status Attribute Chronology: Receiving TS1 Ordered Sets During DPP Tx

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “C” | at Point “D” |
|------------------|--------------------|-----------------|-----------------|-----------------|-----------------|
| status | INITIAL | ACTIVE | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT | ACCEPT |
| payload_status | INITIAL | INITIAL | ACTIVE | ACCEPT | ACCEPT |
| payload_presence | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT |

4.3.6.2.2 During DPP Tx, EDB Result

A TS1 ordered set is received during the transmission of a DPP payload (due to the link partner entering recovery state). The super-speed transmitter aborts production of the payload and ends the DPP with appropriate EDB framing. After recovery, the LGOOD advertisement indicates a header sequence number prior to the DPP in question (such that the DPP packet will be replayed).

Figure 4-12 Packet Chronology: Receiving TS1 Ordered Sets During DPP Tx, EDB Result

At points “A” and “B” the events are identical to [Figure 4-5](#) at points “A” and “B”

At point “C” the following packet attributes are updated:

- ❖ status is set to PARTIAL_ACCEPT
- ❖ payload_status is set to ABORTED
- ❖ payload_presence is set to PAYLOAD_PRESENT_BUT_ABORTED
- ❖ packet_end_time is set to \$realtime

At point “D” the following packet attributes are updated:

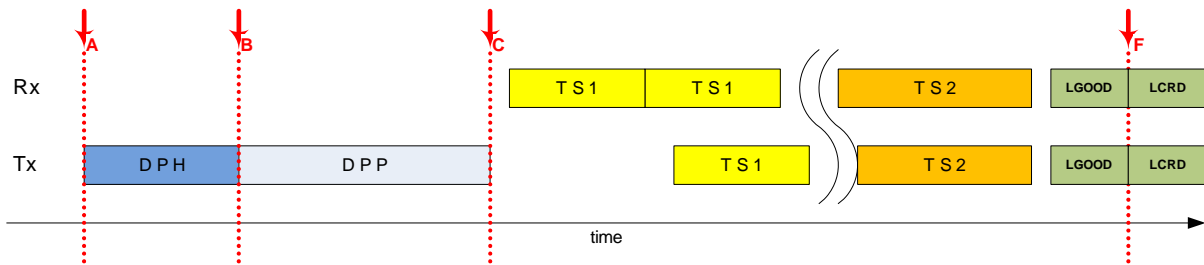
- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

Table 4-9 Status Attribute Chronology: Receiving TS1 Ordered Sets During DPP Tx, EDB Result

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “C” | at Point “D” |
|------------------|--------------------|-----------------|-----------------|-----------------------------|-----------------------------|
| status | INITIAL | ACTIVE | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT | ACCEPT |
| payload_status | INITIAL | INITIAL | ACTIVE | ABORTED | ABORTED |
| payload_presence | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT_BUT_ABORTED | PAYLOAD_PRESENT_BUT_ABORTED |

4.3.6.2.3 After DPP Tx Complete

A TS1 ordered set is received after the complete transmission of a DPP payload (due to the link partner entering recovery state). After recovery, the LGOOD advertisement indicates a header sequence number prior to the DPP in question (such that the DPP packet will be replayed).

Figure 4-13 Packet Chronology: Receiving TS1 Ordered Sets After DPP Tx Complete

At points “A” through “C” the events are identical to [Figure 4-7](#) at points “A” through “C”

At point “F” the following packet attributes are updated:

- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

Table 4-10 Status Attribute Chronology: Receiving TS1 Ordered Sets After DPP Tx Complete

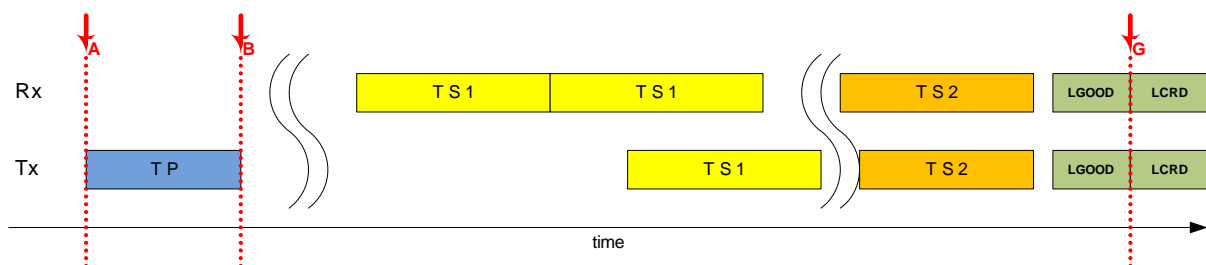
| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “C” | at Point “F” |
|----------------|--------------------|--------------|--------------|----------------|--------------|
| status | INITIAL | ACTIVE | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT | ACCEPT |
| payload_status | INITIAL | INITIAL | ACTIVE | ACCEPT | ACCEPT |

Table 4-10 Status Attribute Chronology: Receiving TS1 Ordered Sets After DPP Tx Complete

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “C” | at Point “F” |
|------------------|--------------------|-----------------|-----------------|-----------------|-----------------|
| payload_presence | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT |

4.3.6.2.4 After HP, No DPP

A TS1 ordered set is received after the complete transmission of Header Packet data (due to the link partner entering recovery state). No payload is associated to the header (this could be a TP, ISOC, ITP or LMP packet). After recovery, the LGOOD advertisement indicates a header sequence number prior to the DPH in question (such that the DPH packet will be replayed).

Figure 4-14 Packet Chronology: Receiving TS1 Ordered Sets After HP, No DPP

At points “A” and “B” the events are identical to [Figure 4-8](#) at points “A” and “B”

At point “G” the following packet attributes are updated:

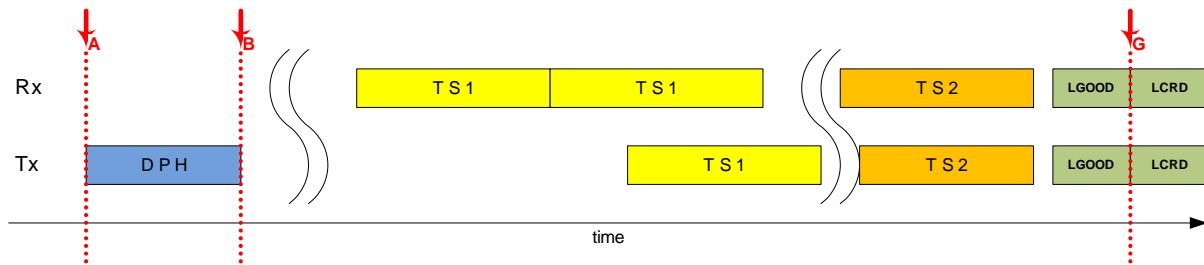
- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

Table 4-11 Status Attribute Chronology: Receiving TS1 Ordered Sets After HP, No DPP

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “G” |
|------------------|---------------------|---------------------|---------------------|---------------------|
| status | INITIAL | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT |
| payload_status | DISABLED | DISABLED | DISABLED | DISABLED |
| payload_presence | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT |

4.3.6.2.5 After Data Packet Replay

A TS1 ordered set is received after the complete transmission of a replay Data Packet; no payload is associated to the replay packet. After recovery, the LGOOD advertisement indicates a header sequence number prior to the DPH in question (such that the DPH packet will be replayed).

Figure 4-15 Packet Chronology: Receiving TS1 Ordered Sets After Data Packet replay

At points “A” and “B” the events are identical to [Figure 4-9](#) at points “A” and “B”

At point “G” the following packet attributes are updated:

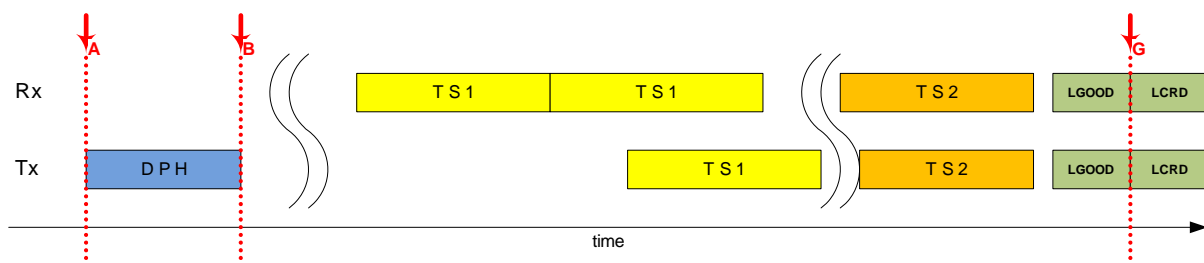
- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

Table 4-12 Status Attribute Chronology: Receiving TS1 Ordered Sets After Data Packet replay

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “G” |
|------------------|--------------------|---------------------|---------------------|---------------------|
| status | RETRY | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | RETRY | ACTIVE | ACCEPT | ACCEPT |
| payload_status | (various possible) | CANCELLED | CANCELLED | CANCELLED |
| payload_presence | (various possible) | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT |

4.3.6.2.6 After Non-Data Packet Replay

A TS1 ordered set is received after the complete transmission of a replay packet (due to the link partner entering recovery state). No payload was previously associated with this header (this could be a TP, ISOC, ITP or LMP packet). After recovery, the LGOOD advertisement indicates a header sequence number prior to the DPH in question (such that the DPH packet will be replayed).

Figure 4-16 Packet Chronology: Receiving TS1 Ordered Sets After Data Packet Replay

At points “A” and “B” the events are identical to [Figure 4-8](#) at points “A” and “B”

At point “G” the following packet attributes are updated:

- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

Table 4-13 Status Attribute Chronology: Receiving TS1 Ordered Sets After Data Packet Replay

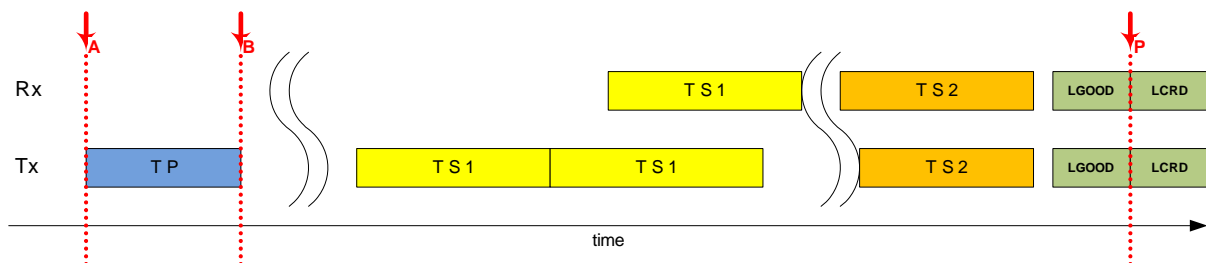
| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “G” |
|------------------|---------------------|---------------------|---------------------|---------------------|
| status | RETRY | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | RETRY | ACTIVE | ACCEPT | ACCEPT |
| payload_status | DISABLED | DISABLED | DISABLED | DISABLED |
| payload_presence | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT |

4.3.6.3 Transmitting TS1 Ordered Sets

4.3.6.3.1 After HP Tx

The link layer can transition to recovery state after transmitting a Header Packet No payload is associated with this header (this could be a TP, ISOC, ITP or LMP packet).

Figure 4-17 Packet Chronology: Transmitting TS1 Ordered Sets After HP Tx



At points “A” and “B” the events are identical to [Figure 4-8](#) at points “A” and “B”

At point “P” the following packet attributes are updated:

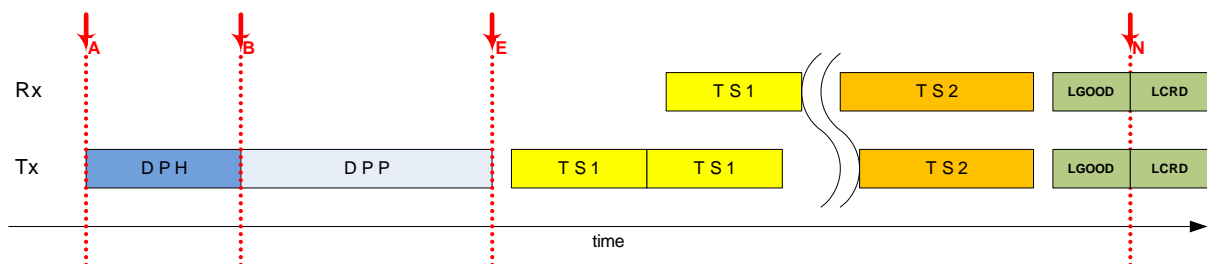
- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

Table 4-14 Status Attribute Chronology: Transmitting TS1 Ordered Sets After HP Tx

| | Prior to Point "A" | at Point "A" | at Point "B" | at Point "G" |
|------------------|---------------------|---------------------|---------------------|---------------------|
| status | INITIAL | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT |
| payload_status | DISABLED | DISABLED | DISABLED | DISABLED |
| payload_presence | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT |

4.3.6.3.2 After DPP Tx

The link layer can transition to recovery state after transmitting a Header Packet with DPP payload.

Figure 4-18 Packet Chronology: Transmitting TS1 Ordered Sets After DPP Tx

At points "A" through "E" the events are identical to [Figure 4-7](#) at points "A" through "E"

At point "N" the following packet attributes are updated:

- ❖ status attribute is set to RETRY
- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

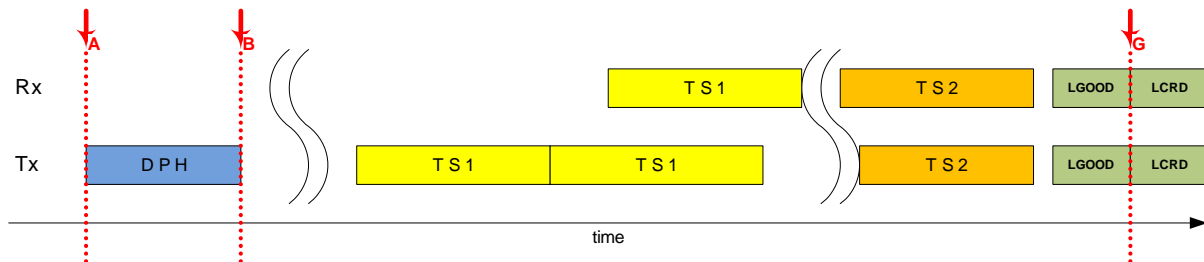
Table 4-15 Status Attribute Chronology: Transmitting TS1 Ordered Sets After DPP Tx

| | Prior to Point "A" | at Point "A" | at Point "B" | at Point "E" | at Point "F" |
|------------------|--------------------|-----------------|-----------------|-----------------|-----------------|
| status | INITIAL | ACTIVE | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT | ACCEPT |
| payload_status | INITIAL | INITIAL | ACTIVE | ACCEPT | ACCEPT |
| payload_presence | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT |

4.3.6.3.3 After Data Packet Replay

The link layer can transition to recovery state after transmitting a replay Data Packet; no payload is associated to the replay packet.

Figure 4-19 Packet Chronology: Transmitting TS1 Ordered Sets After Data Packet Replay



Events at points “A” through “G” are identical to [Figure 4-9](#) events at points “A” through “G”

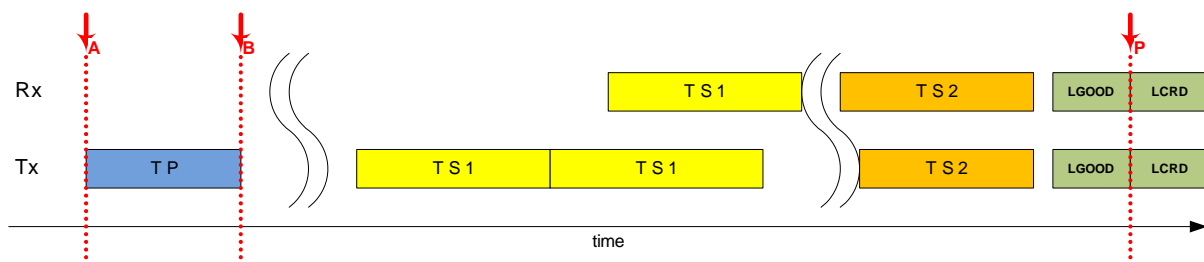
Table 4-16 Status Attribute Chronology: Transmitting TS1 Ordered Sets After Data Packet Replay

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “G” |
|------------------|--------------------|---------------------|---------------------|---------------------|
| status | RETRY | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | RETRY | ACTIVE | ACCEPT | ACCEPT |
| payload_status | (various possible) | CANCELLED | CANCELLED | CANCELLED |
| payload_presence | (various possible) | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT |

4.3.6.3.4 After Non-Data Packet Replay

The link layer can transition to recovery state after transmitting a replay Data Packet. No payload was previously associated with this header (this could be a TP, ISOC, ITP or LMP packet). After recovery, the LGOOD advertisement indicates a header sequence number prior to the DPH in question (such that the DPH packet will be replayed).

Figure 4-20 Packet Chronology: Transmitting TS1 Ordered Sets After Non-Data Packet Replay



At points “A” and “B” the events are identical to [Figure 4-8](#) at points “A” and “B”

At point “P” the following packet attributes are updated:

- ❖ status attribute is set to RETRY

- ❖ trace is stored for transmitted packet
- ❖ packet_start_time attribute is reset to 0
- ❖ packet_header_end_time attribute is reset to 0
- ❖ packet_end_time attribute is reset to 0

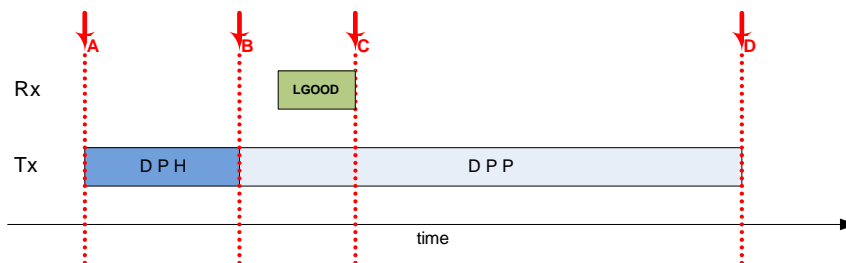
Table 4-17 Status Attribute Chronology: Transmitting TS1 Ordered Sets After Non-Data Packet Replay

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “G” |
|------------------|-------------------------|-------------------------|-------------------------|-------------------------|
| status | RETRY | ACTIVE | PARTIAL_ACCEPT | RETRY |
| header_status | RETRY | ACTIVE | ACCEPT | ACCEPT |
| payload_status | DISABLED | DISABLED | DISABLED | DISABLED |
| payload_presence | PAYLOAD_NOT_PRE SENT | PAYLOAD_NOT_PRE SENT | PAYLOAD_NOT_PRE SENT | PAYLOAD_NOT_PRE SENT |

4.3.6.4 Receiving LGOOD

4.3.6.4.1 During DPP Tx

An LGOOD link command is received (for the currently active packet) during the transmission of a DPP payload.

Figure 4-21 Packet Chronology: Receiving LGOOD During DPP Tx

At points “A” through “C” the events are identical to [Figure 4-5](#) at points “A” through “C”

At point “D” the following packet attributes are updated:

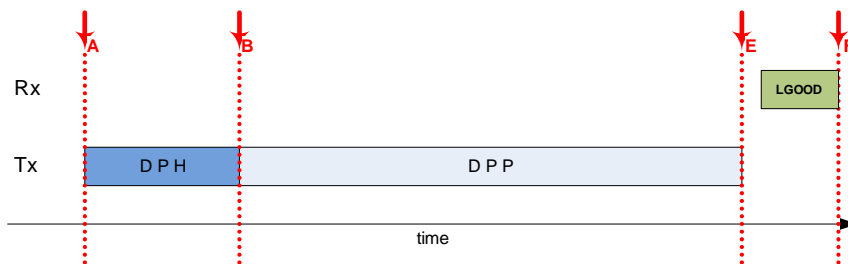
- ❖ status attribute is set to ACCEPT
- ❖ payload_status is set to ACCEPT
- ❖ packet_end_time is set to \$realtime
- ❖ end_time is set to \$realtime
- ❖ trace is stored for transmitted packet
- ❖ notify indicated uvm_event end_event (uvm_transaction’s end_event field) on packet

Table 4-18 Status Attribute Chronology: Receiving LGOOD During DPP Tx

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “C” | at Point “D” |
|------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
| status | INITIAL | ACTIVE | ACTIVE | ACTIVE | ACCEPT |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT | ACCEPT |
| payload_status | INITIAL | INITIAL | ACTIVE | ACTIVE | ACCEPT |
| payload_presence | PAYLOAD_ PRESENT | PAYLOAD_ PRESENT | PAYLOAD_ PRESENT | PAYLOAD_ PRESENT | PAYLOAD_ PRESENT |

4.3.6.4.2 After DPP Tx Complete

An LGOOD link command is received after the complete transmission of a DPP payload.

Figure 4-22 Packet Chronology: Receiving LGOOD After DPP Tx Complete

At points “A” and “B” the events are identical to [Figure 4-5](#) at points “A” and “B”

At point “E” the following packet attributes are updated:

- ❖ status is set to PARTIAL_ACCEPT
- ❖ payload_status is set to ACCEPT
- ❖ packet_end_time is set to \$realtime

At point “F” the following packet attributes are updated:

- ❖ status attribute is set to ACCEPT
- ❖ end_time is set to \$realtime
- ❖ trace is stored for transmitted packet
- ❖ notify on uvm_event end_event (uvm_transaction’s end_event field) on packet

Table 4-19 Status Attribute Chronology: Receiving LGOOD After DPP Tx Complete

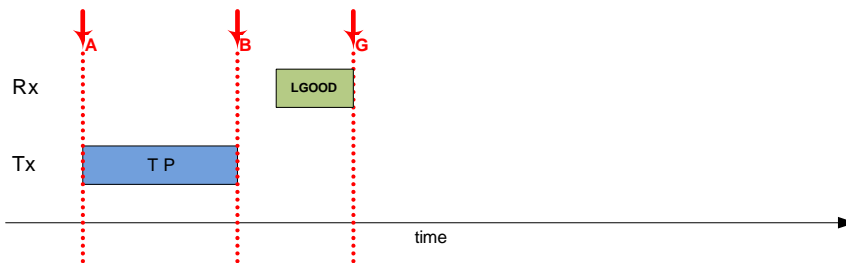
| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “E” | at Point “F” |
|----------------|--------------------|--------------|--------------|--------------------|--------------|
| status | INITIAL | ACTIVE | ACTIVE | PARTIAL_ ACCEPT | ACCEPT |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT | ACCEPT |
| payload_status | INITIAL | INITIAL | ACTIVE | ACCEPT | ACCEPT |

Table 4-19 Status Attribute Chronology: Receiving LGOOD After DPP Tx Complete

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “E” | at Point “F” |
|------------------|--------------------|-----------------|-----------------|-----------------|-----------------|
| payload_presence | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT | PAYLOAD_PRESENT |

4.3.6.4.3 After HP, No DPP

An LGOOD link command is received after the complete transmission of Header Packet data; no payload is associated to the header (this could be a TP, ISOC, ITP or LMP packet).

Figure 4-23 Receiving LGOOD After HP, No DPP

At points “A” and “B” the events are identical to [Figure 4-8](#) at points “A” and “B”

At point “G” the following packet attributes are updated:

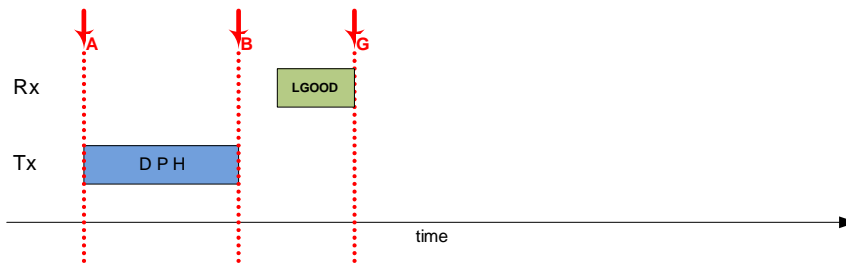
- ❖ status attribute is set to ACCEPT
- ❖ end_time is set to \$realtime
- ❖ trace is stored for transmitted packet
- ❖ notify indicated uvm_event end_event (uvm_transaction’s end_event field) on packet

Table 4-20 Status Attribute Chronology: Receiving LGOOD After HP, No DPP

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “G” |
|------------------|---------------------|---------------------|---------------------|---------------------|
| status | INITIAL | ACTIVE | PARTIAL_ACCEPT | ACCEPT |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT |
| payload_status | DISABLED | DISABLED | DISABLED | DISABLED |
| payload_presence | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT |

4.3.6.4.4 After Data Packet Replay

An LGOOD link command is received after the complete transmission of replay Data Packet; no payload is associated to the replay packet.

Figure 4-24 Packet Chronology: Receiving LGOOD After Data Packet Replay

Events at points “A” and “B” are identical to [Figure 4-9](#) events at points “A” and “B”

At point “G” the following packet attributes are updated:

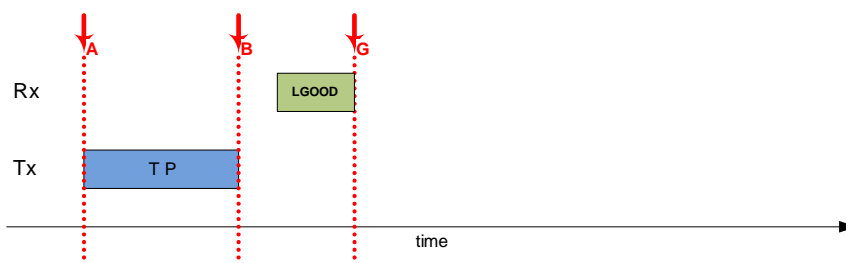
- ❖ status is set to ACCEPT
- ❖ end_time is set to \$realtime
- ❖ trace is stored for transmitted packet
- ❖ notify indicated uvm_event end_event (uvm_transaction’s end_event field) on packet

Table 4-21 Status Attribute Chronology: Receiving LGOOD After Data Packet Replay

| | Prior to Point “A” | at Point “A” | at Point “B” | at Point “G” |
|------------------|--------------------|---------------------|---------------------|---------------------|
| status | INITIAL | ACTIVE | PARTIAL_ACCEPT | ACCEPT |
| header_status | INITIAL | ACTIVE | ACCEPT | ACCEPT |
| payload_status | (various possible) | DISABLED | DISABLED | DISABLED |
| payload_presence | (various possible) | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT | PAYLOAD_NOT_PRESENT |

4.3.6.4.5 After Non-Data Packet Replay

An LGOOD link command is received after the complete transmission of replay Data Packet; no payload was previously associated with this header (this could be a TP, ISOC, ITP or LMP packet).

Figure 4-25 Packet Chronology: Receiving LGOOD After Non-Data Packet Replay

At point “A” and “B” the events are identical to [Figure 4-5](#) at points “A” and “B”

At point “G” the following packet attributes are updated:

- ❖ status is set to ACCEPT

- ❖ end_time is set to \$realtime
- ❖ trace is stored for transmitted packet
- ❖ notify indicated uvm_event end_event (uvm_transaction's end_event field) on packet

Table 4-22 Status Attribute Chronology: Receiving LGOOD After Non-Data Packet Replay

| | Prior to Point "A" | at Point "A" | at Point "B" | at Point "G" |
|------------------|-------------------------|-------------------------|-------------------------|-------------------------|
| status | RETRY | ACTIVE | PARTIAL_ACCEPT | ACCEPT |
| header_status | RETRY | ACTIVE | ACCEPT | ACCEPT |
| payload_status | DISABLED | DISABLED | DISABLED | DISABLED |
| payload_presence | PAYLOAD_ NOT_PRESENT | PAYLOAD_ NOT_PRESENT | PAYLOAD_ NOT_PRESENT | PAYLOAD_ NOT_PRESENT |

4.3.7 Related Topics About Link Component

| FOR INFO ABOUT: | SEE: |
|---|--|
| <ul style="list-style-type: none">Link service commands | <ul style="list-style-type: none">The HTML class reference. |
| <ul style="list-style-type: none">Factories, callbacks, and ports | <ul style="list-style-type: none">Complete list of UVM factories, callbacks, and ports used by the Link Component in the HTML class reference. |

4.4 Physical Component

USB VIP Physical components are component objects in a UVM-compliant verification environment. The USB Physical component object is extended from the `uvm_component` class. This object implements all of the methods specified by UVM for the `uvm_agent` class.

The USB Physical component provides TLM ports to deal with the USB3.0 SuperSpeed and USB 2.0 HS/FS/LS traffic, as well as supporting service request input. This includes:

- ❖ The USB Physical component supports automated exception generation on its input ports.
- ❖ The USB Physical component supports the specification of factory objects for its output ports.
- ❖ The USB Physical component supports the general UVM events (e.g., `begin_event`) as well as several USB VIP specific `uvm_events` (i.e. `POWER_ON_RESET`). For those `uvm_events` where the current `sequence_item` is pertinent, a handle to the current `sequence_item` is included as part of the `uvm_event` data.
- ❖ Physical layer processing: VIP Physical components model the data processing (i.e. 8b10b encoding/decoding) and event generating/detecting (i.e. LFPS transmission/detection) associated with the physical layer of the USB protocol. This includes the ability to inject/detect errors associated with the physical layer (i.e. 8b10b encode/decode error).
- ❖ Verification signal interface: VIP Physical components connect to simulations through a signal interface. Physical components support three types of signal interfaces:
 - ◆ Serial Interface (SS or 2.0)
 - ◆ PIPE3-MAC (PIPE3-MAC interface to Vendor PHY)
 - ◆ PIPE3-PHY (PIPE3-PHY interface to DUT MAC)

The [Class Reference HTML](#) describes Physical component functions and attributes.

4.4.1 Physical Layer Feature Support

“[Physical Layer Features](#)” on page 27 lists the physical layer features supported by the USB VIP. The following is a list of supported protocol layer verification features:

- ❖ SS and 2.0 data input/output ports
- ❖ In/Out Service
 - reset, Vbus ON/OFF, Attach/Detach, Power State, Scrambling ON/OFF, Rx/Tx - Polarity, Rx/Tx LFPS ON/OFF
- ❖ Configurable input port stimulus
 - ◆ Auto connect to link layer
 - ◆ Direct port
- ❖ Error injection
- ❖ Callbacks providing testbench visibility and control

4.4.2 Data Flow Support

4.4.2.1 Physical Component Ports

UVM Ports are the mechanism through which the Physical Layer component connects to other components in the USB VIP sub-environment (svt_usb_agent) and/or to the testbench. The Physical component contains various types of ports:

- ❖ Transfer Input ports. These are used by the sequencer to send sequences into the physical layer. You cannot connect to them. They are only used by the sequencer. Consult UVM documentation on how to use sequence related classes.
- ❖ Observed ports (analysis ports). You take data from these ports and use them for either generating inputs into response (output ports), or for creating scoreboards and coverage checks.
- ❖ Response or Transfer Out ports. You use these ports to place data into the physical layer.

