Verification Continuum™

VC Verification IP
USB
OVM User Guide

Version R-2021.03, March 2021

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

Preface 7
Web Resources 7
Customer Support 7
Synopsys Statement on Inclusivity and Diversity 7

Chapter 1
Introduction 9
1.1 Product Overview 9
1.2 USB Feature Support 10
1.2.1 Protocol Layer Features 11
1.2.2 Link Layer Features 11
1.2.3 Physical Layer Features 12
1.3 OVM Support Provided by USB VIP 12

Chapter 2
Installation and Setup 15
2.1 Verifying the Hardware Requirements 15
2.2 Verifying Software Requirements 15
2.3 Platform/OS and Simulator Software 15
2.4 Synopsys Common Licensing (SCL) Software 16
2.5 Other Third Party Software 16
2.6 Preparing for Installation 16
2.7 Downloading and Installing 16
2.8 Set Up a New VIP 17
2.9 Installing and Setting Up More than One VIP Protocol Suite 17
2.10 Updating an Existing Model 18
2.11 Include and Import Model Files into Your Testbench 18
2.12 Compile and Run Time Options 19
2.13 What's Next? 20
2.14 Licensing Information 20
2.14.1 If Licensing Fails 20
2.14.2 License Polling 20
2.14.3 Simulation License Suspension 21
2.15 Environment Variable and Path Settings 21
2.15.1 Simulator-Specific Settings 21
2.16 Determining Your Model Version 21
2.17 Integrating a Synopsys VIP into Your Testbench 21
2.17.1 Creating a Testbench Design Directory 22
2.17.2 The dw_vip_setup Utility 24
2.17.3 Using DesignWare Verification IP in Your Testbench 26

2.17.4 Running the Example With +incdir+ 26

Chapter 3
General Concepts 29
3.1 USB VIP in a OVM Environment 29
3.1.1 Base Classes 30
3.1.2 OVM Components 30
3.1.3 USB VIP Objects 31
3.1.4 Interfaces and Modports 37
3.1.5 Constraints 38
3.1.6 Factories 40
3.1.7 Messages 41

Chapter 4
The USB Agent Overview 43
4.0.1 OVM Component Stack 43
4.0.2 Stimulus Objects 46
4.0.3 Reporting and Tracking Objects 46
4.0.4 OVM Events 47

Chapter 5
USB Verification OVM IP Stack Components 49
5.1 Protocol Component 49
5.1.1 Protocol Layer Feature Support 49
5.1.2 Protocol Component Ports 50
5.1.3 Data Objects 50
5.1.4 Protocol Component Modes 51
5.1.5 Data Transformation Objects 51
5.1.6 Protocol Component Status 55
5.1.7 Protocol Component SuperSpeed Link Callback, Factory, and Event Flows 56
5.1.8 Protocol Component 2.0 Link Callback, Factory, and OVM Event Flows 69
5.2 Link Component 79
5.2.1 Link Layer Feature Support 79
5.2.2 Link Component Ports 79
5.2.3 Data Objects 80
5.2.4 Data Transformation Objects 81
5.2.5 Link Component SuperSpeed Link Callback, Factory, and OVM Event Flows 83
5.2.6 SuperSpeed Packet Chronology 84
5.2.7 Related Topics About Link Component 102
5.3 Physical Component 102
5.3.1 Physical Layer Feature Support 103
5.3.2 Data Flow Support 103
5.3.3 Interface Options 106
5.3.4 Interface File Features 107
5.3.5 Information Transformation Objects 107
5.3.6 Exception Support 108
5.3.7 OVM Event Support 109
5.3.8 Physical Component Callbacks 109
5.3.9 Related Topics About Physical Component 110

Chapter 6 . . . . .

Using the USB Verification IP 111
6.1 Configuring VIP Using coreConsultant 111
6.2 Creating Transactions Using OVM Sequencers 112
6.3 SuperSpeed Low Power Entry Support 113
6.3.1 Automatic Low-Power Entry Attempts 113
6.3.2 Automatic Low-Power Entry for Upstream Ports 114
6.3.3 Testbench-Initiated Low-Power Entry Attempts 114
6.4 Implementing Functional Coverage 115
6.4.1 Default Functional Coverage 115
6.4.2 Covergroup Organization 116
6.4.3 Range Bins 117
6.4.4 Default Functional Coverage Class Hierarchy 118
6.4.5 Coverage Callback Classes 119
6.4.6 Using Functional Coverage 120
6.4.7 Using the High-Level Verification Plans 120
6.5 Executing Aligned Transfers 121
6.5.1 VIP Acting as a Host 121
6.5.2 VIP Acting as a Device 121
6.6 SuperSpeed Serial LTSSM Flow (SS.Disabled to U0) 122
6.6.1 LTSSM state is Rx.Detect.Reset 123
6.6.2 LTSSM state is Rx.Detect.Active 123
6.6.3 LTSSM state is Polling.LFPS 123
6.6.4 LTSSM state is Polling.RxEQ 124
6.6.5 LTSSM state is Polling.Active 124
6.6.6 LTSSM state is Polling.Configuration 124
6.6.7 LTSSM state is Polling.Idle 124
6.7 USB 2.0 OTG Support 124
6.7.1 OTG Interface Signals 125
6.7.2 Session Request Protocol 127
6.7.3 Role Swapping Using the HNP Protocol 130
6.7.4 Attach Detection Protocol 139
6.8 HSIC Overview 143
6.8.1 Supported HSIC Features 143
6.8.2 Unsupported HSIC Features 143
6.8.3 Configuration Parameters 143
6.8.4 Transactions 144
6.8.5 Exceptions 145
6.8.6 HSIC Interface 146
6.8.7 HSIC Signal Interface 146
6.8.8 Callbacks 146
6.8.9 Notifications 146
6.8.10 Factories 147
6.8.11 Shared Status 147
6.8.12 Usage 147
6.9 Using Test Mode 158
6.9.1 Entering Test Mode 158
6.9.2 Verifying TEST_PACKET 159
6.9.3 Verifying TEST_J 160
6.9.4 Verifying TEST_K 161
6.9.5 Verifying TEST_SE0_NAK 161

6.9.6 Exiting test_mode on Downstream Facing Ports 162
6.9.7 Test Mode Notifications 163
6.9.8 Test Mode Configuration Members 163
6.10 SystemVerilog OVM Example Testbenches 163

Chapter 7
Verification Topologies 165
7.1 USB VIP Host and DUT Device Controller 165
7.2 USB VIP Device and DUT Host Controller 166
7.3 USB VIP Host and DUT Device PHY 167
7.4 USB VIP Host and DUT Device 168
7.5 USB VIP Device and DUT Host 169
7.6 USB VIP Device and DUT Host – Concurrent SS and 2.0 Traffic 170
7.7 USB VIP Device with Hub Emulation and DUT Host – Concurrent Serial Interface Traffic 171

Chapter 8
VIP Tools 173
8.1 Using Native Protocol Analyzer for Debugging 173
8.1.1 Prerequisites 173
8.1.2 Invoking Protocol Analyzer 174
8.1.3 Documentation 174
8.1.4 Limitations 174

Chapter 9
Troubleshooting 175
9.1 Using Trace Files for Debugging 175
9.2 Enabling Tracing 177
9.3 Setting Verbosity Levels 178
9.3.1 Setting Verbosity in the Testbench 178
9.3.2 Setting Verbosity During Run Time 179
9.4 Disabling Specific In-line Checking 181

Appendix A
Reporting Problems 183
A.1 Debug Automation 183
A.2 Enabling and Specifying Debug Automation Features 183
A.3 Debug Automation Outputs 185
A.4 FSDB File Generation 186
A.4.1 VCS 186
A.4.2 Questa 186
A.4.3 Incisive 186
A.5 Initial Customer Information 186
A.6 Sending Debug Information to Synopsys 186
A.7 Limitations 187

Preface

About This Manual

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

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

Web Resources

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

Customer Support

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

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

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

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

Synopsys Statement on Inclusivity and Diversity

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

1 Introduction

The VC VIP for Verification IP supports verification of SoC designs that include interfaces implementing the Universal Serial Bus 3.0 Specification. This document describes the use of this VIP in testbenches that comply with the Open Verification Methodology (OVM). 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 OVM.

See also:

- *Universal Serial Bus 3.0 Specificati*on, 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_ovm_class_reference/html/index.html

For a complete list of examples included with the Verification IP, see SystemVerilog OVM Example Testbenches.

1.1 Product Overview

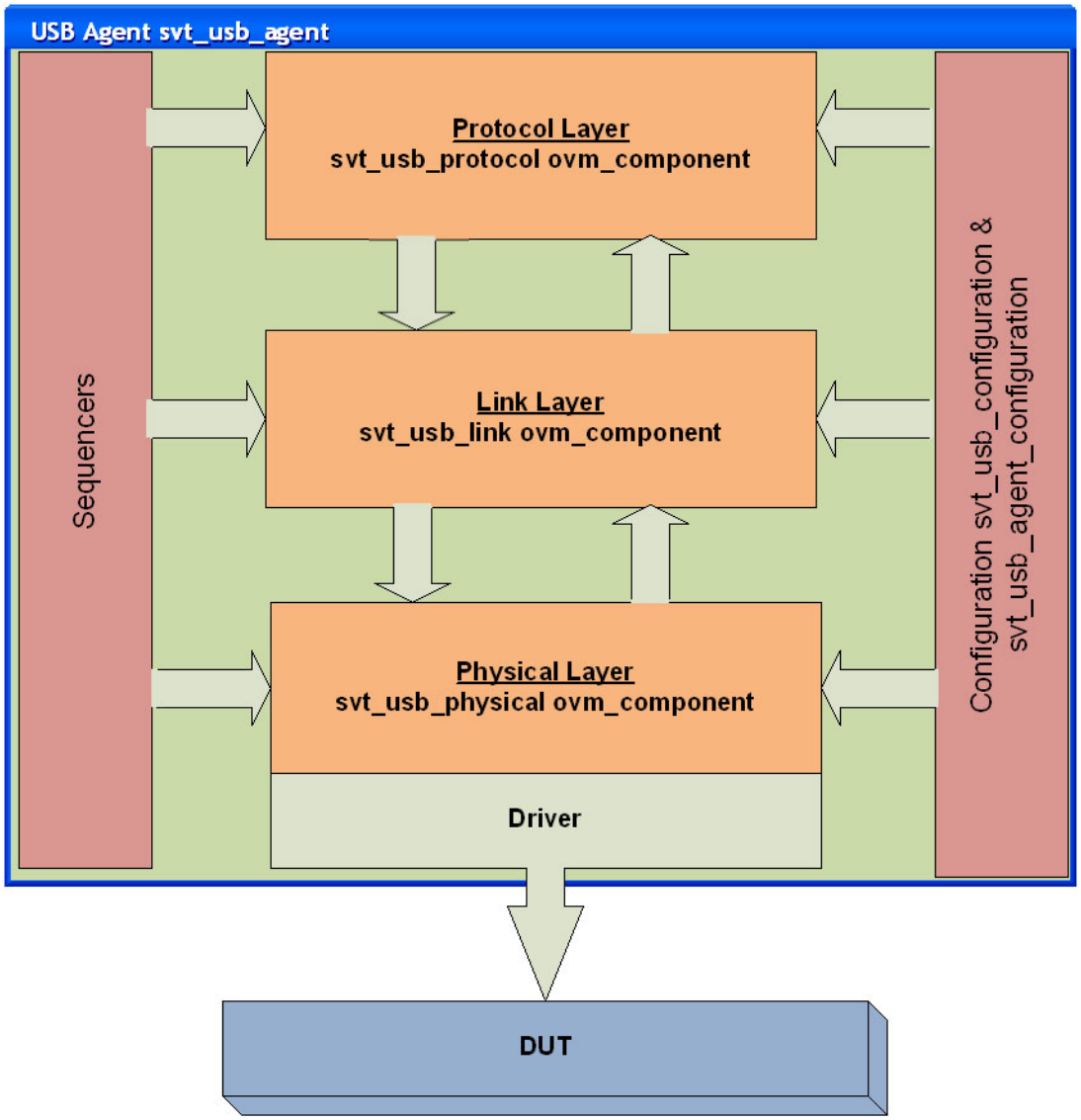
The USB VIP is a suite of OVM-based verification components that are compatible for use with SystemVerilog-Compliant testbenches. The USB VIP suite simulates transfers, transactions, and packets through OVM agents as defined by the USB and USB 2.0 specifications. The VIP defines one agent with three OVM components to implement the USB Specification. The components are

- svt_usb_physical – supports the physical layer of the USB protocol
- svt_usb_link – supports the link layer of the USB protocol

- svt_usb_protocol – supports the protocol layer of the USB protocol.

Figure 1-1 shows the relationship of the three OVM components within the USB agent.

Figure 1-1 USB VIP OVM Agent Architecture



1.2 USB Feature Support

The USB VIP supports the following features:

- SuperSpeed (SS), High-Speed (HS), Full-Speed (FS) and Low-Speed (LS)
- Host, Device, and Hub verification
- PHY interface support

- Power Management: USB and USB 2.0 implementations
- SS PIPE3, SS Serial, and 2.0 Serial interfaces
- USB 2.0 OTG and Embedded Host support for serial interfaces
- USB Layer Stack Feature Support

1.2.1 Protocol Layer Features

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

USB VIP supports the following protocol layer functions:

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

1.2.2 Link Layer Features

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

- USB General Support
  - Host and Device support
  - Speed fallback from SS to FS or HS
  - Speed fall-forward from FS or HS to SS
- USB Support
  - LTSSM – Direct LTSSM state change support
  - SS power management

- ITP support
- Link advertisements
- Link command and packet processing and response

- USB 2.0 Support
  - LPM
  - Suspend and Resume
  - USB2 power management

1.2.3 Physical Layer Features

USB VIP supports the following physical layer functions:

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

1.3 OVM Support Provided by USB VIP

The following is a summary of the supported OVM 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 ovm_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.

2
Installation and Setup

This section leads you through installing and setting up the Synopsys USB VIP. When you complete this checklist, the provided example testbench will be operational and the Synopsys USB VIP will be ready to use.

The quick start consists of the following major steps:

- "Introduction"
- "Licensing Information"
- "Environment Variable and Path Settings"
- "Determining Your Model Version"
- "Integrating a Synopsys VIP into Your Testbench"

If you encounter any problems with installing the Synopsys USB VIP, see Customer Support.

2.1 Verifying the Hardware Requirements

The USB 3.0 Verification IP requires a Solaris or Linux workstation configured as follows:

- 400 MB available disk space for installation
- 1 GB available swap space
- 1 GB RAM (physical memory) recommended

2.2 Verifying Software Requirements

The Synopsys USB VIP is qualified for use with certain versions of platforms and simulators. This section lists software that the Synopsys USB VIP requires.

2.3 Platform/OS and Simulator Software

- **Platform/OS and VCS**: You need versions of your platform/OS and simulator that have been qualified for use. To see which platform/OS and simulator versions are qualified for use with the USB 3.0 VIP, check the support matrix for "SVT-based" VIP in the following document:

  Support Matrix for SVT-Based DesignWare USB 3.0 VIP is in USB Release Notes.

2.4 Synopsys Common Licensing (SCL) Software

- The SCL software provides the licensing function for the Synopsys USB VIP. Acquiring the SCL software is covered here in the installation instructions in Licensing Information.

2.5 Other Third Party Software

- **Adobe Acrobat**: Synopsys USB VIP documents are available in Acrobat PDF files. You can get Adobe Acrobat Reader for free from http://www.adobe.com.
- **HTML browser:** Synopsys USB VIP includes class reference documentation in HTML. The following browser/platform combinations are supported:
  - Microsoft Internet Explorer 6.0 or later (Windows)
  - Firefox 1.0 or later (Windows and Linux)
  - Netscape 7.x (Windows and Linux)

2.6 Preparing for Installation

1. Set `DESIGNWARE_HOME` to the following absolute path where DesignWare USB 3.0 VIP is to be installed:

   ```
   setenv DESIGNWARE_HOME absolute_path_to_designware_home
   ```

2. Ensure that your environment and PATH variables are set correctly, including:
   - `DESIGNWARE_HOME/bin` – The absolute path as described in the previous step.
   - `LM_LICENSE_FILE` – The absolute path to a file that contains the license keys for your third-party tools. Also, include the absolute path to the third party executable in your PATH variable.

     ```
     % setenv LM_LICENSE_FILE <my_license_file|port@host>
     ```

   - `SNPSLMD_LICENSE_FILE` – The absolute path to a file that contains the license keys for Vera and Synopsys Common Licensing software or the *port@host* reference to this file.

     ```
     % setenv SNPSLMD_LICENSE_FILE <my_Synopsys_license_file|port@host>
     ```

   - `DW_LICENSE_FILE` – The absolute path to a file that contains the license keys for VIP product software or the port@host reference to this file.

     ```
     % setenv DW_LICENSE_FILE <my_VIP_license_file|port@host>
     ```

2.7 Downloading and Installing

To receive a new version of the Synopsys USB VIP

1. Enter a call through SolvNetPlus.
   - Go to https://solvnetplus.synopsys.com and open a case.

     Enter the information according to your environment and your issue.
2. Synopsys indicates the FTP location and access instructions for requested run file.
3. Execute the run file:

   % <*vip run file name*>.run

   Answer the prompts that the .run script generates until the install is complete.

2.8 Set Up a New VIP

After you have installed the VIP, you must set up the VIP for project and testbench use. All Discovery VIP suites contain various components such as transceivers, masters, slaves, and monitors depending on the protocol. The setup process gathers together all the required component files you need to incorporate into your testbench and simulation runs.

You have the choice to set up all of them, or only specific ones. For example, the USB 3.0VIP contains the following components.

In OVM the USB 3.0 VIP contains the following component:

- `usb_agent_svt`: This is the name used for the entire set of sub-models.
- `usb_protocol_svt`: Supports protocol layer of USB protocol
- `usb_link_svt`: Supports link layer of USB protocol
- `usb_physical_svt`: Supports physical layer of USB protocol
- `usb_ssic_physical_svt`: Supports SSIC physical layer of USB protocol

You can set up either an individual component, or the entire set of components within one protocol suite. Use the Synopsys provided tool called dw_vip_setup for these tasks. It resides in `$DESIGNWARE_HOME/bin`. To get help on dw_vip_setup, invoke the following:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup --help
```

The following command adds an model <model_svt> to the directory `design_dir`.

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/design_dir -add usb_subenv_svt
-svlog
```

This command sets up all the required files in `/tmp/design_dir`. The utility dw_vip_setup creates three directories under design_dir which contain all the necessary model files. Files for every VIP are included in these three directories.

- **examples**. Each VIP includes example testbenches. The dw_vip_setup utility adds them in this directory, along with a script for simulation. If an example testbench is specified on the command line, this directory contains all files required for model, suite, and system testbenches.
- **include**. Language-specific include files that contain critical information for Synopsys models. This directory "include/sverilog" is specified in simulator commands to locate model files.
- **src**. Synopsys-specific include files This directory "src/sverilog/vcs" must be included in the simulator command to locate model files.

Note that some components are “top level” and will setup the entire suite. You have the choice to set up the entire suite, or just one component such as a monitor.

Attention There ***must*** be only one design_dir installation per simulation, regardless of the number of Synopsys Verification and Implementation IPs you have in your project. Do create this directory in $DESIGNWARE_HOME.

2.9 Installing and Setting Up More than One VIP Protocol Suite

All VIPs for a particular project must be set up in a single common directory once you execute the *.run file. You may have different projects. In this case, the projects can use their own VIP setup directory. However, all the VIPS used by that specific project must reside in a common directory.

The examples in this chapter call that directory `design_dir`, but you can use any name. In this example, assume you have the AXI suite set up in the `design_dir` directory. In addition to the AXI VIP, you require the Ethernet and USB VIP suites.

First, follow the previous instructions on downloading and installing the Ethernet VIP and USB suites.

Once installed, the Ethernet and USB suites must be set up in and located in the same `design_dir` location as AMBA. Use the following commands:

```
// First install AXI.
%unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path  /tmp/design_dir
-add axi_system_env_svt-svlog

//Add Ethernet to the same design_dir as AXI.
%unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path  /tmp/design_dir
-add ethernet_system_env_svt -svlog

// Add USB to he same design_dir as AMBA and Ethernet
%unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path  /tmp/design_dir
-add usb_agent_svt -svlog
```

To specify other model names, consult the VIP docum`% setenv SNPSLMD_LICENSE_FILE <my_Synopsys_license_file|port@host>`

- `DW_LICENSE_FILE` – The absolute path to a file that contains the license keys for VIP product software or the port@host reference to this file.

  `% setenv DW_LICENSE_FILE <my_VIP_license_file|port@host>`

entation.

By default, all of the VIPs use the latest installed version of SVT. Synopsys maintains backward compatibility with previous versions of SVT. As a result, you may mix and match models using previous versions of SVT.

2.10 Updating an Existing Model

To add an update an existing model, do the following:

1. Install the model to the same location as your other VIPs by setting the $DESIGNWARE_HOME environment variable.
2. Issue the following command using design_dir as the location for your project directory.

   ```
   %unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path  /tmp/design_dir
   -add  <model_name>_svt -svlog
   ```

3. You can also update your design_dir by specifying the version number of the model.

   `%unix> dw_vip_setup -path design_dir -add <model_name>_svt -v 3.50a model_vmt -v 3.50a`

2.11 Include and Import Model Files into Your Testbench

After you set up the models, you must include and import various files into your top testbench files to use the VIP. Following is a code list of the includes and imports for USB 3.0:

```
/* include ovm package before VIP includes, If not included elsewhere*/
`include "ovm_pkg.sv"
/* include USB VIP interface */
```

```
`include "svt_usb_if.svi
/** Include the USB SVT OVM package */
`include "svt_usb.ovm.pkg"
/** Import OVM Package */
import ovm_pkg::*;
`include "ovm_macros.svh"
/** Import the SVT OVM Package */
import svt_ovm_pkg::*;
/** Import the USB VIP */
import svt_usb_ovm_pkg::*;
```

These files contain both optional and required switches.

For USB 3.0, following are the contents of each file, listing optional and required switches:

- vcs_build_options
- Required: +define+OVM_PACKER_MAX_BYTES=16384
- Optional: -timescale=1ps/1ps
- Required: +define+SVT_<model>_INCLUDE_USER_DEFINES
- vcs_run_options
- Required: +OVM_TESTNAME=$scenario

Note
Scenario is the OVM test name you pass to VCS.

2.12 Compile and Run Time Options

Every Synopsys provided example has ASCII files containing compile and run time options. The examples for the model are located in:

$DESIGNWARE_HOME/vip/svt/<*model*>/latest/examples/sverilog/example_testbench_name

The files containing the options are:

- sim_build_options (also vcs_build_options)
- sim_run_options (also vcs_run_options)

These files contain both optional and required switches. For <model>, following are the contents of each file, listing optional and required switches:

vcs_build_options are as follows

- Required: +define+UVM_PACKER_MAX_BYTES=16384
- Required: +define+UVM_DISABLE_AUTO_ITEM_RECORDING
- Optional: -timescale=1ps/1ps
- Required: +define+SVT_<model>_INCLUDE_USER_DEFINES

vcs_run_options

- Required: +UVM_TESTNAME=$scenario

Note
Scenario is the UVM test name you pass to VCS.

2.13 What's Next?

The remainder of this chapter describes the details of the different steps you performed during installation and setup, and consists of the following sections:

- Licensing Information
- Environment Variable and Path Settings
- Determining Your Model Version
- Integrating a Synopsys VIP into Your Testbench

2.14 Licensing Information

The Synopsys USB VIP uses the Synopsys Common Licensing (SCL) software to control its usage. You can find general SCL information at:

http://www.synopsys.com/keys

The Synopsys USB VIP uses a licensing mechanism that is enabled by the following license feature:

- Synopsys-USB3-VIP

Only one license is consumed per simulation session, no matter how many Synopsys VIP models are instantiated in the design.

The licensing key must reside in files that are indicated by specific environment variables. For information about setting these licensing environment variables, refer to Environment Variable and Path Settings.

2.14.1 If Licensing Fails

By default, simulations exit with an error when a Synopsys VIP license cannot be secured. Alternatively, the `DW_NOAUTH_CONTINUE` environment variable can be set to allow simulations to continue when one or more VIP models fail to authorize. Unauthorized Synopsys VIP models essentially become disabled when `DW_NOAUTH_CONTINUE` is set to any value.

```
% setenv DW_NOAUTH_CONTINUE
```

Also, some simulation environments allow license polling, which pauses the simulation until a license is available. License polling is described next.

If you encounter problems with licensing, see Customer Support.

2.14.2 License Polling

If you request a license and none are available, license polling allows your request to exist until a license becomes available instead of exiting immediately. To control license polling, you use the `DW_WAIT_LICENSE` environment variable as follows:

- To enable license polling, set the `DW_WAIT_LICENSE` environment variable to 1.

- To disable license polling, unset the `DW_WAIT_LICENSE` environment variable. By default, license polling is disabled.

2.14.3 Simulation License Suspension

All DesignWare Verification IP products support license suspension. Simulators that support license suspension allow a model to check in its license token while the simulator is suspended, then check the license token back out when the simulation is resumed.

This capability is simulator-specific; not all simulators support license check-in during suspension.

2.15 Environment Variable and Path Settings

The following are environment variables and path settings required by the Synopsys USB VIP verification models:

- `DESIGNWARE_HOME` – The absolute path to where the DesignWare VIP is installed.
- `SNPSLMD_LICENSE_FILE` – The absolute path to a file that contains the license keys for Synopsys Common Licensing software or the *port@host* reference to this file.
- `LM_LICENSE_FILE` – The absolute path to a file that contains the license keys for your third-party tools. Also, include the absolute path to the third party executable in your PATH variable.

2.15.1 Simulator-Specific Settings

Your simulation environment and PATH variables must be set as required to support your simulator.

2.16 Determining Your Model Version

The version of the DesignWare USB VIP at time of publication is R-2021.03. The following steps tell you how to check the version of the models you are using.

Verification IP products are released and versioned by the suite and not by individual model. The version number of a model indicates the suite version.

- To determine the versions of Synopsys VIP models installed in your $DESIGNWARE_HOME tree, use the setup utility as follows:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -i home
```

- To determine the versions of Synopsys VIP models in your design directory, use the setup utility as follows:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -p design_dir_path -i design
```

2.17 Integrating a Synopsys VIP into Your Testbench

After installing a Synopsys VIP, follow these procedures to set up the VIP for use in testbenches:

- Creating a Testbench Design Directory
- The dw_vip_setup Utility
- Using DesignWare Verification IP in Your Testbench

2.17.1 Creating a Testbench Design Directory

A *design directory* contains a version of the Synopsys VIP that is set up and ready for use in a testbench. You use the dw_vip_setup utility to create design directories. For the full description of dw_vip_setup, see The dw_vip_setup Utility.

Note If you move a design directory, the references in your testbenches to the include files will need to be revised to point to the new location. Also, any simulation scripts in the examples directory will need to be recreated.

A design directory gives you control over the version of Synopsys VIP in your testbench because it is isolated from the DESIGNWARE_HOME installation. When you want, you can use dw_vip_setup to update the DesignWare VIP in your design directory. Figure 2-1 shows this process and the contents of a design directory.

Figure 2-1 Design Directory Created by dw_vip_setup

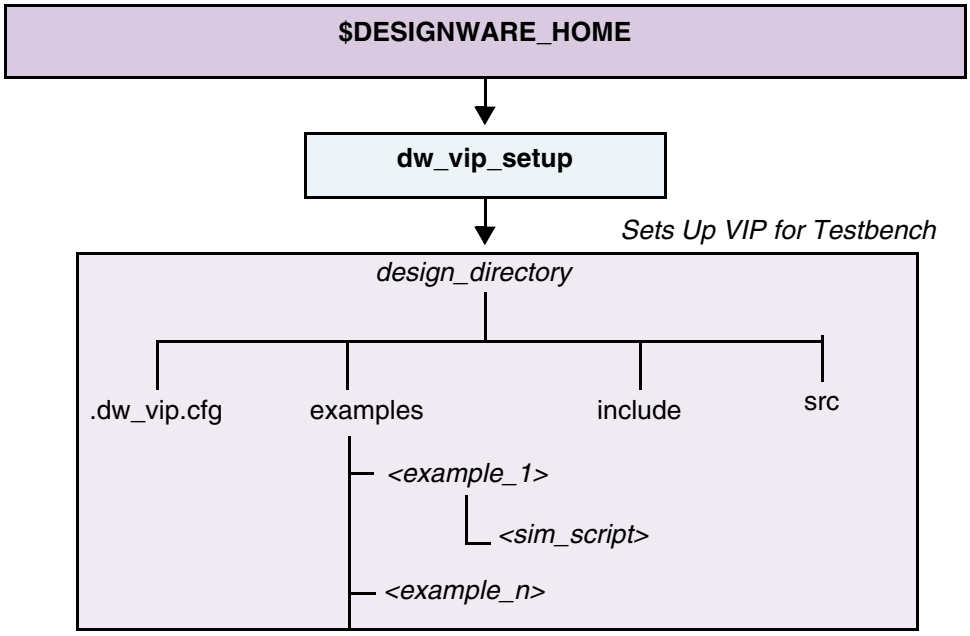


Table 2-1 Design Directory

Directory	Description
examples	Each MIPI VIP includes example testbenches. The dw_vip_setup utility adds them in this directory, along with a script for simulation. If an example testbench is specified on the command line, this directory contains all the files required for model, suite, and system testbenches.
include	Language-specific include files that contain critical information for VIP models. This directory is specified in simulator command lines.
src	VIP-specific include files (not used by all VIP). This directory may be specified in simulator command lines.

Directory	Description
`.dw_vip.cfg`	A database of all the VIP models being used in the testbench. The dw_vip_setup program reads this file to rebuild or recreate a design setup.

Note Do not modify this file because `dw_vip_setup` depends on the original content.

This section contains three examples that show common usage scenarios.

- Adding or Updating Synopsys VIP Models In a Design Directory
- Removing DesignWare VIP Models from a Design Directory
- Reporting Information About DESIGNWARE_HOME or a Design Directory
- Setting Environment Variables
- The dw_vip_setup Command

2.17.1.1 Adding or Updating Synopsys VIP Models In a Design Directory

Synopsys USB VIP models include:

- usb_link_svt
- usb_physical_svt
- usb_protocol_svt
- usb_subenv_svt

The following example adds a Synopsys USB VIP model to a design directory:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -a usb_link_svt -svtb
```

The following example updates a Synopsys USB VIP model in a design directory:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -p design_dir -u usb_link_svt -svtb
```

In these examples, the dw_vip_setup utility does the following:

1. Creates an include directory under the current directory and copies:
   - All files in the usb_link_svt model include directory
   - All include files in the Synopsys VIP suite
   - The latest SVT library include files into the include directory
2. Creates the Synopsys USB VIP suite libraries and SVT libraries.

2.17.1.2 Removing DesignWare VIP Models from a Design Directory

This example shows how to remove all listed models in the design directory at `/d/test2/daily` using the model list in the file del_list in the scratch directory under your home directory. The dw_vip_setup program command line is:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -p /d/test2/daily -r -m ~/scratch/del_list
```

The models in the *del_list* file are removed, but object files and include files are not.

2.17.1.3 Reporting Information About DESIGNWARE_HOME or a Design Directory

In these examples, the setup program sends output to STDOUT.

The following example lists the Synopsys VIP libraries, models, example testbenches, and license version in a DESIGNWARE_HOME installation:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -i home
```

The following example lists the Synopsys VIP libraries, models, and license version in a testbench design directory:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -p design_dir -i design
```

2.17.2 The dw_vip_setup Utility

The `dw_vip_setup` utility:

- Adds, removes, or updates Synopsys VIP models in a design directory
- Adds example testbenches to a design directory, the Synopsys VIP models they use (if necessary), and creates a script for simulating the testbench using any of the supported simulators
- Restores (cleans) example testbench files to their original state
- Reports information about your installation or design directory, including version information

2.17.2.1 Setting Environment Variables

Before running dw_vip_setup, the following environment variables must be set:

- `DESIGNWARE_HOME` – Points to where the Synopsys VIP is installed

2.17.2.2 The dw_vip_setup Command

You invoke dw_vip_setup from the command prompt. The dw_vip_setup program checks command line argument syntax and makes sure that the requested input files exist. The general form of the command is:

```
% dw_vip_setup [-p[ath] directory] switch (model [-v[ersion] latest | version_no] ) …
```

or

```
% dw_vip_setup [-p[ath] directory] switch -m[odel_list] filename
```

where

Table 2-2 dw_vip_setup Command

Command	Description
[-p[ath] *directory*]	The optional `-path` argument specifies the path to your design directory. When omitted, `dw_vip_setup` uses the current working directory.
switch	The switch argument defines `dw_vip_setup` operation. Table 2-3 lists the switches and their applicable sub-switches.

Table 2-3 Setup Program Switch Descriptions

Switch	Description
-a[dd] (*model* [**-v**[ersion] *version*]) …	Adds the specified model or models to the specified design directory or current working directory. If you do not specify a version, the latest version is assumed. The model names are: • usb_link_svt • usb_physical_svt • usb_protocol_svt • usb_subenv_svt The -add switch causes dw_vip_setup to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-r[emove] *model*	Removes ***all versions*** of the specified model or models from the design. The dw_vip_setup program does not attempt to remove any include files used solely by the specified model or models. The model names are: • usb_link_svt • usb_physical_svt • usb_protocol_svt • usb_subenv_svt
-u[pdate] (*model* [**-v**[ersion] *version*]) …	Updates to the specified model version for the specified model or models. The dw_vip_setup script updates to the latest models when you do not specify a version. The model names are: • usb_link_svt • usb_physical_svt • usb_protocol_svt • usb_subenv_svt The -update switch causes dw_vip_setup to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-e[xample] {*scenario* | *model*/*scenario*} [**-v**[ersion] *version*]	The dw_vip_setup script configures a testbench example for a single model or a system testbench for a group of models. The program creates a simulator run program for all supported simulators. If you specify a *scenario* (or system) example testbench, the models needed for the testbench are included automatically and do not need to be specified in the command. **Note:** Use the -info switch to list all available system examples.
-ntb	Not supported.
-svtb	Use this switch to set up models and example testbenches for SystemVerilog VMM. The resulting design directory is streamlined and can only be used in SystemVerilog simulations.
-c[lean] {*scenario* | *model*/*scenario*}	Cleans the specified scenario/testbench in either the design directory (as specified by the *-path* switch) or the current working directory. This switch deletes *all files in the specified directory*, then restores all Synopsys created files to their original contents.

Table 2-3 Setup Program Switch Descriptions (Continued)

Switch	Description
-i[nfo] *design* | *home*	When you specify the -info *design* switch, dw_vip_setup prints a list of all models and libraries installed in the specified design directory or current working directory, and their respective versions. Output from -info design can be used to create a model_list file. When you specify the -info *home* switch, dw_vip_setup prints a list of all models, libraries, and examples available in the currently-defined $DESIGNWARE_HOME installation, and their respective versions. The reports are printed to STDOUT.
-h[elp]	Returns a list of valid dw_vip_setup switches and the correct syntax for each.
model	Synopsys USB VIP models are: usb_link_svt usb_physical_svt usb_protocol_svt usb_subenv_svt The *model* argument defines the model or models that dw_vip_setup acts upon. This argument is not needed with the -info or -help switches. All switches that require the *model* argument may also use a model list. You may specify a version for each listed *model*, using the -version option. If omitted, dw_vip_setup uses the latest version. The -update switch ignores *model* version information.
-m[odel_list] *filename*	The -model_list argument causes dw_vip_setup to use a user-specified file to define the list of models that the program acts on. The model_list, like the *model* argument, can contain model version information. Each line in the file contains: *model_name* [**-v** *version*] –or– **# Comments**

The dw_vip_setup program treats all lines beginning with “#” as comments.

2.17.3 Using DesignWare Verification IP in Your Testbench

For those customers using the USB3 synthesizable IP shipped with the DesignWare corekit, there is an example of using the DesignWare VIP in a testbench. Please refer to the “vmmtb” testbench provided in the coreKit.

2.17.4 Running the Example With +incdir+

In the current setup, you install the VIP under *DESIGNWARE_HOME* followed by creation of a design directory which contains the versioned VIP files.

With every newer version of the already installed VIP requires the design directory to be updated.

This results in:

- Consumption of additional disk space

- Increased complexity to apply patches

The new alternative approach of directly pulling in all the files from *DESIGNWARE_HOME* eliminates the need for design directory creation. VIP version control is now in the command line invocation. The following code snippet shows how to run the basic example from a script:

```
cd /examples/sverilog/usb_svt/tb_usb_svt_ovm_basic_sys/

// To run the example using the generated run script with +incdir+
./run_usb_svt_ovm_basic_sys -verbose -incdir base_test vcsvlog
```

For example, the following compile log snippet shows the paths and defines set by the new flow to use VIP files right out of DESIGNWARE_HOME instead of `design_dir`.

```
vcs -l ./logs/compile.log -q -Mdir=./output/csrc
+define+DESIGNWARE_INCDIR= \
+define+SVT_LOADER_UTIL_ENABLE_DWHOME_INCDIRS \
+incdir+/vip/svt/usb_svt/sverilog/include \
-ntb_opts ovm -full64 -sverilog +define+OVM_PACKER_MAX_BYTES=16000
+define+OVM_PACKER_MAX_BYTES=1500000 \
+define+OVM_DISABLE_AUTO_ITEM_RECORDING -timescale=1ns/1ps +define+SVT_OVM_TECHNOLOGY
+define+SYNOPSYS_SV \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_ovm_basic_sys/
. \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_ovm_basic_sys/
../../env \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_ovm_basic_sys/
../env \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_ovm_basic_sys/
env \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_ovm_basic_sys/
dut \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_ovm_basic_sys/
hdl_interconnect \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_ovm_basic_sys/
lib \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_ovm_basic_sys/
tests \
-o ./output/simvcssvlog -f top_files -f hdl_files
```

Note
For VIPs with dependency, include the `+incdir+` for each dependent VIP.

3 General Concepts

OVM 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.

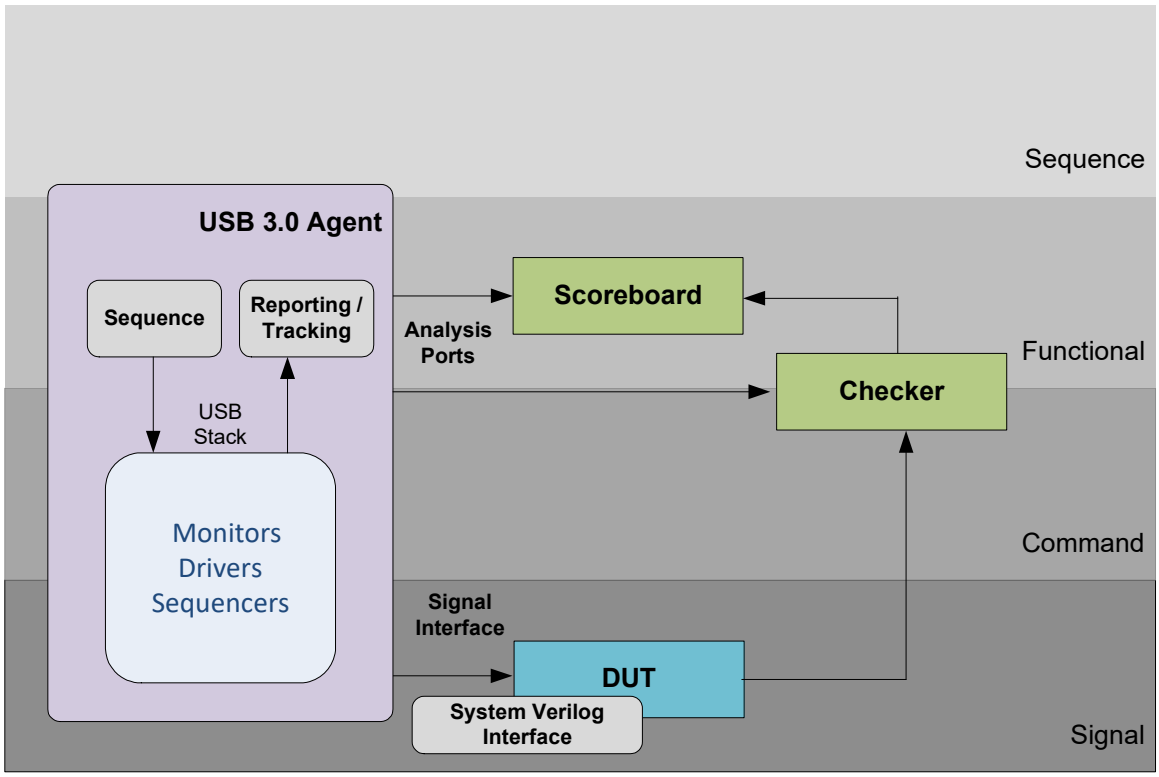
The USB VIP for OVM described in this document provides class libraries that verify features of the USB protocol. The VIP provides a USB OVM agent that contains stacked components that are compatible for use with SystemVerilog-Compliant OVM based testbenches. Agent supports all functionality normally associated with active OVM 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.

3.1 USB VIP in a OVM Environment

Figure 3-1 shows where the USB VIP fits into the OVM methodology. In the layered approach that is typical for OVM, the VIP (purple) fits into the lower levels, which allows you to focus on higher levels of abstraction.

Figure 3-1 USB VIP in Layered Testbench Architecture



3.1.1 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 OVM 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 OVM is not simply a set of guidelines and recommendations. So, instead of writing your own reporting routine, you can reuse the ovm report class. Inheritance, extension, and polymorphism facilitate customization opportunities.

Important OVM base classes used by the USB VIP include:

- ovm_agent
- ovm_component
- ovm_object
- ovm_callback
- ovm_sequencer

3.1.2 OVM Components

Objects derived from ovm_component are objects in a OVM compliant verification environment. The testbench and stack layers exchange transactions through ovm_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 ovm_component supports additional callbacks to access to data at internal dataflow points. Refer to the HTML documentation for a complete callback list.
- **OVM Events**
  - Events are based on the ovm_event class. Some ovm_component signal “significant events” through OVM events and include transactions as ovm_data objects with these events. Testbenches can be configured to wait for ovm_event instances, making a call to the ovm_component’s OVM event service instance. The you can retrieve data by making a call to the get_trigger_data() method on the OVM event.

OVM 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.

3.1.3 USB VIP Objects

The USB VIP defines several classes designed for a OVM 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.

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"

3.1.3.1 Configuration Objects

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

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

Configuration data objects conveys 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 ovm_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 OVM 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 ovm_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 ovm_data class contains information regarding a 'USB SS stream resource'.

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

For more information, see the Class Reference.

3.1.3.2 Sequence Item Data Objects

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

Pre-defined sequence and sequencer classes exist for different OVM 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.

ovm_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.

ovm_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 ovm_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 ovm_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 ovm_component controls USB bus activity using the list of transactions. Alternatively, the testbench can leave the list of data objects empty and the protocol ovm_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 ovm_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.

3.1.3.3 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 ovm_sequence_item base class. These objects implement all of the methods specified by OVM for the ovm_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 OVM 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 ovm_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.

3.1.3.4 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.

3.1.3.5 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 ovm_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 ovm_callback class. These objects implement all of the methods specified by OVM for the ovm_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.

```
ovm_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

For more information, see the Class Reference.

3.1.4 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 OVM the svt_usb_if interface comes from the svt_usb_if.ovm.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).

For more information, see the USB SVT - Interfaces Reference page in the Class Reference.

3.1.5 Constraints

3.1.5.1 Description

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

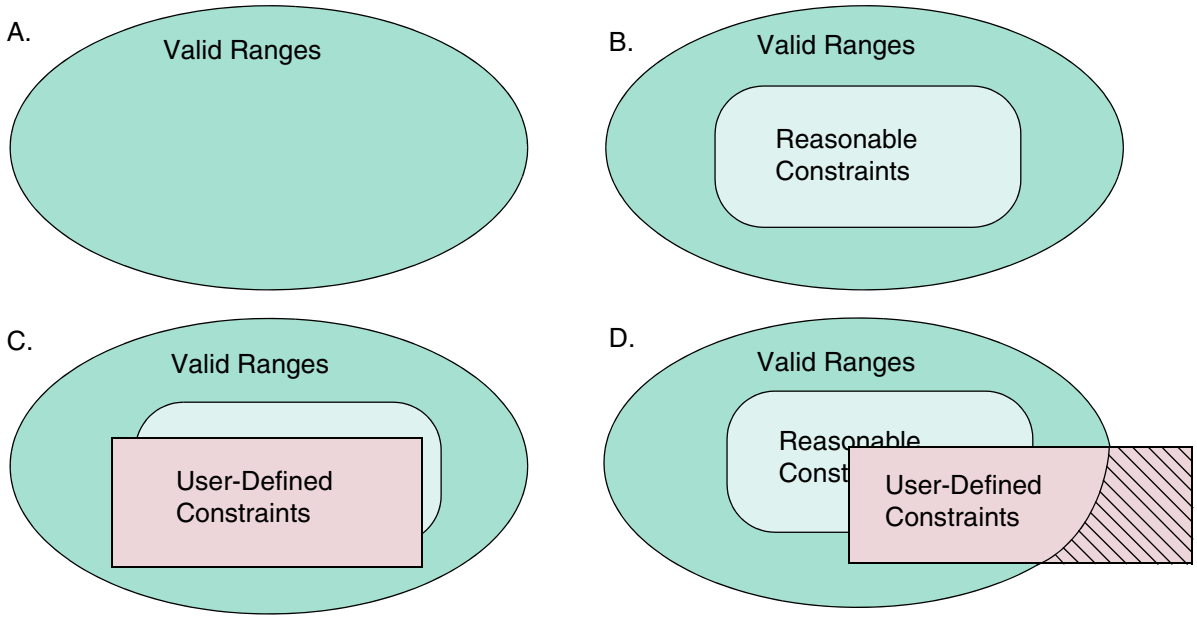
Classes that provide random attributes allow you to constrain the contents of the resulting object. When you call the randomize() method, which is a built-in method, all random attributes are randomized using all constraints that are enabled.

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 3-2 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.

3.1.5.2 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.

For 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.

3.1.6 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 OVM factories for the user sequencers as well as for VIP components should all go through the ovm_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

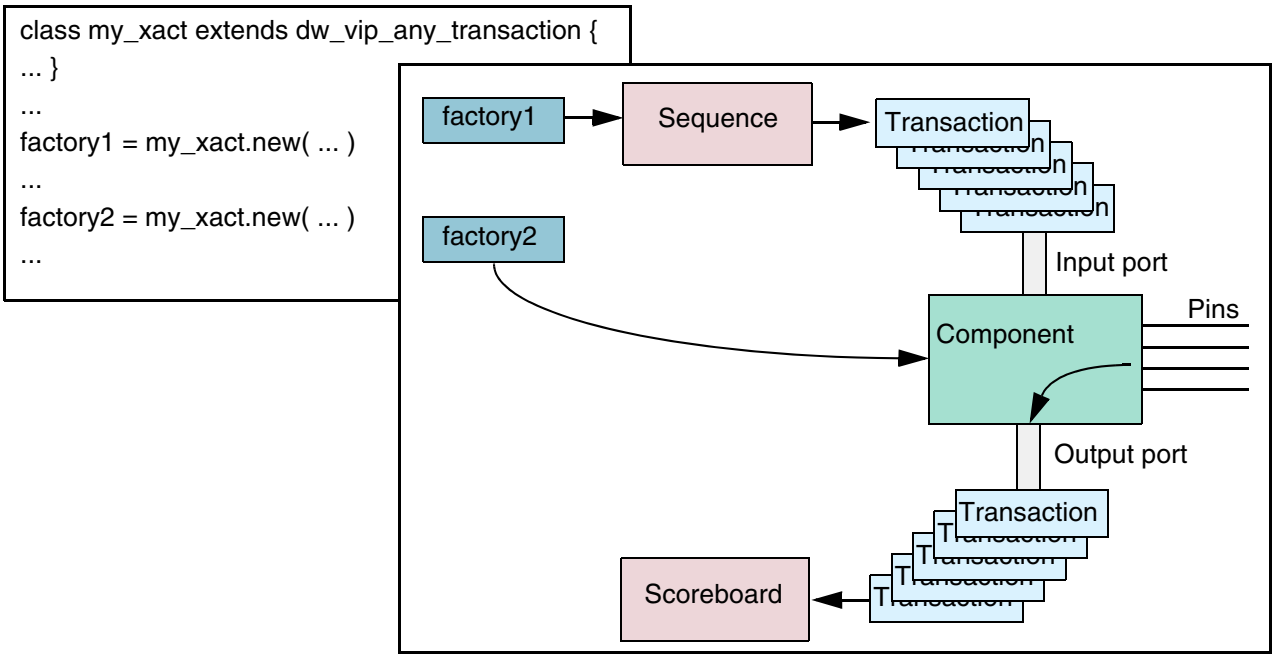
  The nomenclature for this is 'randomized object'; these variables are prefixed with the 'randomized_' string.