UVM Ports utilized for the USB SS data path are:

- ❖ **usb_ss_link_data_in_port**. Input port for USB SS data transactions to be sent by this component.
- ❖ **usb_ss_link_data_observed_port**. Analysis port for USB SS data transactions received by this component.
- ❖ **usb_ss_physical_data_in_port**. Input port for USB SS data transactions received by this component.
- ❖ **usb_ss_physical_data_out_port**. Output port for USB SS data transactions to be sent by this component.
- ❖ **usb_ss_physical_service_in_port**. Input port for USB SS service requests to this component.

UVM Ports utilized for the USB 2.0 data path are:

- ❖ **usb_20_link_data_in_port**. Input port for USB 2.0 data transactions to be sent by this component.
- ❖ **usb_20_link_data_observed_port**. Analysis port for USB 2.0 data transactions received by this component.
- ❖ **usb_20_physical_data_in_port**. Input port for USB 2.0 data transactions received by this component.
- ❖ **usb_20_physical_data_out_port**. Output port for USB 2.0 data transactions to be sent by this component.
- ❖ **usb_20_physical_service_in_port**. Input port for USB 2.0 service requests to this component.

4.4.2.2 Signal Interfaces

The HTML reference documentation contains a complete listing of pins within the signal interfaces listed below.

4.4.2.2.1 PIPE3 Interfaces

This SystemVerilog 'interface' definition declares all of the signals (scoped within an instance of this interface) that are specified by the PIPE3 Specification as being included in a USB PIPE3 connection

The interface declares 'clocking blocks' that define the clock synchronization and directionality of interface signals used by the SVT USB Physical component, and declares 'modports' (which reference the clocking blocks) that are to be used as logical port connections for the SVT USB Physical port `#svt_usb_physical_port_pipe3`.

The following lists the PIPE3 signal interfaces into the model:

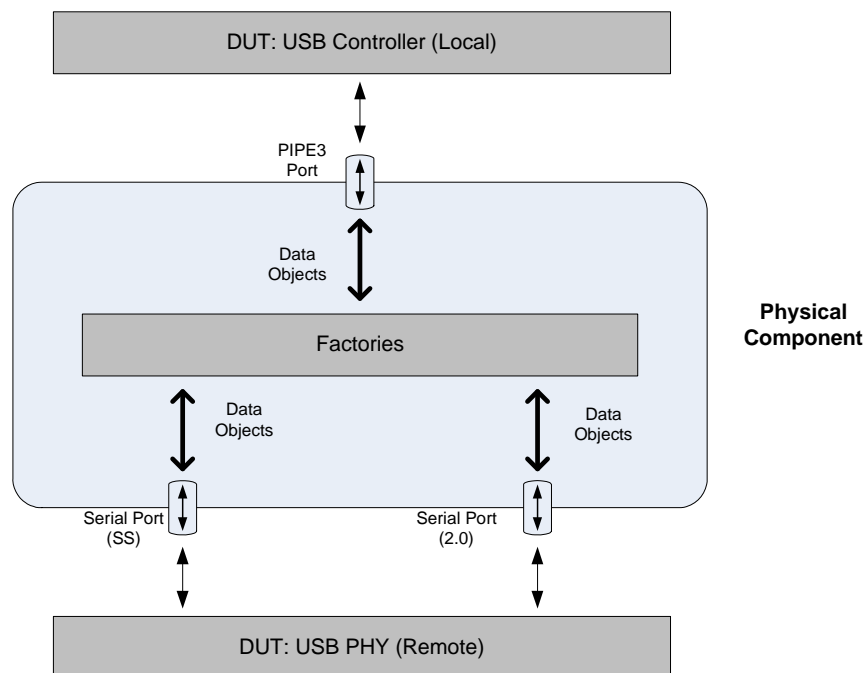
- ❖ `interface svt_usb_pipe3_dut_mac_if()`.
- ❖ `interface svt_usb_pipe3_dut_phy_if()`.

4.4.2.2.2 Serial Interfaces

The following are the serial interfaces supported by the model:

- ❖ `svt_usb_20_hsic_if ()`. This interface defines the USB 2.0 HSIC signal interface used by the SVT USB component to communicate with a DUT.
- ❖ `svt_usb_20_serial_if ()`. This interface defines the USB 2.0 serial signal interface used by the SVT USB component to communicate with a DUT.
- ❖ `svt_usb_if ()`. This interface defines a top-level interface used by the SVT USB component to hold an instance of each supported signal interface that can be used for communication with a DUT.
- ❖ `svt_usb_otg_if ()`. This interface defines a signal interface used by the SVT USB component to support simulation of OTG related behavior within a testbench.
- ❖ `svt_usb_ss_serial_if ()`. This interface defines the USB SS serial signal interface used by the SVT USB component to communicate with a DUT.

Figure 4-26 Physical Component data flow through Port Objects



Signal interfaces use 4-state logic for communicating across physical signals with the following exception:

- ❖ **USB 2.0 serial signal:** This interface models the serial bus using 9-state logic in combination with wired-OR outputs.

This matches the Synopsys nanoPHY specification. Refer to the DesignWare Cores USB 2.0 nanoPHY One-Port Databook.

USB Physical components do not support the simultaneous configuration of a signal interface on link and physical interfaces. Signal interfaces are only supported on one component side at a time.

Refer to [“Interface Options”](#) on page 94 for a description of valid Physical transaction configurations that utilize signal interfaces.

4.4.2.3 Data Objects

The following is a list of objects that represent information the Physical component receives, sends, or processes.

[“Sequence Item Data Objects”](#) on page 22 describes USB data objects

- ❖ **Data Objects:** These objects that represent USB data transfer that flow between the USB Protocol layer and the entity accessing the USB bus.
- ❖ **Physical Service Transaction Objects:** These objects represent service requests that initiate physical layer events or control physical layer operations.

4.4.2.4 Transaction Support

USB Physical components support using transactions to model data flow – byte and non-byte.

- ❖ **Byte data** – Transmission and reception of data
- ❖ **Non-byte data** – Driving and detecting linestate during idle

Physical components support data flow transactions by providing data transformations associated with data flow in the physical layer:

- ❖ USB Transmit Transformations
 - ◆ Scrambles the byte value
 - ◆ 8b10b encodes the scrambled byte value
- ❖ USB Receive Transformations
 - ◆ Addition or removal of SKP sets
 - ◆ 8b10b decodes the encoded byte value
 - ◆ Unscrambles the decoded byte value
- ❖ USB 2.0 Transmit Transformations
 - ◆ Bit stuffs the byte value
 - ◆ NRZI encodes the bit stuffed byte value
 - ◆ Inserts SYNC/EOP at packet start/end
- ❖ USB 2.0 Receive Transformations
 - ◆ Removes SYNC/EOP at packet start/end
 - ◆ Addition or removal of symbols
 - ◆ NRZI decodes the encoded byte value
 - ◆ Bit un-stuffs the decoded byte value

4.4.3 Interface Options

The VIP Physical Layer component supports several verification topologies. The following sections describe these topologies.

4.4.3.1 Connecting the Physical component to a Link component and a remote Physical component

This Physical component configuration is used within the local USB protocol stack when verifying a USB core that is connected to the PHY interface of the remote Physical component; the core is connected to the other side of the remote Physical component, as described in [“Connecting the Physical Component to a Link Component and a Remote DUT”](#) on page 94. When connecting Link layer and a remote Physical Layer components, ports typically exchange information between the components.

4.4.3.2 Connecting the Physical Component to a Link Component and a Remote DUT PHY

This Physical component configuration is used when connecting the USB protocol stack to the testbench or a DUT. The Physical component exchanges information with the testbench through serial bus ports.

4.4.3.3 Connecting the Physical Component to a Link Component and a Remote DUT

This Physical component configuration is typically used to verify a USB controller independently of its PHY. The Physical component exchanges information with the testbench through PIPE3 bus ports.

4.4.3.4 Connecting the Physical Component to a Link Layer Component and a local PHY (DUT)

This Physical component configuration is typically used to verify a USB PHY. The Physical component exchanges information with the testbench through PIPE3 ports.

4.4.4 Interface File Features

Interface files implement features that UVM environments cannot directly model, such as clock generation. Features best suited for modeling in Verilog, such as clock recovery, are also implemented in interface files.

The following interface related features are implemented directly within the appropriate interface files:

- ❖ PIPE3 PCLK generation – component modeling a PHY
- ❖ Serial clock recovery – receive data/clock recovery
- ❖ USB 2.0 serial signal modeling – (9-state signaling)

The following features use state variables in interface files to maintain a compatible usage model with Synopsys PHY IIP simulation models:

- ❖ SS Receiver Detection
- ❖ HS Disconnect Detection

4.4.5 Information Transformation Objects

The following list describes Physical component objects that manipulate data objects.

The physical component creates Factory data objects. To create them, the component uses UVM factory capabilities, in combination with the override capabilities to come up with templates that can be used to create these objects.

The 'overrides' are related to the 'override_by_type' and 'override_by_name' methods in the `uvm_factory` class:

- ❖ `set_inst_override_by_type`

- ❖ `set_inst_override_by_name`
- ❖ `etc.`

You can use one of these override methods to define a "factory" for objects based on a certain name or type.

4.4.5.1 Exception List Factories

USB Physical components support exception list factories associated with each of its input ports. Exception List factories are null by default; null factories are not randomized.

Assign factories to USB Physical components to generate exception lists. Replacement factories are either assigned by the USB Physical component's constructor or replaced later. Factories are generally replaced with an extended version.

- ❖ **link data exception list factory:** These factories randomize data descriptors received from link components. Exception lists resulting from randomization are applied to the data descriptor.

Component attribute names: *randomized_usb_ss_link_data_exception_list*,
randomized_usb_20_link_data_exception_list.

- ❖ **physical data exception list factory:** These factories randomize data descriptors received from physical components. Exception lists resulting from randomization are applied to the data descriptor.

Component attribute names: *randomized_usb_ss_physical_data_exception_list*,
randomized_usb_20_physical_data_exception_list.

4.4.6 Exception Support

Physical components support injection and detection of errors associated with the USB physical layer.

4.4.6.1 Error Injection

USB Physical components inject physical layer-specific exceptions. Exceptions are applied to the data transformation associated with the injected error.

The addition and removal of SKP ordered is implemented through error injection, although they are not errors. Error injection models the differences between transmit and receive clock domains.

- ❖ USB Transmit Error Injection
 - ◆ 8b10b disparity errors
 - ◆ 8b10b encoding errors
- ❖ USB Receive Error Injection
 - ◆ Addition of SKP ordered-sets
 - ◆ Removal of SKP ordered-sets
 - ◆ Forcing buffer overflow
 - ◆ Forcing buffer underflow
- ❖ USB 2.0 Transmit Error Injection
 - ◆ Bit stuff errors
 - ◆ SYNC error (at packet start)
 - ◆ EOP errors (at packet end)
- ❖ USB 2.0 Receive Error Injection

- ◆ Forcing buffer overflow
- ◆ Forcing buffer underflow

4.4.6.2 Error Detection

The USB Physical component detects physical layer-specific exceptions. Exceptions are reported at the data transformation associated with the detected error and recorded in the transaction's `exception_list`.

The detection of SKP ordered additions and removals is implemented through error injection, although they are not errors. Error injection reports the differences between transmit and receive clock domains.

- ❖ USB Receive Error Detection
 - ◆ Detection of the addition of a SKP ordered-sets
 - ◆ Detection of the removal of a SKP ordered-sets
 - ◆ Detection of buffer overflow
 - ◆ Detection of buffer underflow
 - ◆ Detection of 8b10b disparity errors
 - ◆ Detection of 8b10b decoding errors
- ❖ USB 2.0 Receive Error Detection
 - ◆ Detection of SYNC errors (at packet start)
 - ◆ Detection of EOP errors (at packet end)
 - ◆ Detection of buffer overflow
 - ◆ Detection of buffer underflow
 - ◆ Detection of bit stuff errors

4.4.7 UVM Event Support

Physical components use `uvm_events` to facilitate the communication of state and event control/status between other USB stack components and the testbench. UVM events are defined in the shared notify object

The following are examples for using physical-layer `uvm_events`

- ◆ State – Track the VBUS state (on/off)
- ◆ Control – Respond to receiver detect requests
- ◆ Status – Report the LFPS detection

See the HTML Class Reference for more information.

4.4.8 Physical Component Callbacks

The `svt_usb_physical_callback` class is extended from the `svt_xactor_callbacks` base class, which is extended from the `uvm_callback` class. These objects implement all of the methods specified by UVM for the `uvm_callback` class

The Physical component supports the following callbacks

- ❖ **Port input:** These callbacks indicate that the component collected data from an input port.

Physical component callback method names: `post_usb_ss_link_data_in_port_get`,
`post_usb_20_physical_service_in_port_get`, `post_usb_20_link_data_in_port_get`,
`post_usb_20_physical_data_in_port_get`, `usb_20_link_data_in_ended`,
`post_usb_ss_physical_data_in_port_get`, `post_usb_ss_physical_service_in_port_get`,
`usb_20_link_data_in_port_cov`, `usb_20_physical_data_in_port_cov`, `usb_20_physical_service_in_port_cov`,
`usb_20_service_in_ended`, `usb_ss_link_data_in_port_cov`, `usb_ss_link_data_in_ended`,
`usb_ss_physical_data_in_port_cov`, `usb_ss_physical_service_in_port_cov`, `usb_ss_service_in_ended`

- ❖ **Output port:** These callbacks indicate that the component placed data on an output port.

Physical component callback method names: `pre_usb_ss_link_data_out_port_put`,
`pre_usb_ss_physical_data_out_port_put`, `pre_usb_20_link_data_out_port_put`,
`pre_usb_20_physical_data_out_port_put`, `usb_20_link_data_out_port_cov`, `usb_20_link_data_out_ended`,
`usb_20_physical_data_out_port_cov`, `usb_ss_link_data_out_port_cov`, `usb_ss_link_data_out_ended`,
`usb_ss_physical_data_out_port_cov`

- ❖ **Rx Event:** These callbacks indicate an event related to data stream received from the remote Physical component.

Physical component callback method names: `pre_usb_20_rx_drive`, `pre_usb_ss_rx_drive`,
`post_usb_20_rx_sample`, `post_usb_ss_rx_sample`

- ❖ **Tx Event:** These callbacks indicate an event related to the transmission of a data stream to the remote Physical component.

Physical component callback method names: `pre_usb_ss_tx_drive`, `post_usb_20_tx_sample`,
`pre_usb_20_tx_drive`, `post_usb_ss_tx_sample`

- ❖ **Error Injection:** This callback indicate errors were injected on one or more request signals on the interface.

Physical component callback method names: `errors_driven`

- ❖ **Corrupted Transaction:** This callback indicates a corrupted transaction was generated or detected.

Physical component callback method names: `transaction_invalid_traffic`

4.4.9 Related Topics About Physical Component

| FOR INFO ABOUT: | SEE: |
|---|--|
| <ul style="list-style-type: none">Physical service commands | <ul style="list-style-type: none">The HTML class reference. |
| <ul style="list-style-type: none">Factories, callbacks, and ports | <ul style="list-style-type: none">Complete list of UVM factories, callbacks, and ports used by the Physical component in the HTML class reference. |



5

Using the USB Verification IP

5.1 Introduction

This chapter presents UVM concepts and techniques for quickly achieving a basic constrained random testbench that incorporates the USB VIP. Code snippets illustrate these methods in practical use. The testbench shows typical USB VIP and SystemVerilog UVM usage, and highlights the concepts and techniques described. These techniques can be used with any of the VIP products.

5.2 SystemVerilog UVM Example Testbenches

This section describes SystemVerilog UVM example testbenches that show general usage for various applications. A summary of the examples is listed in [Table 5-1](#).

Table 5-1 SystemVerilog Example Summary

| Name | Source in design_dir/Description |
|--|--|
| Basic Example --Demonstrates how to implement a UVM testbench using USB VIP with SS Serial interface. This example consists of a top-level testbench, a Verilog <code>hdl_interconnect</code> , a UVM verification environment, a host, hub and device agent components, one test file with an sequence, and four directed tests. | |
| tb_usb_svt_uvm_basic_router_sys | examples/sverilog/usb_svt/tb_usb_svt_uvm_basic_router_sys/ |
| tb_usb_svt_uvm_basic_router_sys | .../usb_svt/test/sverilog/tb_usb_svt_uvm_basic_router_sys |

Configuring Hub Agent:

Testbench: `tb_usb_svt_uvm_basic_router_sys/env/`.

Has a file named `usb_hub_shared_cfg` consists of Host, Hub, Device Agent configuration as:

```
/**
 * Configuration object used by the HOST Driver objects.
 */
svt_usb_agent_configuration host_cfg;
/**
 * Configuration object used by the HUB Driver objects.
 */
svt_usb_hub_agent_configuration hub_cfg[];
/**
 * Configuration object used by the Device Driver objects.
 */
svt_usb_agent_configuration dev_cfg[];
```

Macro used to define Number of Hub instance per tier (Currently_supported 1 Tier):

```
`SVT_REPLACEABLE_DEFINE(SVT_USB_NUM_HUB_INST, 1)
```

Macro used to define Number of Device on Hub downstreams:

```
`SVT_REPLACEABLE_DEFINE(SVT_USB_NUM_DEVICE_INST, 1)
```

Macro used to define Number of Downstream port on each Hub:

```
`SVT_REPLACEABLE_DEFINE(SVT_USB_NUM_OF_DOWNSTREAM_PORTS_INST, 1)
```

Depending on above macros the configuration of each Host, Device, and Hub Agents are done.

5.2.1 For example (Speed = SS),

```
foreach(hub_cfg[hub_inst])
begin
```

```
        foreach(hub_cfg[hub_inst].downstream_cfg[j])
begin
    hub_cfg[hub_inst].downstream_cfg[j].capability = svt_usb_configuration::PLAIN;
    hub_cfg[hub_inst].downstream_cfg[j].speed = svt_usb_types::SS;
    hub_cfg[hub_inst].downstream_cfg[j].usb_20_signal_interface =
svt_usb_configuration::NO_20_IF;
    hub_cfg[hub_inst].downstream_cfg[j].usb_ss_signal_interface =
svt_usb_configuration::USB_SS_SERIAL_IF;
    hub_cfg[hub_inst].downstream_cfg[j].usb_capability =
svt_usb_configuration::USB_SS_ONLY;
    hub_cfg[hub_inst].downstream_cfg[j].usb_ss_initial_ltssm_state =
svt_usb_types::RX_DETECT;
    hub_cfg[hub_inst].downstream_cfg[j].usb_ss_rx_buffer_mode =
svt_usb_configuration::NOMINAL_EMPTY;
    hub_cfg[hub_inst].downstream_cfg[j].usb_ss_rx_buffer_latency = 0;
end

    hub_cfg[hub_inst].upstream_cfg.capability = svt_usb_configuration::PLAIN;
    hub_cfg[hub_inst].upstream_cfg.speed = svt_usb_types::SS;
    hub_cfg[hub_inst].upstream_cfg.usb_20_signal_interface =
svt_usb_configuration::NO_20_IF;
    hub_cfg[hub_inst].upstream_cfg.usb_ss_signal_interface =
svt_usb_configuration::USB_SS_SERIAL_IF;
    hub_cfg[hub_inst].upstream_cfg.usb_capability =
svt_usb_configuration::USB_SS_ONLY;
    hub_cfg[hub_inst].upstream_cfg.usb_ss_initial_ltssm_state =
svt_usb_types::RX_DETECT;
    hub_cfg[hub_inst].upstream_cfg.usb_ss_rx_buffer_mode =
svt_usb_configuration::NOMINAL_EMPTY;
    hub_cfg[hub_inst].upstream_cfg.usb_ss_rx_buffer_latency = 0;
```

5.2.2 Configuring Endpoints:

```
for (int ep_num = 0; ep_num < max_usb_ss_hub_endpoints; ep_num++)
begin
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[ep_num] =
new();
end

    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].device_address = hub_inst;
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].connected_bus_speed =
svt_usb_types::SS;
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].connected_hub_device_address
= 0;
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].functionality_support =
svt_usb_types::SS;
```

```

        hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].num_endpoints =
max_usb_ss_hub_endpoints;

        if (max_usb_ss_hub_endpoints >= 1)
begin
    /**
     * Configuring End point '0' to CONTROL
     * NOTE: 'direction' is irrelevant for a CONTROL endpoint. Though
     * direction is chosen as 'IN' here, this endpoint will support both
     * IN and OUT transfers
     */
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[0].ep_number =
0;
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[0].direction =
svt_usb_types::IN;
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[0].ep_type =
svt_usb_types::CONTROL;
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[0].interval =
1;

    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[0].max_burst_size = 0;

    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[0].max_packet_size =
`SVT_USB_SS_CONTROL_MAX_PACKET_SIZE ;

    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[0].supports_ustreams =
1'b0;
end

        if (max_usb_ss_hub_endpoints >= 1)
begin
    /**
     * Configuring End point '1' to INTERRUPT IN
     */
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[1].ep_number =
1;
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[1].direction =
svt_usb_types::IN;
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[1].ep_type
= svt_usb_types::INTERRUPT;
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[1].allow_aligned_transf
er_without_zero_length = 0;
    hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[1].interval =
1;

```

```
hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[1].max_packet_size =  
`SVT_USB_SS_MAX_PACKET_SIZE;  
  
hub_cfg[hub_inst].upstream_cfg.local_device_cfg[0].endpoint_cfg[1].supports_ustreams =  
1'b0;  
end  
end
```

5.3 Creating Transactions Using UVM Sequencers

The USB VIP sequence collections define sequences consistent with UVM. This section describes the process of setting up a sequencer to create new USB sequence_item data objects (which may for example be sent as an input to the Physical component's Transmit path).

Do the following:

- ❖ Extend the sequence class (e.g. class my_seq extends svt_usb_data_sequence), and define the body of the sequence to apply user-specific constraints to control generated objects based on that data.
- ❖ Instantiate a svt_usb_data_sequencer object (e.g. my_sequencer) in the testbench. This is one of the classes defined by the VIP, and is derived from uvm_sequencer.
- ❖ Instantiate a my_seq object in the testbench, registering it with my_sequencer. Register my_sequencer as one of the data sequencers (usb_ss_data_sequencer or usb_20_data_sequencer) with the agent.

The UVM sequencers have a number of other features such as 'do' macros, virtual sequencers, and sequence libraries.

5.4 SuperSpeed Low Power Entry Support

5.4.1 Overview

Link power management reduces power consumption when link partners are idle. The following Link Training and Status State Machine (LTSSM) operational states manages link power.

- ❖ U0 (link active): The fully operational link active state. Packets of any type may be communicated over links in the U0 state.
- ❖ U1 (link standby with fast exit): Power saving state characterized by fast transition to the U0 State.
- ❖ U2 (link standby with slower exit): Power saving state characterized by greater power savings at the cost of increased exit latency.
- ❖ U3 (suspend): Deep power saving state where portions of device power may be removed except as needed for limited functions.

After software configuration, the U1 and U2 link states are entered and exited through hardware autonomous control. The U3 link state is entered only under software control, typically after a software inactivity timeout expiry, and is exited either by software (host initiated exit) or hardware (remote wakeup). The U3 state is directly coupled to the device's suspend state.

The USB VIP provides support for both automatic and testbench-initiated low-power entry attempts.

5.4.2 Automatic Low-Power Entry Attempts

Automatic low-power entry from U0 to U1 is enabled by enabling the U1 inactivity timer. Automatic low-power entry from U0 to U2 is enabled by disabling the U1 inactivity timer and enabling the U2 inactivity timer instead. In each, when the designated amount of time elapses with no bus activity, the downstream port attempts to enter a low-power state by transmitting an LGO_U1 or LGO_U2 link command.

The USB protocol and VIP support the autonomous transition from U1 to U2 when the U2 inactivity timer is enabled for both the upstream and downstream ports. This scenario differs from transition attempts from U0 to U1 or from U0 to U2 in that the U1 transitions to U2 with no other traffic on the bus. Because this transition requires no handshaking, the U2 inactivity timeout values must be the same for both ports.

5.4.2.1 Controlling the U1 Inactivity Timer

Two configuration variables control the U1 inactivity timer: `u1_timeout` and `u1_timeout_factor`.

- ❖ `u1_timeout` specifies the U1 inactivity timer enabled status and the U1 inactivity timeout value.
Set `u1_timeout` to 0x00 to disable the U1 inactivity timer. This is the default value.
Set `u1_timeout` to 0xFF to disable the U1 inactivity timer and program the downstream port to reject U1 entry requests initiated by the connected upstream port.
Set `u1_timeout` between 0x01 and 0xFE to enable the U1 inactivity timer.
- ❖ `u1_timeout_factor` specifies the period by which the U1 inactivity timeout value is multiplied to determine the timeout period. The timeout value is

$$(u1_timeout_factor) \times (u1_timeout) \times (1 \mu s)$$

`u1_timeout` is an 8-bit array that corresponds to `U1_TIMEOUT` and `PORT_U1_TIMEOUT` in the USB Specification. While the protocol specifies this 8-bit value is multiplied by 1 μs to determine the timeout value, the USB VIP uses the `u1_timeout_factor` as an additional multiplication factor to facilitate scaling while preserving the original timeout value.

5.4.2.2 Controlling the U2 Inactivity Timer

The U2 timeout value is represented by two variables: `initial_u2_inactivity_timeout` in the configuration, and `u2_inactivity_timeout` in the status object. The U2 inactivity timer can change dynamically because the Link Management Packet (LMP) can modify the timeout value.

Two configuration variables control the U2 inactivity timer: `u2_timeout` and `u2_timeout_factor`.

- ❖ `u2_timeout` specifies the U2 inactivity timer enabled status and the U2 inactivity timeout value.
Set `u2_timeout` to 0x00 to disable the U2 inactivity timer. This is the default value.
Set `u2_timeout` to 0xFF to disable the U2 inactivity timer and program the downstream port to reject U2 entry requests initiated by the connected upstream port.
Set `u2_timeout` between 0x01 and 0xFE to enable the U2 inactivity timer.
- ❖ `u2_timeout_factor` specifies the period by which the U2 inactivity timeout value is multiplied to determine the timeout period. The timeout value is

$$(u2_timeout_factor) \times (u2_timeout) \times (256 \mu s)$$

`u2_timeout` is an 8-bit array that corresponds to `U2_TIMEOUT` and `PORT_U2_TIMEOUT` in the USB Specification. While the protocol specifies this 8-bit value is multiplied by 256 μs to determine the timeout value, the USB VIP uses the `u2_timeout_factor` as an additional multiplication factor to facilitate scaling while preserving the original timeout value.

Because a testbench cannot modify the value of any status object variable, the VIP defines a Link Service Command that can modify the U2 inactivity timeout value. Because the link layer of the VIP does not interpret the contents of a U2 Inactivity Timeout LMP, updating the U2 timeout value in the VIP requires that a Link Service Command must accompany the timeout LMP.

When simulation starts, the `initial_u2_inactivity_timeout` value is copied to `u2_inactivity_timeout`. `USB_SS_U2_TIMEOUT` Link Service Commands perform subsequent U2 inactivity timeout value changes. The new timeout value is specified in by `u2_inactivity_timeout`.

5.4.3 Automatic Low-Power Entry for Upstream Ports

While the USB Specification defines automatic transition attempts from U0 to U1 or from U0 to U2 only for downstream ports, the VIP supports enabling U1 and U2 inactivity timers for upstream ports.

- ❖ Set `u1_inactivity_upstream_enabled` to enable U1 for upstream ports.
- ❖ Set `u2_inactivity_upstream_enabled` to enable U2 for upstream ports.

This feature is not supported by the USB Specification.

5.4.4 Testbench-Initiated Low-Power Entry Attempts

A testbench initiates low-power entry attempt through the following Link Service Commands:

- ❖ `USB_SS_ATTEMPT_U1_ENTRY`
- ❖ `USB_SS_ATTEMPT_U2_ENTRY`

You can configure these service commands to attempt immediate transmission of the LGO Link Command, or to wait for the required sending conditions.

- ❖ If you request immediate transmission of the LGO command when the required conditions are not satisfied, the VIP reports that low-power entry was not attempted and takes no further action.
- ❖ If you request postponing the transmission until conditions permit an LGO transmission, the request is active until the conditions are satisfied, after which the LGO Link Command is sent.

A port can have only one active low-power entry attempt.

Setting either of these Link Service Command variables to 1 causes the VIP to immediately send the LGO:

- ❖ `low_power_entry_ignore_pending_protocol`
- ❖ `low_power_entry_ignore_pending_link`

`USB_SS_CANCEL_LP_ENTRY_ATTEMPT` Link Service Command cancels active low-power entry attempts.

Link Service Commands attempting low-power entry require specific Link Layer and Protocol Layer conditions before an `LGO_U1` or `LGO_U2` is sent. These conditions ensure that there is no bus activity, pending packets, or link command transmissions. By default, these conditions are taken into account. You can configure Link Service Commands to ignore Protocol Layer or Link Layer conditions. The testbench must account for error conditions resulting from ignoring these conditions.

The following Link Service Command variable controls if the low-power entry is attempted immediately, or if it is “queued” until the required conditions are met before attempting low-power entry.

- ❖ `low_power_entry_wait_until_permitted`
If this variable is set to 1, the request is queued.

The following SystemVerilog code creates and submits a Link Service Command requesting a U1 entry attempt when the necessary Link Layer and Protocol Layer conditions are satisfied and to queue the request if the conditions are not satisfied:

```
`vmm_trace($psprintf("Setting U2 Inactivity Timeout to 0x01 @ %0t", $realtime()));
svt_usb_link_service link_service = new(cfg.host_cfg);
link_service.service_type = svt_usb_link_service::LINK_SS_PORT_COMMAND;
link_service.link_ss_command_type = svt_usb_link_service::USB_SS_ATTEMPT_U1_ENTRY;
link_service.low_power_entry_ignore_pending_protocol = 0;
link_service.low_power_entry_ignore_pending_link = 0;
link_service.low_power_entry_wait_until_permitted = 1;
host_agent.link.link_service_in_port.put(link_service);
```

5.5 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.