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

Figure 3-3 Factories with USB VIP



When using USB VIP, the factory object is typically a sequence. In Figure 3-3, 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 OVM, constructors come in entirely through ovm_config_db (that is, all 'randomized_' field)s and/or the factory override subsystem (all randomized_ fields if not found via config_db, as well as all _factory fields in all situations).

3.1.7 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. OVM supports and promotes identifying messages by string matching against a regular expression.

4 The USB Agent Overview

OVM system-level verification environments is constructed with an OVM Agent which contains three OVM Components, each functioning as the Protocol, Link, and Physical layers of the USB protocol.

4.0.1 OVM Component Stack

USB ovm_components are component objects in a OVM-compliant environment that implement a specific layer of the USB protocol. The USB VIP defines four ovm_components, each of which extends from the *ovm_component* class:

- Protocol ovm_component
- Link ovm_component
- Physical ovm_component

In addition, there is another ovm_component layer that models the physical layer of the remote USB stack.

The set of components within a stack are contiguous. 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 local 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 through the ovm_components. The top layer of the component stack receives stimulus objects that drive 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. Refer to the Data Objects section of the respective component descriptions in USB Verification OVM IP Stack Components for detailed information.

Note

- All of the components have all of the necessary 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.
- You should use model provided sequencers. The focus should be for you to provide sequences for the Synopsys provided sequencers.

The bottom layer is a Physical ovm_component that connects to the DUT through a signal interface driver. The USB VIP defines serial and PIPE3 interfaces, allowing the Physical ovm_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.

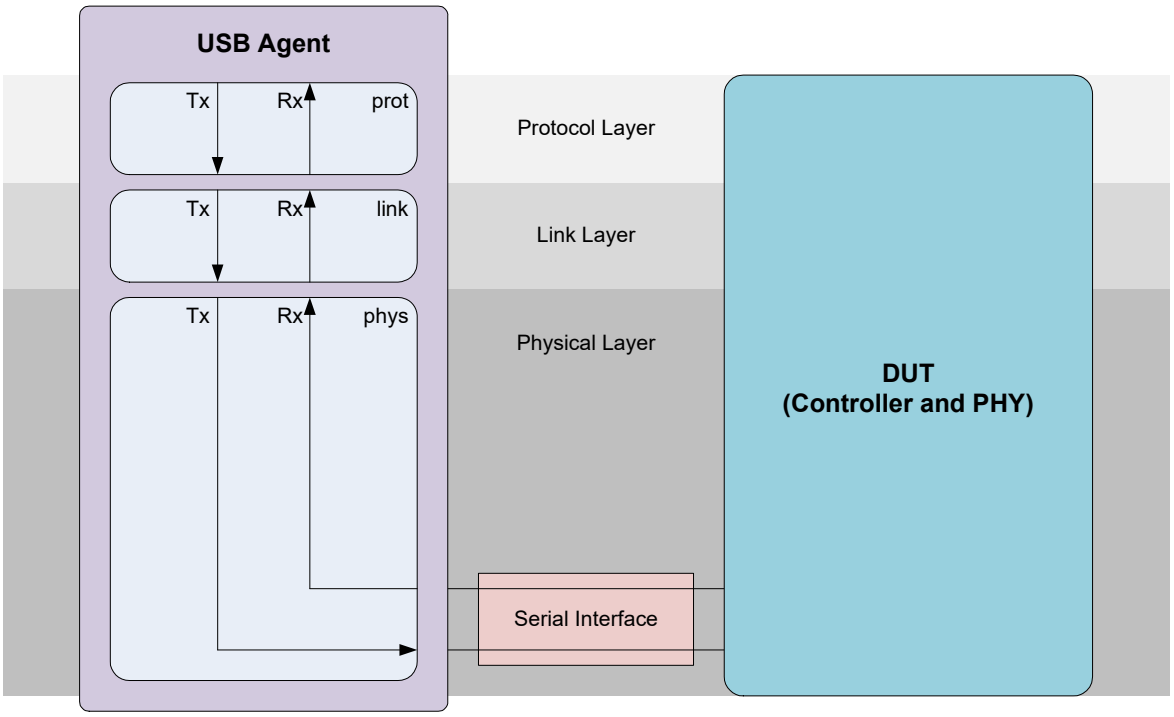
The following examples display ovm_component stack formations in common agent configurations. Refer toVerification Topologies for additional examples.

Example 1: Verifying a Controller-PHY (Figure 4-1)

The VIP agent in Figure 4-1 utilizes a three layer protocol stack, containing protocol, link, and physical component. If the VIP is configured as a host, the protocol lay receives stimulus data through a transfer port; otherwise, the protocol component receives transfers through an input port in response to the Rx data stream.

The Physical layer connects to the DUT through a serial interface. The VIP supports SS and 2.0 serial interfaces.

Figure 4-1 Verifying a USB Controller and PHY

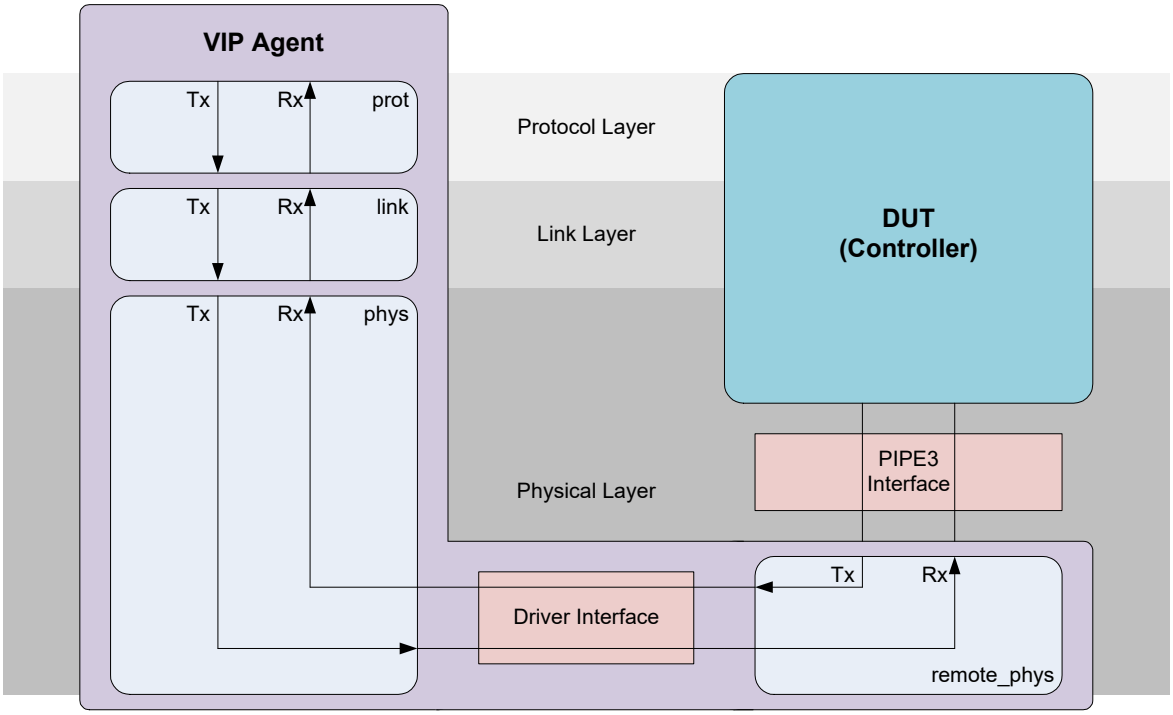


Example 2: Verifying a Controller (Figure 4-2)

The VIP agent in Figure 4-2 utilizes a four layer protocol stack, containing protocol, link, and two physical components. If the VIP is configured as a host, the protocol component receives stimulus data through a transfer port; otherwise, the protocol component receives transfers through an input port in response to the Rx data stream.

The two physical components connect through a port interface. The remote physical component simulates the PHY portion of the Physical layer and connects to the MAC interface of the DUT through a PIPE3 interface. The PIPE3 interface is not symmetrical, unlike the serial interface – the PHY side of the PIPE3 interface differs from the MAC side. Refer to Interface Options for information on Physical component interfaces.

Figure 4-2 Verifying a USB Controller



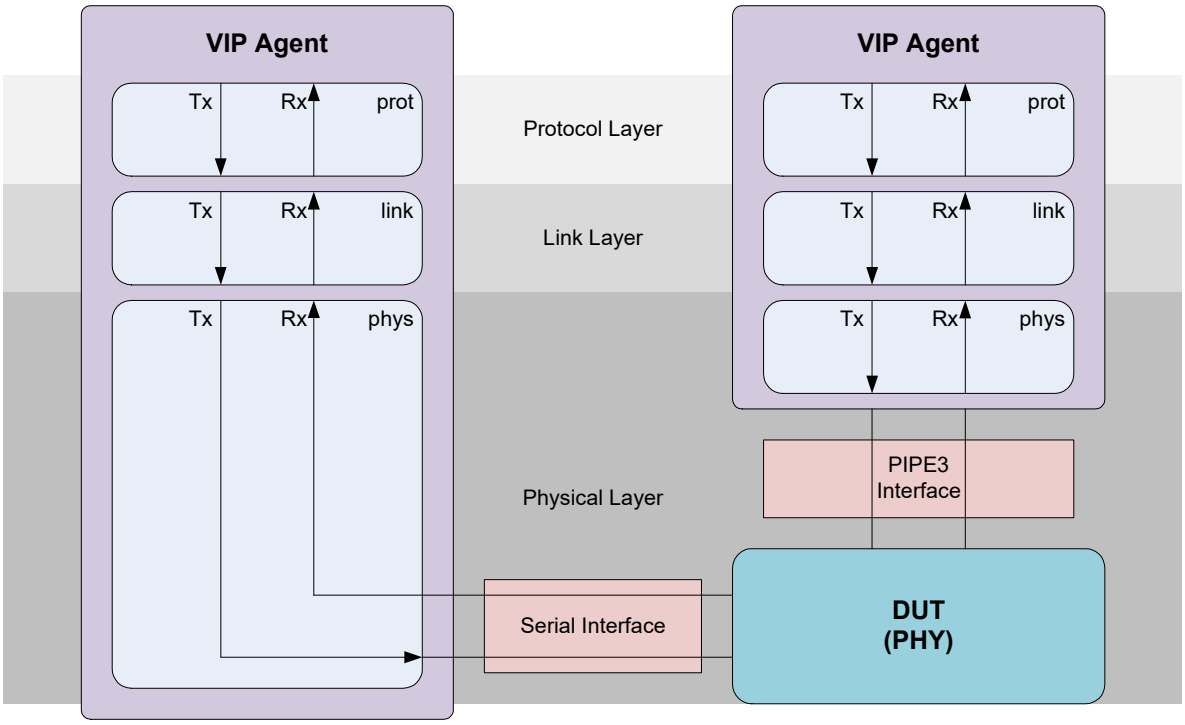
Example 3: Verifying a PHY (Figure 4-3)

The environment Figure 4-3 utilizes two agents. The agent on the left side of Figure 4-3 is remote from the DUT, whereas the agent on the right side is local to the DUT.

The agent that is remote from the DUT contains a three layer component stack, similar to the agent in Figure 4-1. The Physical component connects to the DUT through a serial interface.

The agent that is local to the DUT also contains a three layer component stack. Unlike the Physical component on the remote side, which simulates the MAC and PHY portions of the USB physical layer, the Physical component on the local (right) side simulates only the MAC portion of the USB physical layer and connects to the DUT-PHY through a PIPE3 interface. As in Example 2, the PIPE3 interface is not symmetrical – the MAC side of the interface must connect to the agent and the PHY side of the interface must connect to the DUT.

Figure 4-3 Verifying a USB Controller



4.0.2 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.).

4.0.3 Reporting and Tracking Objects

The agent supports the following reporting and tracking objects:

4.0.3.1 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

For more information for a list of checks provided by these classes, see the class reference.

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

- enable_prot_chk
- enable_link_chk
- enable_phys_chk

4.0.3.2 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

4.0.4 OVM Events

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

The following are the ovm_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

For more information, see the class reference.

5 USB Verification OVM IP Stack Components

This chapter describes the ovm_component objects that the USB VIP supports. For more description of objects, classes, and attributes mentioned in this chapter, see the Class Reference.

5.1 Protocol Component

USB Protocol components are component objects in a OVM-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 *ovm_component* class.

The Class Reference HTML describes Protocol component functions and attributes.

5.1.1 Protocol Layer Feature Support

Protocol Layer Features 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

5.1.2 Protocol Component Ports

OVM 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 use by the sequencer to send sequences into the protocol layer. You cannot connect to them. They are only used by the sequencer. Consult OVM 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 OVM 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).

5.1.3 Data Objects

The following is a list of objects that represent information the Protocol component receives, sends, or processes. 5.1.3 displays the flow of information objects within the Protocol component.

Sequence Item Data Objects 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.

5.1.4 Protocol Component Modes

5.1.4.1 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.

5.1.4.2 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.

5.1.5 Data Transformation Objects

The following list describes Protocol component objects that manipulate data objects. Figure 5-1 displays the objects that affect Protocol component data flow objects.

5.1.5.1 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.

5.0.6.2 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 OVM
- Setting via the 'inst_override' and 'type_override' capabilities in OVM

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 OVM interface helps it to create it as an instance of the override class (e.g., my_xfer_class).

In OVM 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 OVM, 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 5-1 displays a functional representation of the Protocol component factory objects

5.1.5.3 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 5-1 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.*

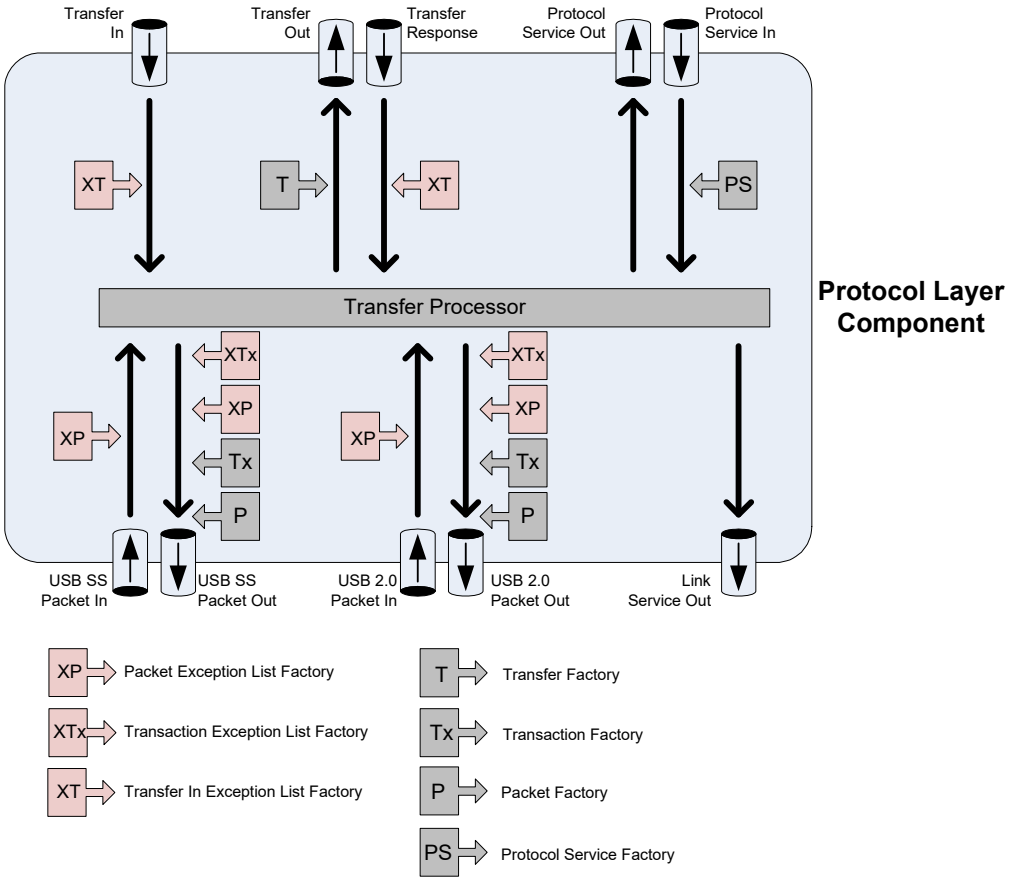
- **Transaction exception list factory:** These randomization factories create exceptions that are injected into USB transactions.

  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 5-1 Protocol Component Factories – Objects and Exception List



5.1.5.4 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*

5.1.5.5 OVM Protocol Events

The USB Protocol component supports OVM and USB VIP specific OVM events. OVM event instances for which the current transaction is pertinent include a handle to the current data entity (transfer, transaction, or packet) with the ovm_event, which is accessible via the 'ovm_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 OVM-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 OVM 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.

5.1.6 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.

5.1.7 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.

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

5.1.7.1.1 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 d*iscarded_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.

5.1.7.1.2 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.

5.1.7.1.3 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.

5.1.7.1.4 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).

5.1.7.2 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.

5.1.7.3 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.

5.1.7.3.1 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.

5.1.7.3.2 Stream Endpoint Manager Flow

Figures 5-2 and 5-3 illustrate the stream endpoint manager flow for host and device respectively.

Figure 5-2 Host Stream Endpoint Manager Flow

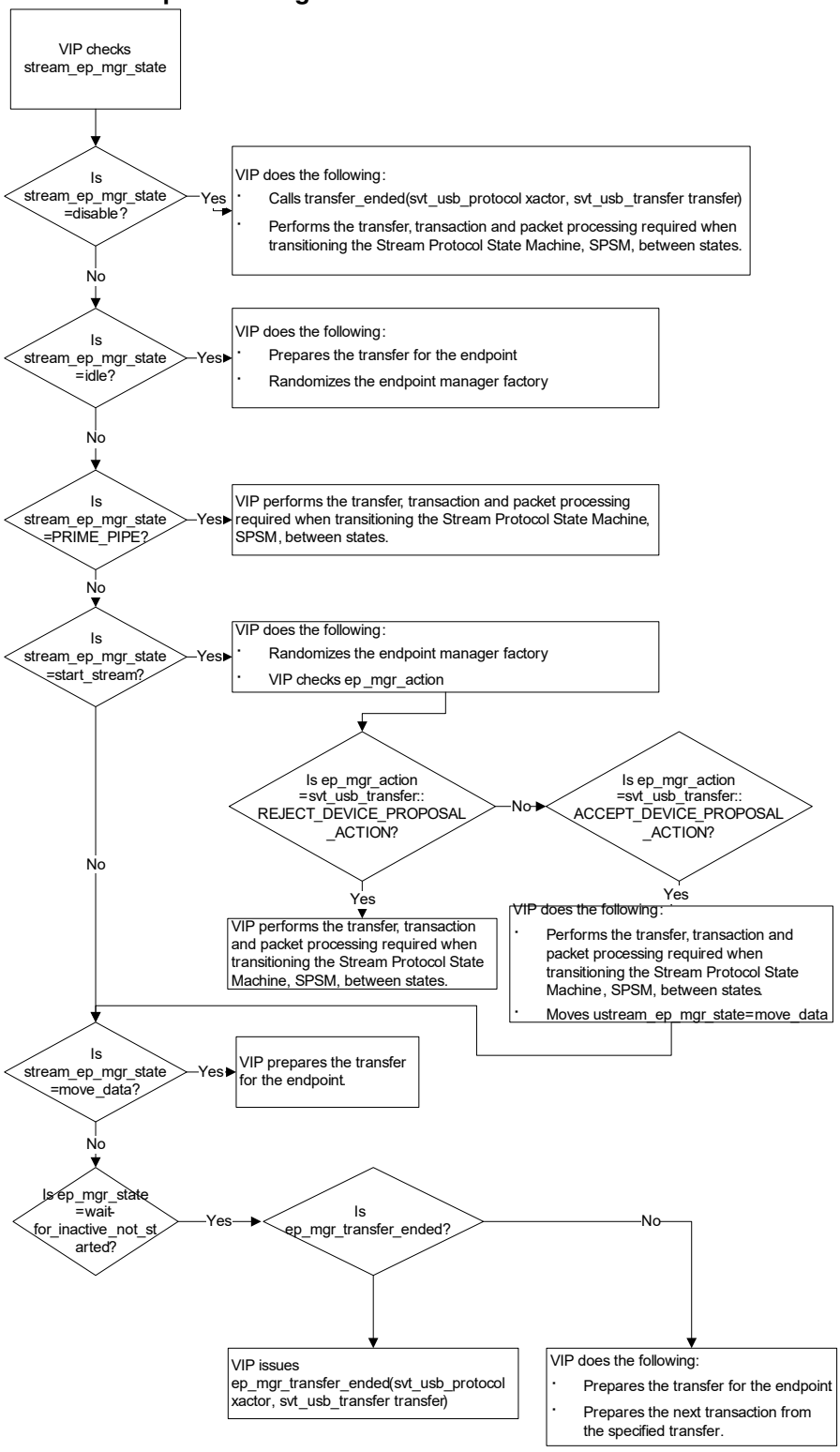
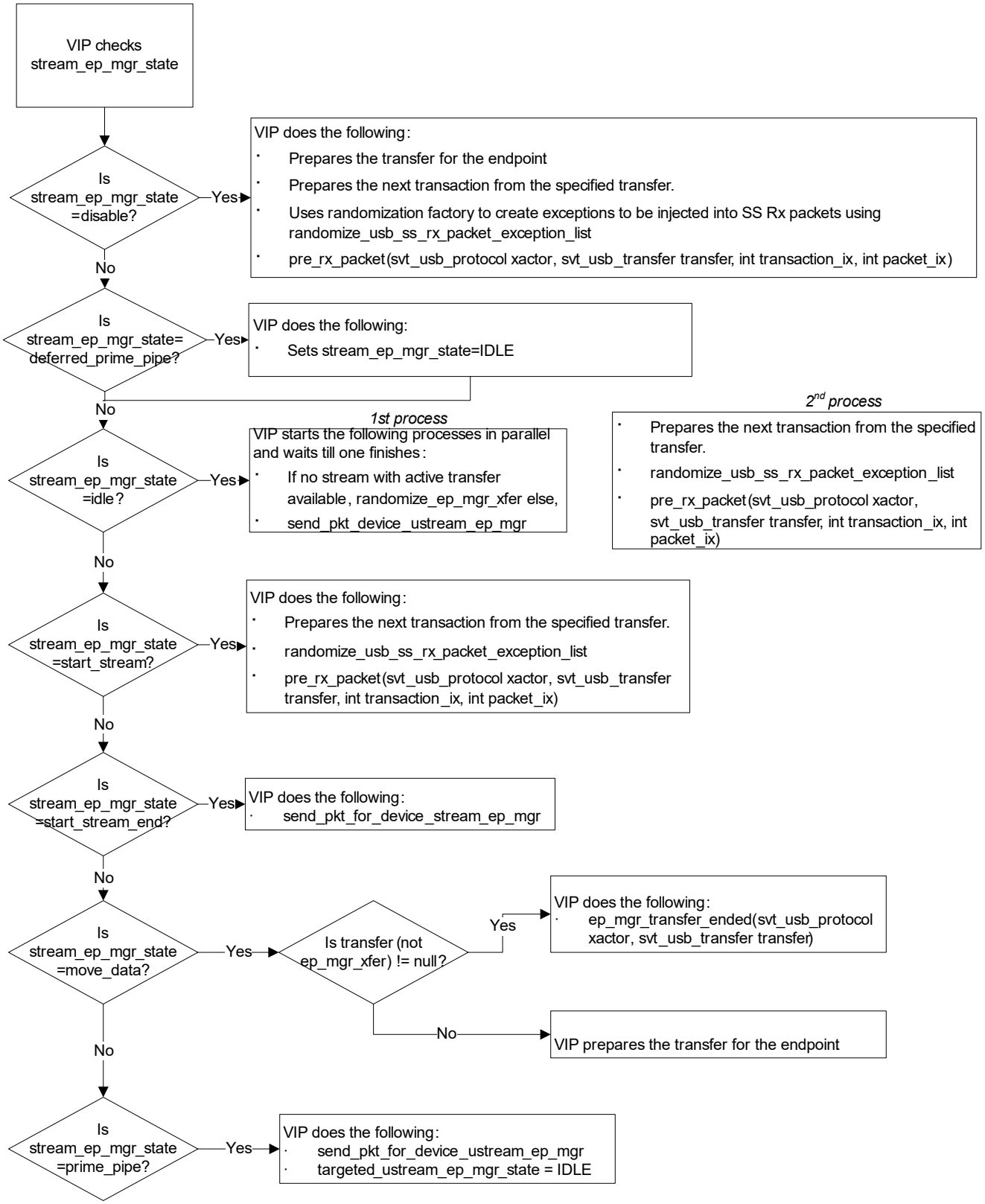


Figure 5-3 Device Stream Endpoint Manager Flow



5.1.8 Protocol Component 2.0 Link Callback, Factory, and OVM Event Flows

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

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

5.1.8.1.1 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 ovm_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).*

5.1.8.1.2 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 ovm_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 ovm_event that the transaction has ENDED using *transfer.end_tr().*
3. Provide a copy of the ENDED transaction with the top protocol ovm_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 ovm_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).*

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

5.1.8.2.1 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 ovm_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 ovm_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 ovm_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 *ovm_event begin_event (ovm_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)*.