5.5.1 Default Functional Coverage

The USB Verification IP supports protocol and link layer functional coverage for Super Speed (SS) and USB20 modes of operation. The protocol and link layer functional coverage items are provided by host and device components via the component callback classes.

Table 5-2 lists the default functional coverage features provided with USB3.0 Verification IP.

Table 5-2 Default Functional Coverage Features

| Layer/Coverage | SuperSpeed | USB20 |
|----------------|---|---|
| Protocol | <ul style="list-style-type: none"> Bulk transfers Interrupt transfers Control transfers ISOC transfers | <ul style="list-style-type: none"> Bulk Transfers Interrupt transfers Control transfers ISOC transfers Split transfers |
| Link | Link Management and Flow control <ul style="list-style-type: none"> Header packet Link Command Low Power Management DPP Link Initialization Link Training and Status State Machine | <ul style="list-style-type: none"> 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.

**Note**

Detailed information about covergroups in USB3.0 Verification IP can be found in the class reference HTML located at:

[\\$DESIGNWARE_HOME/vip/svt/usb_svt/latest/doc/usb_svt_public/html/index.html](#)

5.5.2 Covergroup Organization

For the coverage based on sequences (such as Protocol layer coverage), the covergroup is organized as the collection of coverpoints and the cross of these coverpoints.

The coverpoints are divided into the following four categories:

- ❖ “Normal Behavior Sequences” on page 107
- ❖ “Error Condition Sequences” on page 107
- ❖ “Packet Field Range Coverpoints” on page 107
- ❖ “Data Collection Coverpoints” on page 107

5.5.2.1 Normal Behavior Sequences

These are the sequences, in which the responder entity (host or device) responds without any error for a transaction.

Example: `svt_usb_ack_dp_packet_sequence`

5.5.2.2 Error Condition Sequences

These are the sequences, in which errors are injected in the packets sent by the host or device. Depending on the error situation there may or may not be a response sequence.

Example: `svt_usb_invalid_ack_no_response_packet_sequence`

5.5.2.3 Packet Field Range Coverpoints

These are coverpoints that define ranges for the field values in a packet. These are typically used to define data length ranges, and sequence numbers.

Example: `data_length_range_low1_mid_high1`

5.5.2.4 Data Collection Coverpoints

These are all the data points that are captured as `svt_usb_packet` attribute field (such as flow control situations, and error injections).

Examples: `eob_lpf_bit`, `rtty_bit`, `after_inactive_flow_control`, and `preexisting_flow_control_state`

These attributes can take values either 0 or 1 (`rtty_bit`, `eob_lpf_bit`, `setup_bit`), or enumerated values (`preexisting_flow_control_state`, `cov_flow_control_cause`)

5.5.2.5 Crosses

You can define crosses by crossing one or more of the coverpoints in the above categories.

Examples:

- ❖ For normal traffic, normal behavior sequences are crossed with packet field range coverpoints.
cross of (`svt_usb_ack_dp_packet_sequence`) and (`data_length_range_low1_mid_high1`)

- ❖ For erroneous traffic, the error condition sequences are crossed with data collection coverpoints.
cross of (svt_usb_invalid_ack_no_response_packet_sequence) and (invalid_hp_sequence)
- ❖ For specific scenarios (for example, flow control state) the normal behavior sequences are crossed with data collection coverpoints.
cross of (ack_nrdy_sequence) and (flow_control_cause) and (after_flow_controlled)



Note For all the cover points participating in the cross coverage, the options.weight is set to zero. This means that the individual cover points do not contribute to the coverage score in those cover groups.

5.5.3 Range Bins

Range bins are defined by dividing the total range of possible values into a few buckets. The buckets are divided as low, mid and high values. The low and high carry the min and max values. The mid values may contain all mid values as one or two buckets.

Example 1

For data length associated with bulk transfers is defined as follows:

```
data_length_range_low1_mid_high1 : coverpoint cov_data_length_range {
bins data_length_range_low1[] = { 0 };
bins data_length_range_mid = { [1:`SVT_USB_SS_MAX_PACKET_SIZE-1] };
bins data_length_range_high1[] = { `SVT_USB_SS_MAX_PACKET_SIZE }; }
```

In the above example, the data_length_range_low1_mid_high1 is divided into three buckets with low (0 value), high (MPS) and mid (1 to MPS-1), where MPS is the Max Packet Size.

Example 2

For data length with babble associated with bulk transfer is defined as follows:

```
data_length_range_low1_mid_high1_babble : coverpoint cov_data_length_range {
bins data_length_range_low1[] = { 0 };
bins data_length_range_mid = { [1:`SVT_USB_SS_MAX_PACKET_SIZE-1] };
bins data_length_range_high1[] = { `SVT_USB_SS_MAX_PACKET_SIZE };
bins data_length_range_babble = { 1030 }; }
```

Example 3

For USB20 high speed packet field range is defined as follows:

```
hs_bulk_max_packet_size_min_mid_max : coverpoint cov_20_max_packet_size {
bins max_packet_size_min = { 0 }
bins max_packet_size_mid = { [1:(`SVT_USB_STATIC_HS_BULK_MAX_PACKET_SIZE-1)] }
bins max_packet_size_max = { `SVT_USB_STATIC_HS_BULK_MAX_PACKET_SIZE }; }
```

5.5.4 Default Functional Coverage Class Hierarchy

The basic classes are:

- ❖ Pattern sequences
- ❖ Component coverage data callbacks
- ❖ Component coverage callbacks

5.5.4.1 Pattern Sequences

These classes are not part of coverage hierarchy but used to define the patterns of the sequences to be compared in the simulation data for default coverage. The protocol layer sequence objects are extended from `svt_pattern` class. The link layer sequence objects are extended from `svt_usb_pattern` class.

5.5.4.2 Component Coverage Data Callbacks

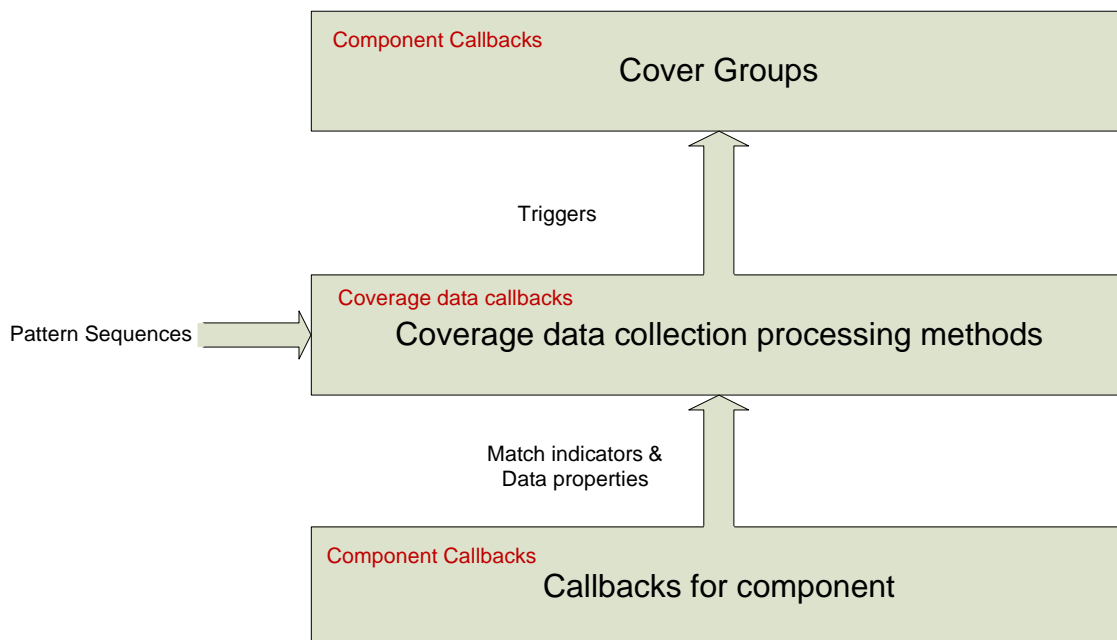
This callback class defines default data and event information that are used to implement the coverage groups. The naming convention uses “`def_cov_data`” in the class names for easy identification of these classes. This class also includes implementations of the coverage methods that respond to the coverage requests by setting the coverage data and triggering the coverage events. This implementation does not include any coverage groups. The `def_cov_data` callbacks classes are extended from component callbacks.

5.5.4.3 Component Coverage Callbacks

This class is extended from the component coverage data class. The naming convention uses “`def_cov`” in the class names for easy identification of these classes. It includes default cover groups based on the data and events defined in the data class.

The class hierarchy of default functional coverage described above is as shown in [Figure 5-1](#).

Figure 5-1 Default Functional Coverage Class Hierarchy



5.5.5 Coverage Callback Classes

The following naming convention is used for coverage callback classes:

`svt_usb_<layer>_<speed>_<VIP config>_def_cov_<type>_callbacks`

Where:

<layer>: protocol, link, physical

<speed>: ss or 20

<VIP config>: host or device

<type>: data for callback classes defining the default data, methods and event information used for coverage
none for callback classes defining default cover groups

Table 5-3 lists the coverage callback classes.

Table 5-3 Coverage Callback Classes

| Class Name | Type | Layer | Configuration |
|---|--------------|----------|-------------------|
| svt_usb_protocol_ss_host_def_cov_data_callbacks | Data | Protocol | SuperSpeed Host |
| svt_usb_protocol_ss_host_def_cov_callbacks | Cover Groups | Protocol | SuperSpeed Host |
| svt_usb_protocol_ss_device_def_cov_data_callbacks | Data | Protocol | SuperSpeed Device |
| svt_usb_protocol_ss_device_def_cov_callbacks | Cover Groups | Protocol | SuperSpeed Device |
| svt_usb_protocol_20_host_def_cov_data_callbacks | Data | Protocol | USB20 Host |
| svt_usb_protocol_20_host_def_cov_callbacks | Cover Groups | Protocol | USB20 Host |
| svt_usb_protocol_20_device_def_cov_data_callbacks | Data | Protocol | USB20 Device |
| svt_usb_protocol_20_device_def_cov_callbacks | Cover Groups | Protocol | USB20 Device |
| svt_usb_link_ss_def_cov_callbacks | Cover Groups | Link | SS Host or Device |
| svt_usb_link_ss_def_cov_data_callbacks | Data | Link | SS Host or Device |
| svt_usb_link_20_host_def_cov_callbacks | Cover Groups | Link | USB20 Host |
| svt_usb_link_20_host_def_cov_data_callbacks | Data | Link | USB20 Host |



Note Information about coverage callbacks in USB3.0 Verification IP can be found in the class reference HTML located at: [\\$DESIGNWARE_HOME/vip/svt/usb_svt/latest/doc/usb_svt_public/html/index.html](#)

5.5.6 Using Functional Coverage

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

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

5.5.6.1 Coverage Extensions

You can extend the coverage callback classes to specify user-defined coverage in addition to default coverage provided in the model. Callbacks are added using the following syntax.


```
uvm_callbacks#(T)::add(obj,cb);
```