5.1.8.2.2 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 ovm_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 ovm_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 ovm_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 -> *ovm_event begin_event (ovm_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 ovm_event begin_event (ovm_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 ovm_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).*

5.2 Link Component

USB Link component are component objects in a OVM-compliant verification environment. The USB Link component object extends from the *ovm_component* class, which extends from the *ovm_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.

5.2.1 Link Layer Feature Support

Link Layer Features 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

5.2.2 Link Component Ports

OVM 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 use by the sequencer to send sequences into the link layer. You cannot connect to them. They are only used by the sequencer. Consult OVM 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 (recieved).
- **usb_ss_packet_observed_port**. Super-speed Packet analysis port used to give testbench access to packets consumed (recieved).
- **usb_20_packet_in_port**. 2.0 Packet input port used to supply stimulus packets for the VIP to produce (transmit).

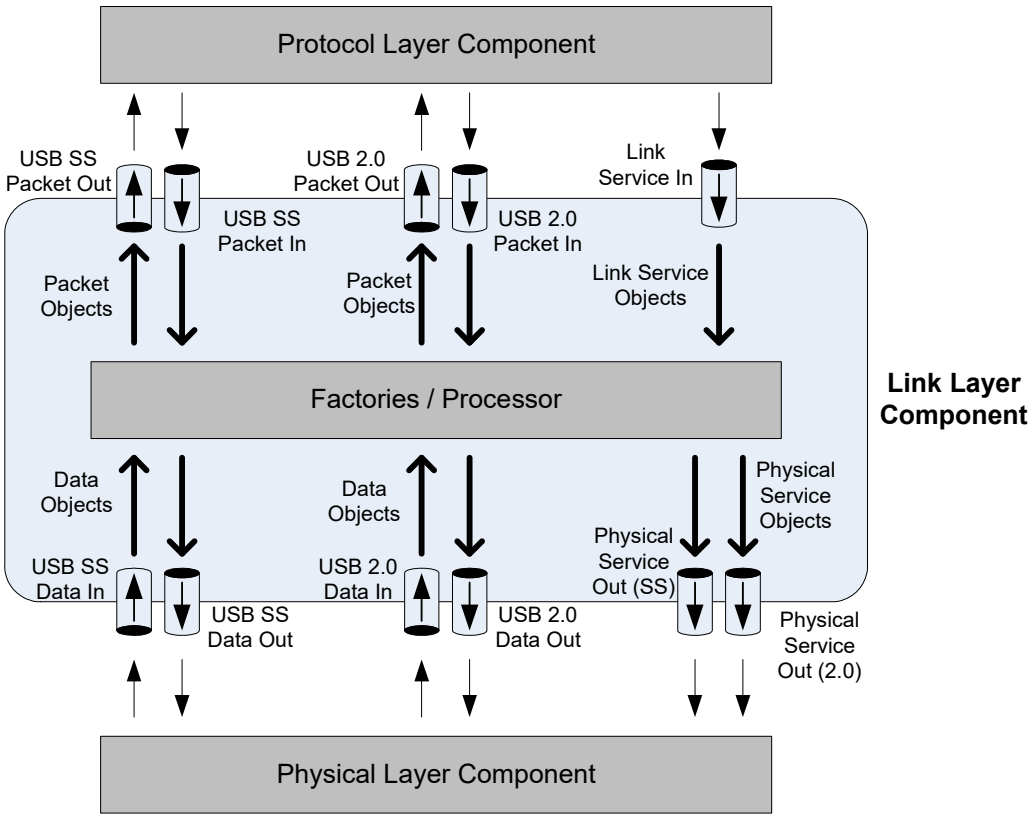
- **usb_20_packet_out_port** . 2.0 Packet get_peek output port used to give upper-layer access to packets consumed (recieved).
- **usb_20_packet_observed_port**. 2.0 Packet analysis port used to give testbench access to packets consumed (recieved).
- **usb_ss_data_in_port**. Rx Data objects coming from 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.

5.2.3 Data Objects

The following is a list of objects that represent information the Link component receives, sends, or processes. Figure 5-4 displays the flow of information objects within the Link component.

Sequence Item Data Objects describes USB data objects

Figure 5-4 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.

5.2.4 Data Transformation Objects

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

5.2.4.1 Factory Objects

The link component creates these Factory data objects. To create them, the componenet uses OVM 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 ovm_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.*

5.2.4.2 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.

5.2.4.3 Callbacks

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

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:

```
ovm_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*

5.2.4.4 OVM Events

USB Link components support general OVM 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 OVM 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*

5.2.5 Link Component SuperSpeed Link Callback, Factory, and OVM 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.

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

USB 2.0 Service Command Handling

Table 5-1 lists the various USB 2.0 link service commands that are available.

Table 5-1 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 5-1 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.

5.2.6 SuperSpeed Packet Chronology

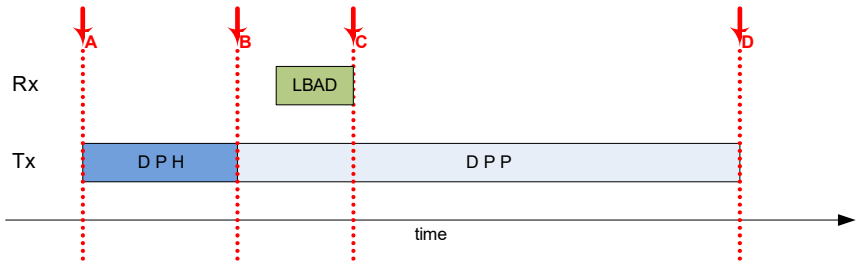
This section describes the timing of packets on the bus. Possible outcomes of low-level packet exchange on the bus are represented by the events and status values of the packet data objects within the VIP.

5.2.6.1 Receiving LBAD

5.2.6.1.1 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 5-5 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

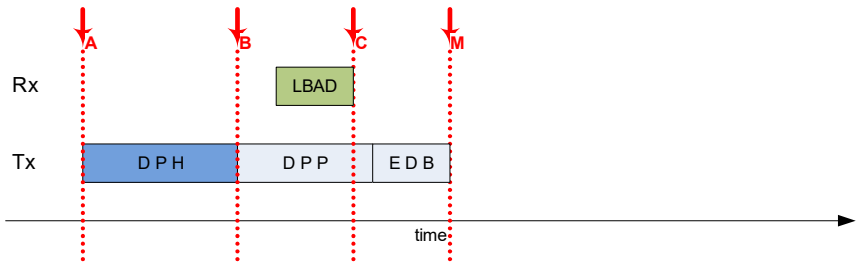
Table 5-2 Status Attribute Chronology: Receiving LBAD During DPP Tx

	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

5.2.6.1.2 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 5-6 Packet Chronology: Receiving LBAD During DPP Tx, EDB Result



At points “A” through “C” the events are identical to Figure 5-5 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

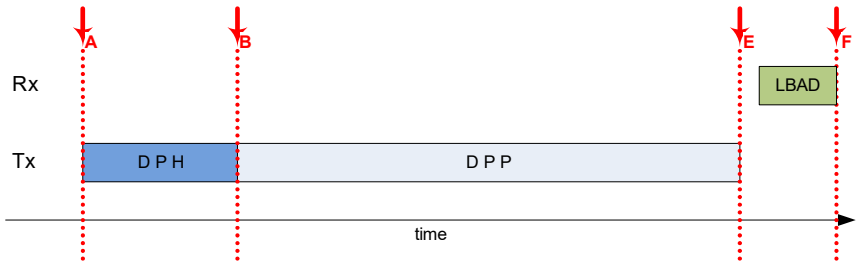
Table 5-3 Status Attribute Chronology: Receiving LBAD During DPP Tx, EDB Result

	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
payload_presence	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT_ BUT_ABORTED

5.2.6.1.3 After DPP Tx Complete

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

Figure 5-7 Packet Chronology: Receiving LBAD After DPP Tx Complete



At points "A" and "B" the events are identical to Figure 5-5 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

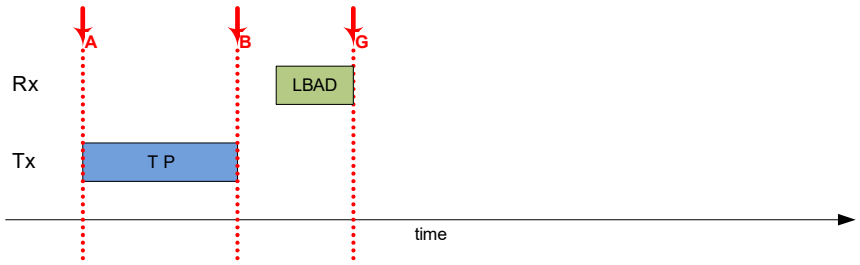
Table 5-4 Status Attribute Chronology: Receiving LBAD After DPP Tx Complete

	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

5.2.6.1.4 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 5-8 Packet Chronology: Receiving LBAD After HP, No DPP



At point "A" the events are identical to Figure 5-5 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

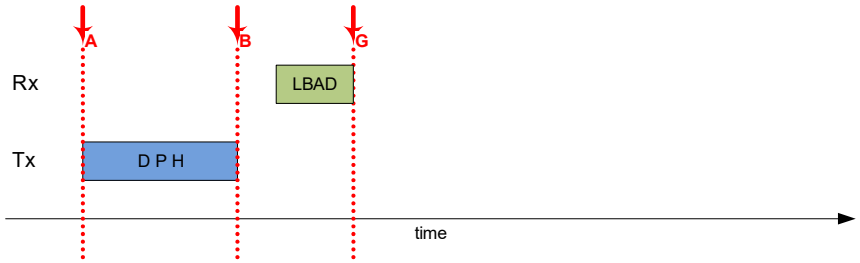
Table 5-5 Status Attribute Chronology: Receiving LBAD After HP, No DPP

	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

5.2.6.1.5 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 5-9 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

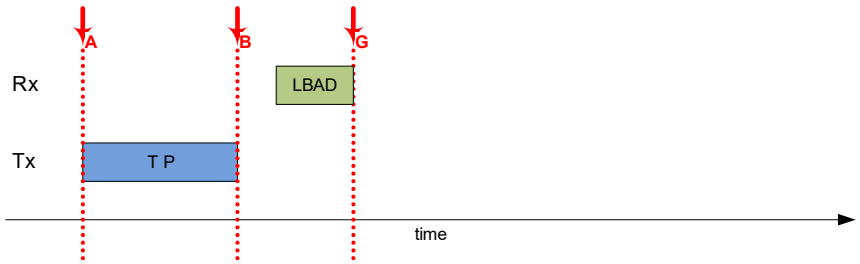
Table 5-6 Status Attribute Chronology: Receiving LBAD After Data Packet replay

	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
payload_presence	(various possible)	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT

5.2.6.1.6 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 5-10 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 5-7 Status Attribute Chronology: Receiving LBAD After Non-Data Packet Replay

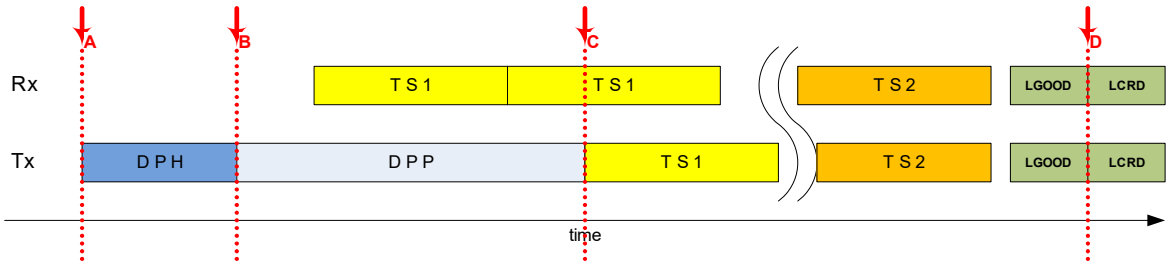
	Prior to Point "A"	at Point "A"	at Point "B"	at Point "G"
status	RETRY	ACTIVE	PARTIAL_ACCEPT	RETRY
header_status	RETRY	ACTIVE	ACCEPT	ACCEPT
payload_status	DISABLED	DISABLED	DISABLED	DISABLED
payload_presence	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT

5.2.6.2 Receiving TS1 Ordered Sets

5.2.6.2.1 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 DPH in question (such that the DPH packet will be replayed).

Figure 5-11 Packet Chronology: Receiving TS1 ordered sets During DPP Tx



At points "A" and "B" the events are identical to Figure 5-5 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

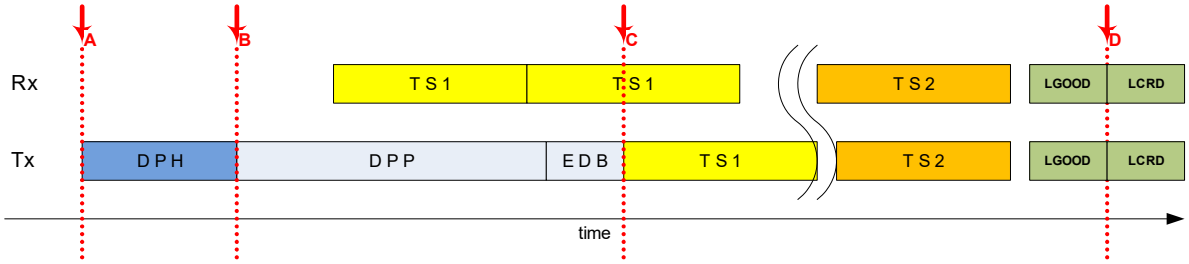
Table 5-8 Status Attribute Chronology: Receiving TS1 Ordered Sets During DPP Tx

	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

5.2.6.2.2 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 DPH in question (such that the DPH packet will be replayed).

Figure 5-12 Packet Chronology: Receiving TS1 Ordered Sets During DPP Tx, EDB Result



At points "A" and "B" the events are identical to Figure 5-5 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

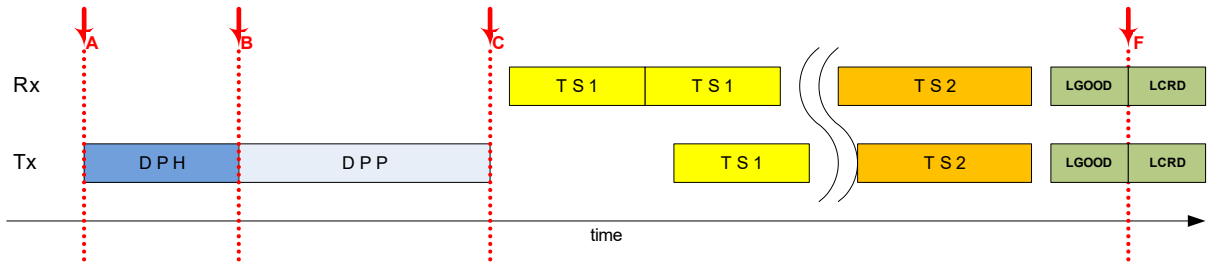
Table 5-9 Status Attribute Chronology: Receiving TS1 Ordered Sets During DPP Tx, EDB Result

	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

5.2.6.2.3 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 DPH in question (such that the DPH packet will be replayed).

Figure 5-13 Packet Chronology: Receiving TS1 Ordered Sets After DPP Tx Complete



At points "A" through "C" the events are identical to Figure 5-7 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 5-10 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

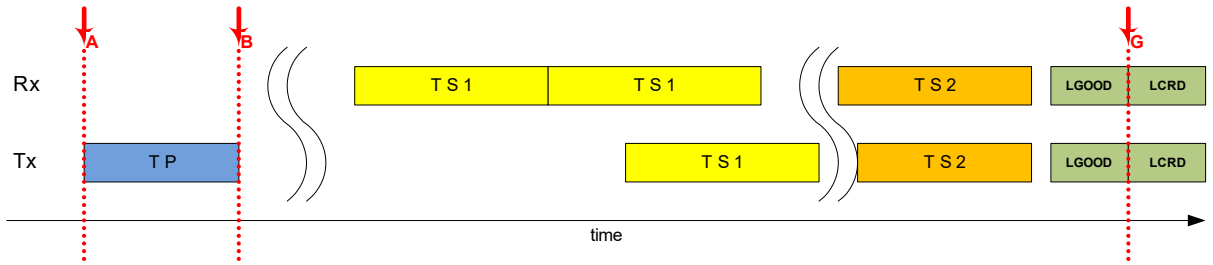
Table 5-10 Status Attribute Chronology: Receiving TS1 Ordered Sets After DPP Tx Complete

	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

5.2.6.2.4 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 5-14 Packet Chronology: Receiving TS1 Ordered Sets After HP, No DPP



At points "A" and "B" the events are identical to Figure 5-8 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

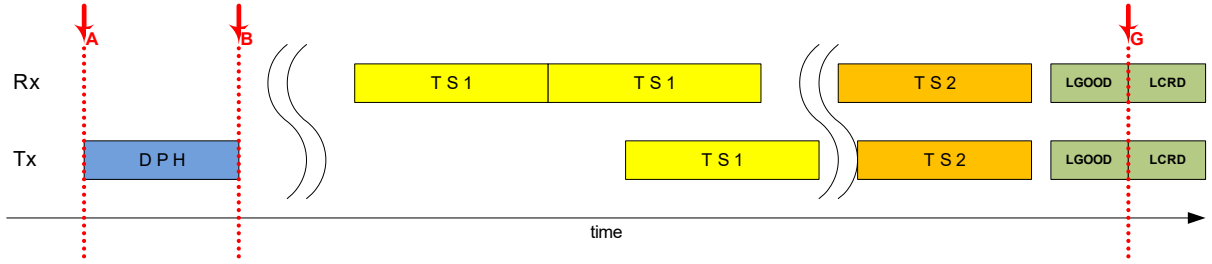
Table 5-11 Status Attribute Chronology: Receiving TS1 Ordered Sets After HP, No DPP

	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

5.2.6.2.5 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 5-15 Packet Chronology: Receiving TS1 Ordered Sets After Data Packet replay



At points “A” and “B” the events are identical to Figure 5-9 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

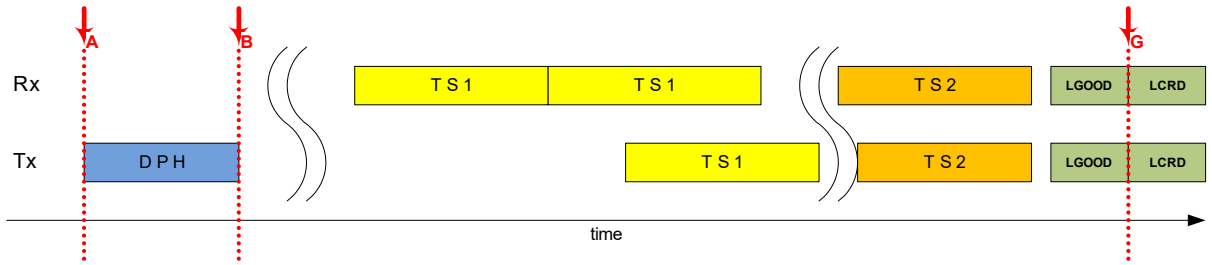
Table 5-12 Status Attribute Chronology: Receiving TS1 Ordered Sets After Data Packet replay

	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

5.2.6.2.6 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 5-16 Packet Chronology: Receiving TS1 Ordered Sets After Data Packet Replay



At points “A” and “B” the events are identical to Figure 5-8 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 5-13 Status Attribute Chronology: Receiving TS1 Ordered Sets After Data Packet Replay

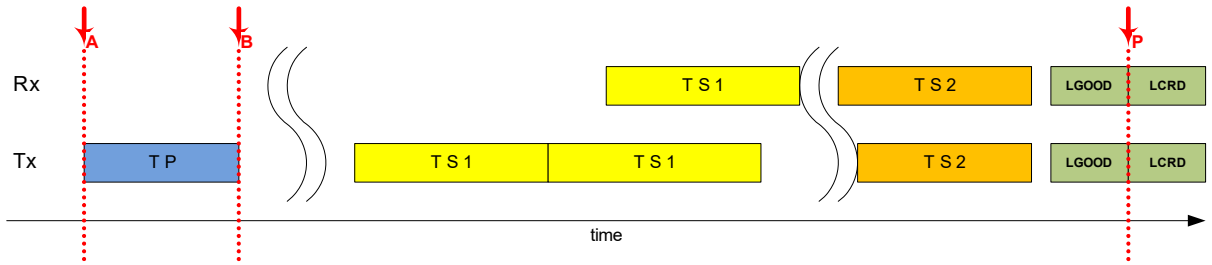
	Prior to Point “A”	at Point “A”	at Point “B”	at Point “G”
status	RETRY	ACTIVE	PARTIAL_ACCEPT	RETRY
header_status	RETRY	ACTIVE	ACCEPT	ACCEPT
payload_status	DISABLED	DISABLED	DISABLED	DISABLED
payload_presence	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT

5.2.6.3 Transmitting TS1 Ordered Sets

5.2.6.3.1 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 5-17 Packet Chronology: Transmitting TS1 Ordered Sets After HP Tx



At points “A” and “B” the events are identical to Figure 5-8 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

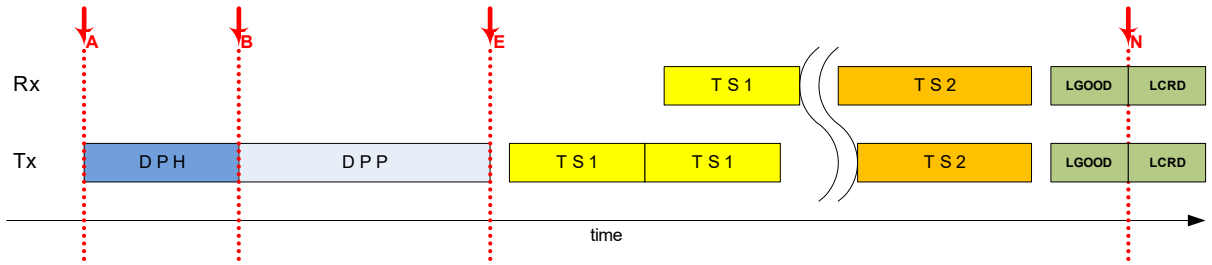
Table 5-14 Status Attribute Chronology: Transmitting TS1 Ordered Sets After HP Tx

	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

5.2.6.3.2 After DPP Tx

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

Figure 5-18 Packet Chronology: Transmitting TS1 Ordered Sets After DPP Tx



At points "A" through "E" the events are identical to Figure 5-7 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

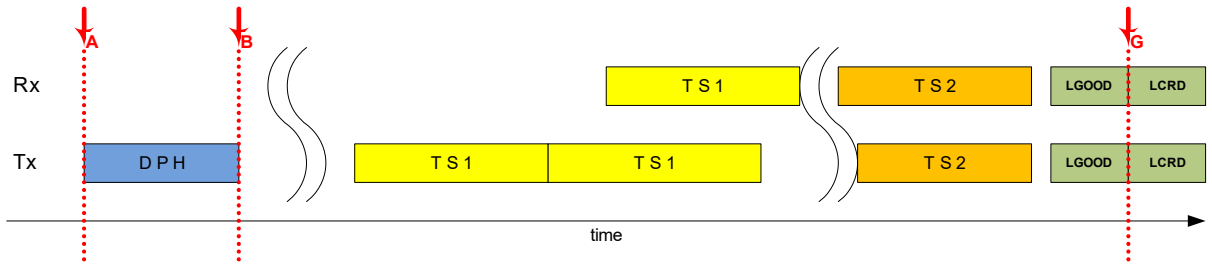
Table 5-15 Status Attribute Chronology: Transmitting TS1 Ordered Sets After DPP Tx

	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

5.2.6.3.3 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 5-19 Packet Chronology: Transmitting TS1 Ordered Sets After Data Packet Replay



Events at points “A” through “G” are identical to Figure 5-9 events at points “A” through “G”

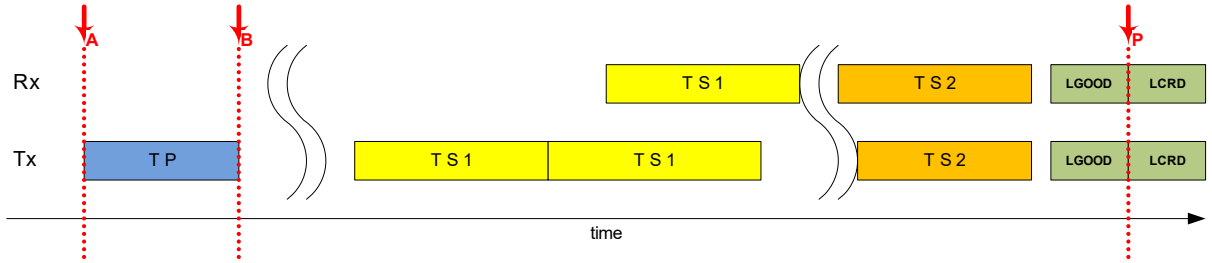
Table 5-16 Status Attribute Chronology: Transmitting TS1 Ordered Sets After Data Packet Replay

	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

5.2.6.3.4 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 5-20 Packet Chronology: Transmitting TS1 Ordered Sets After Non-Data Packet Replay



At points “A” and “B” the events are identical to Figure 5-8 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 5-17 Status Attribute Chronology: Transmitting TS1 Ordered Sets After Non-Data Packet Replay

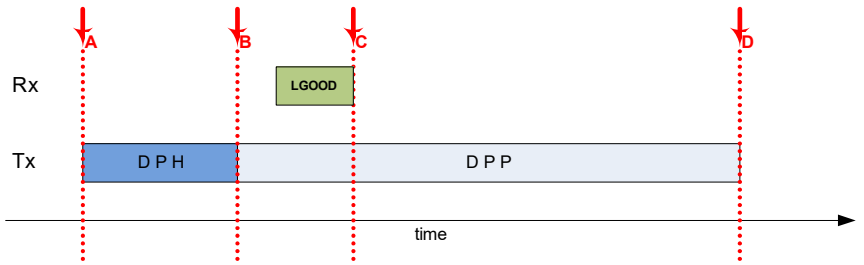
	Prior to Point "A"	at Point "A"	at Point "B"	at Point "G"
status	RETRY	ACTIVE	PARTIAL_ACCEPT	RETRY
header_status	RETRY	ACTIVE	ACCEPT	ACCEPT
payload_status	DISABLED	DISABLED	DISABLED	DISABLED
payload_presence	PAYLOAD_NOT_PRESENT	PAYLOAD_NOT_PRESENT	PAYLOAD_NOT_PRESENT	PAYLOAD_NOT_PRESENT

5.2.6.4 Receiving LGOOD

5.2.6.4.1 During DPP Tx

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

Figure 5-21 Packet Chronology: Receiving LGOOD During DPP Tx



At points "A" through "C" the events are identical to Figure 5-5 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 ovm_event end_event (ovm_transaction's end_event field) on packet

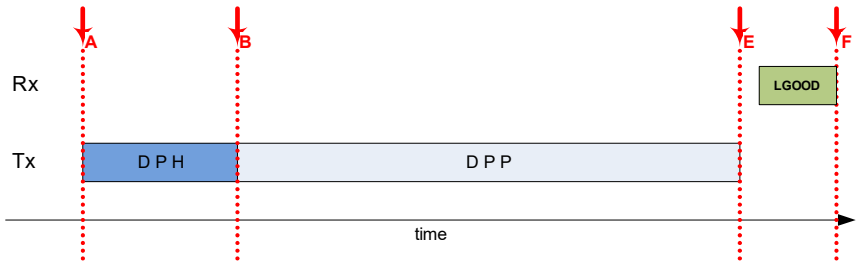
Table 5-18 Status Attribute Chronology: Receiving LGOOD During DPP Tx

	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

5.2.6.4.2 After DPP Tx Complete

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

Figure 5-22 Packet Chronology: Receiving LGOOD After DPP Tx Complete



At points "A" and "B" the events are identical to Figure 5-5 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 ovm_event end_event (ovm_transaction's end_event field) on packet

Table 5-19 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

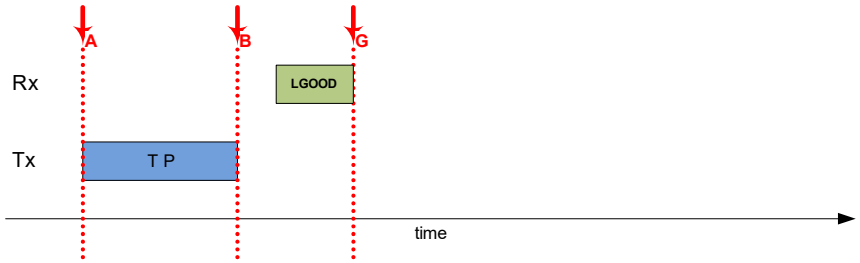
Table 5-19 Status Attribute Chronology: Receiving LGOOD After DPP Tx Complete

	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

5.2.6.4.3 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 5-23 Receiving LGOOD After HP, No DPP



At points "A" and "B" the events are identical to Figure 5-8 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 ovm_event end_event (ovm_transaction's end_event field) on packet

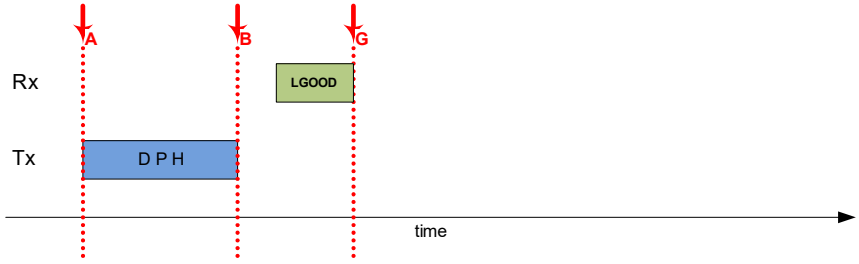
Table 5-20 Status Attribute Chronology: Receiving LGOOD After HP, No DPP

	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

5.2.6.4.4 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 5-24 Packet Chronology: Receiving LGOOD After Data Packet Replay



Events at points "A" and "B" are identical to Figure 5-9 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 ovm_event end_event (ovm_transaction's end_event field) on packet

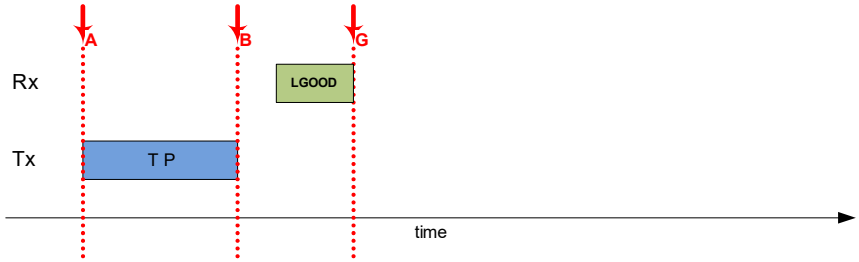
Table 5-21 Status Attribute Chronology: Receiving LGOOD After Data Packet Replay

	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

5.2.6.4.5 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 5-25 Packet Chronology: Receiving LGOOD After Non-Data Packet Replay



At point "A" and "B" the events are identical to Figure 5-5 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 ovm_event end_event (ovm_transaction's end_event field) on packet

Table 5-22 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

5.2.7 Related Topics About Link Component

FOR INFO ABOUT:	SEE:
• Link service commands	• The HTML class reference.
• Factories, callbacks, and ports	• Complete list of OVM factories, callbacks, and ports used by the Link Component in the HTML class reference.

5.3 Physical Component

USB VIP Physical components are component objects in a OVM-compliant verification environment. The USB Physical component object is extended from the ovm_component class. This object implements all of the methods specified by OVM for the ovm_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 OVM events (e.g., begin_event) as well as several USB VIP specific ovm_events (i.e. POWER_ON_RESET). For those ovm_events where the current sequence_item is pertinent, a handle to the current sequence_item is included as part of the ovm_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.

5.3.1 Physical Layer Feature Support

Physical Layer Features 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

5.3.2 Data Flow Support

5.3.2.1 Physical Component Ports

OVM 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 use by the sequencer to send sequences into the physical layer. You cannot connect to them. They are only used by the sequencer. Consult OVM 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 physcial layer.

OVM 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.

OVM 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.

5.3.2.2 Signal Interfaces

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

5.3.2.2.1 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:

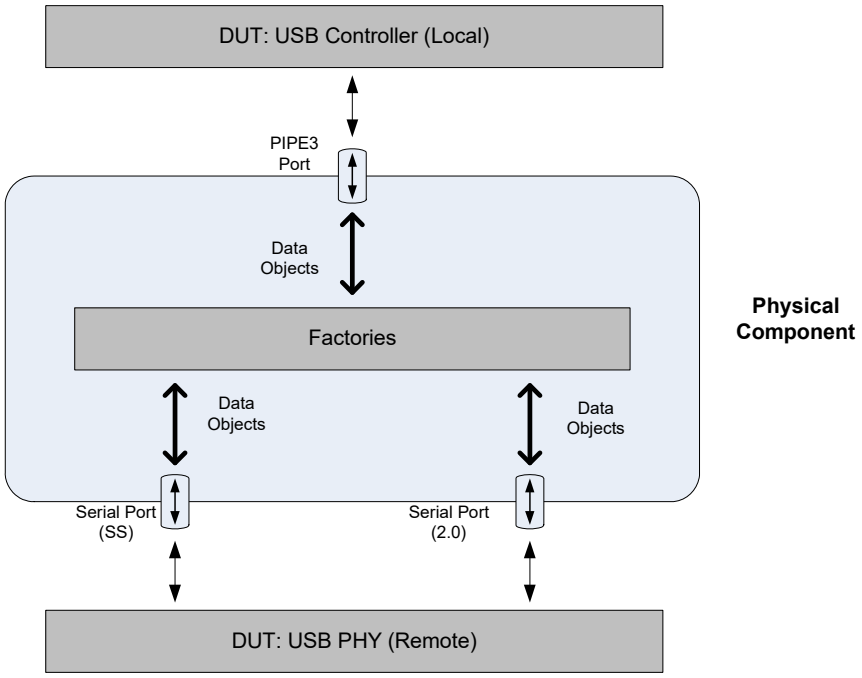
- interface svt_usb_pipe3_dut_mac_if().
- interface svt_usb_pipe3_dut_phy_if().

5.3.2.2.2 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 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 5-26 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 for a description of valid Physical transaction configurations that utilize signal interfaces.

5.3.2.3 Data Objects

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

Sequence Item Data Objects 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.

5.3.2.4 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

5.3.3 Interface Options

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

5.3.3.1 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. When connecting Link layer and a remote Physical Layer components, ports typically exchange information between the components.

5.3.3.2 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.

5.3.3.3 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.

5.3.3.4 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.

5.3.4 Interface File Features

Interface files implement features that OVM 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

5.3.5 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 componenet uses OVM 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 ovm_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.

5.3.5.1 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*.

5.3.6 Exception Support

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

5.3.6.1 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

5.3.6.2 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

5.3.7 OVM Event Support

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

The following are examples for using physical–layer ovm_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.

5.3.8 Physical Component Callbacks

The svt_usb_physical_callback class is extended from the svt_xactor_callbacks base class, which is extended from the ovm_callback class. These objects implement all of the methods specified by OVM for the ovm_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*

5.3.9 Related Topics About Physical Component

FOR INFO ABOUT	SEE
Physical service commands	The HTML class reference.
Factories, callbacks, and ports	Complete list of OVM factories, callbacks, and ports used by the Physical component in the HTML class reference.

6 Using the USB Verification IP

This chapter presents OVM 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 OVM usage, and highlights the concepts and techniques described. These techniques can be used with any of the VIP products.

6.1 Configuring VIP Using coreConsultant

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

Requirements

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

- Latest version of coreConsultant installed.

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

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

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

Configuring VIP Using coreConsultant

To configure VIP using coreConsultant, complete the following steps:

1. Open coreConsultant.
2. Select **File > Install coreKit** and provide the path to the USB3Config.coreKit file.
3. In the **Specify Configuration tab** in the "Activity List", you can change the default configuration values.

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

4. Ensure that <design_dir> is set to $DESIGNWARE_HOME and then run the dw_vip_setup utility. For example:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path $DESIGNWARE_HOME -svtb -e
usb_svt/tb_usb_svt_ovm_basic_sys
```

5. In the **Edit > Tool Installation Roots > VCS** text box, add VCS_HOME.
6. Click **Apply** to generate the configuration file.
7. Click **Results File** to see the output file and validation report.

The <configuration_name>.cfg is the file generated that can be loaded into VIP. The <configuration_name>.report shows the validation result.

To load the *.cfg file into the VIP, use the load_prop_vals() method. The method definition is on svt_data object and user should call it on svt_usb_agent_configuration object instance. The following code snippet illustrates how to use the load_prop_vals() method to load a configuration.

```
      svt_usb_agent_configuration cfg = new();
      if (cfg.load_prop_vals(filename)) begin
        $display("Successfully loaded svt_usb_agent_configuration using '%0s'.",
filename);
        if (cfg.is_valid(0)) begin
          $display("svt_usb_agent_configuration loaded using '%0s' is valid.",
filename);
        end else begin
          $display("ERROR: svt_usb_agent_configuration loaded using '%0s' is NOT
valid.", filename);
        end
      end else begin
        $display("ERROR: Failed attempting to load svt_usb_agent_configuration using
'%0s'.", filename);
      end
    end
```

For more information, see <workspace>/sim/in_valid_test.sv.

6.2 Creating Transactions Using OVM Sequencers

The USB VIP sequence collections define sequences consistent with OVM. 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 ovm_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 OVM sequencers have a number of other features such as 'do' macros, virtual sequencers, and sequence libraries.

6.3 SuperSpeed Low Power Entry Support

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.

6.3.1 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.

6.3.1.1 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) x (u1_timeout) x (1 µs)

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.

6.3.1.2 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) x (u2_timeout) x (256 µs)

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.

6.3.2 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.

6.3.3 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);
```

6.4 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.

6.4.1 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 6-1 lists the default functional coverage features provided with USB3.0 Verification IP.

Table 6-1 Default Functional Coverage Features

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

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

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_ovm_class_reference/html/index.html

6.4.2 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
- Error Condition Sequences
- Packet Field Range Coverpoints
- Data Collection Coverpoints

6.4.2.1 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

6.4.2.2 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

6.4.2.3 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

6.4.2.4 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, rty_bit, after_inactive_flow_control, and preexisting_flow_control_state

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

6.4.2.5 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)

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.

6.4.3 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 };}
```

6.4.4 Default Functional Coverage Class Hierarchy

The basic classes are:

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

6.4.4.1 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.

6.4.4.2 Component Coverage Data Callbacks

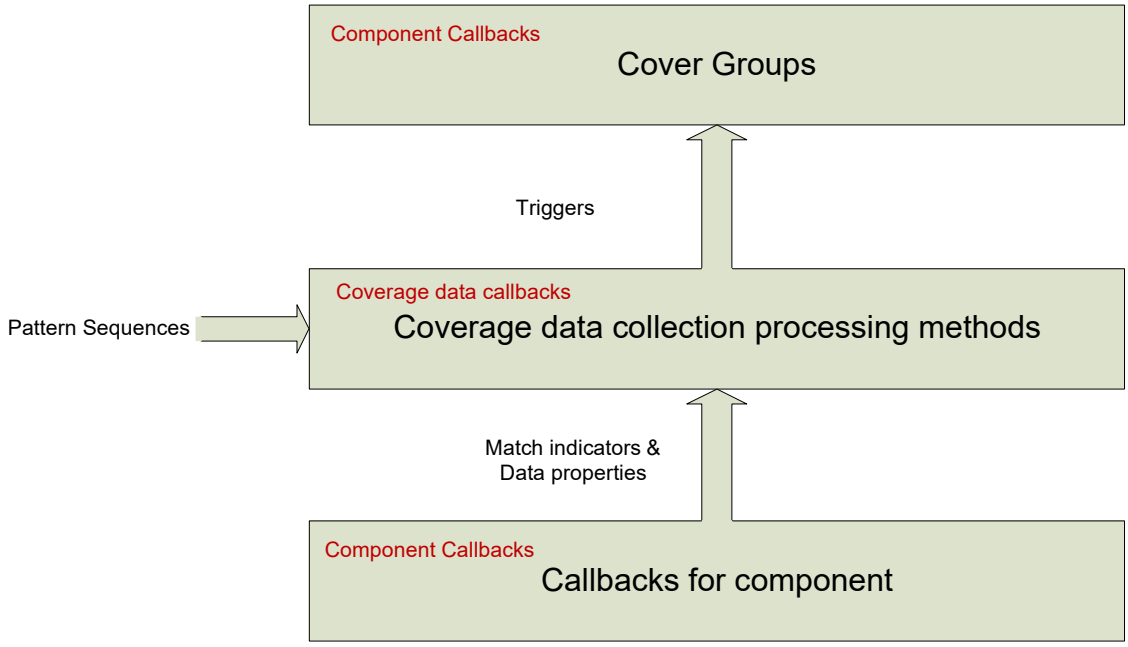
This callback class defines default data and event information that are used to implement the coverage groups. The naming convention uses “def_cov_data” in the class names for easy identification of these classes. This class also includes implementations of the coverage methods that respond to the coverage 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.

6.4.4.3 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 6-1.

Figure 6-1 Default Functional Coverage Class Hierarchy



6.4.5 Coverage Callback Classes

The following naming convention is used for coverage callback classes:

```
svt_usb_<layer>_<speed>_<VIP config>_def_cov_<type_>callbacks
```

Where:

Table 6-2

Convention	Description
`<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 6-3 lists the coverage callback classes.

Table 6-3 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

Table 6-3 Coverage Callback Classes

Class Name	Type	Layer	Configuration
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

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_ovm_class_reference/html/index.html

6.4.6 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

6.4.6.1 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.