5.5.7 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>
```

5.6 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.

5.6.1 VIP Acting as a Host

[Table 5-4](#) describes the VIP behavior when it is acting as a host.

Table 5-4 Behavior of VIP Acting as a Host for Different Endpoint Configurations and Transfers

| Transfer | Endpoint Configuration | Endpoint Configuration |
|--|--|--|
| | <code>allow_aligned_transfers_without_zero_length=0</code> | <code>allow_aligned_transfers_without_zero_length=1</code> |
| <code>aligned_transfer_with_zero_length=0</code> | <p>VIP performs a check, and if <code>txfer::payload_intended_count</code> is aligned</p> <ul style="list-style-type: none"> OUT: VIP reports an error that endpoint configuration and transfer attributes are inconsistent. VIP then issues one or more OUT transactions to transmit <code>payload_count</code> 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 <code>payload_intended_count</code> or <code><max_packet_size></code> length payload is received (whichever occurs first), hence does not require a 0-length data packet. | <p>VIP performs a check, and if <code>txfer::payload_intended_count</code> is aligned</p> <ul style="list-style-type: none"> OUT: VIP issues one or more OUT transactions to transmit <code>payload_count</code> bytes, and does not end with a 0-length data packet. IN: VIP continues to issue IN transactions until it receives all <code>payload_intended_count</code> or <code><max_packet_size></code> length payload is received (whichever occurs first), hence does not require a 0-length data packet. |
| | <p>VIP performs a check, and if <code>txfer::payload_intended_count</code> is not aligned, it reports the following constraint solver error:</p> <pre>txfer:: aligned_transfer_ends_with_zero_length must be set to 1 when payload is not aligned.</pre> | |
| <code>aligned_transfer_with_zero_length=1</code> | <ul style="list-style-type: none"> OUT: VIP issues one or more OUT transactions to transmit <code>payload_intended_byte_count</code> number of bytes, and always ends with a <code><max_packet_size></code> (0 or short) length data packet. IN: VIP continues to issue IN transactions until a <code><max_packet_size></code> (0 or short) length payload is received. | |

5.6.2 VIP Acting as a Device

Table 5-5 describes the VIP behavior when it is acting as a host.

Table 5-5 Behavior of VIP Acting as a Device for Different Endpoint Configurations and Transfers

| Transfer | Endpoint Configuration | Endpoint Configuration |
|--|---|---|
| | <code>allow_aligned_transfers_without_zero_length=0</code> | <code>allow_aligned_transfers_without_zero_length=1</code> |
| <code>aligned_transfer_with_zero_length=0</code> | <p>VIP performs a check, and if <code>txfer::payload_intended_count</code> is aligned</p> <ul style="list-style-type: none"> OUT: VIP reports an error that endpoint configuration and transfer attributes are inconsistent. VIP then continues to receive data packets until it receives all <code>payload_intended_count</code> or <code><max_packet_size></code> 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 <code>payload_intended_count</code>, and does not transmit a 0-length data packet. | <p>VIP performs a check, and if <code>txfer::payload_intended_count</code> is aligned</p> <ul style="list-style-type: none"> OUT: VIP continues to receive data packets until it receives all <code>payload_intended_count</code> or <code><max_packet_size></code> 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 <code>payload_intended_count</code>, and does not transmit a 0-length data packet. |
| | <p>VIP performs a check, and if <code>txfer::payload_intended_count</code> is not aligned, it reports the following constraint solver error:</p> <pre>txfer:: aligned_transfer_ends_with_zero_length must be set to 1 when payload is not aligned.</pre> | |
| <code>aligned_transfer_with_zero_length=1</code> | <ul style="list-style-type: none"> OUT: VIP continues to receive data packets until a data packet with <code><max_packet_size></code> (0 or short) length payload is received. IN: VIP continues to provide data packets to IN transactions until it transmits all of <code>payload_intended_count</code> and always ends with a <code>< max_packet_size></code> (0 or short) length data packet. | |

5.7 UTMI+ Support

This section provides information on USB VIP support for UTMI+.

5.7.1 Port Interface

The USB VIP supports the complete UTMI+ signal set. The signals can be found in the interfaces:

- ❖ svt_usb_utmi_dut_mac_if
- ❖ svt_usb_utmi_dut_phy_if

The host and peripheral devices connected via the UTMI protocol are required to be operating on the same clock. That is, one clock domain for transceivers. When the VIP is acting as a PHY, it is capable of generating the clock for the DUT, as with any real world PHY.

Synopsys has added a pin called "Vip_Speed" in addition to the pins specified by the protocol. This pin relays the speed information as specified by the svt_usb_configuration object. The pin should only be used for internal testbench uses.

For detailed information on the interfaces, consult the HTML on-line documentation at:

[\\$DESIGNWARE_HOME/vip/svt/usb_svt/latest/doc/usb_svt_public.uvm/html/index.html](#)

5.7.2 Configuring the UTMI+ Interface

Table 5-6 lists all the available configuration parameters you can use to set UTMI behavior.

Table 5-6 Configuration Parameters to Control the UTMI Interface

| Parameter | Legal values | Description |
|-------------------------|---|--|
| usb_20_signal_interface | Enum: UTMI_IF | Determines the interface to be used by the VIP |
| utmi_data_width | Integer: 8 or 16 | This parameter determines the width of the UTMI+ interface. It is speed dependent: <ul style="list-style-type: none"> • HS/FS -- 8-bit interface • HS/FS -- 16-bit interface • FS Only -- 8-bit interface • LS Only -- 8-bit interface. <p>This attribute is dependent on the speed attribute.</p> |
| speed | Enum: svt_usb_types::HS, svt_usb_types::FS svt_usb_types::LS | This attribute is used to control the UTMI+ interface speed. NOTE: For the VIP to support HS_FS functionality speed must be set to HS. |
| component_subtype | ENUM MAC PHY | Represents the behavior of the VIP relative to the signal interface. |

5.7.3 Error Injection

The VIP in the UTMI+ mode supports all protocol, link and physical exceptions that are supported at the USB 2.0 serial interface.

The VIP supports control over RxError by doing the following:

- ❖ Injecting bit errors in the local PHY transmission (using callbacks) so that that the error is translated into an abort1 or abort2 error on the remote PHY side.
- ❖ Injecting BUFFER_UNDERRUN (type of packet exception) error into the packet objects (using callbacks).

5.7.4 Attach / Detach

Attach/Detach is supported through service calls to the PHY model.

5.7.5 L1, L2 (Suspend), Resume, Remote wake-up

The UTMI+ protocol supports entry into either the L1 (LPM) sleep state or the L2 suspend state. Exit from either state can be accomplished through resume signaling initiated by the host on the bus, or by remote wake-up signaling initiated by the device/peripheral on the bus.

5.7.6 UTMI+ Messages

The UTMI+ interface will issue error and warning messages under the following conditions:

- ❖ Error
 - ◆ When xcvrselect is set to a value of 2 b11, which requests a feature of UTMI+ level 3 that is not supported.
 - ◆ A device tries to attach using the UTMI+ interface with an active high speed transceiver. The device is not attached.

❖ Notes

- ◆ Transceiver reset completed. Current state of the reset pin is 1'b0.
- ◆ Attach command called when OpMode does not equal 2'b01 (non-driving mode). Waiting for Opmode to return to 2'b01.
- ◆ Attach requested when reset is being driven. Command ignored.
- ◆ Data bus width is changed to 16 (or 8) (the most recently sample value of databus16_8 at the transceiver reset).

5.8 USB 2.0 OTG Support

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 Interface Signals” on page 117
- ❖ “Session Request Protocol” on page 119
- ❖ “Role Swapping Using the HNP Protocol” on page 123
- ❖ “Attach Detection Protocol” on page 133

5.8.1 OTG Interface Signals

The VIP USB OTG interface (svt_usb_otg_if) includes the following different types of signals

5.8.1.1 Link Level 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 this OTG interface is used in conjunction with 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 signals within this interface 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 5-7 lists and describes the OTG link interface signals.

Table 5-7 OTG Link Interface Signals

| Signal | Direction | Polarity | Default | Size | Description and Values |
|-----------|-----------|----------|-----------|------|--|
| VbusValid | Output | High | Tri-state | 1 | VBUS Valid signal <ul style="list-style-type: none">1'b0: VBUS voltage < VBUS Valid threshold1'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) <ul style="list-style-type: none">1'b0: VBUS voltage < VBUS A-device Session Valid threshold1'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) <ul style="list-style-type: none">1'b0: VBUS voltage < VBUS B-device Session Valid threshold1'b1: VBUS voltage >= VBUS B-device Session Valid threshold |

Table 5-7 OTG Link Interface Signals

| Signal | Direction | Polarity | Default | Size | Description and Values |
|----------------|-----------|----------|-----------|------|---|
| SessEnd | Output | High | Tri-state | 1 | Session End signal <ul style="list-style-type: none"> 1'b0: VBUS voltage \geq VBUS Session End threshold 1'b1: VBUS voltage is not above VBUS Session End threshold |
| DrvVbus | Input | High | - | 1 | VBUS Drive signal <ul style="list-style-type: none"> 1'b0: VBUS is not driven to 5V 1'b1: VBUS is driven to 5V |
| ChrgVbus | Input | High | - | 1 | VBUS Charging signal <ul style="list-style-type: none"> 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 <ul style="list-style-type: none"> 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 <ul style="list-style-type: none"> 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 <ul style="list-style-type: none"> 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 <ul style="list-style-type: none"> 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 <ul style="list-style-type: none"> 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 <ul style="list-style-type: none"> 1'b0: Pull-down resistor is not connected to DM 1'b1: Pull-down resistor is connected to DM |

5.8.1.2 Serial Interface Signals

The OTG interface (svt_usb_otg_if) includes VIP-defined signals to 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 this OTG interface is used in conjunction with 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.

5.8.1.2.1 ADP Interface Signals

Table 5-8 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 5-8 OTG ADP Serial Interface Signals

| Signal | Direction | Polarity | Default | Size | Description and Values |
|--------------|-----------|----------|---------|------|---|
| vip_attached | Output | High | 1 | 1 | VIP is attached <ul style="list-style-type: none">1'b0: VIP is not attached1'b1: VIP is attached |
| vip_adp_prb | Output | High | 0 | 1 | VIP ADP probe is active <ul style="list-style-type: none">1'b0: VIP is not performing an ADP probe1'b1: VIP is performing an ADP probe |
| dut_attached | Input | High | – | 1 | DUT is attached <ul style="list-style-type: none">1'b0: DUT is not attached1'b1: DUT is attached |
| dut_adp_prb | Input | High | – | 1 | DUT ADP probe is active <ul style="list-style-type: none">1'b0: DUT is not performing an ADP probe1'b1: DUT is performing an ADP probe |

5.8.1.2.2 SRP Interface Signals

Table 5-9 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 5-9 OTG SRP Serial Interface Signals

| Signal | Direction | Polarity | Default | Size | Description and Values |
|--------------|-----------|----------|---------|------|--|
| vip_sess_vld | Output | High | 0 | 1 | VIP VBUS level detector - VOTG_SESS_VLD <ul style="list-style-type: none">1'b0: VIP detected/driven VBUS <= VOTG_SESS_VLD1'b1: VIP detected/driven VBUS > VOTG_SESS_VLD |
| dut_sess_vld | Input | High | 0 | 1 | DUT VBUS level detector - VOTG_SESS_VLD <ul style="list-style-type: none">1'b0: DUT detected/driven VBUS <= VOTG_SESS_VLD1'b1: DUT detected/driven VBUS > VOTG_SESS_VLD |

5.8.2 Session Request Protocol

Session Request Protocol (SRP) is a mechanism used to conserve 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_configuration` object.

5.8.2.1 SRP Protocol Flow When the VIP is Configured as an A-Device

[Figure 5-2](#) explains the SRP process flow when the VIP is configured as an A-device.

Figure 5-2 SRP Protocol Flow when the VIP is Configured as an A-Device

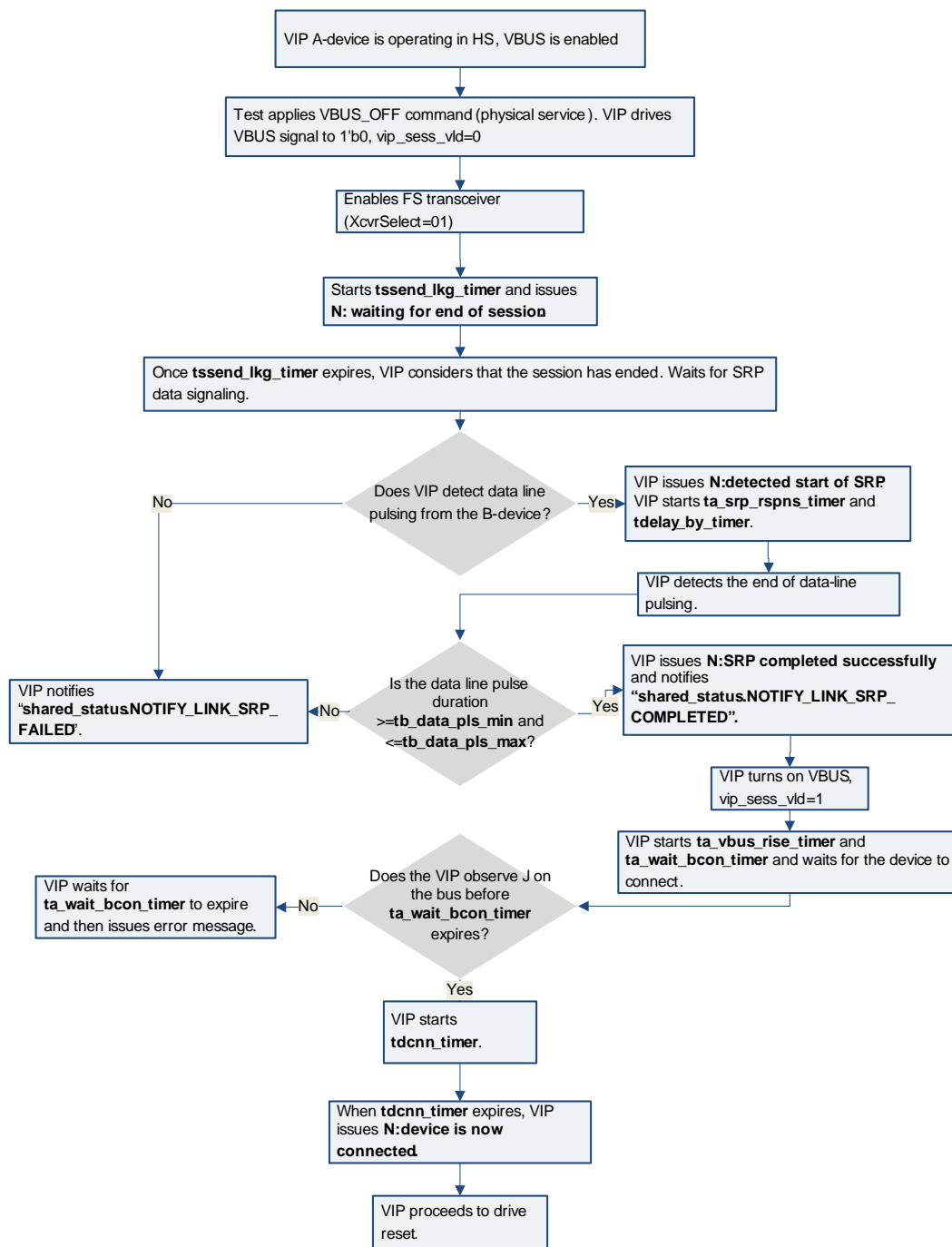


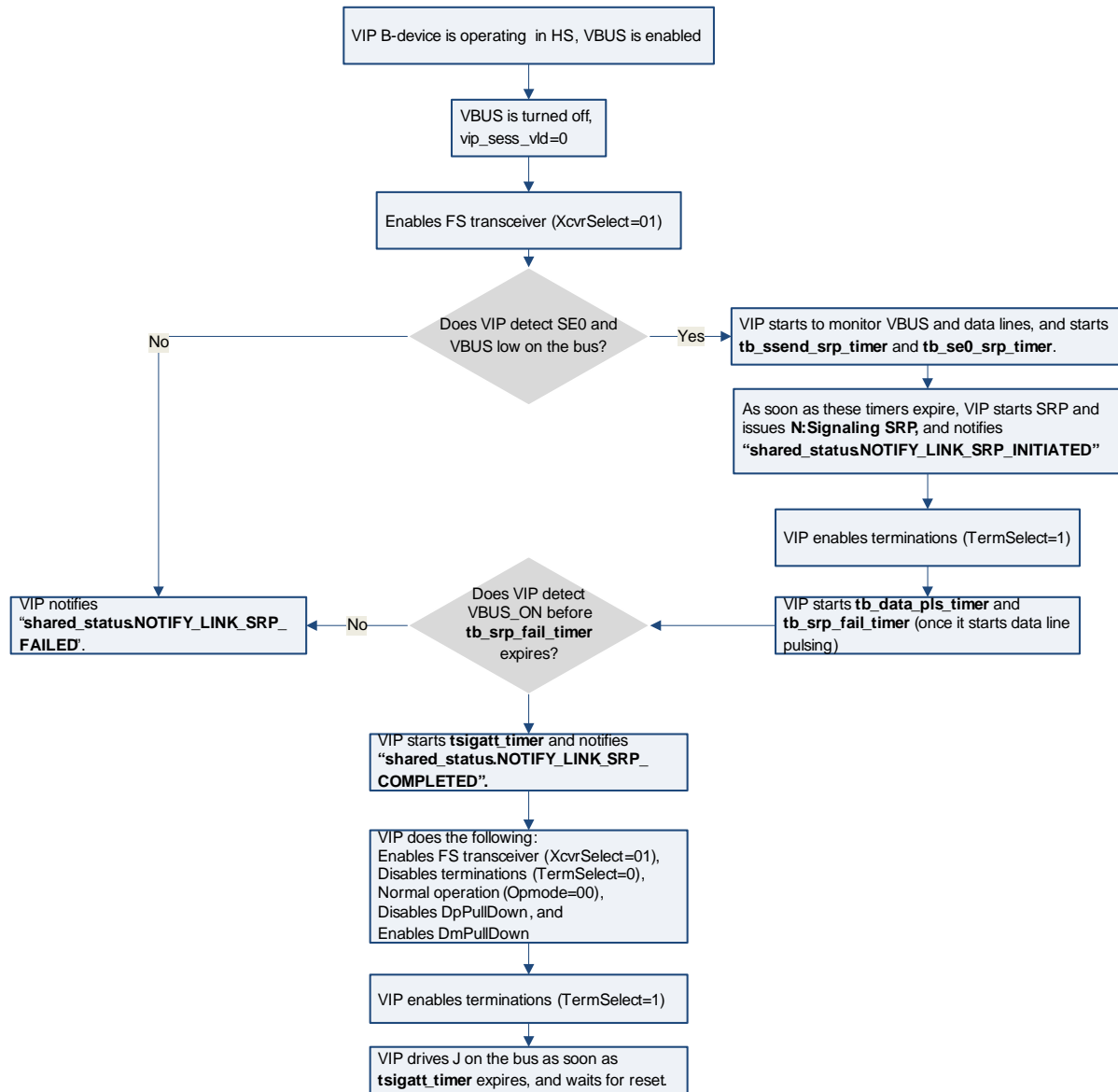
Table 5-10 expands on the abbreviated messages that are used in Figures 5-2 and 5-3.

Table 5-10 Legend Explaining the Abbreviated Messages

| Abbreviated Messages Used in Figure 5-2 | 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 5-3 | Actual Message Issued by VIP |
| N: signaling SRP | B-device signaling SRP by generating data line pulsing. |

5.8.2.2 SRP Protocol Flow When the VIP is Configured as a B-Device

Figure 5-3 explains the SRP process flow when the VIP is configured as a B-device. The abbreviated messages listed in Figure 5-3 are expanded in detail in Table 5-10.

Figure 5-3 SRP Protocol Flow when the VIP is Configured as a B-Device

5.8.3 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.

5.8.3.1 Initializing and Storing Configurations for OTG Roles

To successfully support role swapping, the VIP agent 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 agent is constructed. The swapped configuration is stored through the `set_otg_cfg` function bit.

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` task.

The agent also notifies the testbench whether the role swap was successful or not through one of the following `uvm_events`:

- ❖ `NOTIFY_OTG_ROLE_SWAP_SUCCEEDED` or
- ❖ `NOTIFY_OTG_ROLE_SWAP_FAILED`

After successful role swap, the VIP agent and the component stack is reconfigured using the `reconfigure()` command. When `svt_usb_agent::reconfigure()` is called by the role-swap process, the stored `initial_cfg` or `swapped_cfg` configuration is used as the argument to the `reconfigure()` method. This means that the initial and swapped configurations are retained across role swaps. Changes to the active configuration by using the `svt_usb_agent::reconfigure()` method directly by the user (testbench) are not retained by the agent 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 by the `set_otg_cfg()` method 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 call `svt_usb_agent::set_otg_cfg()` for the inactive role while the current active role is in effect.

5.8.3.2 Role Swapping Process Overview

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

1. Testbench creates two `svt_usb_agent_configuration` instances, one each representing the configuration for the initial OTG role (`initial_cfg`) and the swapped OTG role (`swapped_cfg`).
2. Testbench constructs (using the `new()` method) the VIP agent using the `initial_cfg` configuration data object that represents its initial role (based on `component_type` value).
3. Testbench stores the `swapped_cfg` configuration data object using `set_otg_cfg()`.
4. If the VIP agent 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 agent reconfigures the component 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` `uvm_event`.

- ◆ If the role swap fails the agent does not change anything, but notifies the testbench through the `NOTIFY_OTG_ROLE_SWAP_FAILED uvm_event`.
- 5. If VIP agent initially has the OTG Host role, and the DUT OTG Device initiates a role swap attempt, the agent notifies the testbench that a role swap attempt has started.
 - ◆ If the role swap succeeds, the agent reconfigures the component 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 uvm_event`.
 - ◆ If the role swap fails the agent does not change anything, but notifies the testbench through the `NOTIFY_OTG_ROLE_SWAP_FAILED uvm_event`.
- 6. When a successful role swap back to the initial OTG role occurs, (after step 4 or 5), the agent reconfigures the component stack (and itself) using the `initial_cfg` configuration data object.

5.8.3.3 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 configured 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.

5.8.3.3.1 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()` commands 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 `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.

5.8.3.3.2 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 whose 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 agent'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.

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

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

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

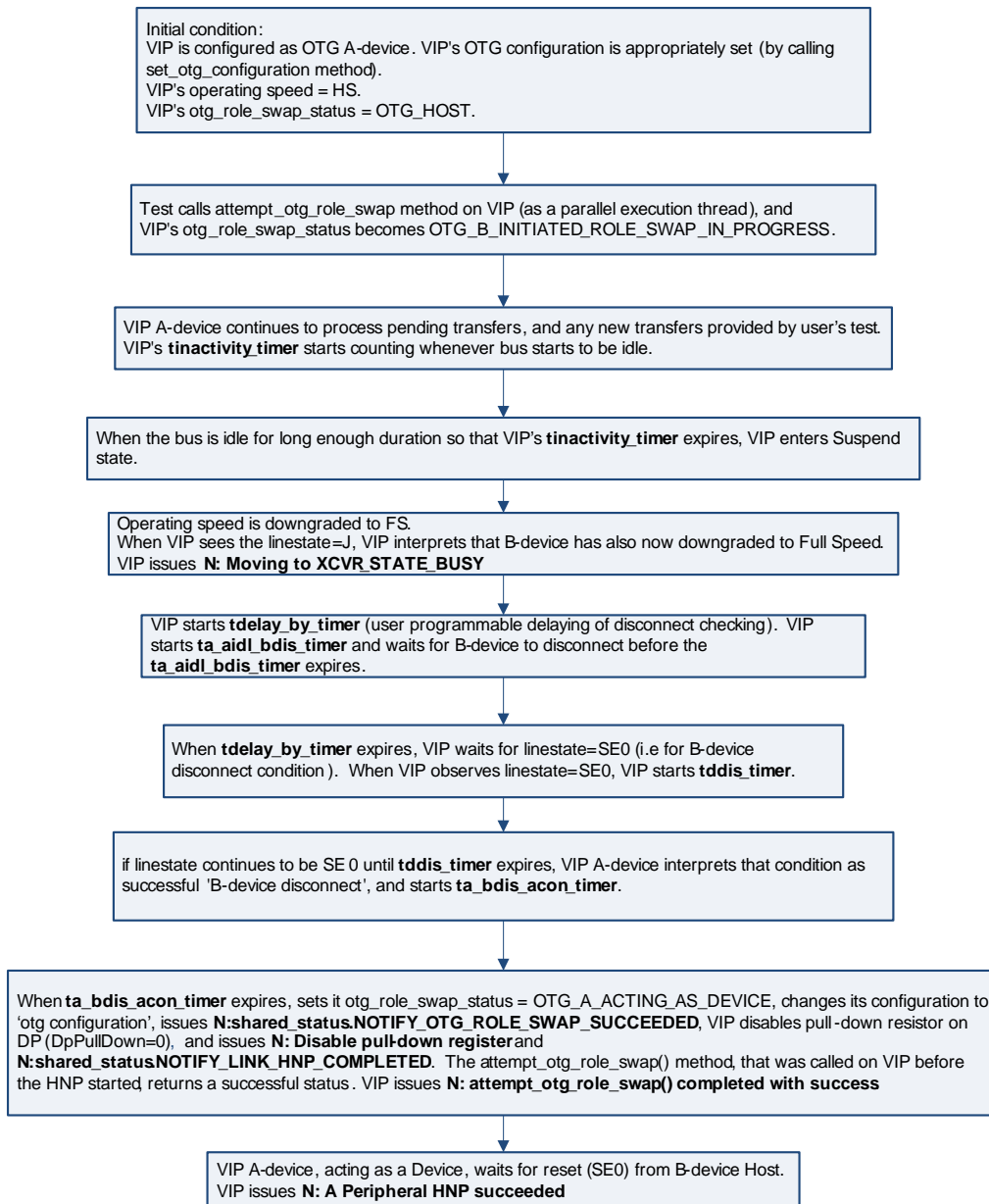


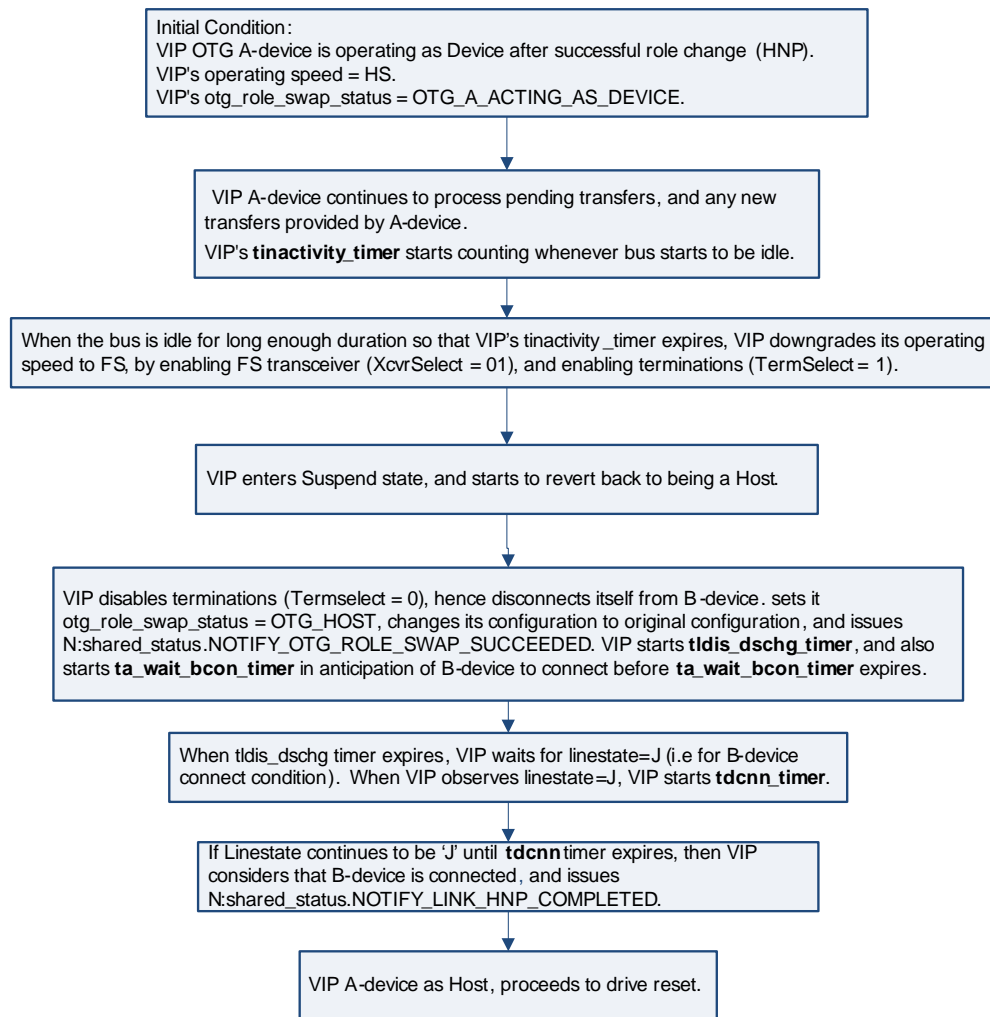
Figure 5-5 Reverse HNP Flow when the VIP A-Device Reverts Back to Acting as a Host

Table 5-11 expands on the abbreviated messages that are used in Figures 5-4 and 5-5.

Table 5-11 Legend Explaining the Abbreviated Messages Used in Figures 5-4 and 5-5

| Abbreviated Messages Used in Figure 5-4 | 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 |
| Abbreviated Message Used in Figure 5-5 | Actual Message Issued by VIP |
| N: Enable FS transceiver | Enable FS transceiver (XcvtSelect = 01) |
| N: Enable terminations | Enable terminations (TermSelect = 1) |

Table 5-11 Legend Explaining the Abbreviated Messages Used in Figures 5-4 and 5-5

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

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

Figure 5-6 explains the HNP flow when the VIP is configured as a B-device.

Figure 5-6 HNP Flow When VIP is Configured as a B-Device

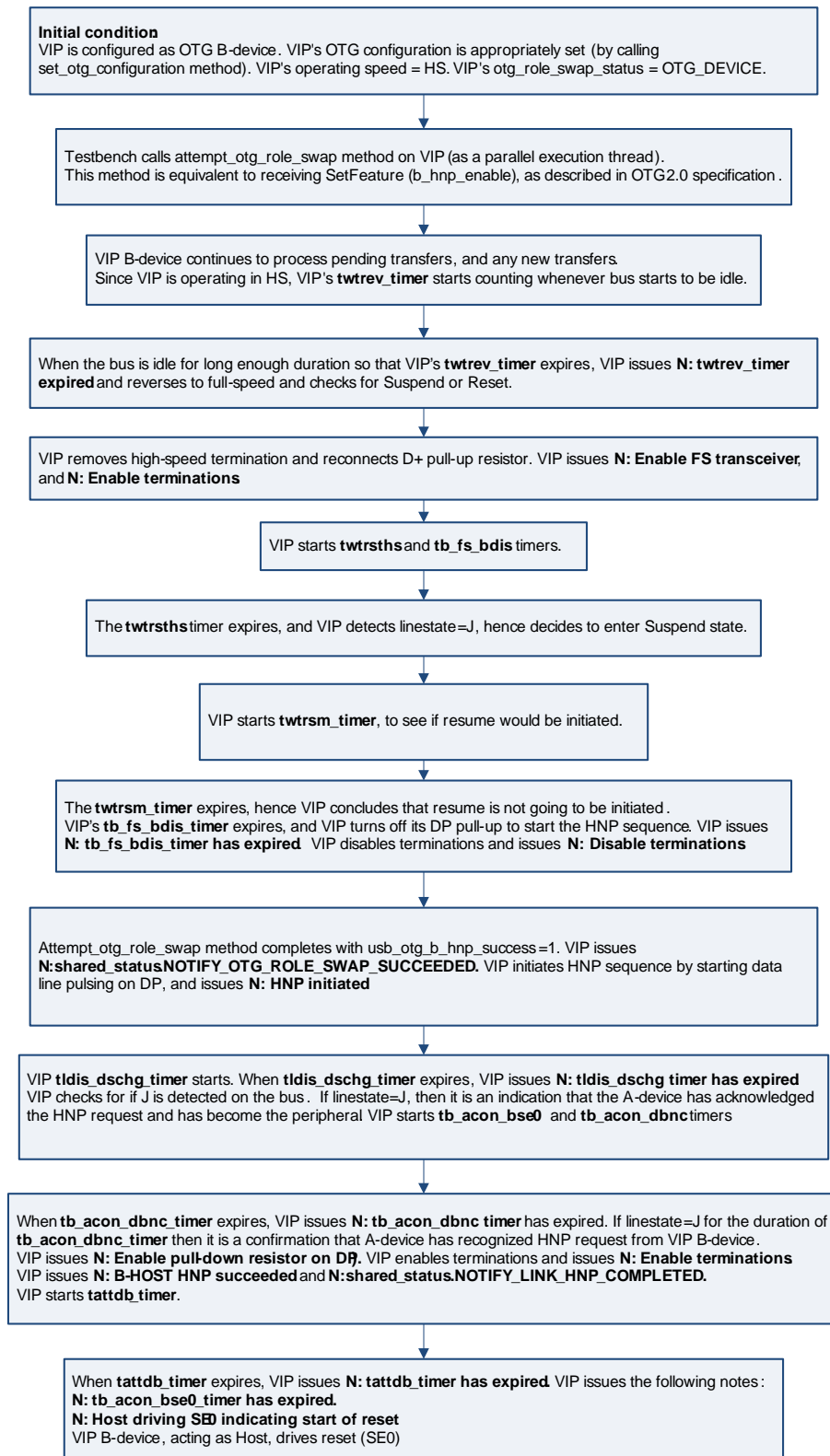


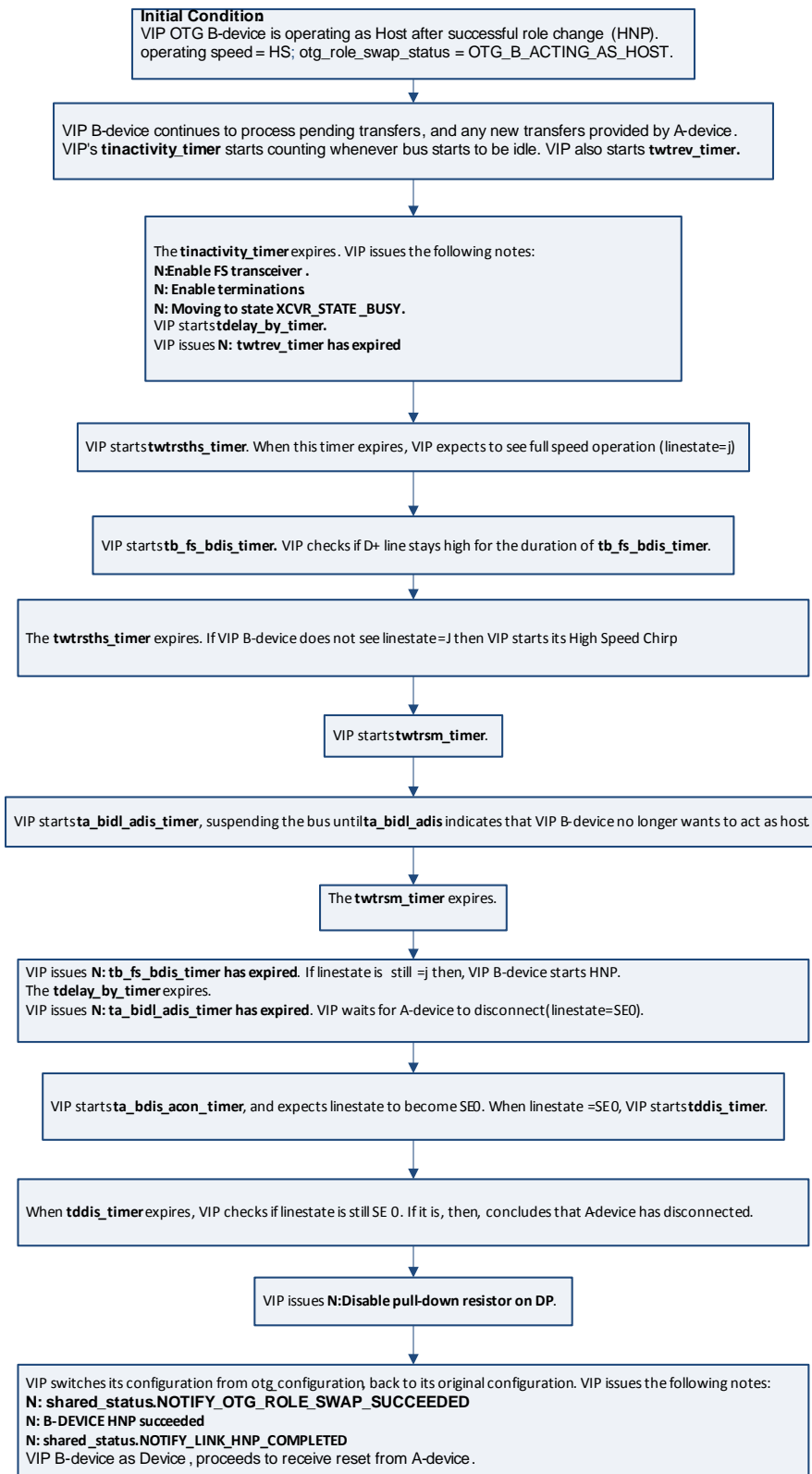
Figure 5-7 Reverse HNP Flow When the VIP B-Device Reverts Back to Acting as a Device

Table 5-12 expands on the abbreviated messages that are used in Figures 5-6 and 5-7.

Table 5-12 Legend Explaining the Abbreviated Messages Used in Figures 5-6 and 5-7

| Abbreviated Messages Used in Figure 5-6 | Actual Messages Issued by VIP |
|--|---|
| N: twtrev_timer expired | N: twtrev_timer expired |
| N: Enable FS transceiver | N: Enable FS transceiver (XcvtSelect = 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 5-7 | 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 |

5.8.4 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

5.8.4.1 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)

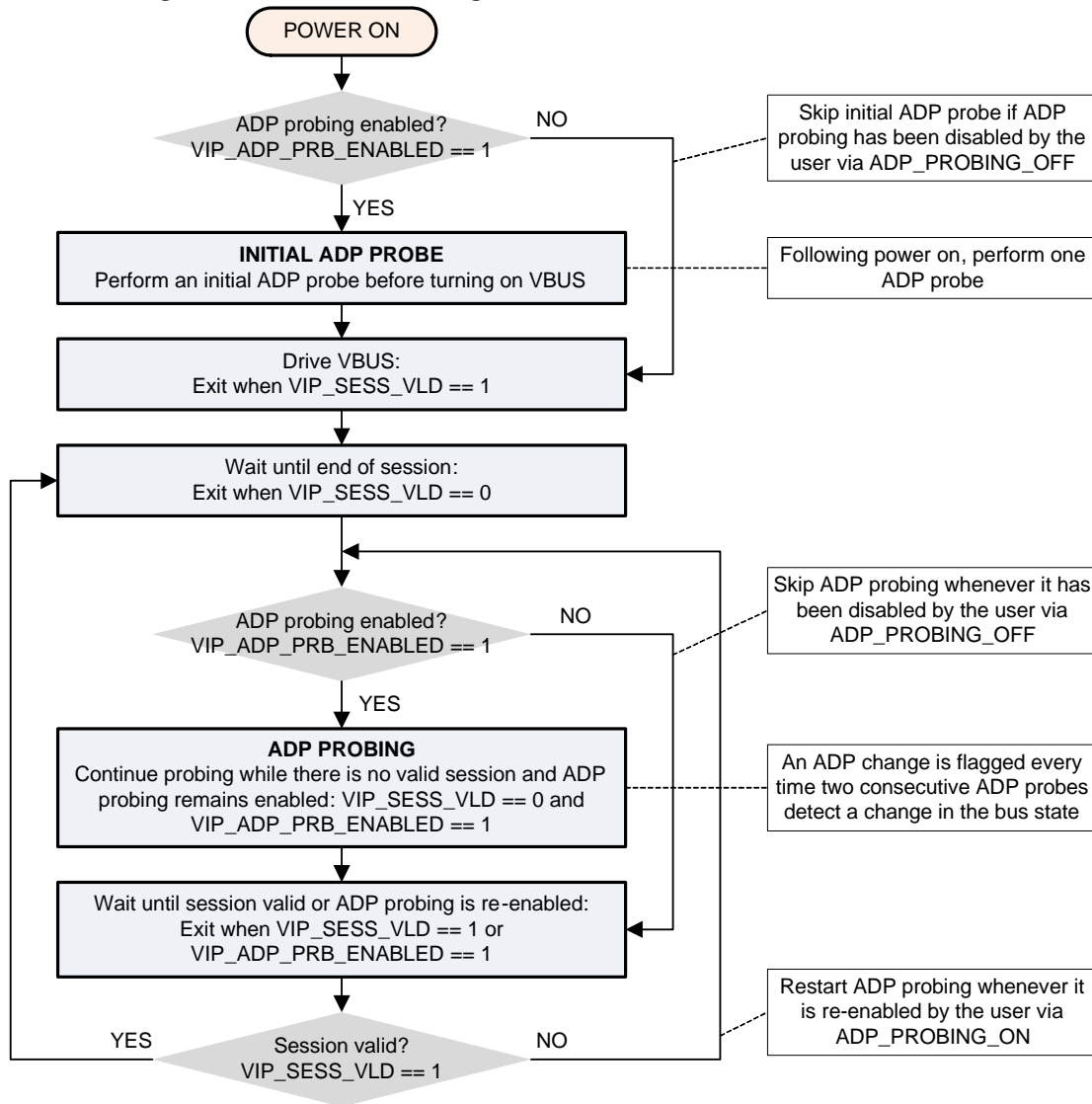
5.8.4.1.1 ADP Probing When VIP is Configured as an A-Device

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 5-8](#) illustrates a simple ADP Probing process for VIP as an A-device or an Embedded Host using a waveform.

Figure 5-8 ADP Probing Flow when VIP is Configured as an A-Device or Embedded Host

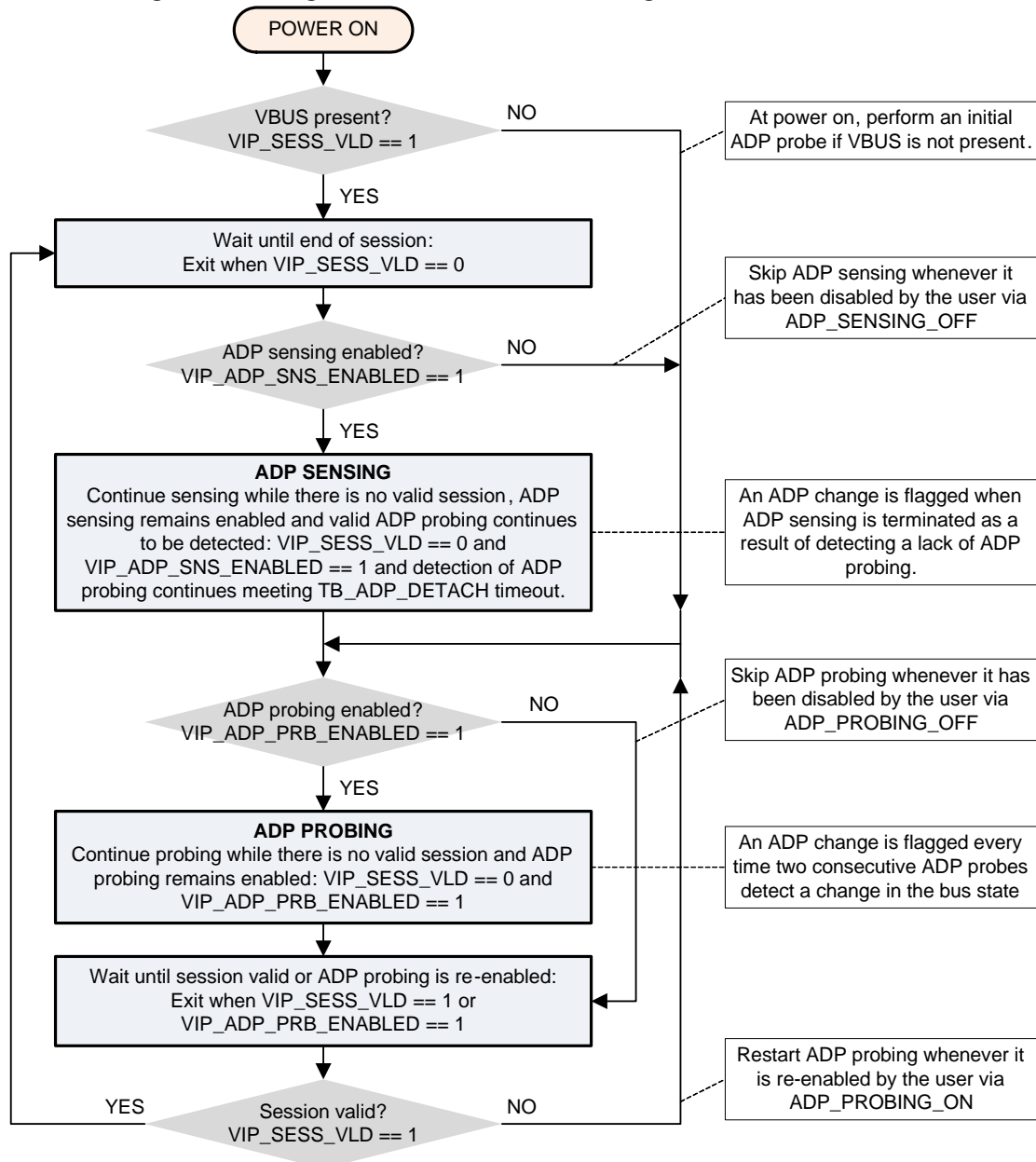


5.8.4.1.2 ADP Probing When VIP is Configured as a B-Device

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 5-9 explains the ADP control when VIP is configured as a B-device.

Figure 5-9 ADP Probing and Sensing Flow when the VIP is Configured as a B-Device

5.8.4.2 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 detecting ADP probing by the host

5.8.4.2.1 ADP Sensing When VIP is Configured 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 5-9](#).

5.8.4.3 ADP UVM Events

VIP issues the following `uvm_events` 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 `uvm_event` 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` value).
- ❖ `NOTIFY_PHYSICAL_VBUS_CHANGE` - General `uvm_event` issued whenever a change on VBUS (driven or received) is detected.
- ❖ `NOTIFY_PHYSICAL_USB_20_LINESTATE_CHANGE` - General `uvm_event` issued whenever a change in received linestate is detected.

5.8.4.4 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 you to override the automatic default behavior of the VIP.

Use the following commands to enable and disable ADP probing:

- ❖ `ADP_PROBING_ON` - The `ADP_PROBING_ON` command enables ADP probing by the VIP. This command re-enables the VIP's default ADP probing behavior.
- ❖ `ADP_PROBING_OFF` - The `ADP_PROBING_OFF` 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:

- ❖ `ADP_SENSING_ON` - The `ADP_SENSING_ON` command enables ADP sensing by the VIP. This command re-enables the VIP's default ADP sensing behavior.

- ❖ **ADP_SENSING_OFF** - The **ADP_SENSING_OFF** 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.

5.9 HSIC Overview

The this section covers HSIC Support.

5.9.1 Supported HSIC Features

The model supports the following HSIC features:

- ❖ HS bus traffic
- ❖ Reset/Suspend/Resume/Remote Wakeup
- ❖ HSIC Discovery process
- ❖ Reception and transmission of inverted packet data
- ❖ Modeling of the powered off state.

5.9.2 Unsupported HSIC Features

When a HSIC interface is selected, some standard USB 2.0 features become unavailable. Following is a list of VIP features that are not supported when using a HSIC interface:

- ❖ FS/LS bus traffic
- ❖ OTG
- ❖ Disconnect (can neither drive nor detect disconnection)
- ❖ Attach/Detach (can neither attach nor detach)

5.9.3 Configuration Parameters

You control much of the HSIC interface through various configuration members. The members also define the available feature set. The following table shows the most important HSCI configuration members. For a listing of the valid and reasonable constraints on the configuration members, consult the online HTML documentation..

Table 5-13 HSIC Configuration Parameters

| Configuration Parameter | Purpose |
|--------------------------------|--|
| usb_20_signal_interface | Selects the USB 2.0 interface type |
| usb_20_hsic_strobe_period | Width of a strobe period. |
| usb_20_hsic_auto_enabled_delay | Delay from host Power On to Enabled state (set negative to request manual initiation of the process). |
| usb_20_hsic_auto_connect_delay | Delay from peripheral IDLE detect to Connect (set negative to request manual initiation of the process). |
| usb_20_hsic_connect_time | Length of time CONNECT is driven during Connect signaling. |

Table 5-13 HSIC Configuration Parameters (Continued)

| Configuration Parameter | Purpose |
|--------------------------------------|---|
| usb_20_hsic_connect_idle_time | Length of time IDLE is driven at the end of Connect signaling. |
| "usb_20_hsic_bus_keeper_time | Length of time peripheral checks for Bus Keepers to maintain IDLE following an IDLE driven at the end of signaling. |
| usb_20_hsic_idle_drive_time | Length of time IDLE is driven after signaling (except during Connect signaling). |
| usb_20_hsic_strobe_data_rx_skew | Defines the maximum expected skew between Strobe and Data during detection of bus state transitions. |
| usb_20_hsic_strobe_data_tx_skew | Defines the skew between Strobe and Data during the driving of bus state transitions. |
| usb_20_hsic_fast_eop_to_idle_disable | Disables fastest transition from driving EOP to driving IDLE. |
| usb_20_hsic_bus_keeper_on_count | Number of half-strobe periods from IDLE detect to Bus Keepers being enabled. |
| usb_20_hsic_bus_keeper_off_count | Number of half-strobe periods from non-IDLE detect to Bus Keepers being disabled. |
| usb_20_hsic_tx_inversion_enable | Enables transmission of inverted packet data. |
| usb_20_hsic_dut_is_legacy_device | Used to identify when the DUT is a legacy device. A legacy device is allowed to transmit inverted packet data. |

5.9.4 Transactions

The HSIC interface uses data transactions to model the flow of data exchanged across the HSIC interface, and uses service transactions to provide test bench and inter-layer control of the transactor's operation.

5.9.4.1 Data Transactions

There are no HSIC-specific data transaction requirements; the HSIC interface uses the current set of USB SVT transactions associated with the standard USB 2.0 HS data path.

5.9.4.2 Physical Service Transactions

The following HSIC-specific physical service commands are intended for test bench use to manually initiate a HSIC operation.

- ❖ **DRIVE_HSI_ENABLED** - Request to a host to "present" the IDLE bus state indicating the HSIC Enabled On state.
- ❖ **DRIVE_HSI_CONNECT** - Request to a peripheral to drive Connect signaling.

The following general use USB 2.0 physical service commands are also supported when configured for a HSIC interface:

- ❖ **VBUS_OFF** - When configured for HSIC, moves the VIP into the powered off state.
- ❖ **VBUS_ON** - When configured for HSIC, moves the VIP into the powered on state.

The following general use USB 2.0 physical service commands are not supported when configured for a HSIC interface:

- ❖ ATTACH_DEVICE - No support for attach/detach.
- ❖ DETACH_DEVICE - No support for attach/detach.
- ❖ REMOTE_ATTACH_DEVICE - Only supported for remote link interfaces.
- ❖ REMOTE_DETACH_DEVICE - Only supported for remote link interfaces.
- ❖ REMOTE_VBUS_OFF - Only supported for remote link interfaces.
- ❖ REMOTE_VBUS_ON - Only supported for remote link interfaces.
- ❖ ADP_PROBING_OFF - No support for OTG.
- ❖ ADP_PROBING_ON - No support for OTG.
- ❖ ADP_SENSING_OFF - No support for OTG.
- ❖ ADP_SENSING_ON - No support for OTG.

5.9.5 Exceptions

The HSIC interface uses USB SVT transaction exceptions to inject/report data-based flow and/or content errors and uses USB SVT error checks to report on protocol-related failures involving timing and/or signaling.

5.9.5.1 Protocol Errors

Outside of the HSIC Discovery process, the HSIC interface uses the current set of USB 2.0 protocol checks (i.e. reset timing, inter-packet delays).

Specific to HSIC Discovery process and to HSIC operation, the HSIC interface adds some new layer-specific protocol checks (using standard USB SVT error checks based on the `svt_err_check` class).

5.9.5.2 Physical Level Protocol Checks

The HSIC interface adds the following physical-level HSIC protocol checks:

- ❖ `usb_20_hsic_connect_check` - Identifies Connect signaling issues detected by the VIP when configured as a HSIC host. This check, performed by the VIP when configured as a host, verifies that a DUT peripheral signals CONNECT only after enabled, and that it drives CONNECT for the proper length of time.