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

6.4.7 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>
```

6.5 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.

6.5.1 VIP Acting as a Host

Table 6-4 describes the VIP behavior when it is acting as a host.

Table 6-4 Behavior of VIP Acting as a Host for Different Endpoint Configurations and Transfers

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

6.5.2 VIP Acting as a Device

Table 6-5 describes the VIP behavior when it is acting as a host.

Table 6-5 Behavior of VIP Acting as a Device for Different Endpoint Configurations and Transfers

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

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

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

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

LTSSM state is SS.Disabled

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

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

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

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

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

6.6.1 LTSSM state is Rx.Detect.Reset

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

6.6.2 LTSSM state is Rx.Detect.Active

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

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

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

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

Note

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

6.6.3 LTSSM state is Polling.LFPS

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

- polling_lfps_burst_time
- polling_lfps_repeat_time
- tx_lfps_duty_cycle
- tx_lfps_period

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

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

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

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

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

6.6.4 LTSSM state is Polling.RxEQ

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

6.6.5 LTSSM state is Polling.Active

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

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

6.6.6 LTSSM state is Polling.Configuration

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

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

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

6.6.7 LTSSM state is Polling.Idle

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

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

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

6.7 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
- Session Request Protocol
- Role Swapping Using the HNP Protocol
- Attach Detection Protocol

6.7.1 OTG Interface Signals

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

6.7.1.1 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 6-6 lists and describes the OTG link interface signals.

Table 6-6 OTG Link Interface Signals

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

Table 6-6 OTG Link Interface Signals

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

6.7.1.2 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.

6.7.1.2.1 ADP Interface Signals

Table 6-7 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 6-7 OTG ADP Serial Interface Signals

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

6.7.1.2.2 SRP Interface Signals

Table 6-8 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 6-8 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 • 1'b0: VIP detected/driven VBUS <= VOTG_SESS_VLD • 1'b1: VIP detected/driven VBUS > VOTG_SESS_VLD
dut_sess_vld	Input	High	0	1	DUT VBUS level detector - VOTG_SESS_VLD • 1'b0: DUT detected/driven VBUS <= VOTG_SESS_VLD • 1'b1: DUT detected/driven VBUS > VOTG_SESS_VLD

6.7.2 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.

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

Figure 6-2 explains the SRP process flow when the VIP is configured as an A-device.

Figure 6-2 SRP Protocol Flow when the VIP is Configured as an A-Device

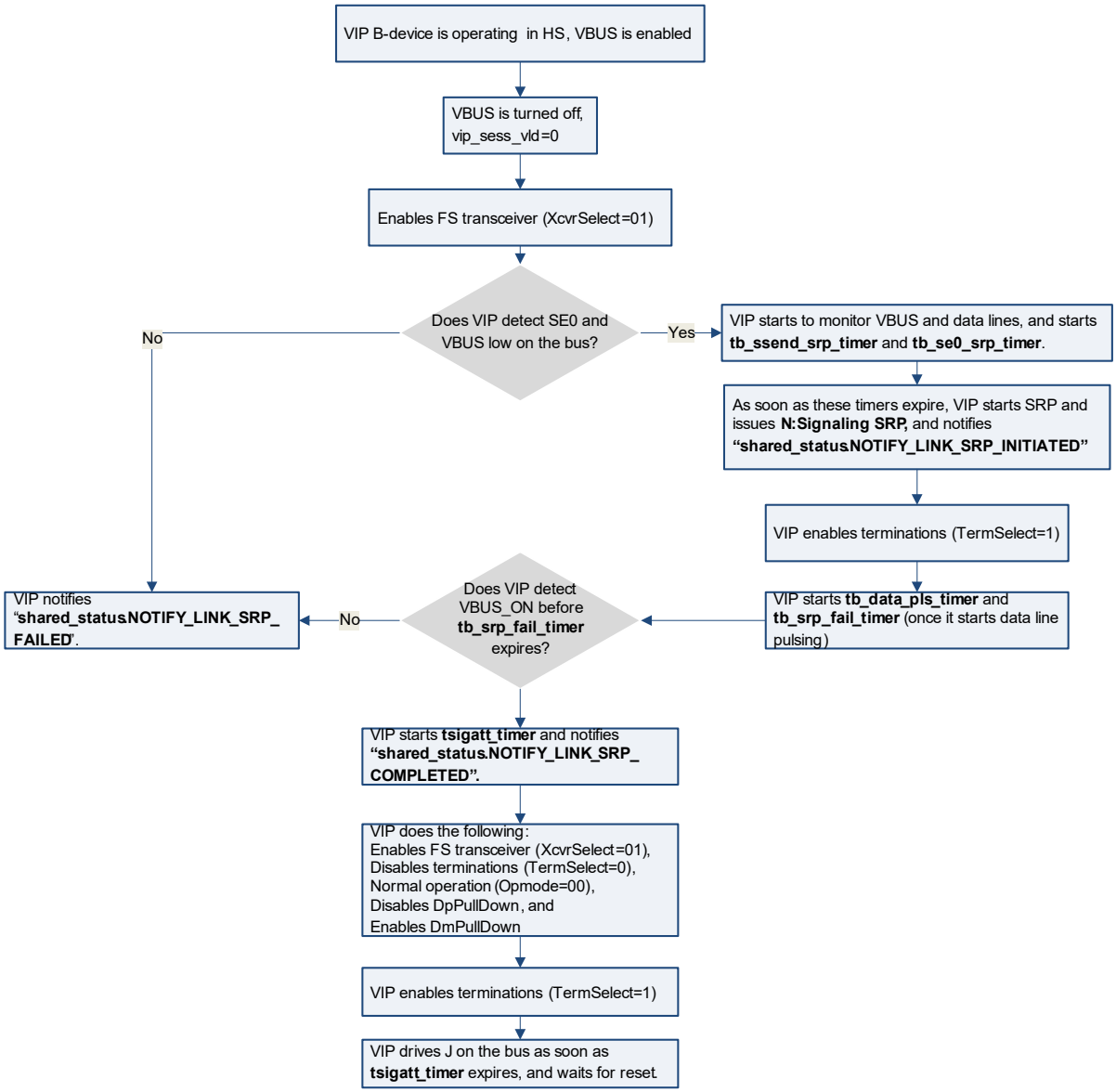


Table 6-9 expands on the abbreviated messages that are used in Figures 6-2 and 6-3.

Table 6-9 Legend Explaining the Abbreviated Messages

Abbreviated Messages Used in Figure 6-2	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

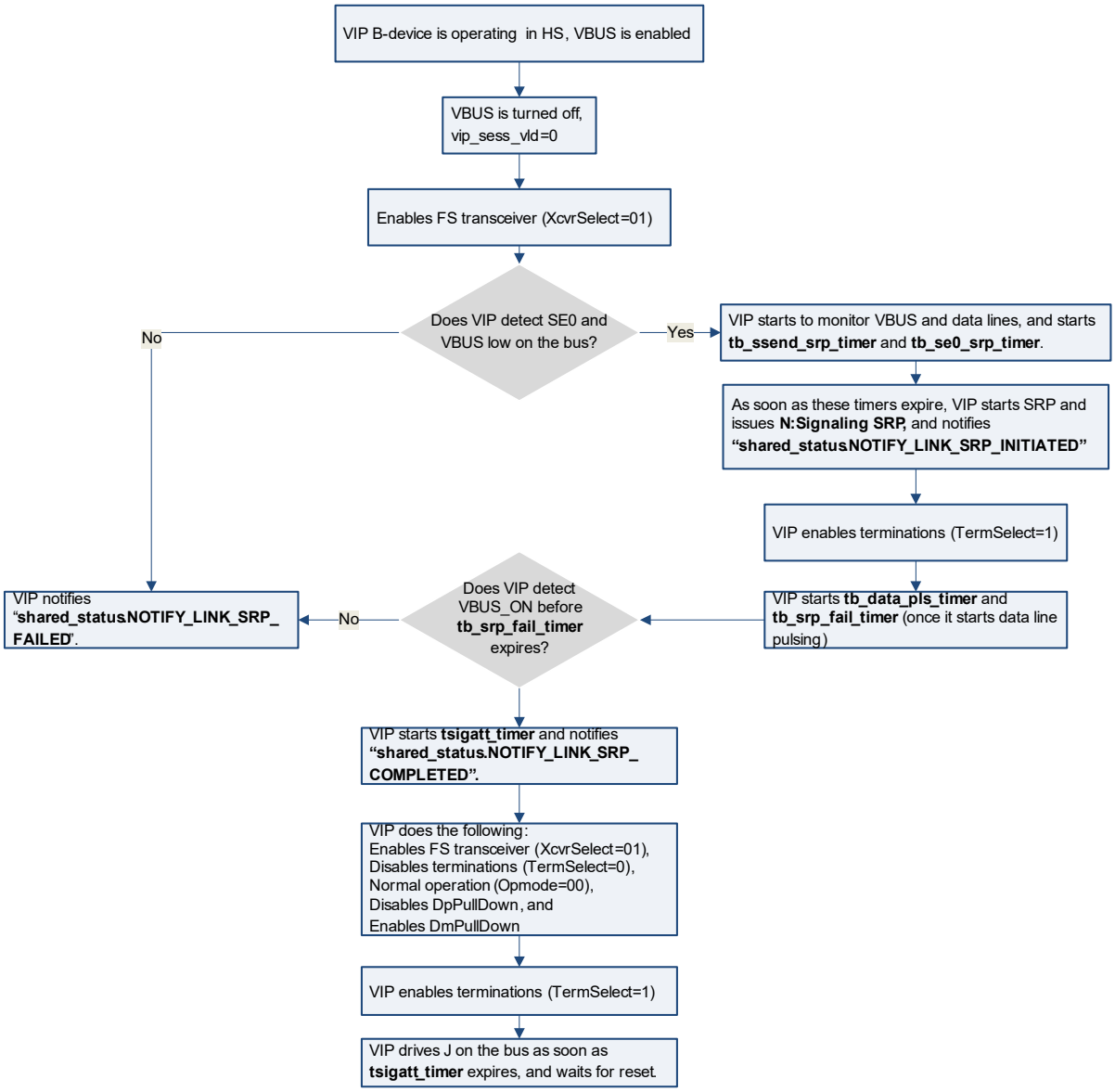
Table 6-9 Legend Explaining the Abbreviated Messages

Abbreviated Messages Used in Figure 6-2	Actual Messages Issued by VIP
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 6-3	**Actual Message Issued by VIP**
N: signaling SRP	B-device signaling SRP by generating data line pulsing.

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

Figure 6-3 explains the SRP process flow when the VIP is configured as a B-device. The abbreviated messages listed in Figure 6-3 are expanded in detail in Table 6-9.

Figure 6-3 SRP Protocol Flow when the VIP is Configured as a B-Device



6.7.3 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.

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.

6.7.3.1 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 ovm_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.

6.7.3.2 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 ovm_event.
   - If the role swap fails the agent does not change anything, but notifies the testbench through the NOTIFY_OTG_ROLE_SWAP_FAILED ovm_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 ovm_event.
   - ✦ If the role swap fails the agent does not change anything, but notifies the testbench through the NOTIFY_OTG_ROLE_SWAP_FAILED ovm_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.

6.7.3.3 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.

6.7.3.3.1 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 agent 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.

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

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

- ❖ The value of the b_hnp_supported bit in the VIP's device configuration does not automatically result in the proper bit being set in the values returned for the GetDescriptor() control transfer. It is up to the testbench to interpret the control transfer, and control the return value based on the value of the b_hnp_supported bit in the VIP's device configuration.

- When a SetFeature() control transfer is received who's intent is to set the b_hnp_enable bit in the VIP, the corresponding bit in the VIP's device status is not automatically set. It is up to the testbench to interpret the control transfer and then call the 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.

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

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

Figure 6-4 HNP Flow When VIP is Configured as an A-Device

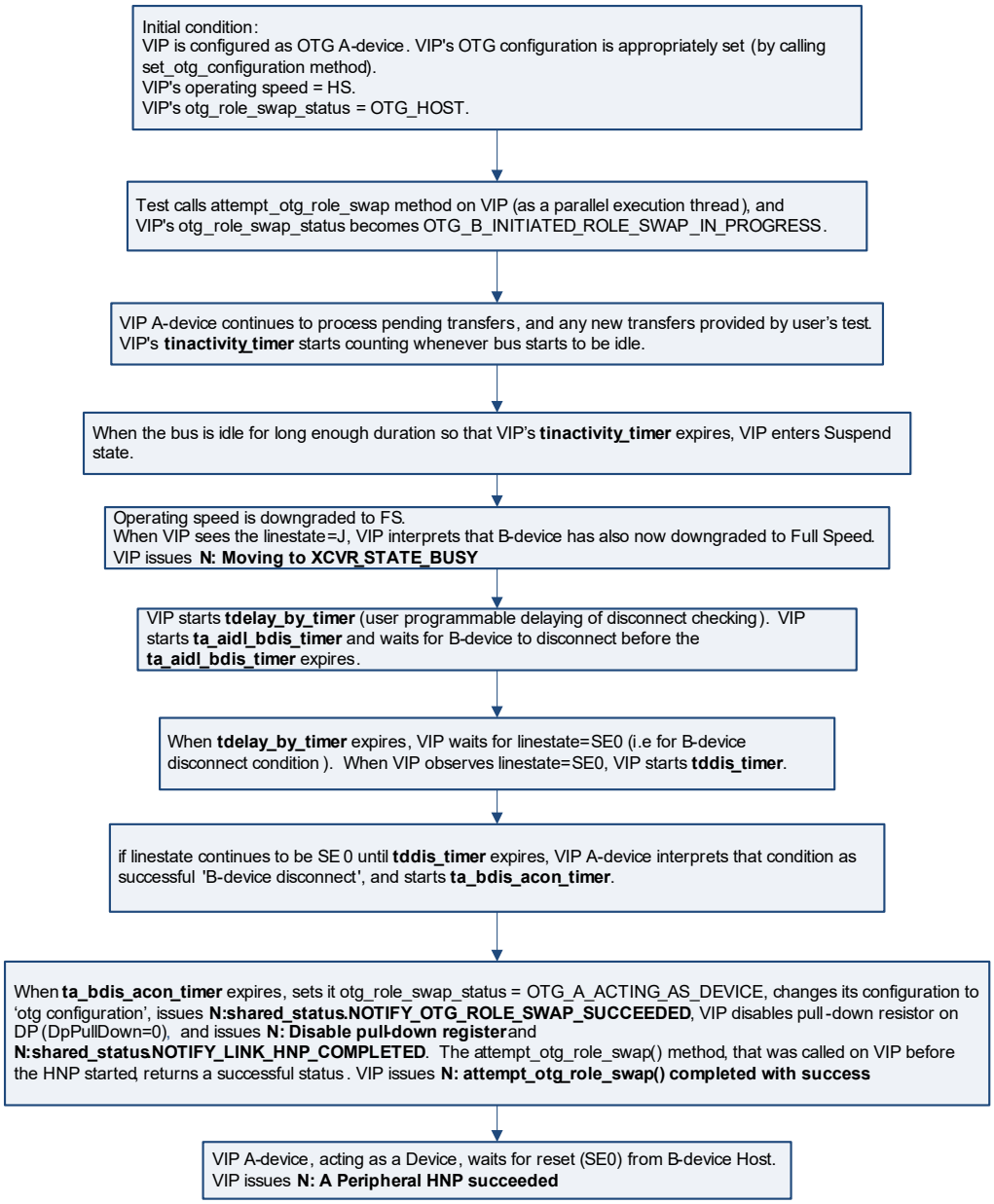


Figure 6-5 Reverse HNP Flow when the VIP A-Device Reverts Back to Acting as a Host

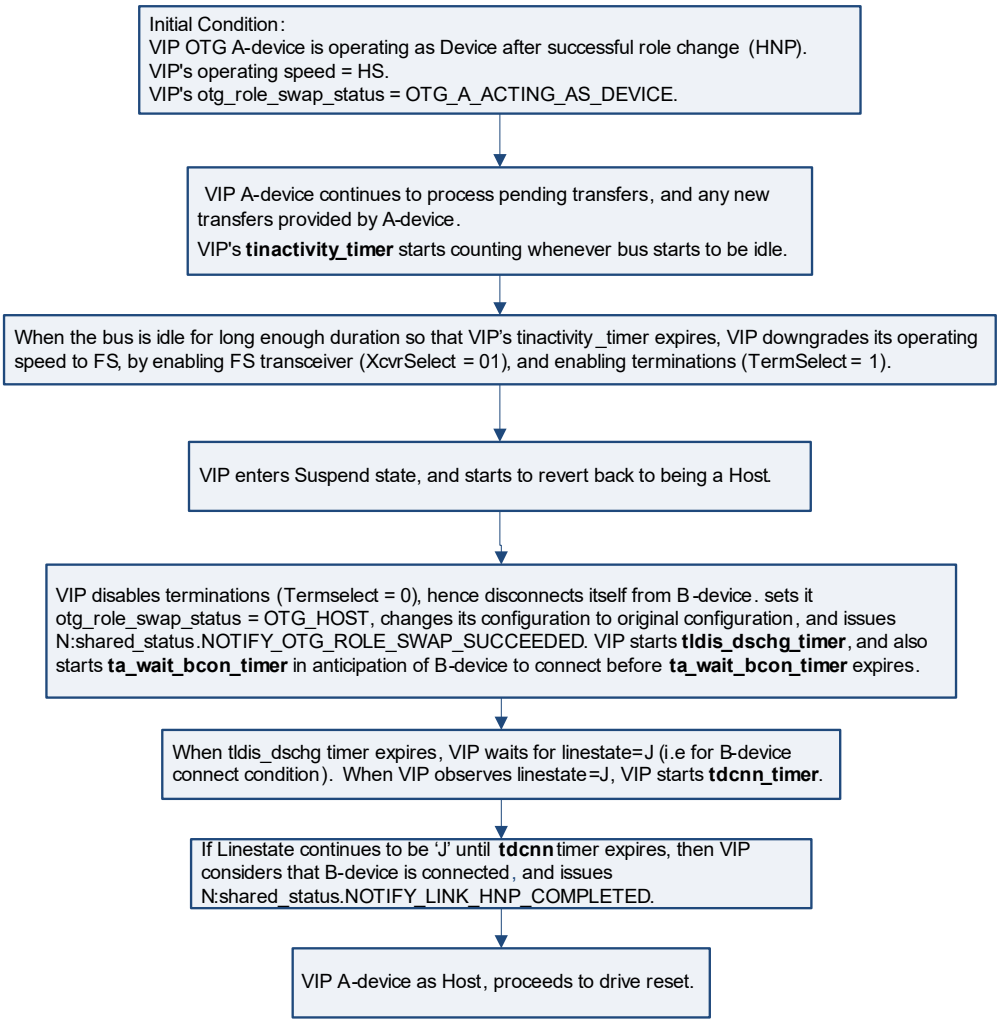


Table 6-10 expands on the abbreviated messages that are used in Figures 6-4 and 6-5.

Table 6-10 Legend Explaining the Abbreviated Messages Used in Figures 6-4 and 6-5

Abbreviated Messages Used in Figure 6-4	**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 6-5	**Actual Message Issued by VIP**
N: Enable FS transceiver	Enable FS transceiver (XcvrSelect = 01)
N: Enable terminations	Enable terminations (TermSelect = 1)

Table 6-10 Legend Explaining the Abbreviated Messages Used in Figures 6-4 and 6-5

Abbreviated Messages Used in Figure 6-4	Actual Messages Issued by VIP
N: Disable terminations	Disable terminations (TermSelect = 0)
N: Reverse HNP Complete	Reverse HNP Complete on A-Device

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

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

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

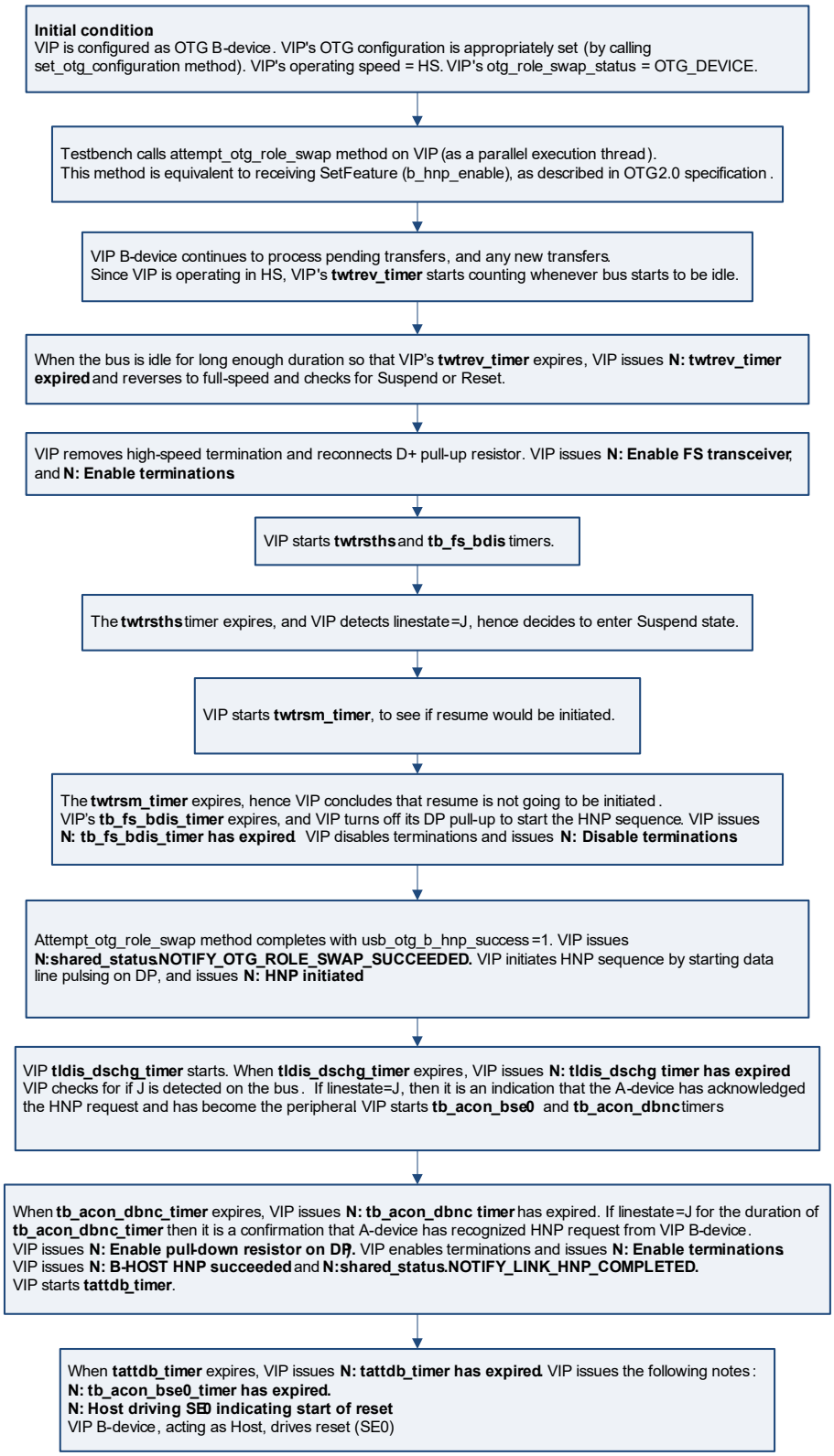


Figure 6-7 Reverse HNP Flow When the VIP B-Device Reverts Back to Acting as a Device

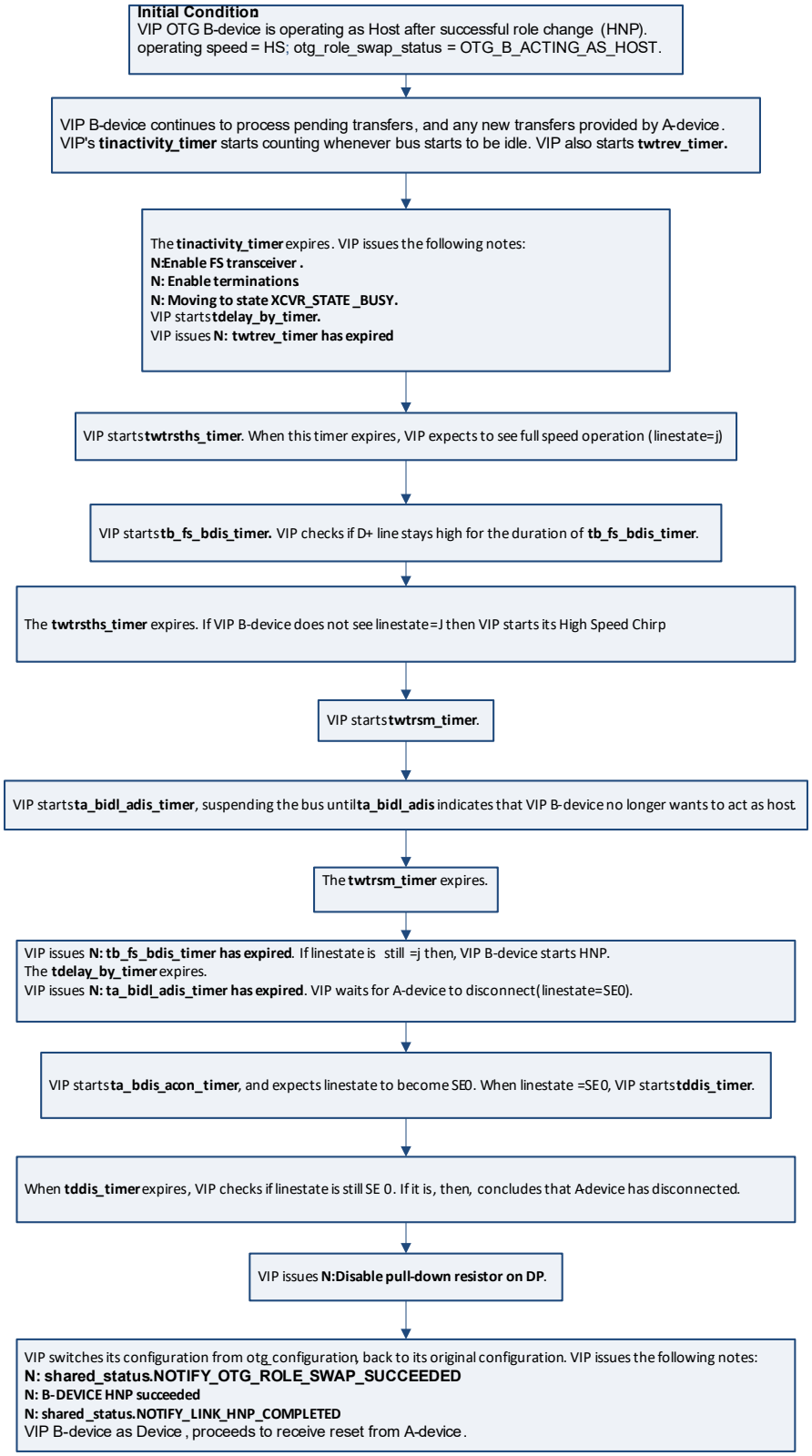


Table 6-11 expands on the abbreviated messages that are used in Figures 6-6 and 6-7.

Table 6-11 Legend Explaining the Abbreviated Messages Used in Figures 6-6 and 6-7

Abbreviated Messages Used in Figure 6-6	Actual Messages Issued by VIP
N: twtrev_timer expired	N: twtrev_timer expired
N: Enable FS transceiver	N: Enable FS transceiver (XcvrSelect = 01)
N: Enable terminations	N: Enable terminations (TermSelect = 1)
N: tb_fs_bdis_timer has expired	N: tb_fs_bdis_timer has expired
N: Disable terminations	N: Disable terminations (TermSelect = 0)
N: HNP initiated	N: HNP initiated
N: Enable pull-down resistor on DP	N: Enable pull-down resistor on DP (DpPullDown =1)
N: B-HOST HNP succeeded	N: B-HOST HNP succeeded
Additional Abbreviated Messages Used in Figure 6-7	**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

6.7.4 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

6.7.4.1 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)

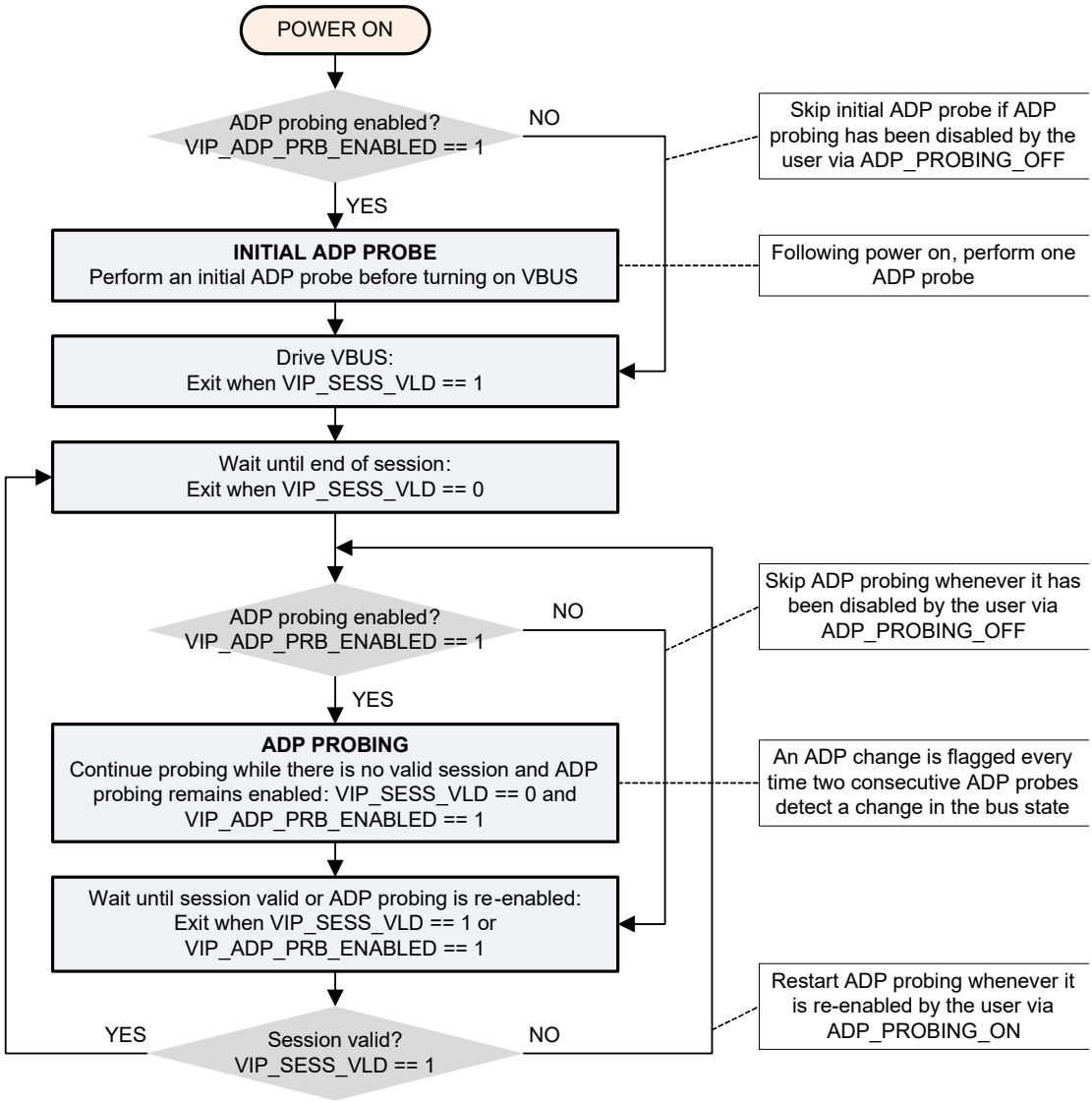
6.7.4.1.1 ADP Probing When VIP is Configured as an A-Device

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 6-8 illustrates a simple ADP Probing process for VIP as an A-device or an Embedded Host using a waveform.

Figure 6-8 ADP Probing Flow when VIP is Configured as an A-Device or Embedded Host



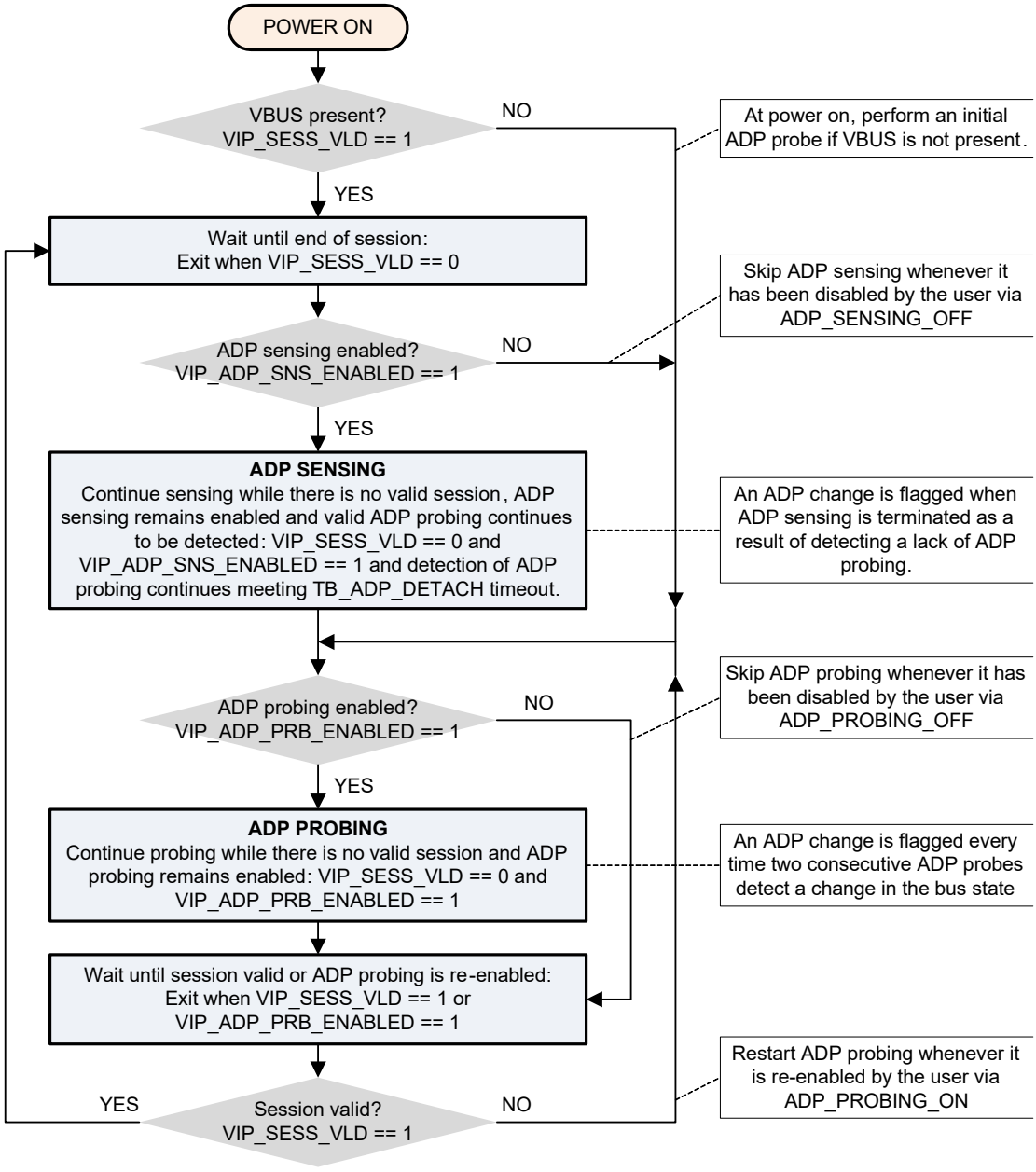
6.7.4.1.2 ADP Probing When VIP is Configured as a B-Device

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 6-9 explains the ADP control when VIP is configured as a B-device.

Figure 6-9 ADP Probing and Sensing Flow when the VIP is Configured as a B-Device



6.7.4.2 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

6.7.4.2.1 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 6-9.

6.7.4.3 ADP OVM Evemts

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

6.7.4.4 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.

6.8 HSIC Overview

The this section covers HSIC Support.

6.8.1 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.

6.8.2 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)

6.8.3 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 6-12 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).

Table 6-12 HSIC Configuration Parameters (Continued)

Configuration Parameter	Purpose
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.
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.

6.8.4 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.

6.8.4.1 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.

6.8.4.2 Physical Service Transations

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

- DRIVE_HSIC_ENABLED - Request to a host to "present" the IDLE bus state indicating the HSIC Enabled On state.
- DRIVE_HSIC_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.

6.8.5 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.

6.8.5.1 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).

6.8.5.2 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.

6.8.6 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.

6.8.7 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 6-13 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.

6.8.8 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.

6.8.9 Notifications

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

Table 6-14

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.

Table 6-14

Notification	Use
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.

6.8.10 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.

6.8.11 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.

6.8.12 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:

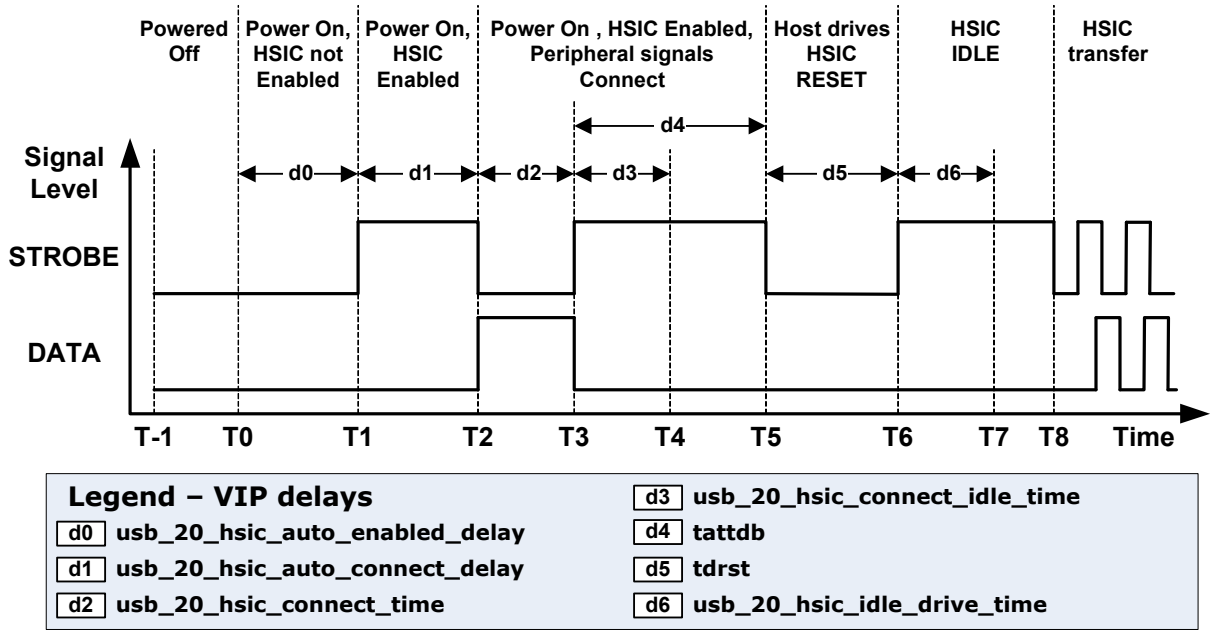
- Discovery Process - The HSIC protocol for establishing a HSIC connection.
- Bus Keepers - The HSIC host feature used to maintain IDLE on the bus.
- Data Inversion - Support for enabling transmission and reception of inverted data.
- "Data EOP Signaling - Support for configuring the end-of-data signaling pattern.
- Modeling Power Off - Support for modeling power off entry and exit.
- Controlling Skew - Support for configuring skew between Strobe and Data signals.

6.8.12.1 Discovery Process

Discovery is the HSIC power up and connection sequence used to establish communication between two HSIC devices. Figure 6-10 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 6-10 Discovery Process with VIP Set Values



6.8.12.2 VIP as Host

Using Figure 6-10 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: • 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: • 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	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 based on the value of usb_20_hsic_auto_enabled_delay: • If usb_20_hsic_auto_enabled_delay is non-negative, the VIP moves to T1 after implementing the delay specified by the value. • If usb_20_hsic_auto_enabled_delay is negative, the VIP will not move to T1 until it receives a DRIVE_HSIC_ENABLED 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	The VIP stops measuring the period CONNECT that is on the bus. • If it detected assertion of the CONNECT bus state for the required minimum period of time (as controlled by the configuration property usb_20_hsic_connect_time_min), the VIP issues the NOTIFY_USB_20_HSIC_CONNECT_DETECTED notification. • If it detected assertion of the CONNECT for less than the minimum requirement, the VIP will issue a usb_20_hsic_connect_check failure and remain in T3 until the VIP is either restarted or powered down. 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.
Exit	The VIP moves to T-1 if it receives a request to power down (via physical service command VBUS_OFF). Based on whether or not the detected CONNECT was valid: • If valid, the VIP moves to T5 after implementing the delay specified by the tattdb. • 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.

6.8.12.3 VIP as Peripheral

Using Figure 6-10as 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.

T-1 - Powered Off state	
Entry	The VIP enters this state under one of the following conditions: • 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: • 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	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: • 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	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 after implementing the delay specified by the usb_20_hsic_connect_time.

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	The VIP issues the NOTIFY_USB_20_HSIC_CONNECT_COMPLETED notification and stops driving the bus. The VIP starts to monitor the Strobe and Data lines for a RESET bus state. 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.

T4 - Power On, HSIC Enabled, Peripheral signals Connect 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 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	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 when it detects the bus transition to the IDLE bus state.

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.

6.8.12.4 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).

6.8.12.5 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.

6.8.12.6 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 6-11 and Figure 6-12. 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 eights of a Strobe-Period period after the last rising edge of Strobe.

Figure 6-11 End of signaling scenarios with DATA low

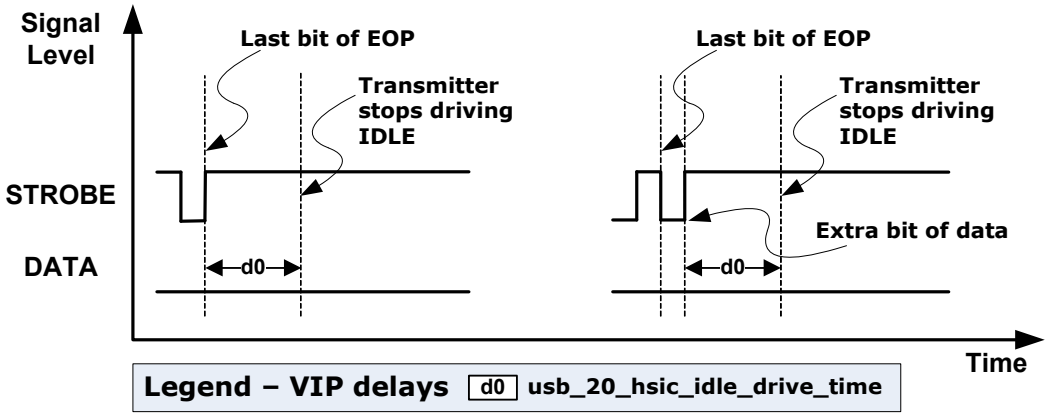
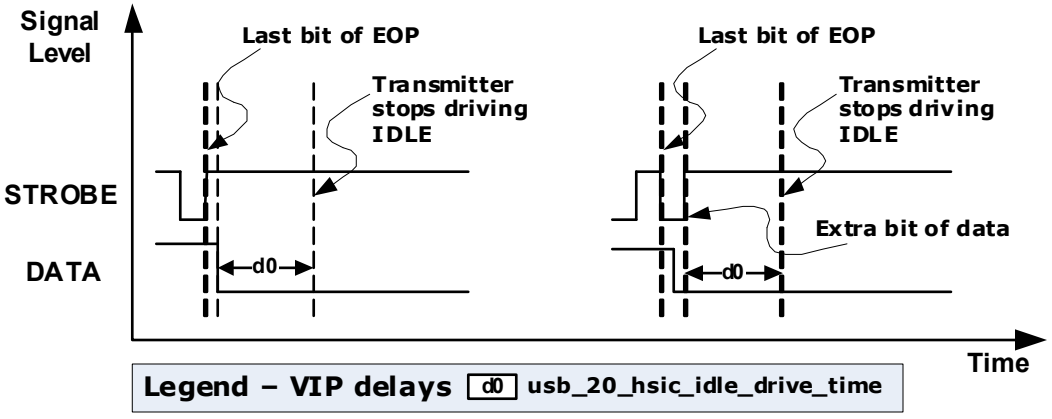
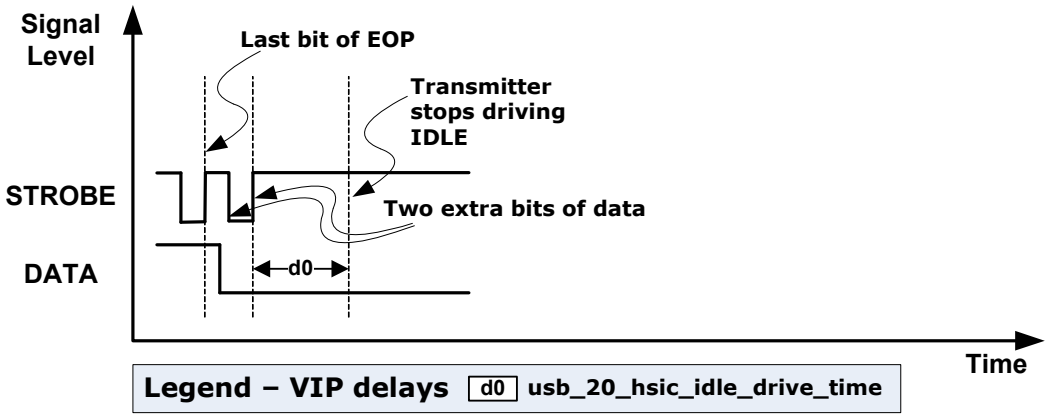


Figure 6-12 End of signaling scenarios with DATA high (fast EOP)



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 6-13.

Figure 6-13 End of signaling scenarios with DATA high (fast EOP disabled)



6.8.12.7 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.

6.8.12.8 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 eight of a Strobe-period prior to driving an edge on Strobe, and holds data for three eights of a Strobe-period following the driving of an edge on Strobe.

6.8.12.9 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.

6.8.12.10 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.

6.9 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.

6.9.1 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.

6.9.1.1 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.

6.9.1.2 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

6.9.2 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.

6.9.2.1 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.

6.9.2.2 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.

6.9.3 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.”

6.9.3.1 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.

6.9.3.2 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

6.9.4 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.”

6.9.4.1 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.

6.9.4.2 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

6.9.5 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.”

6.9.5.1 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.

6.9.5.2 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

6.9.6 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.

6.9.7 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.

6.9.8 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.

6.10 SystemVerilog OVM Example Testbenches

This section describes SystemVerilog OVM example testbenches that show general usage for various applications. A summary of the examples is listed in Table 6-15.

Table 6-15 SystemVerilog Example Summary

Name	Source in design_dir/Description
Basic Example--Demonstrates how to implement a OVM testbench using USB VIP with SS Serial interface. This example consists of a top-level testbench, a Verilog hdl_interconnect, a OVM verification environment, a host and device agent components, one test file with an sequence, and two directed tests.	
tb_usb_svt_ovm_basic_sys	examples/sverilog/usb_svt/tb_usb_svt_ovm_basic_sys/
Intermediate Example--Builds on the Basic OVM Example by adding coverge and a scoreboard.	
tb_usb_svt_ovm_intermediate_syst	examples/tb_usb_svt_ovm_intermediate_syst

7 Verification Topologies

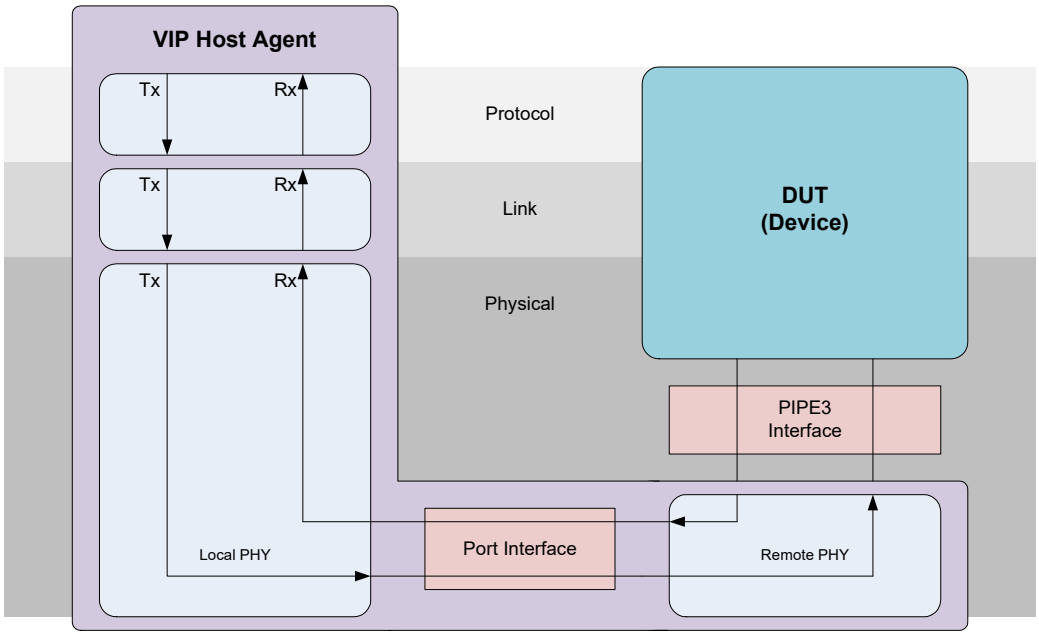
This chapter presents several agent testbench topologies. In addition to a brief description of the testbench topologies, each section lists configuration parameters required to instantiate the VIP agent.

7.1 USB VIP Host and DUT Device Controller

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

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 7-1 USB VIP Host and DUT Device Controller



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;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT;
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;
```

- **remote_cfg – representing the remote phy's properties**

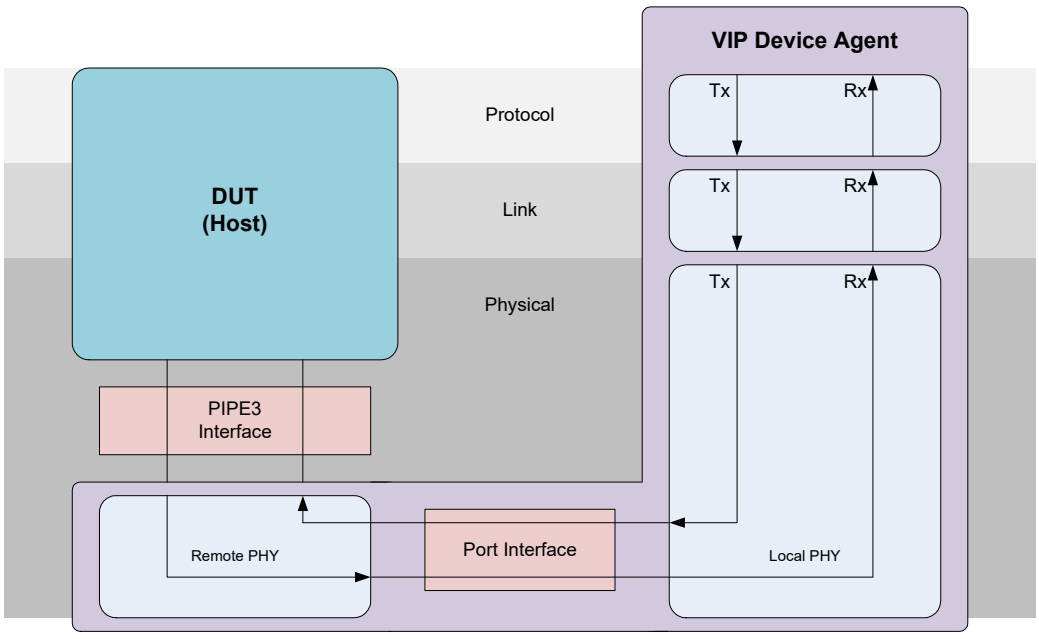
```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS;
component_subtype_enum component_subtype = PHY;
usb_ss_signal_interface_enum usb_ss_signal_interface = PIPE_IF;
usb_capability_enum usb_capability = USB_SS_ONLY;
```