- ❖ `usb_20_hsic_bus_keeper_check` - Identifies Bus Keeper timing issues detected by the VIP when configured as a HSIC peripheral. This check attempts to verify that after a non-IDLE to IDLE transition, a DUT host's Bus Keepers get enabled within the proper amount of time, and that they continue to hold the bus in IDLE once it is no longer being driven IDLE. This check is only attempted following the success of the post-signaling IDLE drive check (`usb_20_hsic_drive_idle_check`).
- ❖ `usb_20_hsic_drive_idle_check` - Identifies end-of-signaling drive IDLE timing issues detected by the VIP when the receiving HSIC device. This check attempts to verify that after a non-IDLE to IDLE transition, a DUT drives IDLE for the proper amount of time.
- ❖ `usb_20_hsic_data_inversion_check` - Identifies when inverted packet data is detected by the VIP when configured to support HSIC. This check is used to flag the reception of inverted packet data from a non-legacy device.

5.9.6 HSIC Interface

The HSIC signal interface, as with other supported signal interfaces, resides as an instance within the top-level SVT USB interface (`svt_usb_if.svi`). The `usb_20_hsic_if` interface includes all of the signals that make up

the supported USB signal connections for a HSIC interface. It includes any clocking blocks and modports necessary to provide logical connections for the transactors interacting via the supported signal connections. This also includes modports which provide debug access to the different transactors.

5.9.7 HSIC Signal Interface

Defined within the `svt_usb_20_hsic_if.svi` file, the HSIC interface is a simple two signal serial interface consisting of the data and strobe signals. Additionally, this interface includes an input clock signal, `clk`; this input clock is used by the VIP to derive its transmit clock and to support clock/data recovery. See the following table.

Table 5-14 HSIC Signals

| Signal Name | Direction | Polarity | Default | Size | Description/Values |
|-------------|-----------|----------|---------|------|---|
| data | In/Out | High | 0 | 1 | Bi-directional DDR data signal. During data communication, this signal is used to communicate data. Outside of data communication, this signal is used in combination with the strobe signal to control state. |
| strobe | In/Out | High | 0 | 1 | Bi-directional data strobe signal. During data communication, this signal is used as the clock for data. Outside of data communication, this signal is used in combination with the data signal to control state. |
| clk | Input | High | 0 | 1 | Input clock from the testbench. Requires a stable clock running at 4 times the desired VIP transmission rate (8x the desired strobe-period rate). This clock is used to derive the VIP transmit clock and to support clock recovery during reception. |

5.9.8 Callbacks

There are no HSIC-specific callback requirements; the HSIC interface uses the current set of callbacks associated with the standard USB 2.0 data path.

5.9.9 Notifications

The HSIC interface adds the following HSIC-specific shared status notifications:

Table 5-15

| Notification | Use |
|---|---|
| NOTIFY_USB_20_HSIC_ENABLED_ASSERTED | A HSIC host, moving from the Powered On state to the Enabled On state, has started asserting IDLE on the bus. |
| NOTIFY_USB_20_HSIC_ENABLED_DETECTED | A HSIC peripheral, in the Powered On state, has detected assertion of IDLE on the bus. |
| NOTIFY_USB_20_HSIC_CONNECT_INITIATED | A HSIC peripheral, moving from the Powered On state to the Connect state, has started driving the CONNECT portion of Connect signaling. |
| NOTIFY_USB_20_HSIC_CONNECT_DRIVE_IDLE | A HSIC peripheral, moving from the Powered On state to the Connect state, has started driving the IDLE portion of Connect signaling. |
| NOTIFY_USB_20_HSIC_CONNECT_COMPLETED | A HSIC peripheral, in the Connect state, has completed driving Connect signaling. |
| NOTIFY_USB_20_HSIC_CONNECT_DETECTED | A HSIC host, in the Enabled state, has detected Connect signaling. |
| NOTIFY_USB_20_HSIC_INVERTED_SYNC_DETECTED | A HSIC device, while in an idle state, has detected reception of an inverted SYNC. |
| NOTIFY_USB_20_HSIC_INVERTED_SYNC_TRANSMIT | A HSIC device, in the process of transmitting an inverted packet, has started transmission of that packet's inverted SYNC. |

5.9.10 Factories

There are no HSIC-specific factory requirements; the HSIC interface uses the current set of factories associated with the standard USB 2.0 data path.

5.9.11 Shared Status

The HSIC interface adds the following HSIC-specific shared status properties:

- ❖ `usb_20_hsic_enabled` - Identifies when the HSIC device is in the Enabled state. When a host, set to 1 once it starts asserting IDLE on the bus after Power On. When a peripheral, set to 1 once it detects IDLE on the bus after Power On.
- ❖ `usb_20_hsic_connected` - Identifies when the HSIC device is in the Connected state. When a peripheral, set to 1 once it completes asserting CONNECT on the bus after Enabled. When a host, set to 1 once it detects valid Connect signaling on the bus after Enabled.

5.9.12 Usage

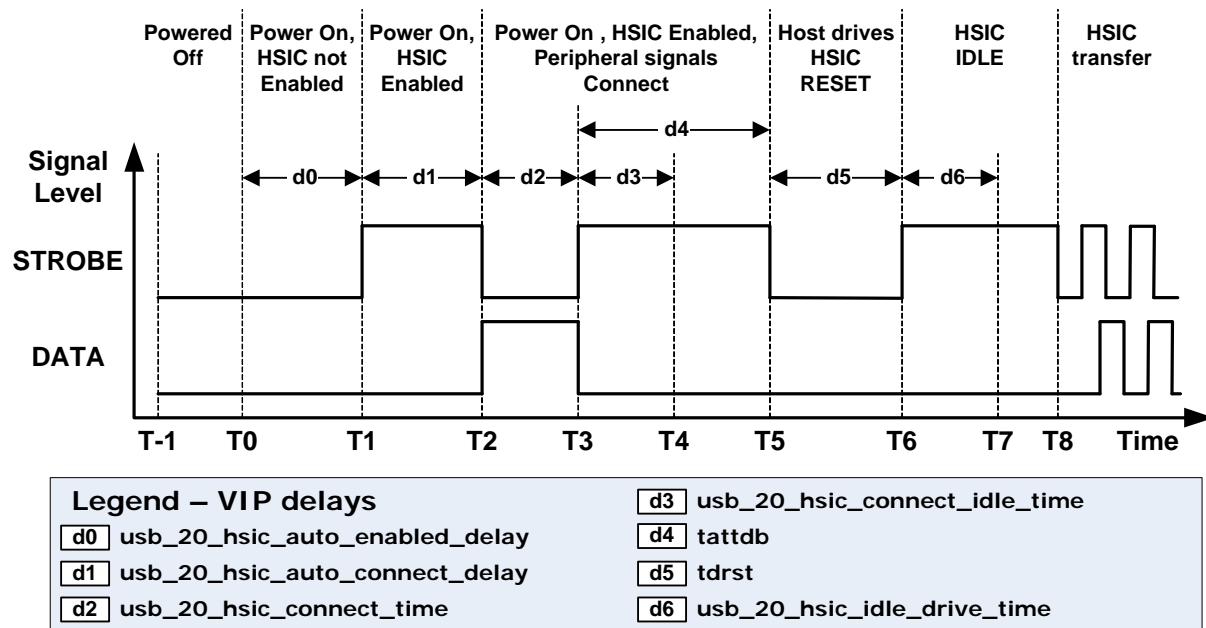
Following is a discussion of the HSIC-specific usage scenarios and how users may control and/or interact with them. The HSIC-specific usage scenarios are:

- ❖ Discovery Process - The HSIC protocol for establishing a HSIC connection.
- ❖ Bus Keepers - The HSIC host feature used to maintain IDLE on the bus.
- ❖ Data Inversion - Support for enabling transmission and reception of inverted data.
- ❖ "Data EOP Signaling - Support for configuring the end-of-data signaling pattern.
- ❖ Modeling Power Off - Support for modeling power off entry and exit.
- ❖ Controlling Skew - Support for configuring skew between Strobe and Data signals.

5.9.12.1 Discovery Process

Discovery is the HSIC power up and connection sequence used to establish communication between two HSIC devices. [Figure 5-10](#) illustrates the basic flow of the Discovery process. Included in the figure are annotations used in the following sections to help describe how the VIP operates during the Discovery process.

Figure 5-10 Discovery Process with VIP Set Values



5.9.12.2 VIP as Host

Using [Figure 5-10](#) as a reference, this section describes how the VIP operates as a HSIC host during the Discovery process.

T1 - Powered Off state

| | |
|-------------|--|
| Description | This state models a HSIC host in the "Power state is OFF" state. |
| Entry | The VIP enters this state under one of the following conditions: <ul style="list-style-type: none"> • The VIP has not yet been started. • The VIP has been reset (via reset() after being started). • The VIP has received a request to power down (via physical service command VBUS_OFF after being started). |

T1 - Powered Off state

| | |
|--------|---|
| Action | In this state, the VIP's HSIC interface provides weak pull-downs to prevent the Strobe and Data signals from floating. |
| Exit | <p>Based on its entry into this state, the VIP exits this state under one of the following conditions:</p> <ul style="list-style-type: none"> • An un-started VIP will move to the T0 when it is started (via start()). • A reset VIP will move to the T0 when it is restarted (via start()). • A started VIP will move to the T0 when it receives a request to power up (via physical service command VBUS_ON). |

T0 - Power On, HSIC not Enabled state

| | |
|-------------|---|
| Description | This state models a HSIC host in the "Power state is ON, but HSIC is not enabled" state. |
| Entry | From state T-1. |
| Action | In this state, the requirement for the host to "assert Bus Keepers on both Strobe and Data to provide a logic '0' state" is provided by the interface's weak pull-downs. |
| Exit | <p>The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF).</p> <p>The VIP moves to T1 based on the value of <code>usb_20_hsic_auto_enabled_delay</code>:</p> <ul style="list-style-type: none"> • If <code>usb_20_hsic_auto_enabled_delay</code> is non-negative, the VIP moves to T1 after implementing the delay specified by the value. • If <code>usb_20_hsic_auto_enabled_delay</code> is negative, the VIP will not move to T1 until it receives a <code>DRIVE_HSIC_ENABLED</code> physical service command. |



T1 - Power On, HSIC Enabled state

| | |
|-------------|--|
| Description | This state models a HSIC host in the "Power state is ON, HSIC is enabled" state. |
| Entry | From state T0. |
| Action | The VIP issues the NOTIFY_USB_20_HSIC_ENABLED_ASSERTED notification, asserts it Bus Keepers to provide an IDLE bus state and then starts to monitor the Strobe and Data lines for the CONNECT bus state. |
| Exit | The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF). The VIP moves to T2 when it detects the bus transition to the CONNECT bus state. |

T2 - Power On, HSIC Enabled, Peripheral signals Connect state

| | |
|-------------|---|
| Description | This state models a HSIC host in the "Power state is ON, HSIC is enabled, and Peripheral signals a CONNECT" state following the detection of CONNECT. |
| Entry | From state T1. |
| Action | The VIP starts measuring the period CONNECT that is on the bus. |
| Exit | The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF). The VIP moves to T3 when it detects the bus transition out of the CONNECT bus state. |

T3 - Power On, HSIC Enabled, Peripheral signals Connect state

| | |
|-------------|--|
| Description | This state models a HSIC host in the "Power state is ON, HSIC is enabled, and Peripheral signals a CONNECT" state following the detection of end of CONNECT. |
| Entry | From state T2. |
| Action | <p>The VIP stops measuring the period CONNECT that is on the bus.</p> <ul style="list-style-type: none"> • If it detected assertion of the CONNECT bus state for the required minimum period of time (as controlled by the configuration property <code>usb_20_hsic_connect_time_min</code>), the VIP issues the <code>NOTIFY_USB_20_HSIC_CONNECT_DETECTED</code> notification. • If it detected assertion of the CONNECT for less than the minimum requirement, the VIP will issue a <code>usb_20_hsic_connect_check</code> failure and remain in T3 until the VIP is either restarted or powered down. <p>NOTE: At this point, the VIP starts operating as a standard USB 2.0 device. It begins using standard USB 2.0 configuration timers and state.</p> |
| Exit | <p>The VIP moves to T-1 if it receives a request to power down (via physical service command <code>VBUS_OFF</code>). Based on whether or not the detected CONNECT was valid:</p> <ul style="list-style-type: none"> • If valid, the VIP moves to T5 after implementing the delay specified by the <code>tattdb</code>. • If not valid, the VIP will remain in T3 until the VIP is either restarted or powered down. |

T4 - Power On, HSIC Enabled, Peripheral signals Connect state

| | |
|-------------|---|
| Description | This state models the point at which a HSIC peripheral stops driving IDLE following the end of driving CONNECT. |
| Entry | none |
| Action | none |
| Exit | n/a |



T5 - Host drive HSIC RESET state

| | |
|-------------|---|
| Description | This state models a HSIC host in processes of driving RESET. |
| Entry | From state T3. |
| Action | The VIP drives RESET on the bus. |
| Exit | The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF). The VIP moves to T6 after implementing the delay specified by the tdrst. |

T6 - HSIC IDLE

| | |
|-------------|--|
| Description | This state models a HSIC host driving the IDLE state following driving RESET. |
| Entry | From state T5. |
| Action | The VIP drives IDLE on the bus. |
| Exit | The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF). The VIP moves to T7 after implementing the delay specified by the usb_20_hsic_idle_drive_time. |

T7 - HSIC IDLE

| | |
|-------------|--|
| Description | This state models a HSIC host in the idle state following driving post-RESET IDLE. The host Bus Keepers maintain IDLE on the bus. |
| Entry | From state T6. |
| Action | The VIP stops driving the bus. NOTE: At this point, the VIP is ready to start exchanging data across the bus (as illustrated at T8). |
| Exit | The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF). The VIP moves to T8 if packet transmission or reception is started. |

T8 - HSIC transfer

| | |
|-------------|---|
| Description | This state models the transfer of data across the bus. |
| Entry | From state T7 |
| Action | The VIP is transmitting or receiving packet data. |
| Exit | The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF). The VIP moves back to T7 when packet transmission/reception has completed. |

5.9.12.3 VIP as Peripheral

Using [Figure 5-10](#) as a reference, this section describes how the VIP operates as a HSIC peripheral during the Discovery process.

T-1 - Powered Off state

| | |
|-------------|--|
| Description | This state models a HSIC peripheral in the "Power state is OFF" state. |
| Entry | The VIP enters this state under one of the following conditions: <ul style="list-style-type: none"> • The VIP has not yet been started. • The VIP has been reset (via reset() after being started). • The VIP has received a request to power down (via physical service command VBUS_OFF after being started). |
| Action | In this state, the VIP's HSIC interface provides weak pull-downs to prevent the Strobe and Data signals from floating. |
| Exit | Based on its entry into this state, the VIP exits this state under one of the following conditions: <ul style="list-style-type: none"> • An un-started VIP will move to the T0 when it is started (via start()). • A reset VIP will move to the T0 when it is restarted (via start()). • A started VIP will move to the T0 when it receives a request to power up (via physical service command VBUS_ON). |



T0 - Power On, HSIC not Enabled state

| | |
|-------------|---|
| Description | This state models a HSIC peripheral in the "Power state is ON, but HSIC is not enabled" state. |
| Entry | From state T-1. |
| Action | In this state, the VIP starts to monitor the Strobe and Data lines for an IDLE bus state. |
| Exit | The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF).The VIP moves to T1 when it detects the bus transition to the IDLE bus state. |

T1 - Power On, HSIC Enabled state

| | |
|-------------|--|
| Description | This state models a HSIC peripheral in the "Power state is ON, HSIC is enabled" state. |
| Entry | From state T0. |
| Action | The VIP issues the NOTIFY_USB_20_HSIC_ENABLED_DETECTED notification. |
| Exit | <p>The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF).The VIP moves to T2 based on the value of usb_20_hsic_auto_connect_delay:</p> <ul style="list-style-type: none">• If usb_20_hsic_auto_connect_delay is non-negative, the VIP moves to T2 after implementing the delay specified by the value.• If usb_20_hsic_auto_connect_delay is negative, the VIP will not move to T2 until it receives a DRIVE_HSIC_CONNECT physical service command. |

T2 - Power On, HSIC Enabled, Peripheral signals Connect state

| | |
|-------------|--|
| Description | This state models a HSIC peripheral in the "Power state is ON, HSIC is enabled, and Peripheral signals a CONNECT" state and starting to drive CONNECT. |
| Entry | From state T1. |
| Action | The VIP issues the NOTIFY_USB_20_HSIC_CONNECT_INITIATED notification and drives CONNECT on the bus. |
| Exit | <p>The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF).</p> <p>The VIP moves to T3 after implementing the delay specified by the usb_20_hsic_connect_time.</p> |

T3 - Power On, HSIC Enabled, Peripheral signals Connect state

| | |
|-------------|---|
| Description | This state models a HSIC peripheral in the "Power state is ON, HSIC is enabled, and Peripheral signals a CONNECT" state and starting to drive IDLE. |
| Entry | From state T2. |
| Action | The VIP issues the NOTIFY_USB_20_HSIC_CONNECT_DRIVE_IDLE notification and starts to drive IDLE on the bus. |
| Exit | The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF). The VIP moves to T4 after implementing the delay specified by the usb_20_hsic_connect_idle_time. |

T4 - Power On, HSIC Enabled, Peripheral signals Connect state

| | |
|-------------|--|
| Description | This state models the point at which a HSIC peripheral stops driving IDLE following the end of driving CONNECT. |
| Entry | From state T3. |
| Action | <p>The VIP issues the NOTIFY_USB_20_HSIC_CONNECT_COMPLETED notification and stops driving the bus.</p> <p>The VIP starts to monitor the Strobe and Data lines for a RESET bus state.</p> <p>NOTE: At this point, the VIP starts operating as a standard USB 2.0 device. It begins using standard USB 2.0 configuration timers and state.</p> |
| Exit | The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF).The VIP moves to T5 when it detects the bus transition to the RESET bus state. |

T5 - Host drive HSIC RESET state

| | |
|-------------|--|
| Description | This state models a HSIC host in processes of driving RESET. |
| Entry | From state T4. |
| Action | The VIP starts to monitor the Strobe and Data lines for the end of RESET. |
| Exit | <p>The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF).</p> <p>The VIP moves to T6 when it detects the bus transition to the IDLE bus state.</p> |

| T6 - HSIC IDLE | |
|----------------|---|
| Description | This state models a HSIC peripheral driving the IDLE state following driving RESET. |
| Entry | From state T6. |
| Action | The VIP starts to monitor the Strobe and Data lines for the end of IDLE.NOTE: At this point, the VIP is ready to start exchanging data across the bus (as illustrated at T8). |
| Exit | The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF). The VIP moves to T8 if packet transmission or reception is started. |

| T7 - HSIC IDLE | |
|----------------|--|
| Description | This state models a HSIC host in the idle state following driving post-RESET IDLE. |
| Entry | none |
| Action | none |
| Exit | n/a |

| T8 - HSIC transfer | |
|--------------------|---|
| Description | This state models the transfer of data across the bus. |
| Entry | From state T6 |
| Action | The VIP is receiving or transmitting packet data. |
| Exit | The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF). The VIP moves back to T6 when packet reception/transmission has completed. |

5.9.12.4 Bus Keepers

When operating as a Host, the HSIC device is responsible for providing Bus Keepers that maintain IDLE on the bus when the bus is not being driven. When configured as a HSIC Host, the VIP provides the ability to configure the timing of its Bus Keepers. The VIP provides independent Bus Keeper enable and disable timers with half-strobe resolution.

Control of the VIP's Bus Keeper timers is through the following configuration properties:

- ❖ `usb_20_hsic_bus_keeper_on_count` - Number of half-strobe periods from IDLE detect to Bus Keepers being enabled.

- ❖ `usb_20_hsic_bus_keeper_off_count` - Number of half-strobe periods from non-IDLE detect to Bus Keepers being disabled.

By default, these timers are set to the value 3 (which translates into the nominal HSIC Bus Keeper timing requirement of 1.5 Strobe-periods).

5.9.12.5 Data Inversion

The original HSIC specification failed to identify any correlation between the data values transmitted on the HSIC bus and standard HS USB J and K values. The HSIC ECN that followed does clearly state this correlation. Some "legacy" devices, those designed before the ECN, may transmit and receive what is now considered inverted packet data (i.e. the

SYNC pattern terminating with two J's, rather than with two K's). "Non-legacy" devices, those designed after the ECN are required to support reception of inverted packet data, but are not allowed to transmit inverted packet data.

The VIP supports the ability to both transmit and receive inverted packet data. Control of these features is through the following configuration properties:

- ❖ `usb_20_hsic_tx_inversion_enable` - Enables transmission of inverted packet data.
- ❖ `usb_20_hsic_dut_is_legacy_device` - Used to identify when the DUT is a legacy device. A legacy device is allowed to transmit inverted packet data.

Whenever the `usb_20_hsic_tx_inversion_enable` property is set false (the default), the VIP transmits normal packets; whenever set true, the VIP transmits inverted packets.

Whenever the `usb_20_hsic_dut_is_legacy_device` property is set false (the default), the VIP will register a failure for each inverted packet it receives; whenever set true, the VIP will receive inverted packets without complaint.

5.9.12.6 Data EOP Signaling

Following the completion of data signaling, the transmitting device is expected to drive IDLE for 2 Strobe-Periods. However, since transmitting the last bit of a packet's EOP does not always leave the bus in an IDLE state (STROBE=1 DATA=0), the transmitter may be required to move the bus to IDLE first.

By default, the VIP will drive the end of data signaling as illustrated in [Figure 5-11](#) and [Figure 5-12](#). In the case where the last bit of EOP leaves the bus in the IDLE state, the VIP starts driving IDLE immediately. In the case where STROBE ends low, the VIP will start driving IDLE on the bus when it drives STROBE high at the next half Strobe-Period boundary, resulting in the transmission of one extra bit of data. In the case where both STROBE and DATA end high, the VIP starts driving IDLE on the bus when it drives DATA low three eighths of a Strobe-Period period after the last rising edge of Strobe.

Figure 5-11 End of signaling scenarios with DATA low

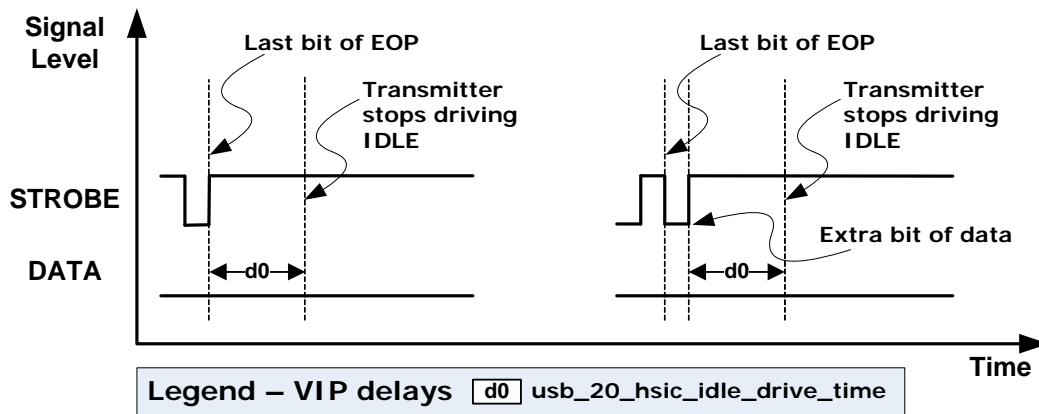
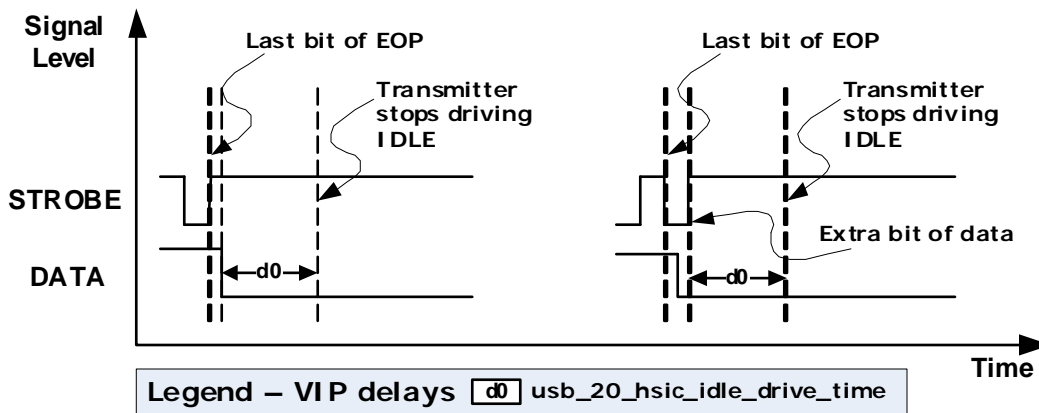
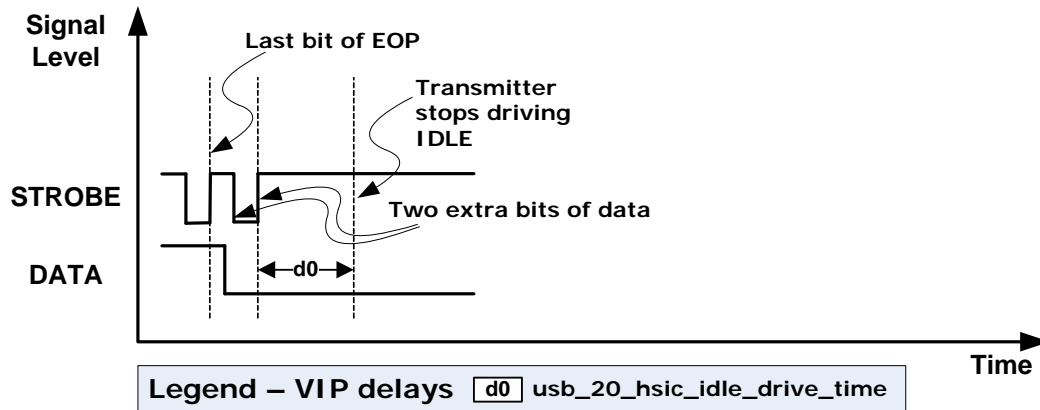


Figure 5-12 End of signaling scenarios with DATA high (fast EOP)



When `usb_20_hsic_fast_eop_to_idle_disable` is set to 1, the VIP will always terminate data signaling by entering IDLE on the rising edge of the STROBE signal. The one case affected by this is when the data signaling completes with both DATA and STROBE high. In this case, two extra bits of data will be generated following the last bit of transmitted data. This case is illustrated in [Figure 5-13](#).

Figure 5-13 End of signaling scenarios with DATA high (fast EOP disabled)

5.9.12.7 Modeling Power Off

The HSIC specification identifies each HSIC device as being self-powered. The VIP, considered to be in powered off state prior to being started, immediately enters the powered on state when started. After being started, the VIP can move into and out of the powered off state using the following physical service commands:

- ❖ `VBUS_OFF` - When configured for HSIC, moves the VIP into the powered off state.
- ❖ `VBUS_ON` - When configured for HSIC, moves the VIP into the powered on state.

5.9.12.8 Controlling Skew

The HSIC specification identifies a maximum limit on the allowable "Circuit Board Trace propagation skew" between the Strobe and Data signals. In essence, this parameter is specified to help ensure "signal timing is within specification limits at the receiver". The VIP provides the ability to configure both the skew it expects during detection of bus state transitions, and the skew it uses during the driving of bus state transitions.

Control of the VIP's Strobe/Data skew is through the following configuration properties:

- ❖ `usb_20_hsic_strobe_data_rx_skew` - Defines the maximum expected skew between Strobe and Data during detection of bus state transitions.
- ❖ `usb_20_hsic_strobe_data_tx_skew` - Defines the skew that is driven between Strobe and Data during the driving of bus state transitions.

Note these properties only effect skew during non-data signaling; they have no effect on the timing of Strobe and Data during data signaling. When transmitting data, the VIP sets up Data one eighth of a Strobe-period prior to driving an edge on Strobe, and holds data for three eighths of a Strobe-period following the driving of an edge on Strobe.

5.9.12.9 Controlling RX Skew

The `usb_20_hsic_strobe_data_rx_skew` value is used by the VIP to control the maximum amount of expected skew between Strobe and Data whenever the VIP is detecting a bus state transition. This value identifies the amount of time the Strobe signal may lead or lag the Data signal during detection of bus state transitions: When this timing is met, the VIP will detect a single state transition. When the timing is not met, the VIP will detect multiple state transitions.

For example, in the case of the VIP detecting a transition from SUSPEND (Strobe/Data = 10) to RESUME (Strobe/Data = 01), the VIP expects to see one of the following three scenarios:

- ❖ A bus transition where Strobe and Data change together: 10->01
- ❖ A bus transition where Strobe leads Data: 10->00->01
- ❖ A bus transition where Strobe lags Data: 10->11->01

In each of the preceding scenarios, the `usb_20_hsic_strobe_data_rx_skew` value is used to determine whether or not one or more states transitions get detected: If the end-to-end timing of the completed transition is less than or equal to the value of `usb_20_hsic_strobe_data_rx_skew`, the VIP detects a single state transition. If the end-to-end timing is greater than `usb_20_hsic_strobe_data_rx_skew`, the VIP detects multiple state transitions.

5.9.12.10 Controlling TX Skew

The `usb_20_hsic_strobe_data_tx_skew` value is used to control the amount of skew the inserts between Strobe and Data whenever it drives a bus state transition. This property, treated as a signed value, identifies the amount of time the Strobe signal will lead the Data signal: When positive, this value defines the length of time by which the Strobe will lead the Data signal. When negative, this value defines the length of time by which the Strobe will lag the Data signal. When zero, the Strobe and Data signals are driven simultaneously.

For example, in the case of the VIP driving a transition from SUSPEND (Strobe/Data = 10) to RESUME (Strobe/Data = 01). The `usb_20_hsic_strobe_data_tx_skew` value would have the following effect:

- ❖ If zero, the VIP drives the following Strobe/Data transition: 10->01
- ❖ If positive, the VIP drives the following Strobe/Data transition: 10->00->01
- ❖ If negative, the VIP drives the following Strobe/Data transition: 10->11->01

In each of the preceding scenarios, the `usb_20_hsic_strobe_data_tx_skew` value defines the end-to-end timing of the completed transition.

5.10 Using the Passive Monitor

You can use the VIP as standalone passive monitor or can use it together as an active component of the VIP. All the steps for both the components are similar, except the instance as you will have to define the instance active or passive for their respective components.



Note

It is recommended to set the configuration parameter with DUT.

The following test bench examples explains passive monitor as Host Passive Monitor or as Device Passive Monitor:

5.10.1 VIP as Host Passive Monitor

Perform the following steps to use VIP as standalone passive monitor:

1. Create the following agent:

```
svt_usb_agent                                host_mon_agent;
```

2. Set the VIPs configuration similar to host(dut) configuration (assuming host configuration instance is `host_cfg`)

```
uvm_config_db#(svt_usb_agent_configuration)::set(this, "host_mon_agent", "cfg",  
cfg.host_cfg);
```

3. Set the VIP as passive component

```
uvm_config_db#(bit)::set(this, "host_mon_agent", "is_active", 0);
```

4. Set the following interface (for example, when host interface instance is `host_usb_if`)

```
uvm_config_db#(USB_IF)::set(this, "host_mon_agent", "usb_ss_if",
this.host_usb_if);
```

5.10.2 VIP as Device Passive Monitor

Perform the following steps to use VIP as standalone passive monitor:

1. Create the following agent:

```
svt_usb_agent dev_mon_agent;
```

2. Set the VIPs configuration same as device(dut) configuration (assuming device configuration instance is `dev_cfg`)

```
uvm_config_db#(svt_usb_agent_configuration)::set(this, "dev_mon_agent", "cfg",
cfg.dev_cfg);
```

3. Set the VIP as passive component

```
uvm_config_db#(bit)::set(this, "dev_mon_agent", "is_active", 0);
```

4. Set the following interface (for example, when device interface instance is `dev_usb_if`)

```
uvm_config_db#(USB_IF)::set(this, "dev_mon_agent", "dev_ss_if", this.dev_usb_if);
```



Note

Passive Monitor support is not available for physical as top layer.

5.11 Using Test Mode

This section documents on how to use Test Mode. All high-speed capable devices/hubs must support Test Mode requests. These requests are not supported for non-high-speed devices. Otherwise, the VIP will issue a warning and Abort the protocol service.

5.11.1 Entering Test Mode

You enter Test Mode of a port by using a device specific standard request (for an upstream facing port) or a port specific hub class request (for a downstream facing port):

- ❖ The device standard request SetFeature (TEST_MODE) as defined in Section 9.4.9 of the USB protocol.
- ❖ The hub class request SetPortFeature (PORT_TEST) as defined in Section 11.24.2.13 of the USB protocol.

5.11.1.1 Using the UTMi VIP as a Device MAC

1. Issue a control transfer (set_feature with feature_selector set to TEST_MODE) command to Host (VIP/DUT).
2. The VIP will not interpret the control transfer. As a result, on *successful* completion of this transfer issue the Protocol_service to enter test mode.
 - a. Set Protocol_service with service_type to TEST_MODE and the protocol_20_command_type to one of the following values:
 - ✧ USB_20_TEST_MODE_TEST_SE0_NAK
 - ✧ USB_20_TEST_MODE_TEST_J
 - ✧ USB_20_TEST_MODE_TEST_K
 - ✧ USB_20_TEST_MODE_TEST_PACKET
 - b. The protocol_20_command_type enum value should match whatever has been passed in the control transfer.

5.11.1.2 Using the UTMi VIP as a Host MAC

1. Issue Protocol_service to enter test mode.
2. Set protocol_service with service_type set to TEST_MODE, and protocol_20_command_type set to one of the following values:
 - ✧ USB_20_TEST_MODE_TEST_SE0_NAK
 - ✧ USB_20_TEST_MODE_TEST_J
 - ✧ USB_20_TEST_MODE_TEST_K
 - ✧ USB_20_TEST_MODE_TEST_PACKET

5.11.2 Verifying TEST_PACKET

According to the USB 2.0 specification “Upon command, a port must repetitively transmit the following test packet until the exit action is taken. This enables the testing of rise and fall times, eye patterns, jitter, and any other dynamic waveform specifications.”

The test packet is created by concatenating the following strings. (Note: For J/K NRZI data, and for NRZ data, the bit on the left is the first one transmitted).

- ❖ "S" indicates that a bit stuff occurs, which inserts an "extra" NRZI data bit. "
- ❖ *N" is used to indicate N occurrences of a string of bits or symbols.

A port in Test_Packet mode must send this packet repetitively. The inter-packet timing must be no less than the minimum allowable inter-packet gap as defined in Section 7.1.18 and no greater than 125.

5.11.2.1 Using the UTMI VIP as a Device MAC

1. Issue a control transfer (set_feature with feature_selector set to TEST_MODE & TEST_SELECTOR set to TEST_PACKET) command to the host UTMI VIP using a remote phyHost (VIP/DUT).
2. The VIP does not interpret the control transfer. As a result, on a *successful* completion of this transfer, issue Protocol_service to enter test mode.
 - a. Set the Protocol_service with service_type set to TEST_MODE and protocol_20_command_type to USB_20_TEST_MODE_TEST_PACKET.
 - b. On reception of this protocol service, the device will create link_service and send it to the link layer. The link_service with service_type to LINK_20_PORT_COMMAND and link_20_command_type to USB_20_PORT_TEST_MODE_TEST_PACKET are put into the link layer service channel.
 - c. On successful completion of link_service command, the VIP will create TEST_PACKET and send it to the link layer. The Packet sent to link layer is 53 byte long, with PID set to DATA0. The payload will contain following.

```
00 00 00 00 00 00 00 00
00 AA AA AA AA AA AA AA
AA EE EE EE EE EE EE EE
EE FE FF FF FF FF FF FF
FF FF FF FF FF 7F BF DF
EF F7 FB FD FC 7E BF DF
EF F7 FB FD 7E
```

The Protocol layer will wait for a packet ENDED notification. On receiving notification it will put the same packet again into the link layer packet channel. It is the link layer responsibility to schedule the packet after the expiration of the interpacket delay.

The Protocol layer will continue to send test_packet to the link layer until TEST_Mode is exited.

5.11.2.2 Using the UTMI VIP as a Host MAC

1. Issue Protocol_service to enter test mode.
2. Set Protocol_service with service_type to TEST_MODE and protocol_20_command_type set to USB_20_TEST_MODE_TEST_PACKET. On reception of this protocol service, the device creates a link service and sends it to the link layer.
3. link_service with service_type to LINK_20_PORT_COMMAND and link_20_command_type to USB_20_PORT_TEST_MODE_TEST_PACKET is put into link layer service channel.

4. On completion of the link_service command, the host VIP creates TEST_PACKET and sends it to the link layer. The packet sent to link layer is 53 byte long, with PID set to DATA0. The payload will contain following.

```
00 00 00 00 00 00 00 00
00 AA AA AA AA AA AA AA
AA EE EE EE EE EE EE EE
EE FE FF FF FF FF FF FF
FF FF FF FF FF 7F BF DF
EF F7 FB FD FC 7E BF DF
EF F7 FB FD 7E
```

The Protocol layer will wait for a packet ENDED notification. On receiving the notification, it will put the same packet again into link layer packet channel. It is the link layer responsibility to schedule the packet after the expiration the of interpacket delay.

The Protocol layer will continue to send test_packet to the link layer until TEST_Mode is exited.

5.11.3 Verifying TEST_J

According to USB 2.0 specification regarding Test mode Test_J: "Upon command, a port's transceiver must enter the high-speed J state and remain in that state until the exit action is taken. This enables the testing of the high output drive level on the D+ line."

5.11.3.1 Using the UTMI VIP as a Device MAC

1. Issue a control transfer (set_feature with feature_selector set to TEST_MODE and TEST_SELECTOR set to TEST_J) command to the Host (VIP/DUT).
2. The VIP does not interpret the control transfer. As a result, on *successful* completion of this transfer model enters test mode.
 - a. Set Protocol_service with service_type to TEST_MODE and protocol_20_command_type to USB_20_TEST_MODE_TEST_J.
 - b. On reception of protocol service, the device vip will create a link_service and send it to the link layer.
 - c. The link_service with service_type set to LINK_20_PORT_COMMAND and link_20_command_type set to USB_20_PORT_TEST_MODE_TEST_J is put into link layer service channel.

5.11.3.2 Using the UTMI VIP as a Host MAC

1. Issue Protocol_service to enter test mode. Set Protocol_service with service_type set to TEST_MODE and protocol_20_command_type set to USB_20_TEST_MODE_TEST_J.
2. On reception of the protocol service, the device VIP will create a link_service and send it to the link layer.
3. Link_service with service_type set to LINK_20_PORT_COMMAND and set link_20_command_type to USB_20_PORT_TEST_MODE_TEST_J is put into link a layer service channel

5.11.4 Verifying TEST_K

According to USB 2.0 specification regarding Test Mode Test_K: "Upon command, a port's transceiver must enter the high-speed K state and remain in that state until the exit action is taken. This enables the testing of the high output drive level on the D- line."

5.11.4.1 Using the UTMI VIP as a Device MAC

1. Issue a control transfer (set_feature with feature_selector set to TEST_MODE and TEST_SELECTOR set to TEST_K) command to the Host (VIP/DUT).
2. The VIP does not interpret the control transfer. As a result, on *successful* completion of this transfer issue a Protocol_service request to enter test mode.
 - a. Issue Protocol_service with service_type set to TEST_MODE, and protocol_20_command_type set to USB_20_TEST_MODE_TEST_K.
 - b. On reception of this protocol service request, the device VIP will create a link_service and send it to the link layer.
 - c. link_service (with service_type set to LINK_20_PORT_COMMAND and link_20_command_type set to USB_20_PORT_TEST_MODE_TEST_K) is put into link layer service channel.

5.11.4.2 Using the UTMI VIP as a Host MAC

1. Issue Protocol_service to enter test mode. Protocol_service request with service_type set to TEST_MODE and protocol_20_command_type set to USB_20_TEST_MODE_TEST_K.
2. On reception of this protocol service, the host VIP creates link_service and sends it to the link layer.
3. The link_service with service_type set to LINK_20_PORT_COMMAND and link_20_command_type set to USB_20_PORT_TEST_MODE_TEST_K is put into link layer service channel

5.11.5 Verifying TEST_SE0_NAK

According to USB 2.0 specification regarding Test mode Test_SE0_NAK: “Upon command, a port's transceiver must enter the high-speed receive mode and remain in that mode until the exit action is taken. This enables the testing of output impedance, low level output voltage, and loading characteristics. In addition, while in this mode, upstream facing ports (and only upstream facing ports) must respond to any IN token packet with a NAK handshake (only if the packet CRC is determined to be correct) within the normal allowed device response time. This enables testing of the device squelch level circuitry and, additionally, provides a general purpose stimulus/response test for basic functional testing.”

5.11.5.1 Using the UTMI VIP as a Device MAC

1. Issue a control transfer (set_feature with feature_selector set to TEST_MODE & TEST_SELECTOR set to TEST_SE0_NAK) command to the Host (VIP/DUT).
2. The VIP does not interpret the control transfer. As a result, on *successful* completion of this transfer issue Protocol_service enters test mode.
 - a. Issue Protocol_service with service_type set to TEST_MODE and protocol_20_command_type set to USB_20_TEST_MODE_TEST_SE0_NAK.
 - b. On reception of this protocol service, the device VIP creates link_service and sends it to the link layer.
 - c. link_service with service_type set to LINK_20_PORT_COMMAND and link_20_command_type set to USB_20_PORT_TEST_MODE_TEST_SE0_NAK is put into link layer service channel
 - d. If the user wants to test a NAK response, they put IN TOKEN PACKET into the packet input channel of the link layer of the host.
 - e. For any IN TOKEN PACKET received during a TEST_SE0_NAK state, the device will respond with NAK if CRC is found to be correct.

5.11.5.2 Using the UTMI VIP as a Host MAC

1. Issue a control transfer (set_feature with feature_selector set to TEST_MODE & TEST_SELECTOR set to TEST_SE0_NAK) command to the host.
2. The VIP does not interpret the control transfer. As a result, on *successful* completion of this transfer issues Protocol_service to enter test mode.
 - a. Issue Protocol_service with service_type set to TEST_MODE and protocol_20_command_type set to USB_20_TEST_MODE_TEST_SE0_NAK.
 - b. On reception of this protocol service, the host VIP creates link_service and sends it to the link layer.
 - c. link_service with service_type set to LINK_20_PORT_COMMAND and link_20_command_type set to USB_20_PORT_TEST_MODE_TEST_SE0_NAK is put into a link layer service channel

5.11.6 Exiting test_mode on Downstream Facing Ports

According to USB 2.0 specification for a downstream facing port, the exit action is to reset the hub, as defined in Section 11.24.2.13. After the test is completed, the hub (with the port under test) must be reset by the host or user. This must be accomplished by manipulating the port of the parent hub to which the hub under test is attached. This manipulation can consist of one of the following:

- ❖ Issuing a SetPortFeature (PORT_RESET) to port of the parent hub to which the hub under test is attached.
- ❖ Issuing a ClearPortFeature (PORT_POWER) and SetPortFeature (PORT_POWER) to cycle power of a parent hub that supports per port power control.
- ❖ Disconnecting and re-connecting the hub under test from its parent hub port.
- ❖ For a root hub under test, a reset of the Host Controller may be required as there is no parent hub of the root hub.

To exit TEST_MODE, issue a port_reset command. Set link_service with service_type set to LINK_20_PORT_COMMAND and link_20_command_type set to SVT_USB_20_PORT_RESET.

According to USB 2.0 specification for an upstream facing port, the exit action is to power cycle the device. It is the host/testbench responsibility to power cycle the device VIP.

5.11.7 Test Mode Notifications

The following notifications in svt_usb_status support Test Mode:

- ❖ NOTIFY_PORT_TEST_MODE_ENTERED
- ❖ NOTIFY_PORT_TEST_MODE_EXITED
- ❖ NOTIFY_PORT_TEST_MODE_ENTERED is issued by the link layer on reception of a link_service command, and on expiration of ttest_mode_entry_delay. It is issued when TEST_MODE is entered.
- ❖ NOTIFY_PORT_TEST_MODE_EXITED is issued when TEST_MODE is exited.

5.11.8 Test Mode Configuration Members

- ❖ ttest_mode_entry_delay in the svt_usb_configuration class. Models the delay between successful completion of a status stage and the host/device entering TEST_MODE.
- ❖ ttest_mode_entry_delay. Defaults to `SVT_USB_20_USER_TEST_MODE_ENTRY_DELAY_MAX` which is set to 3ms. User can override this define.

5.12 SuperSpeed InterChip Physical (SSIC) Usage (Beta Feature)

The USB Verification IP supports SSIC RMMI interfaces connected to an M-Phy supporting an RMMI interface. The following sections show general usage of the SSIC Physical transactor.

The `svt_usb_subenv::new()` constructor takes a `svt_usb_agent_configuration` object as the `cfg` argument and a `svt_usb_configuration` object as the `remote_cfg` argument. These parameters represent

- ❖ The `cfg` object represents the VIP's view of its configuration with respect to its local side of the connection.
- ❖ The `remote_cfg` object represents the VIP's view of the remote Phy's configuration.

Note, the subsections focus on configuring parameter values `cfg` and `remote_cfg` at instantiation time for several connection scenarios.

5.12.1 Configuring the VIP for a DUT Without a SSIC Physical Layer

If the VIP USB Host/Device is being used to test a DUT which does not contain an SSIC Phy (i.e. if the DUT's connection is from the MAC perspective of an SSIC MAC/Phy RMMI interface) then the VIP requires two `svt_usb_subenv_configuration` objects: `cfg` and `remote_cfg`. In this case, the VIP can be thought of as containing two *virtual* Physical layers (one local, and the other remote):

5.12.1.1 cfg Parameter Settings

The following settings would be typical for the `cfg` parameter discussed previously:

- ❖ `ssic_profile` = <desired profile to use>
- ❖ `ssic_pairs` = must equal the number of pairs required by `ssic_profile`.
- ❖ `bottom_layer` = SUB_PHYSICAL
- ❖ `top_layer` >= PROTOCOL (assuming VIP functionality up through protocol layer *transfers* is desired)
- ❖ `component_subtype` = MAC
- ❖ `usb_ss_signal_interface` = USB_SSIC_TLM
- ❖ `usb_mphy_rx_cfg[$]` and `usb_mphy_tx_cfg[$]` have settings of:
 - ◆ Must size to have `cfg.ssic_pairs` entries in each array
 - ◆ `mphy_component_type` == `svt_mphy_configuration::PHY`
 - ◆ `mphy_interface_type` == `svt_mphy_configuration::RMMI_REMOTE`
 - ◆ `mphy_tlm_if_type` == `svt_mphy_configuration::TLM_LOCAL`

With UVM, the interfaces are assigned through the `config_db`.

5.12.1.2 remote_cfg Parameter Settings

The following settings would be typical for the `remote_cfg` parameter discussed previously:

- ❖ `ssic_profile` = Must match the value in `cfg`
- ❖ `ssic_pairs` = Must match the value in `cfg`
- ❖ `component_subtype` = PHY
- ❖ `usb_ss_signal_interface` = SSIC_RMMI_IF
- ❖ `usb_mphy_rx_cfg[$]` and `usb_mphy_tx_cfg[$]` have settings of:

- ◆ Must size to have `cfg.ssic_pairs` entries in each array
- ◆ `mphy_interface_type == svt_mphy_configuration::RMMI_REMOTE`
- ◆ `mphy_tlm_if_type == svt_mphy_configuration::TLM_REMOTE`
- ◆ `mphy_component_type == PHY`

With UVM, the interfaces are assigned through the `config_db`.

5.12.2 Configuring the VIP for a DUT With a SSIC Physical Layer

In this section, configuration options are shown for a DUT with a SSIC Physical Layer.

If the VIP USB Host/Device is being used to test a DUT which contains an SSIC Phy if the DUT's connection is directly to the SSIC Serial interface), then the VIP requires one `svt_usb_subenv_configuration` object `cfg` (and no `remote_cfg`). In this case, the VIP can be thought of as containing one virtual Phy (its local phy). The `cfg` object represents the VIP's view of its configuration with respect to its local side of the connection, while the `remote_cfg` object represents the VIP's view of the remote Phy's configuration.

5.12.2.1 `cfg` Parameter Settings

- ❖ `ssic_profile` = <desired profile to use>
- ❖ `ssic_pairs` = must equal the number of pairs required by `ssic_profile`.
- ❖ `bottom_xactor` (UVM) / `bottom_layer` (UVM) = `SUB_PHYSICAL`
- ❖ `top_xactor` (UVM) / `top_layer` (UVM) `>= PROTOCOL` (assuming VIP functionality up through protocol layer *transfers* is desired)
- ❖ `component_subtype` = `MAC`
- ❖ `usb_ss_signal_interface` = `SSIC_SERIAL_IF`
- ❖ `usb_mphy_rx_cfg[$]` and `usb_mphy_tx_cfg[$]` have settings of:
 - ◆ Must size to have `cfg.ssic_pairs` entries in each array
 - ◆ `mphy_component_type == svt_mphy_configuration::PHY`
 - ◆ `mphy_interface_type == svt_mphy_configuration::SERIAL`
 - ◆ `mphy_tlm_if_type == svt_mphy_configuration::TLM_NONE`

With UVM, the interfaces are assigned through the `config_db`.

5.12.2.2 `remote_cfg` Parameter Settings

For this type of connection (an SSIC Serial connection), the parameter `remote_cfg` remains null.

5.13 SystemVerilog UVM Example Testbenches

This section describes SystemVerilog UVM example testbenches that show general usage for various applications. A summary of the examples is listed in [Table 5-1](#).

Table 5-16 SystemVerilog Example Summary

| Name | Source in design_dir/Description |
|--|---|
| Basic Example --Demonstrates how to implement a UVM testbench using USB VIP with SS Serial interface. This example consists of a top-level testbench, a Verilog hdl_interconnect, a UVM verification environment, a host and device agent components, one test file with an sequence, and two directed tests. | |
| tb_usb_svt_uvm_basic_sys | examples/sverilog/usb_svt/tb_usb_svt_uvm_basic_sys/ |
| Intermediate Example --Builds on the Basic UVM Example by adding coverage and a scoreboard. | |
| tb_usb_svt_uvm_intermediate_syst | examples/tb_usb_svt_uvm_intermediate_syst |

6

Verification Topologies

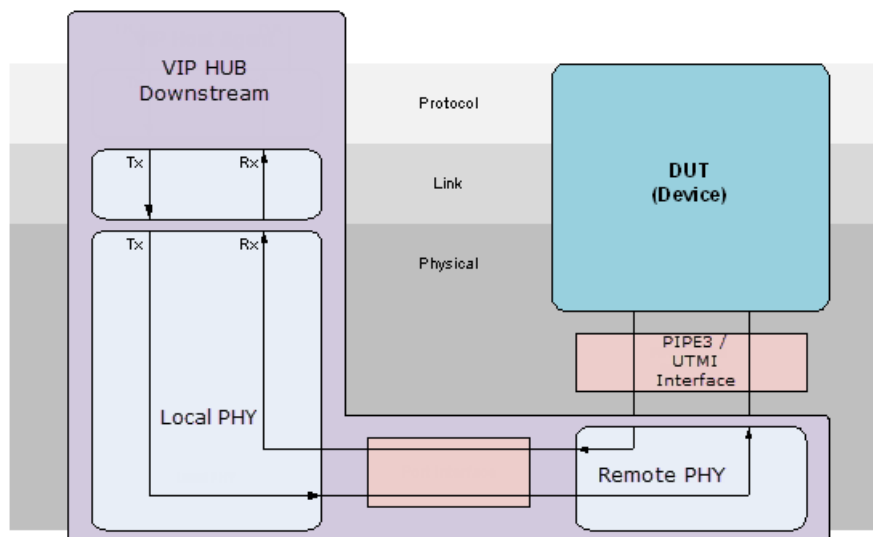
This chapter presents several agent testbench topologies. In addition to a brief description of the testbench topologies, each section lists configuration parameters required to instantiate the VIP agent.

6.1 USB VIP Downstream 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.

In addition to a valid `cfg` object of type `svt_usb_agent_configuration`, the presence of a remote physical layer requires the end user to provide a handle to a valid `remote_cfg` object of type `svt_usb_configuration`.

Figure 6-1 USB HUB VIP Downstream and DUT Device Controller



Implementation of this topology requires the setting of the following properties for 3.0:

- ❖ **cfg object (representing the Downstream hub vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
```

```

component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = 1;

```

❖ **remote_cfg - representing the remote phy's properties**

```

svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = PHY;
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PIPE3_IF; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_UTMI_IF
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;

```

❖ **cfg object (representing the Host vip)**

```

svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = `SVT_USB_NUM_DEVICE_INST + `SVT_USB_NUM_HUB_INST;

```

❖ **cfg object (representing the Upstream hub vip)**

```

svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;

usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 2; (Control and Interrupt IN)
int remote_device_cfg_size = 1;

```

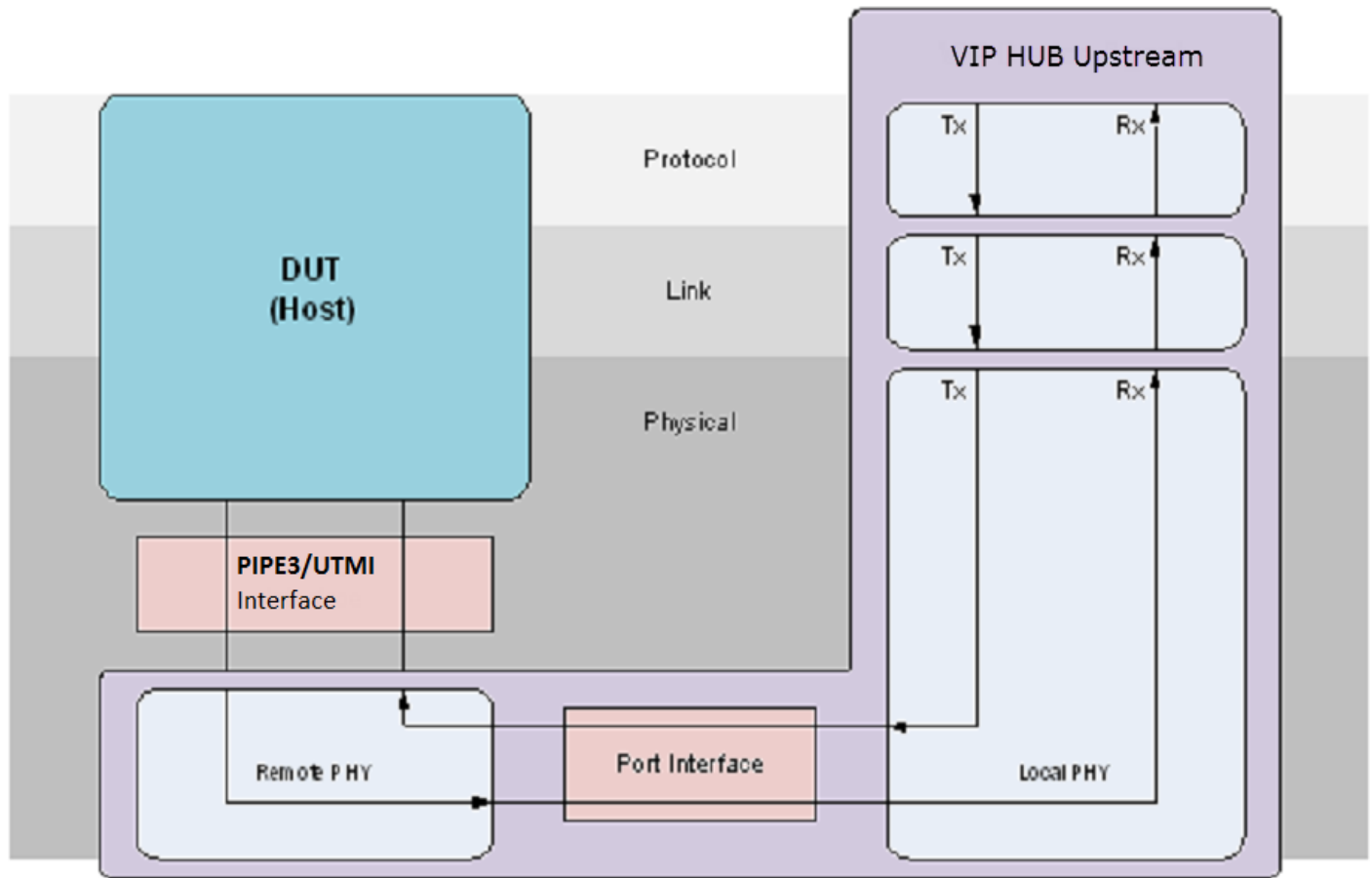
❖ **remote_cfg - representing the remote phy's properties**

```

svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = PHY;
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PIPE3_IF; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_UTMI_IF
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;

```

Figure 6-2 USB VIP Upstream and DUT Host Controller



Implementation of this topology requires the setting of the following properties for 3.0:

❖ **cfg object (representing the Downstream hub vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = 1;
```

❖ **remote_cfg - representing the remote phy's properties**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = PHY;
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PIPE3_IF; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_UTMI_IF

usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
```


❖ **cfg object (representing the Device vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = `SVT_USB_NUM_DEVICE_INST + `SVT_USB_NUM_HUB_INST;
```

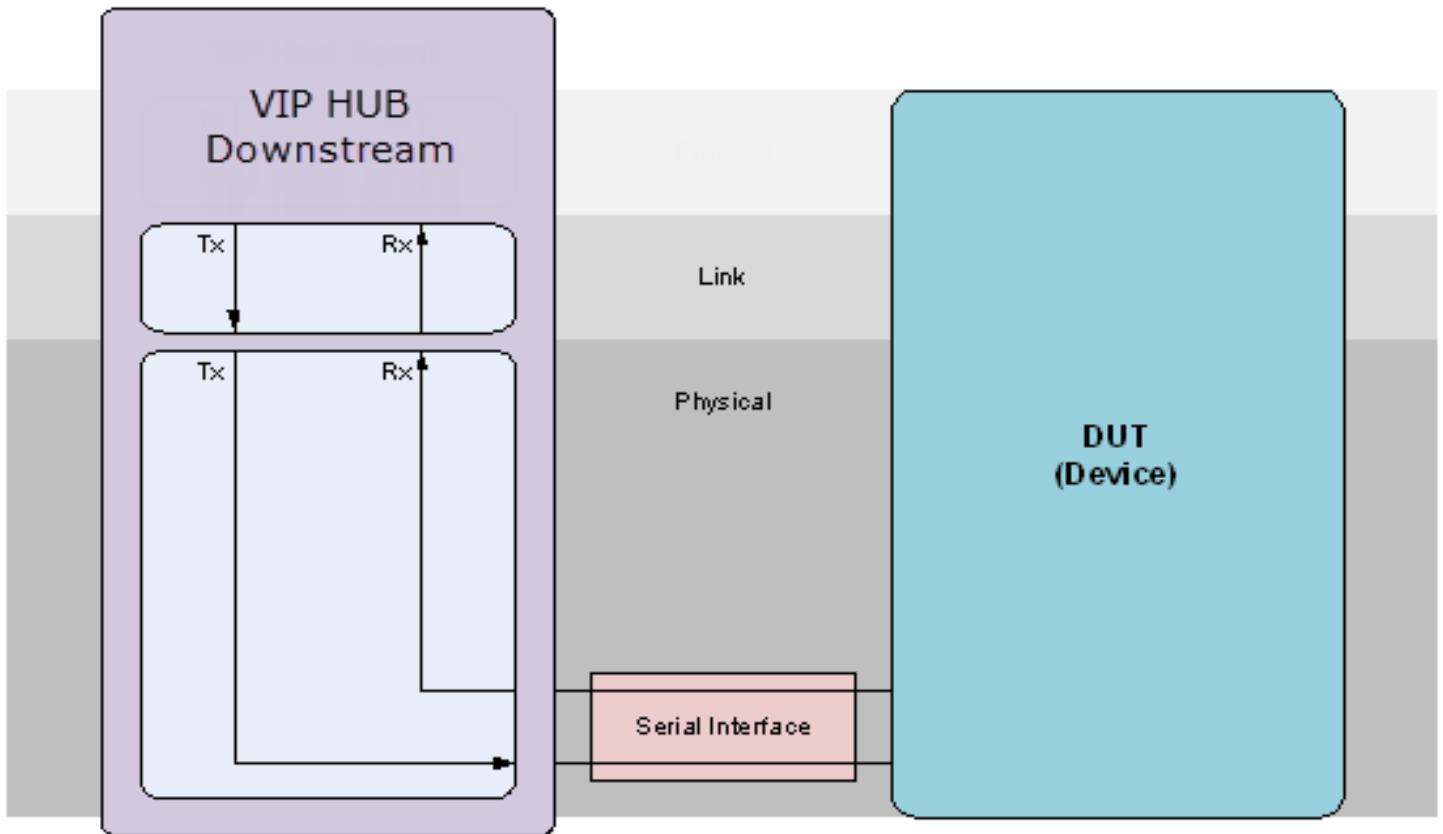
❖ **cfg object (representing the Upstream hub vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 2; (Control and Interrupt IN)
int remote_device_cfg_size = 1;
```

❖ **remote_cfg - representing the remote phy's properties**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = PHY;
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PIPE3_IF; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_UTMI_IF
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
```

Figure 6-3 USB3 HUB VIP Downstream and DUT Device PHY



Implementation of this topology requires the setting of the following properties:

❖ **cfg object (representing the host vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$] ;
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = `SVT_USB_NUM_DEVICE_INST + `SVT_USB_NUM_HUB_INST; (size
needs to be greater than or equal to 1)
```

❖ **cfg object (representing the Upstream vip)**

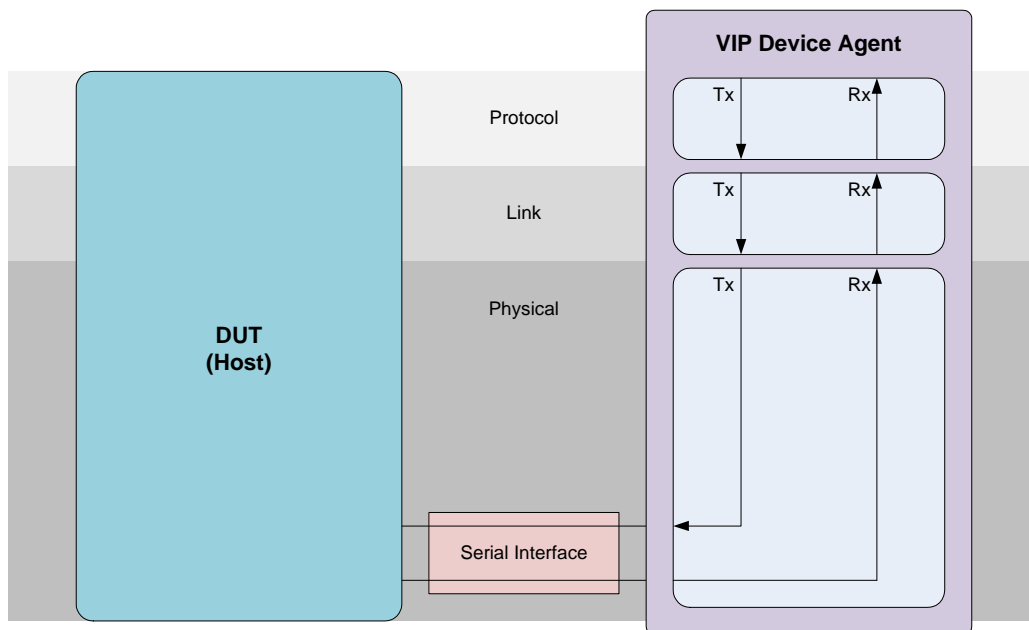
```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_SERIAL_IF; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_SERIAL_IF
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration local_device_cfg[$] ;
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 2; (Control and Interrupt IN)
int remote_device_cfg_size = 1; (size needs to be greater than or equal to 1)
```

❖ **cfg object (representing the Downstream vip)**

```

svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$] ;
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = 1; (size needs to be equal to 1)

```

Figure 6-4 USB Upstream and DUT Host Controller

Implementation of this topology requires the setting of the following properties:

❖ **Simulating a SS (only) serial bus**

```

svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_SERIAL_IF;
usb_capability_enum usb_capability = USB_SS_ONLY;
svt_usb_device_configuration remote_device_cfg[$] ;
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = 1; (size needs to be greater than or equal to 1)

```

❖ **Simulating a 2.0 (only) serial bus**

```

svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::HS; (other speeds FS/LS)
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_20_SERIAL_IF;
usb_capability_enum usb_capability = USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];

```

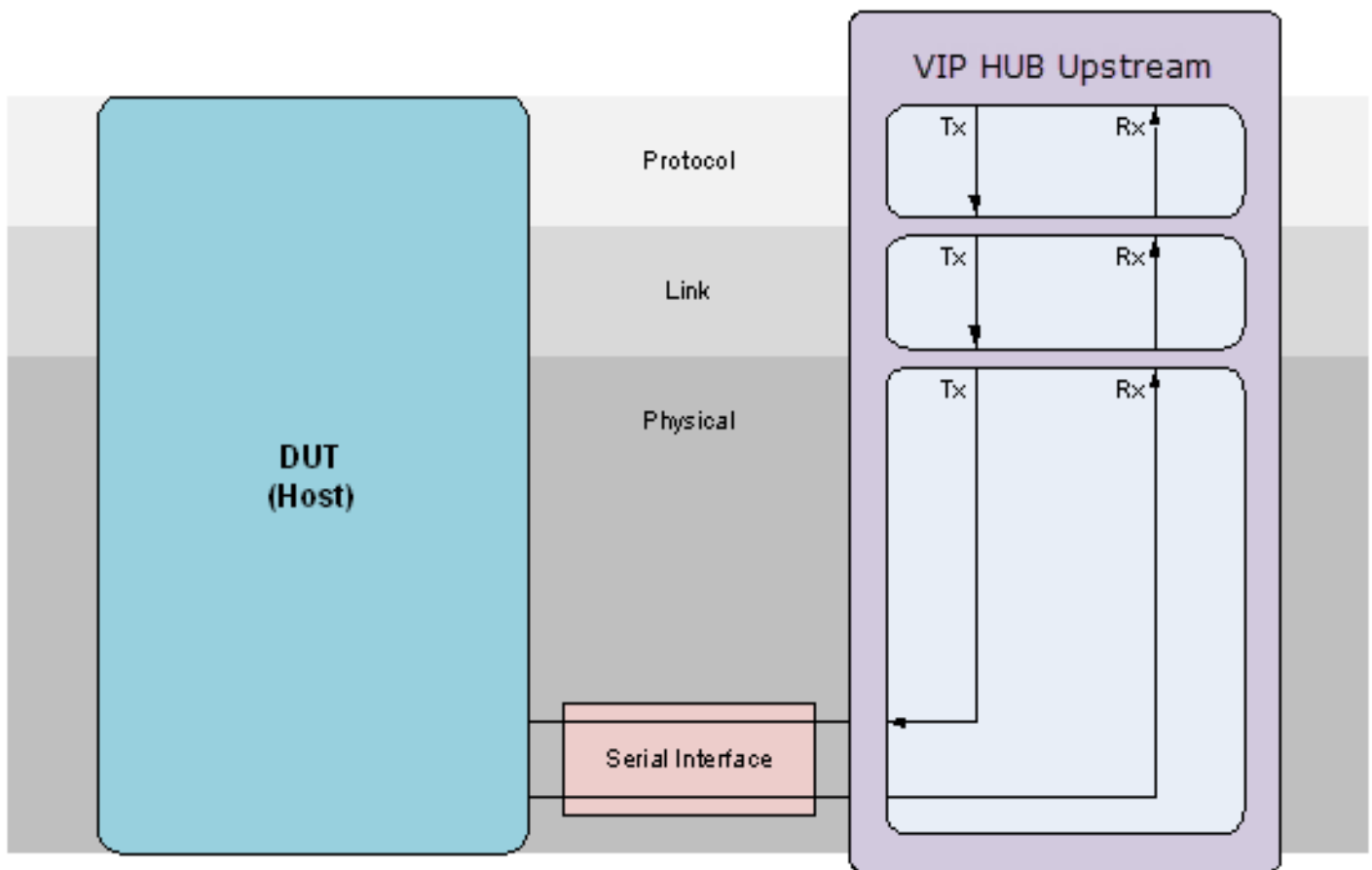
```
svt_usb_endpoint_configuration endpoint_cfg[$];  
int unsigned num_endpoints = 1;  
int remote_device_cfg_size = 1; (size needs to be greater than or equal to 1)
```

6.2 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.

The VIP instance requires a valid `cfg` object of type `svt_usb_agent_configuration`. This section provides configuration objects for SS and 2.0 serial interfaces.

Figure 6-5 USB HUB Upstream VIP and DUT Host



Implementation of this topology requires the setting of the following properties:

❖ **cfg object (representing the device vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;  
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;  
component_subtype_enum component_subtype = MAC  
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;  
svt_usb_device_configuration remote_device_cfg[$] ;  
svt_usb_endpoint_configuration endpoint_cfg[$];  
int unsigned num_endpoints = 1;
```

```
int remote_device_cfg_size = 1; (size needs to be greater than or equal to 1)
```

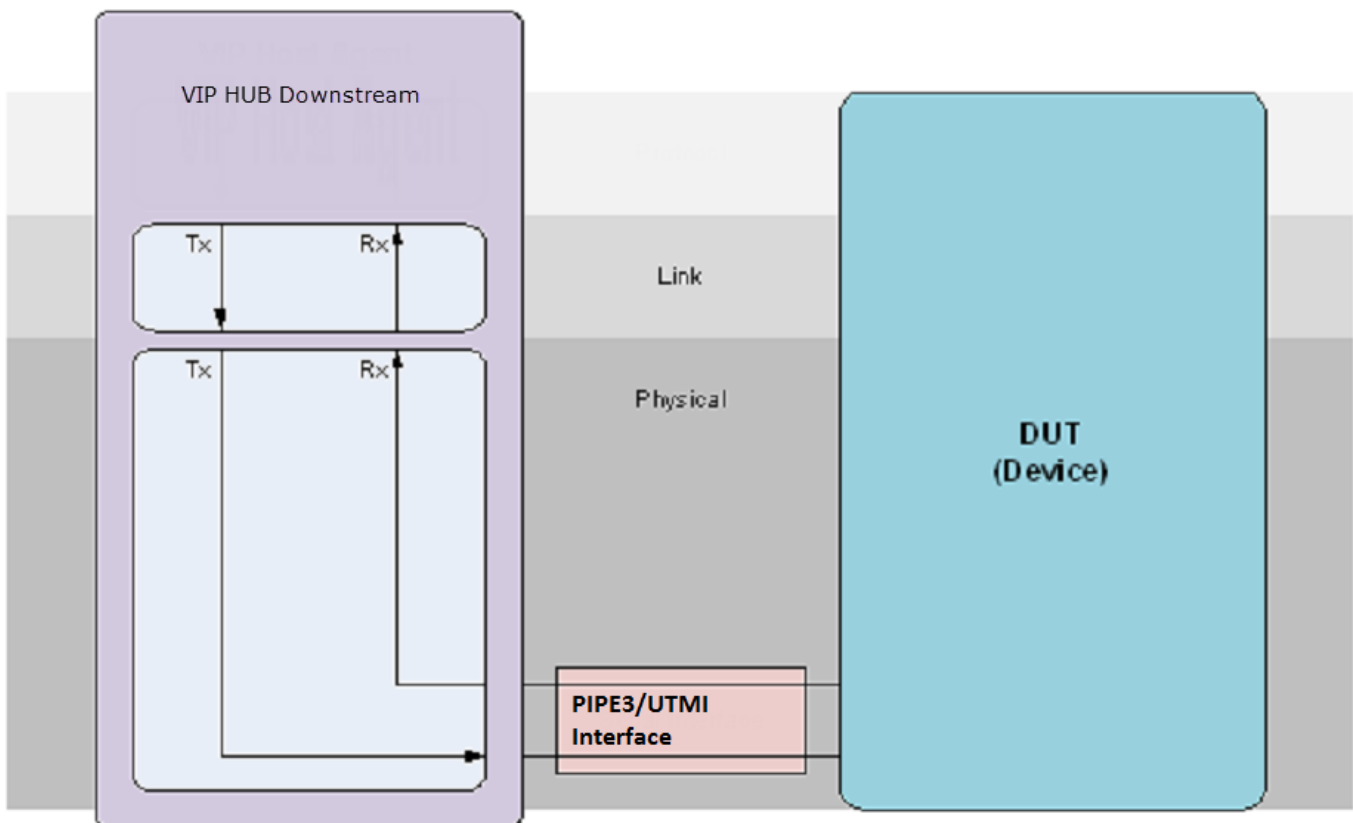
❖ **cfg object (representing the Upstream vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_SERIAL_IF; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_SERIAL_IF
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration local_device_cfg[$] ;
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 2; (Control and Interrupt IN)
int remote_device_cfg_size = 1; (size needs to be greater than or equal to 1)
```

❖ **cfg object (representing the Downstream vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$] ;
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = 1; (size needs to be equal to 1)
```

Figure 6-6 USB3 HUB VIP Downstream and DUT Device



Implementation of this topology requires the setting of the following properties for 3.0:

❖ **cfg object (representing the Host vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = `SVT_USB_NUM_DEVICE_INST + `SVT_USB_NUM_HUB_INST;
```

❖ **remote_cfg - representing the remote phy's properties**

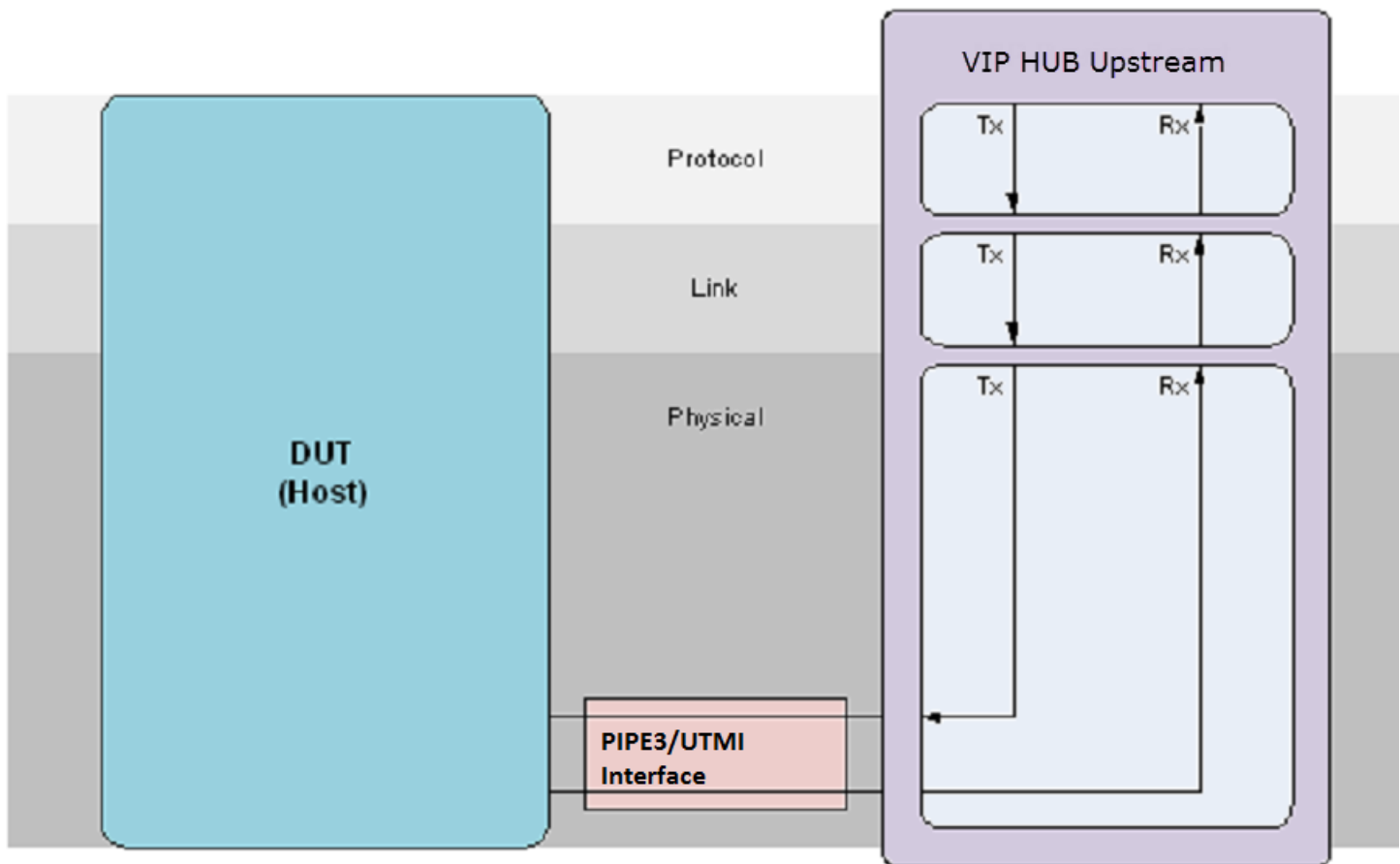
```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = PHY;
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PIPE3_IF; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_UTMI_IF
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
```

❖ **cfg object (representing the Downstream vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = 1;
```

❖ **cfg object (representing the Upstream hub vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 2; (Control and Interrupt IN)
int remote_device_cfg_size = 1;
```

Figure 6-7 USB3 HUB VIP Upstream and DUT Host

Implementation of this topology requires the setting of the following properties for 3.0:

❖ **cfg object (representing the host vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = 1;
??remote_cfg - representing the remote phy's properties
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = PHY;
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PIPE3_IF; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_UTMI_IF
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
```

❖ **cfg object (representing the device vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
```

```
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 1;
int remote_device_cfg_size = 1;
```

❖ **remote_cfg – representing the remote phy's properties**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = PHY;
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PIPE3_IF; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_UTMI_IF
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
```

❖ **cfg object (representing the Upstream hub vip)**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS/HS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT; or
usb_20_signal_interface_enum usb_20_signal_interface = USB_20_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY/USB_20_ONLY;
svt_usb_device_configuration remote_device_cfg[$];
svt_usb_endpoint_configuration endpoint_cfg[$];
int unsigned num_endpoints = 2; (Control and Interrupt IN)
int remote_device_cfg_size = 1;
```


7

Integration Steps for HUB Model

This chapter provides integration steps for the HUB model.

7.1 USB HUB Model Integration

The following code snippets provides the integration steps for USB HUB model.

Figure 7-1 Instantiate

```
/**
 * Instantiate Serial Interface for Host Side USB Port
 */
svt_usb_if usb_host_20_serial_if ();

/**
 * Instantiate Serial Interface's for HUB Side USB Port
 */
svt_usb_hub_if usb_hub_20_serial_if [SVT_USB_NUM_HUB_INST]();

/**
 * Instantiate Serial Interface's for Device Side USB Port
 */
svt_usb_if usb_dev_20_serial_if[SVT_USB_NUM_DEVICE_INST]();

/**
 * Instantiate HDL Interconnect with Host-side USB interface on one side and
 * Device-side USB interface on the other side
 */
usb_20_serial_host_dut_hub_upstream_sv_wrapper usb_20_serial_host_dut_hub_upstream_inst(usb_host_20_serial_if, usb_hub_20_serial_if[0].usb_20_upstream_if);
usb_20_serial_device_dut_hub_downstream_sv_wrapper usb_20_serial_device_dut_hub_downstream_inst(usb_dev_20_serial_if, usb_hub_20_serial_if);
```

Figure 7-2 Assign Generated Clock

```
/**
 * Assign generated clock to the Device VIPs' clock input, "clk", of USB 2.0 serial interface
 * Store usb_dev_20_serial_if[SVT_USB_NUM_DEVICE_INST] in Configuration Database
 */
string string_dev_if;
genvar dev_inst_number;
generate for(dev_inst_number=0;dev_inst_number<SVT_USB_NUM_DEVICE_INST;dev_inst_number++) begin : dev_inst
  assign usb_dev_20_serial_if[dev_inst_number].usb_20_serial_if.clk = usb_20_serial_clock_for_VIP;
  initial
  begin
    string_dev_if = $psprintf("dev_usb_if[%5d]",dev_inst_number);
    uvm_config_db#(virtual svt_usb_if)::set(uvm_root::get(),"uvm_test_top.env",string_dev_if,usb_dev_20_serial_if[dev_inst_number]);
  end
end
endgenerate

/**
 * Assign generated clock to the HUB VIPs' internal interfaces clock input, "clk", of USB 2.0 serial interface
 * Store usb_hub_20_serial_if[SVT_USB_NUM_HUB_INST] in Configuration Database
 */
string string_hub_if;
genvar hub_inst_number,dwnstrm_inst_number;
generate for(hub_inst_number=0;hub_inst_number<SVT_USB_NUM_HUB_INST;hub_inst_number++) begin : hub_inst
  assign usb_hub_20_serial_if[hub_inst_number].usb_20_upstream_if.usb_20_serial_if.clk = usb_20_serial_clock_for_VIP;
  for(dwnstrm_inst_number=0;dwnstrm_inst_number<SVT_USB_NUM_OF_DOWNSTREAM_PORTS_INST;dwnstrm_inst_number++) begin : dwnstrm_inst
    assign usb_hub_20_serial_if[hub_inst_number].usb_20_downstream_if[dwnstrm_inst_number].usb_20_serial_if.clk = usb_20_serial_clock_for_VIP;
  end
  initial
  begin
    string_hub_if = $psprintf("hub_usb_if[%5d]",hub_inst_number);
    uvm_config_db#(virtual svt_usb_hub_if)::set(uvm_root::get(),"uvm_test_top.env",string_hub_if,usb_hub_20_serial_if[hub_inst_number]);
  end
end
endgenerate
```

Figure 7-3 HUB Environment Configurations

```

/**
 * If we made it this far, the cfg is good -- pull out and register the host and device cfgs
 */
uvm_config_db#(svt_usb_agent_configuration)::set(this, "host_agent", "cfg", cfg.host_cfg);

foreach(cfg.dev_cfg[i]) begin
    string_dev_cfg = $psprintf("dev_agent[%0d]", i);
    uvm_config_db#(svt_usb_agent_configuration)::set(this, string_dev_cfg, "cfg", cfg.dev_cfg[i]);
end

foreach(cfg.hub_cfg[i]) begin
    string_hub_cfg = $psprintf("hub_agent[%0d]", i);
    uvm_config_db#(svt_usb_hub_agent_configuration)::set(this, string_hub_cfg, "hub_cfg", cfg.hub_cfg[i]);
end

if (this.cfg.enable_host_mon == 1) begin
    uvm_config_db#(svt_usb_agent_configuration)::set(this, "host_mon_agent", "cfg", cfg.host_cfg);
    uvm_config_db#(bit)::set(this, "host_mon_agent", "is_active", 0);
end
foreach(this.cfg.enable_dev_mon[i]) begin
    string_dev_mon_cfg = $psprintf("dev_mon_agent[%0d]", i);
    if (this.cfg.enable_dev_mon[i] == 1) begin
        uvm_config_db#(svt_usb_agent_configuration)::set(this, string_dev_mon_cfg, "cfg", cfg.dev_cfg[i]);
        uvm_config_db#(bit)::set(this, string_dev_mon_cfg, "is_active", 0);
    end
end
end

```

Figure 7-4 Extract HUB Agent Configurations

```

// Extract 'hub_cfg' from configuration database .
if (svt_config_object_db#(svt_usb_hub_agent_configuration)::get(this, "", "hub_cfg", obj_reg_cfg) && (obj_reg_cfg != null)) begin
    `svt_verbose(name, "cfg" set with an svt_usb_hub_agent_configuration provided via the UVM/OVM configuration infrastructure.");
end else begin
    `svt_fatal(name, "cfg" is null. An svt_usb_hub_agent_configuration object was not set via the UVM/OVM configuration infrastructure. Unable to continue.");
end

// Cast recovered configuration to obj_req_cfg .
if (!$cast(this.cfg, obj_req_cfg)) begin
    `svt_fatal(name, "Unable to use 'cfg' to create a new configuration which can be stored as 'cfg'. Unable to continue.");
end

// Extract remote_cfg's for upstream and downstream from configuration database .
if (svt_config_object_db#(svt_usb_configuration)::get(this, "", "upstream_remote_cfg", obj_reg_upstream_remote_cfg) && (obj_reg_upstream_remote_cfg != null)) begin
    `svt_verbose(name, "Recovered Upstream Remote cfg from configuration database ");
end

for(int downstream_inst_num = 0; downstream_inst_num < "SVT_USB_NUM_OF_DOWNSTREAM_PORTS_INST"; downstream_inst_num++) begin
    string_downstream_agent = $psprintf("downstream_remote_cfg[%0d]", downstream_inst_num);
    if (svt_config_object_db#(svt_usb_configuration)::get(this, "", string_downstream_agent, obj_reg_downstream_remote_cfg(downstream_inst_num)) && (obj_reg_downstream_remote_cfg(downstream_inst_num) != null)) begin
        `svt_verbose(name, $psprintf("Recovered Downstream Remote cfg number [%0d] from configuration database ", downstream_inst_num));
    end
end
end

```

Figure 7-5 Passing Extracted Configurations to HUB Sub-Agent

```

// Passing the recovered cfg and remote_cfg to upstream and downstream agent's cfg and remote_cfg.
svt_config_object_db#(svt_usb_agent_configuration)::set(this, "upstream_agent", "cfg", obj_reg_cfg.upstream_cfg);
if (obj_reg_upstream_remote_cfg != null)
    obj_reg_upstream_remote_cfg.is_upstream = 1;
svt_config_object_db#(svt_usb_configuration)::set(this, "upstream_agent", "remote_cfg", obj_reg_upstream_remote_cfg);
foreach(obj_reg_cfg.downstream_cfg[j]) begin
    string_downstream_agent = $psprintf("downstream_agent[%0d]", j);
    if (obj_reg_downstream_remote_cfg[j] != null)
        obj_reg_downstream_remote_cfg[j].is_downstream = 1;
    svt_config_object_db#(svt_usb_agent_configuration)::set(this, string_downstream_agent, "cfg", obj_reg_cfg.downstream_cfg[j]);
    svt_config_object_db#(svt_usb_configuration)::set(this, string_downstream_agent, "remote_cfg", obj_reg_downstream_remote_cfg[j]);
end
end