7.2 USB VIP Device and DUT Host Controller

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

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 7-2 USB VIP Device and DUT Host Controller



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;
```

```
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_PORT;
usb_capability_enum usb_capability = USB_SS_ONLY;
svt_usb_host_configuration remote_host_cfg;
svt_usb_device_configuration local_device_cfg[$];
```

- **remote_cfg (representing the remote phy's properties)**

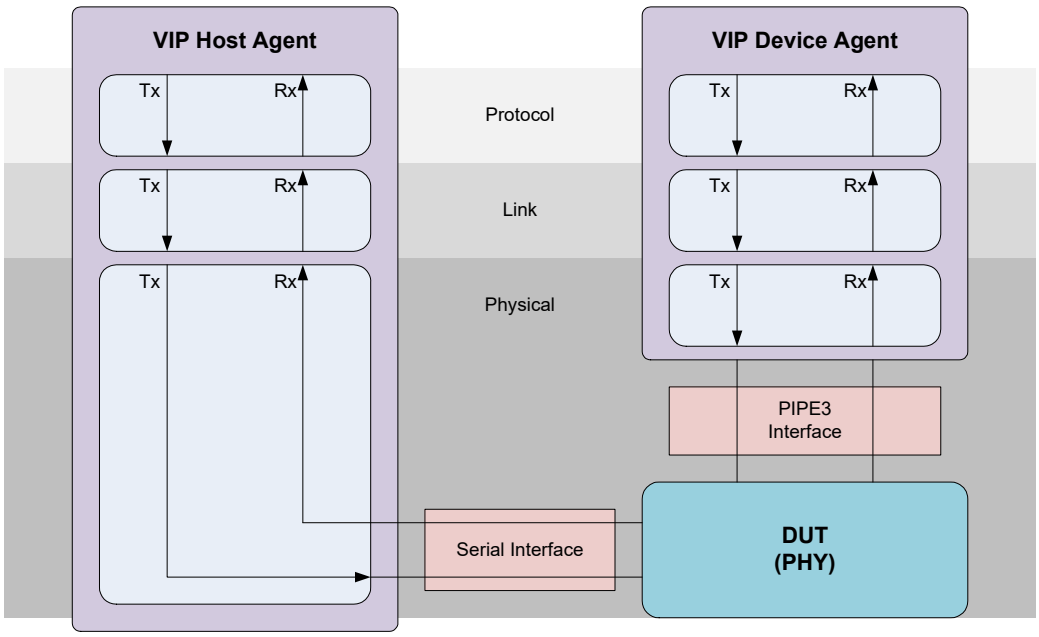
```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS;
component_subtype_enum component_subtype = PHY;
usb_ss_signal_interface_enum usb_ss_signal_interface = PIPE3_IF;
usb_capability_enum usb_capability = USB_SS_ONLY;
```

7.3 USB VIP Host and DUT Device PHY

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

Each VIP instance requires a valid cfg object of type svt_usb_agent_configuration.

Figure 7-3 USB3 VIP Host 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;
component_subtype_enum component_subtype = MAC
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)
```

- **cfg object (representing the device vip)**

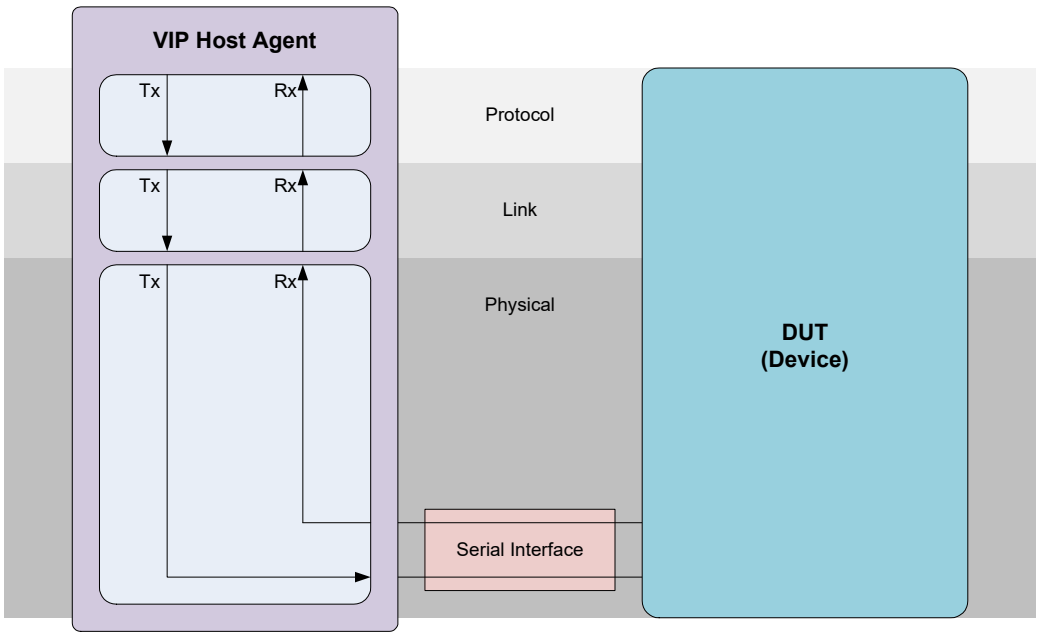
```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = PIPE_IF;
usb_capability_enum usb_capability = USB_SS_ONLY;
svt_usb_device_configuration local_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)
```

7.4 USB VIP Host and DUT Device

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

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 7-4 USB3 VIP Host and DUT Device



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**

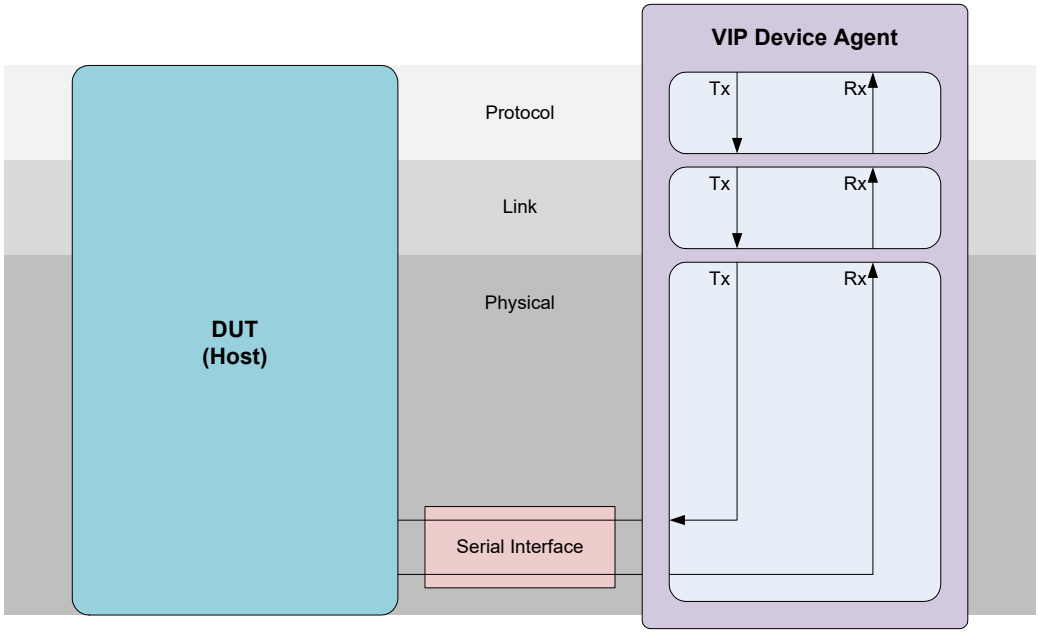
```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::HS;  (other speeds FS/LS)
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_20_SERIAL_IF;
usb_capability_enum usb_capability = 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)
```

7.5 USB VIP Device and DUT Host

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

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 7-5 USB3 VIP Device and DUT Host



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::DEVICE;
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_host_cfg;
```

```
svt_usb_device_configuration local_device_cfg[$];
```

- **Simulating a 2.0 (only) serial bus**

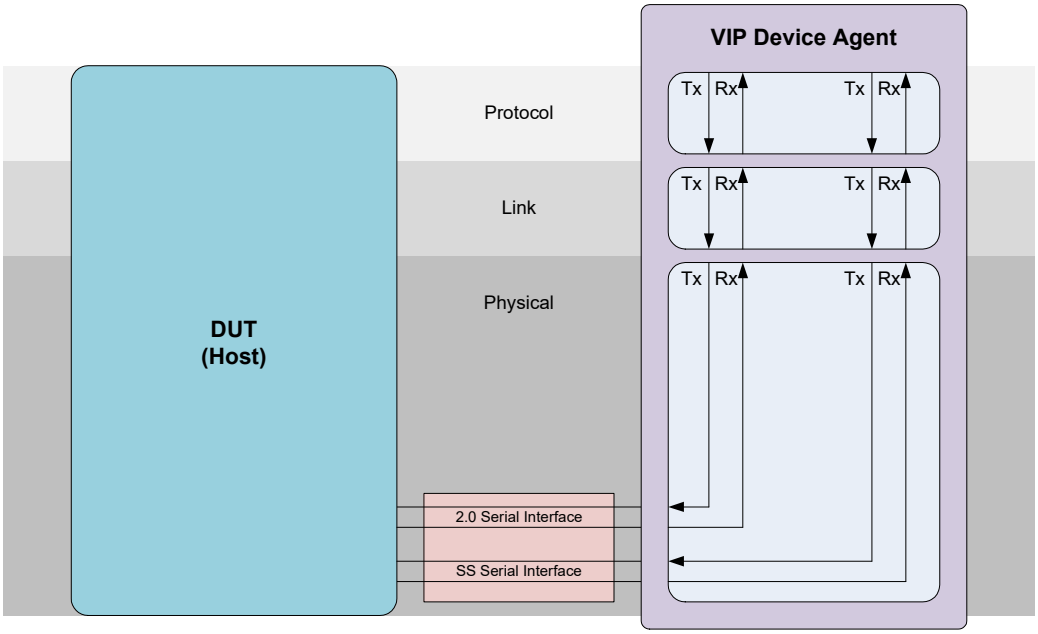
```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::HS;  (other speeds FS/LS)
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_20_SERIAL_IF;
usb_capability_enum usb_capability = USB_20_ONLY;
svt_usb_host_configuration remote_host_cfg ;
svt_usb_device_configuration local_device_cfg ;
int local_device_cfg_size = 1; (size needs to be greater than or equal to 1)
```

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

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

The VIP instance requires a valid cfg object of type svt_usb_agent_configuration.

Figure 7-6 USB3 VIP Device and DUT Host – Concurrent SS and 2.0 Traffic



Implementation of this topology requires the setting of the following properties:

- **Simulating a SS and 20 serial bus**

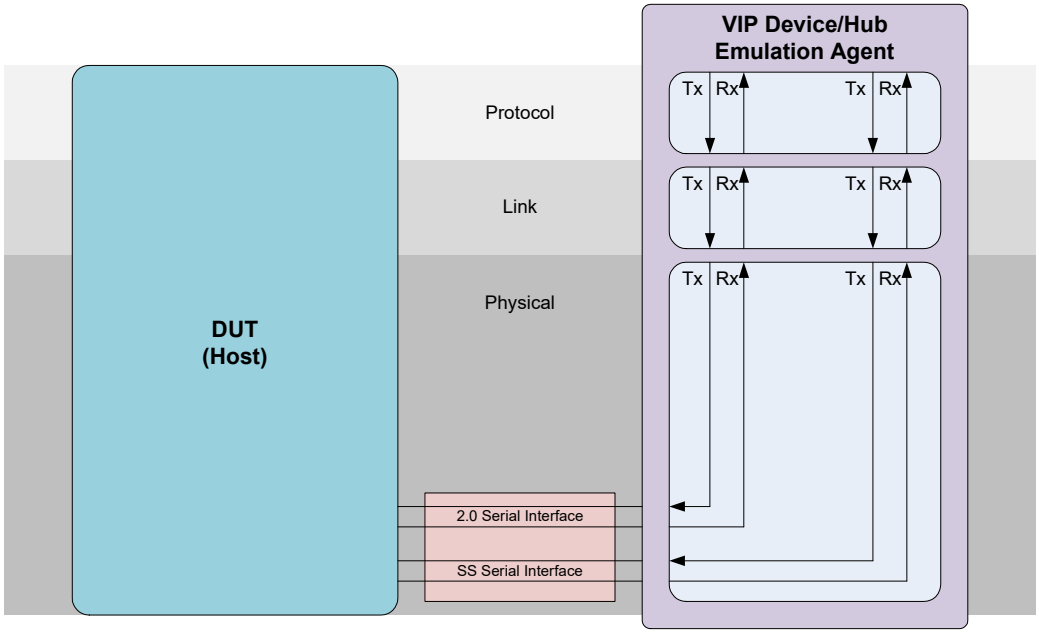
```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_SERIAL_IF;
usb_ss_signal_interface_enum usb_20_signal_interface = USB_20_SERIAL_IF;
usb_capability_enum usb_capability = USB_SS_CAPABLE;
svt_usb_device_configuration remote_host_cfg;
svt_usb_device_configuration local_device_cfg[$];
int local_device_cfg_size = 1; (size needs to be greater than or equal to 1)
```

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

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

The VIP instance requires a valid cfg object of type svt_usb_agent_configuration.

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



Implementation of this topology requires the setting of the following properties:

- **Simulating a SS and 20 serial bus between Host downstream and Hub's upstream port.**

```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
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_ss_signal_interface_enum usb_20_signal_interface = USB_20_SERIAL_IF;
usb_capability_enum usb_capability = USB_SS_CAPABLE;
svt_usb_device_configuration remote_host_cfg;
svt_usb_device_configuration local_device_cfg[$];
int local_device_cfg_size = 1; (size needs to be greater than or equal to 1)
bit hub_emulation_mode = 1'b1;
```

8 VIP Tools

8.1 Using Native Protocol Analyzer for Debugging

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

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

8.1.1 Prerequisites

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

Compile Time Options

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

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

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

$VERDI_HOME/doc/linking_dumping.pdf

Configuration Variable Setting:

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

For Example:

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

Similarly for USB receiver:

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

8.1.2 Invoking Protocol Analyzer

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

Post-processing Mode

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

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

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

Interactive Mode

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

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

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

8.1.3 Documentation

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

$VERDI_HOME/doc/Verdi_Transaction_and_Protocol_Debug.pdf

8.1.4 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 VIP. This chapter discusses the following topics:

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

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.

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:

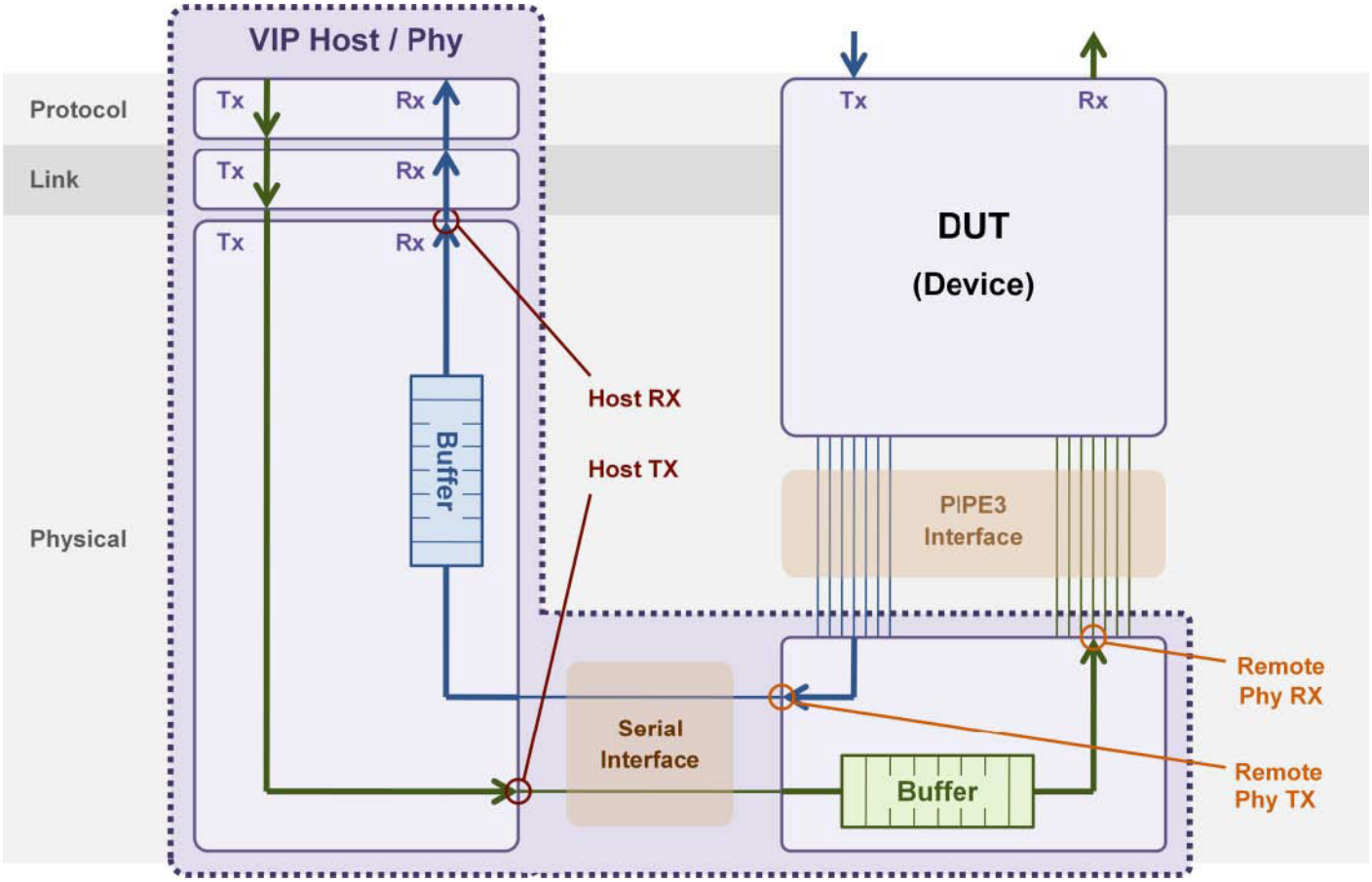
```
vip_agent.phys.SS.RX.data_trace
vip_agent.phys.SS.TX.data_trace
vip_agent.link.SS.RX.packet_trace
vip_agent.link.SS.TX.packet_trace
vip_agent.prot.SS.transaction_trace
vip_agent.prot.SS.transfer_trace
```

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_instrance>.prot.SS.TX.transfer_trace and <vip_agent_instrance>.prot.SS.RX.transfer_trace files are created with nested transfer, transaction, data. and packet information.

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-OVM 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 OVM-specified log-levels in the code. The components are extended from ovm_report_object. You can use the following ovm_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>:

- OVM_NONE. Report is always printed. Verbosity level setting can not disable it.
- OVM_LOW. Report is issued if configured verbosity is set to OVM_LOW or above.
- OVM_MEDIUM. Report is issued if configured verbosity is set to OVM_MEDIUM or above.
- OVM_HIGH. Report is issued if configured verbosity is set to OVM_HIGH or above.

- OVM_FULL. Report is issued if configured verbosity is set to OVM_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> +ovm_log_default=OVM_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 OVM directly:

```
+ovm_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:OVM_LOW,svt_usb_link_ss_ltssm_base:OVM_HIGH,
svt_usb_physical:OVM_HIGH,svt_usb_link_ss_tx:OVM_HIGH
```

The levels can be:

```
`define SVT_FATAL_VERBOSITY   OVM_NONE
`define SVT_ERROR_VERBOSITY   OVM_NONE
`define SVT_WARNING_VERBOSITY OVM_NONE
`define SVT_NORMAL_VERBOSITY  OVM_LOW
`define SVT_TRACE_VERBOSITY   OVM_MEDIUM
`define SVT_DEBUG_VERBOSITY   OVM_HIGH
`define SVT_VERBOSE_VERBOSITY OVM_FULL
```

Table 9-2 Valid VIP Sub-Units

Layer	Sub-unit or Component
Entire agent	svt_usb_agent
Protocol Layer - SS:	• svt_usb_protocol • svt_usb_protocol_block • svt_usb_protocol_device • svt_usb_protocol_processor • svt_usb_protocol_ss_host • svt_usb_protocol_ss_host_non_isoc_ep_processor • svt_usb_protocol_ss_host_isoc_ep_processor • svt_usb_protocol_ss_device • svt_usb_protocol_ss_device_non_isoc_ep_processor • svt_usb_protocol_ss_device_isoc_ep_processor • svt_usb_protocol_scheduler • svt_usb_protocol_host_scheduler • svt_usb_protocol_device_scheduler • svt_usb_protocol_ss_host_isoc_ep_processor • svt_usb_protocol_ss_device_isoc_ep_processor • svt_usb_protocol_ss_lmp_processor • svt_usb_protocol_ss_itp_processor
Protocol Layer - USB 2.0	• svt_usb_protocol_processor • svt_usb_protocol_20_host_non_isoc_ep_processor • svt_usb_protocol_20_host_isoc_ep_processor • svt_usb_protocol_20_device_non_isoc_ep_processor • svt_usb_protocol_20_device_isoc_ep_processor • svt_usb_protocol_block • svt_usb_protocol_20_host • svt_usb_protocol_20_device • svt_usb_protocol_20_lpm_processor • svt_usb_protocol_20_sof_processor
Link layer-SS:	• svt_usb_link_ss_tx • svt_usb_link_ss_rx • svt_usb_link_ss_ltssm_base • svt_usb_link_ss_lcm
Link layer-USB 2.0	• svt_usb_link_20 • svt_usb_link_20_device_a_sm • svt_usb_link_20_device_b_sm • svt_usb_link_20_timer
Physical Layer	svt_usb_physical

9.4 Disabling Specific In-line Checking

By default, all VIP error checking is enabled by default. For more information on chk_cov_mgr, see the class reference.

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

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

A.1 Debug Automation

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

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

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

A.2 Enabling and Specifying Debug Automation Features

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

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

Note the following about the plusarg:

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

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

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

The following table explains each control string:

Table A-1 Control Strings for Debug Automation plusarg

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

Examples:

Enable on all VIP instances with default options:

```
+svt_debug_opts
```

Enable on all instances:

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

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

Enable on all instances:

- starting at time 1000 until time 1500:

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

Enable debug feature on all instances using default options:

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

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

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

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

A.3 Debug Automation Outputs

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

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

The following are the output files generated:

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

A.4 FSDB File Generation

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

A.4.1 VCS

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

```
-debug_access
```

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

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

$VERDI_HOME/doc/linking_dumping.pdf

A.4.2 Questa

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

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

A.4.3 Incisive

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

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

A.5 Initial Customer Information

Follow these steps when you call the Synopsys Support Center:

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

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

A.6 Sending Debug Information to Synopsys

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

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

- DUT languages (Verilog)

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

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

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

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

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

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

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

A.7 Limitations

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

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