```

Figure 7-6 Interfaces Inside USB HUB Interface

```
interface svt_usb_hub_if();  
//-----  
// USB HUB VIP Internally Supported Interfaces  
//-----  
  
`ifdef SVT_USB_IF_NO_PARAMS  
  
    svt_usb_if usb_20_upstream_if ();  
  
    svt_usb_if usb_20_downstream_if[`SVT_USB_NUM_OF_DOWNSTREAM_PORTS_INST] ();  
  
    svt_usb_if usb_ss_upstream_if ();  
  
    svt_usb_if usb_ss_downstream_if[`SVT_USB_NUM_OF_DOWNSTREAM_PORTS_INST] ();  
  
    /** Get svt_usb_if operating as HUB VIP Upstream USB20 Interface */  
    function virtual svt_usb_if get_usb_20_upstream_if();  
        get_usb_20_upstream_if = svt_usb_hub_if.usb_20_upstream_if;  
    endfunction  
  
    /** Get svt_usb_if operating as HUB VIP Downstream USB20 Interface */  
    function virtual svt_usb_if get_usb_20_downstream_if(int downstream_port);  
        case (downstream_port)  
            `ifdef SVT_USB_DOWNSTREAM_PORT_1  
                0 : begin get_usb_20_downstream_if = svt_usb_hub_if.usb_20_downstream_if[0]; end  
            `endif  
            `ifdef SVT_USB_DOWNSTREAM_PORT_2  
                1 : begin get_usb_20_downstream_if = svt_usb_hub_if.usb_20_downstream_if[1]; end  
            `endif  
            `ifdef SVT_USB_DOWNSTREAM_PORT_3  
                2 : begin get_usb_20_downstream_if = svt_usb_hub_if.usb_20_downstream_if[2]; end  
            `endif  
        endcase  
    endfunction  
  
`endif
```

Figure 7-7 HUB Model Agents

```
/**  
 * An instance of VIP AGENT to act as a USB HOST.  
 */  
svt_usb_agent          host_agent;  
  
/**  
 * An array of VIP AGENT to act as a USB HUB.  
 */  
svt_usb_hub_agent      hub_agent[`SVT_USB_NUM_HUB_INST];  
  
/**  
 * An array of VIP AGENT to act as a USB DEVICE.  
 */  
svt_usb_agent          dev_agent[`SVT_USB_NUM_DEVICE_INST];
```

Figure 7-8 HUB Sub-Agent

```
/* Upstream (Device) Agent inside Hub Agent */  
svt_usb_agent upstream_agent;  
  
/* Downstream (Host) Agents inside Hub Agent */  
svt_usb_agent downstream_agent[];
```

Figure 7-9 Virtual Interfaces in Environment

```

/**
 * Register the interfaces
 */
if (this.host_usb_if != null) begin
  if (cfg.host_cfg.usb_ss_signal_interface != svt_usb_configuration::NO_SS_IF) begin
    uvm_config_db#(USB_IF)::set(this, "host_agent", "usb_ss_if", this.host_usb_if);
    if (this.cfg.enable_host_mon == 1) uvm_config_db#(USB_IF)::set(this, "host_mon_agent", "usb_ss_if", this.host_usb_if);
  end
  if (cfg.host_cfg.usb_20_signal_interface != svt_usb_configuration::NO_20_IF) begin
    uvm_config_db#(USB_IF)::set(this, "host_agent", "usb_20_if", this.host_usb_if);
    if (this.cfg.enable_host_mon == 1) uvm_config_db#(USB_IF)::set(this, "host_mon_agent", "usb_20_if", this.host_usb_if);
  end
end
foreach(dev_usb_if[i]) begin
  string_dev_cfg = $psprintf("dev_agent[%0d]", i);
  string_dev_mon_cfg = $psprintf("dev_mon_agent[%0d]", i);
  if (cfg.dev_cfg[i].usb_ss_signal_interface != svt_usb_configuration::NO_SS_IF) begin
    uvm_config_db#(USB_IF)::set(this, string_dev_cfg, "usb_ss_if", this.dev_usb_if[i]);
    if (this.cfg.enable_dev_mon[i] == 1) uvm_config_db#(USB_IF)::set(this, string_dev_mon_cfg, "usb_ss_if", this.dev_usb_if[i]);
  end
  if (cfg.dev_cfg[i].usb_20_signal_interface != svt_usb_configuration::NO_20_IF) begin
    uvm_config_db#(USB_IF)::set(this, string_dev_cfg, "usb_20_if", this.dev_usb_if[i]);
    if (this.cfg.enable_dev_mon[i] == 1) uvm_config_db#(USB_IF)::set(this, string_dev_mon_cfg, "usb_20_if", this.dev_usb_if[i]);
  end
end
foreach(this.hub_usb_if[i]) begin
  string_hub_cfg = $psprintf("hub_agent[%0d]", i);
  uvm_config_db#(USB_HUB_IF)::set(this, string_hub_cfg, "usb_hub_if", this.hub_usb_if[i]);
end

```

Figure 7-10 Build Phase: Instantiating USB HUB Model Agents

```

/**
 * Construct host_agent.
 */
host_agent = svt_usb_agent::type_id::create("host_agent", this);

/**
 * Construct dev_agent.
 */
foreach(cfg.dev_cfg[i]) begin
  string_dev_cfg = $psprintf("dev_agent[%0d]", i);
  dev_agent[i] = svt_usb_agent::type_id::create(string_dev_cfg, this);
end

/**
 * Construct hub_agent.
 */
foreach(cfg.hub_cfg[i]) begin
  string_hub_cfg = $psprintf("hub_agent[%0d]", i);
  hub_agent[i] = svt_usb_hub_agent::type_id::create(string_hub_cfg, this);
end

```


Figure 7-11 Retrieving USB HUB Interfaces

```
// Extract "usb_hub_if" from configuration database .
if(svt_config_vif_db#(USB_HUB_IF)::get(this, "", "usb_hub_if", hub_usb_if)) begin
  `svt_verbose(name, "usb_hub_if successfully recovered from configuration database ");
  usb_20_upstream_if = hub_usb_if.get_usb_20_upstream_if();
  usb_ss_upstream_if = hub_usb_if.get_usb_ss_upstream_if();
  if(usb_20_upstream_if == null) begin
    `svt_fatal(name, "no usb_20_upstream_if Interface exists in usb_hub_if ");
  end
  if(usb_ss_upstream_if == null) begin
    `svt_fatal(name, "no usb_ss_upstream_if Interface exists in usb_hub_if ");
  end
  for(int downstream_inst_num = 0; downstream_inst_num < `SVT_USB_NUM_OF_DOWNSTREAM_PORTS_INST; downstream_inst_num++) begin
    usb_20_downstream_if[downstream_inst_num] = hub_usb_if.get_usb_20_downstream_if(downstream_inst_num);
    usb_ss_downstream_if[downstream_inst_num] = hub_usb_if.get_usb_ss_downstream_if(downstream_inst_num);
    `svt_verbose(name, "usb_20_downstream_if[downstream_inst_num] successfully recovered from configuration database ");
    if(usb_20_downstream_if[downstream_inst_num] == null) begin
      `svt_fatal(name, $sprintf("no usb_20 Downstream Interface number [%0d] exists in usb_hub_if ", downstream_inst_num));
    end
    if(usb_ss_downstream_if[downstream_inst_num] == null) begin
      `svt_fatal(name, $sprintf("no usb_ss Downstream Interface number [%0d] exists in usb_hub_if ", downstream_inst_num));
    end
  end
end
else begin
  `svt_fatal(name, "no usb_hub_if Interface exists in env ");
end
```

Figure 7-12 Passing extracted interface to Upstream/Downstream Hub Agents

```
// Passing the recovered upstream interface to upstream agent's usb_20_if and usb_ss_if .
svt_config_int_db#(int)::set(this, "upstream_agent", "is_upstream", 1);
if(usb_20_upstream_if != null)
  svt_config_vif_db#(USB_IF)::set(this, "upstream_agent", "usb_20_if", usb_20_upstream_if);
if(usb_ss_upstream_if != null)
  svt_config_vif_db#(USB_IF)::set(this, "upstream_agent", "usb_ss_if", usb_ss_upstream_if);

// Passing the recovered downstream interface to downstream agent's usb_20_if and usb_ss_if .
for(int downstream_inst_num=0; downstream_inst_num < `SVT_USB_NUM_OF_DOWNSTREAM_PORTS_INST; downstream_inst_num++) begin
  string_downstream_agent = $sprintf("downstream_agent[%0d]", downstream_inst_num);
  svt_config_int_db#(int)::set(this, string_downstream_agent, "is_downstream", 1);
  if(usb_20_downstream_if[downstream_inst_num] != null) begin
    svt_config_vif_db#(USB_IF)::set(this, string_downstream_agent, "usb_20_if", usb_20_downstream_if[downstream_inst_num]);
    `svt_verbose(name, "usb_20_downstream_if[downstream_inst_num] successfully set in configuration database ");
  end
  else
  begin
    `svt_verbose(name, "usb_20_downstream_if[downstream_inst_num] FAILED TO set in configuration database ");
  end
  if(usb_ss_downstream_if[downstream_inst_num] != null) begin
    svt_config_vif_db#(USB_IF)::set(this, string_downstream_agent, "usb_ss_if", usb_ss_downstream_if[downstream_inst_num]);
    `svt_verbose(name, "usb_ss_downstream_if[downstream_inst_num] successfully set in configuration database ");
  end
  else
  begin
    `svt_verbose(name, "usb_ss_downstream_if[downstream_inst_num] FAILED TO set in configuration database ");
  end
end
```

❖ Testbench

```
.../usb_svt/test/sverilog/tb_usb_svt_uvm_basic_router_sys
```

**Note**

Default Macro passed in the vcs_build_options as SVT_USB_ENABLE_HUB_MODE

❖ Basic transfer Tests HS/SS

```
ts.basic_20_hub_enumeration_serial.sv
```

```
ts.basic_additional_20_utmi_host_mac_device_mac_hub_phy.sv
```

```
ts.basic_additional_20_utmi_host_phy_device_phy_hub_mac.sv  
  
ts.basic_ss_hub_enumeration_serial.sv  
ts.basic_additional_ss_pipe3_host_mac_device_mac_hub_phy.sv  
ts.basic_additional_ss_pipe3_host_phy_device_phy_hub_mac.sv
```

❖ Sequence which initiate in basic transfer tests

All tests call `usb_directed_transfers_sequence`

- a. Consists of `SET_ADDRESS` followed by `SET_CONFIGURATION` transfer for Hub and Devices connected.
- b. Followed by `BULK_IN_TRANSFER` and `BULK_OUT_TRANSFER`.

❖ Command for example to run 2 port.



Note

By default environment is in single port

```
gmake clean basic_ss_hub_enumeration_serial CMDLINE_VCSFLAGS='+define+  
SVT_USB_NUM_DEVICE_INST=2\'+define+SVT_USB_NUM_OF_DOWNSTREAM_PORTS_INST=2\  
+define+SVT_USB_DOWNSTREAM_PORT_2
```

8

VIP Tools

8.1 Using Native Protocol Analyzer for Debugging

8.1.1 Introduction

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.

8.1.2 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`
- ❖ Use any of the following options to enable Verdi PLI libraries:
`-P ${NOVAS_HOME}/share/PLI/VCS/${novas_platform} /novas.tab ${NOVAS_HOME}/share/PLI/VCS/${novas_platform}/pli.a`

OR

`-debug_access+all`

You can dump the transaction database either by setting the `pa_format_type` configuration variable or by passing the runtime switch as shown below:

Configuration Variable Setting:

Set `pa_xml_generation_enable` parameter of USB configuration class `svt_usb_configuration` to enable the generation of XML 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
```

8.1.3 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 and select the Protocol Analyzer option in the main window to invoke 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.

8.1.4 Documentation

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

`$DESIGNWARE_HOME/vip/tools/pa/latest/doc/protocol_analyzer_getting_started.pdf`

8.1.5 Limitations

Interactive support is available only for VCS.

9

Troubleshooting

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

- ❖ [“Using Trace Files for Debugging”](#) on page 185
- ❖ [“Enabling Tracing”](#) on page 187
- ❖ [“Setting Verbosity Levels”](#) on page 188
 - ◆ [“Method 1: To enable the specified severity in the VIP, DUT, and testbench”](#) on page 189
 - ◆ [“Method 2: To enable the specified severity to specific sub-classes of VIP”](#) on page 189
- ❖ [“Disabling Specific In-line Checking”](#) on page 191
- ❖ [“Disabling Specific In-line Checking”](#) on page 191

9.1 Using Trace Files for Debugging

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

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



Note

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

In a typical VIP agent configuration, where three components (protocol, link, and physical) are included with SS operation, the following trace files can be generated:

```
vip_agent.phys.SS.RX.data_trace
vip_agent.phys.SS.TX.data_trace
vip_agent.link.SS.RX.packet_trace
vip_agent.link.SS.TX.packet_trace
vip_agent.prot.SS.transaction_trace
```

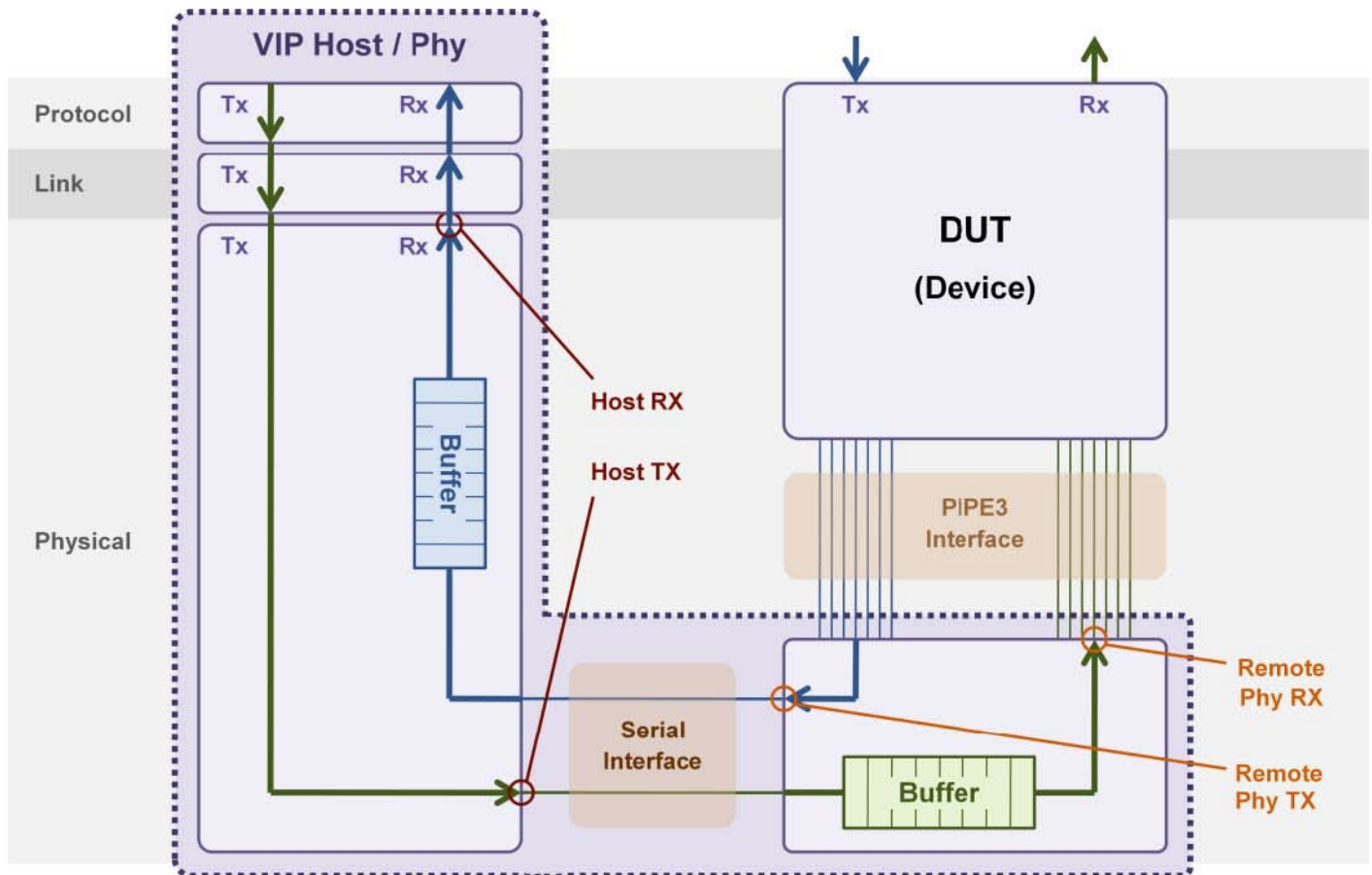
```
vip_agent.prot.SS.transfer_trace
```

In a VIP agent configuration, where a fourth component ('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_agent.remote_phys.SS.RX.data_trace
vip_agent.remote_phys.SS.TX.data_trace
```

Figure 9-1 shows the output ports of each component, where objects are captured for corresponding trace files content.

Figure 9-1 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

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

where:

agent instance name: is taken from the instance name of the corresponding VIP agent.

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

<agent instance name>.link.<speed>.<direction>.packet_trace

where:

agent instance name: is taken from the instance name of the corresponding VIP agent.

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

<agent instance name>.prot.<speed>.transaction_trace

where:

agent instance name: is taken from the instance name of the corresponding VIP agent.

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

Transfer Trace Files

<agent instance name>.prot.<speed>.transfer_trace

where:

agent instance name: is taken from the instance name of the corresponding VIP agent.

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

For example, if the instance name of the VIP agent is “vip_agent”. then the simulation of this environment produces the files listed in [Table 9-1](#):

Table 9-1 Generated Trace Files

| Trace File Type | Trace File Name |
|------------------------|--|
| Data trace files | vip_agent.phys.SS.RX.data_trace vip_agent.phys.SS.TX.data_trace vip_agent.remote_phys.SS.RX.data_trace vip_agent.remote_phys.SS.TX.data_trace |
| Packet trace file | vip_agent.link.SS.RX.packet_trace vip_agent.link.SS.TX.packet_trace |
| Transfer trace file | vip_agent.prot.SS.transfer_trace |
| Transaction trace file | vip_agent.prot.SS.transaction_trace |

9.2 Enabling Tracing

Tracing is enabled or disabled through the “enable_phys_tracing” variable that is defined in the “svt_usb_agent_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 agent. 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_agent_configuration class.

Code Sample 1:

```
vip_cfg.enable_phys_tracing = 1;
vip_cfg.enable_link_tracing = 1;
vip_cfg.enable_prot_tracing = 1;
```

Code Sample 2:

```
// enable nested tracing: packet and corresponding data
vip_cfg.enable_link_tracing = 2;
```

Code Sample 3:

```
// enable nested tracing: transfer, corresponding transactions and corresponding
packets
vip_cfg.enable_prot_tracing = 3;
```

Code Sample 4:

```
// enable nested tracing: transfer, corresponding transactions,
// corresponding packet and corresponding data
vip_cfg.enable_prot_tracing = 4;
```

In this last code sample, <vip_agent_instance>.prot.SS.TX.transfer_trace and <vip_agent_instance>.prot.SS.RX.transfer_trace files are created with nested transfer, transaction, data, and packet information.



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-UVM test examples provided along with the VIP.

9.3 Setting Verbosity Levels

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

9.3.1 Setting Verbosity in the Testbench

To set the verbosity level in the testbench, use the UVM-specified log-levels in the code. The components are extended from uvm_report_object. You can use the following uvm_report_object method to change the verbosity for the host (for example):

```
vip_usb_host.set_report_verbosity_level(<level>);
```

Where the following define all of the possible <levels>:

- ❖ UVM_NONE. Report is always printed. Verbosity level setting can not disable it.
- ❖ UVM_LOW. Report is issued if configured verbosity is set to UVM_LOW or above.
- ❖ UVM_MEDIUM. Report is issued if configured verbosity is set to UVM_MEDIUM or above.

- ❖ UVM_HIGH. Report is issued if configured verbosity is set to UVM_HIGH or above.
- ❖ UVM_FULL. Report is issued if configured verbosity is set to UVM_FULL or above.

9.3.2 Setting Verbosity During Run Time

To set the verbosity level during run-time, you can use one of the following methods:

- ❖ “Method 1: To enable the specified severity in the VIP, DUT, and testbench”
- ❖ “Method 2: To enable the specified severity to specific sub-classes of VIP”

9.3.2.1 Method 1: To enable the specified severity in the VIP, DUT, and testbench

Use Example: VCS

```
vcs <other run time options> +uvm_log_default=UVM_HIGH
```

9.3.2.2 Method 2: To enable the specified severity to specific sub-classes of VIP

Note: only applies to 'types' (e.g., svt_usb_physical). [Table 9-2](#) lists the valid VIP sub-units that you can use.

Instance verbosity is supported by UVM directly:

```
+uvm_set_verbosity=component_name,id,verbosity,phase_name,optional_time
```

Use Example: VCS

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

The levels can be:

```
`define SVT_FATAL_VERBOSITY    UVM_NONE  
`define SVT_ERROR_VERBOSITY    UVM_NONE  
`define SVT_WARNING_VERBOSITY  UVM_NONE  
`define SVT_NORMAL_VERBOSITY   UVM_LOW  
`define SVT_TRACE_VERBOSITY    UVM_MEDIUM  
`define SVT_DEBUG_VERBOSITY    UVM_HIGH  
`define SVT_VERBOSE_VERBOSITY  UVM_FULL
```

Table 9-2 Valid VIP Sub-Units

| Layer | Sub-unit or Component |
|--------------------------|---|
| Entire agent | svt_usb_agent |
| Protocol Layer - SS: | <ul style="list-style-type: none"> 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_imp_processor svt_usb_protocol_ss_itp_processor |
| Protocol Layer - USB 2.0 | <ul style="list-style-type: none"> 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: | <ul style="list-style-type: none"> 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 | <ul style="list-style-type: none"> 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 |

9.4 Disabling Specific In-line Checking

By default, all VIP error checking is enabled by default. Refer to `chk_cov_mgr` in the [Class Reference HTML](#) for valid checks defined.

The following code snippet illustrates the enabled default VIP error checking:

```
svt_err_check_stats temp_check;  
temp_check = usb_host.link.chk_cov_mgr.find("skp_symbol_ratio_check");  
temp_check.set_is_enabled(0);
```




A

Reporting Problems

A.1 Introduction

This chapter outlines the process for working through and reporting SVT component 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.

A.2 Initial Customer Information

If you think an SVT component or supporting object is not working properly, you will need to contact a Synopsys Support Center. Follow these steps:

1. Before you contact technical support, be prepared to provide the following:
 - ◆ A description of the issue under investigation (for information, see [page 194](#))
 - ◆ A description of your verification environment (for information, see [page 194](#))
2. Create a value change dump (VCD) file (for information, see [page 194](#))
3. Generate a log file for the simulation (for information, see [page 194](#))

Providing this information will help ensure accurate diagnosis of the problem.

If the requested information is not sufficient to determine the cause of your issue, the Synopsys Support Center may request a simple testbench demonstrating the issue. This is the most effective way to debug issues, but if an example cannot be provided, Synopsys may request additional information that could include regenerating the log file using different settings.

If your testbench initializes the random seed (see 'srandom()' in the VCS manual) in the host simulator, then a seed that can be used to demonstrate the problem must be included in the testbench or provided as information for executing the testbench.

A.2.1 Describing Your Issue

Compose a detailed 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.

A.2.2 Describing Your Verification Environment Description

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)

A.2.3 Creating a Waveform File

- ❖ For a Verilog or SystemVerilog simulation, generate a VCD waveform file by specifying the following in your top level testbench.

```
initial begin
    $dumpvars;
end
```

When the simulation is completed, a VCD file named *verilog.dump* will be present in the runtime directory.

A.2.4 Creating a Log File

Log file generation relies on basic SVT logging capabilities. The only specific requirement for issue submission is that the log be verbose without being too verbose. There are basically three steps to accomplishing this:

1. Set the verbosity for logging to VERBOSE.
2. Make sure that the **display_kind** display field is set to -1 so that SVT data objects use the less verbose *RELEVANT* display algorithms (as opposed to *COMPLETE*).
3. Make sure that the **max_array_elem_display** display field is set to 16 so that SVT data objects limit their display of array elements.

The **display_kind** and **max_array_elem** data fields are static fields defined on the **svt_data** object. Once they are set, they apply to all SVT data instances. This includes all transaction and configuration instances.

We suggest that you use one of the configuration instances to set these static display fields because configuration instances are readily available in the environment or via the component. Or, any SVT data instance can be used to set this value across all SVT data objects.

The details for performing these steps can be found below, according to your testbench domain:

- ❖ [“SystemVerilog Testbench Users”](#)
- ❖ [“Verilog Testbench Users”](#)

Note that, by default, the log is simply sent to the shell transcript. You must also make sure that the contents of the transcript are written to a file. This file is what is then transmitted to Synopsys.

A.2.4.1 SystemVerilog Testbench Users

vmm_log::set_verbosity is used to set the simulation verbosity in SystemVerilog. The following line will set the verbosity to verbose for all log files in the simulation, regardless of which log instance is used to make the call.

```
log.set_verbosity(vmm_log::VERBOSE_SEV, "/./", "/./", 1);
```

While it is possible to localize the verbosity setting by choosing specific log files for increased verbosity, setting the verbosity for the entire simulation is preferable for issue generation in a SystemVerilog environment. This is because its better to have “too much” in the initial log file, than to have “too little”.

As suggested above, **display_kind** and **max_array_elem** should be set through one of the configuration instances. The following lines demonstrate how to do this for a configuration instance named 'cfg'.

```
cfg.display_kind = -1;  
cfg.max_array_elem = 16;
```

All of the above should be done early in the test execution, preferably within the UVM environment's build phase.

A.2.4.2 Verilog Testbench Users

The **log_msg_val** and **log_set_verbosity** tasks on the component are used to set the simulation verbosity in a Verilog testbench using the SVT command interface. The following lines will set the verbosity to verbose for all messaging done by the 'xactor_1' component instance.

```
xactor_1.log_msg_val(msg_valid, verbosity_val, "severity", "VERBOSE_SEV");  
xactor_1.log_set_verbosity(verbosity_val);
```

Note that the verbosity must be updated for all SVT components which are involved the issue.

As suggested above, **display_kind** and **max_array_elem** should be set through one of the component configurations. The following lines demonstrate how to do this for a configuration represented by 'cfghandle', associated with xactor_1.

```
xactor_1.set_data_prop(is_valid, cfghandle, "display_kind", -1, 0);  
xactor_1.set_data_prop(is_valid, cfghandle, "max_array_elem", 16, 0);
```

Unlike the verbosity, the **display_kind** and **max_array_elem** values only need to be set once, via any one of the SVT components, for the entire Verilog simulation.

All of the above should be done in the initial phases of the test, when the component is initialized.

A.2.5 Identifying an Instance

The generated log file may have many components as well as messages from these many components. Issue resolution relies heavily on reviewing the log file and analyzing the messages from the individual components. Based on this, you must insure that all components in the design are uniquely named. You must also identify which component(s) are failing when you submit your issue report.

The component instance name is defined at component creation by the user testbench. The next few sections describe where the name derives from in SystemVerilog and HDL testbench situations.

A.2.5.1 SystemVerilog Testbench Users

In SystemVerilog the instance name comes from the 'inst' value on the configuration. For example:

```
mstr_1_cfg.inst = "master_1";  
mstr_1 = new(mstr_1_port, mstr_1_cfg, , , , , );
```

In this situation the instance name is 'master_1', as can be seen in one of the resulting messages.

```
Trace [Intern] on usb_svt_physical(master_1) at 7625 ns:
Reset
```

A.2.5.2 Verilog Testbench Users

In Verilog the instance name is name provided in the initial **set_data_prop** call. For example:

```
master_1.set_data_prop(
    is_valid, dummy, "inst", "master_1", `MASTER_1_STREAM);
```

In this situation the instance name is 'master_1', as can be seen in one of the resulting messages.

```
Trace [Intern] on usb_svt_physical(master_1) at 7625 ns:
Reset
```

A.2.6 Impact of Generating Verbose Logs

When verbose logging is enabled, the component prints debug information to the log and to the transcript. As a result, simulation performance is degraded. The length of the simulation and the amount of simulation activity will impact the size of the log file. To make it easier to transmit your information and analyze your issue, try to minimize the size of the log file by logging only the relevant portion of the simulation.

When the following are set, the corresponding layer's trace and report file are generated. For details refer to the `svt_usb_agent_configuration` page in the Class Reference HTML.

- ❖ `svt_usb_agent_configuration::enable_phys_tracing`
- ❖ `svt_usb_agent_configuration::enable_link_tracing`
- ❖ `svt_usb_agent_configuration::enable_protocol_tracing`
- ❖ `svt_usb_agent_configuration::enable_phys_reporting`
- ❖ `svt_usb_agent_configuration::enable_link_reporting`
- ❖ `svt_usb_agent_configuration::enable_protocol_reporting`

A.3 Debug Automation

This section outlines the process for working through and reporting problems. It shows you how to use Debug Automation to enable all the debug capabilities of any SVT-based VIP. In addition, the VIP provides a case submittal tool to help you pack and send all pertinent debug information to Synopsys Support.

Every Synopsys model contains a feature called “debug automation”. Debug Automation allows you to enable all relevant debug information. 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.

- ❖ Generates a file named *svt_debug.out* which contains information about the debug feature itself, along with information about the environment that the VIP is operating in (such as package timeunits and VIP versions). The file is located in the directory where you invoke the simulator.

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* steps. You can use the generated debug data for your own debugging, or as debug data you would send to Synopsys support.

A.3.1 Enabling and Specifying Debug Automation Features

Debug Automation is enabled through the use of a run-time plusarg named *+svt_debug_opts*. This 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 below.

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_string>, 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_string | Identifies the VIP instance to apply the debug automation features to. 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 SVT VIP instances. |
| type | Identifies an SVT class type to apply the debug automation features to. 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. |
| 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 in. 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 in. 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 have been enabled for debug will have their messages routed to a file named <i>svt_debug.transcript</i> . |

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, Trace files of USB.

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



Note

The `gmake clean` command helps to remove old files created.

A.3.2 Debug Automation Outputs

The Automated Debug feature generates a file named *svt_debug.out*. 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

A.3.3 Sending Debug Information to Synopsys

To help you debug testing issues, follow the instructions below 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. Create a value change dump (VCD) file. For a Verilog or SystemVerilog simulation, generate a VCD waveform file by specifying the following in your top level testbench.

```
initial begin
    $dumpvars;
end
```

When the simulation is completed, a VCD file named *verilog.dump* will be present in the runtime directory.

4. 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 "<username>.<uniqid>.svd" file is a TAR type file that contains all pertinent transcripts, trace files, dump files and/or debug output files that exist in the current directory at the time of tool invocation.

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

5. You will be prompted by the case submittal tool with the option to include additional files within the SVD file.
6. The case submittal tool will display options on how to send the file to Synopsys.
7. To unpack the *.svd file, use the following command:

```
tar zxf <svd_file>
```