Verification Continuum™

VC Verification IP
USB
VMM User Guide

Version R-2021.03, March 2021

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

Preface

About This Manual

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

From the R-2021.03 release onwards, the documentation updates for the guides that are based on the SystemVerilog VMM 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 SystemVerilog Verification Methodology Manual (VMM). 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 VMM.

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_vmm_class_reference/html/index.html

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

$DESIGNWARE_HOME/vip/svt/usb_svt/latest/examples/svtb/index.html

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

1.1 Product Overview

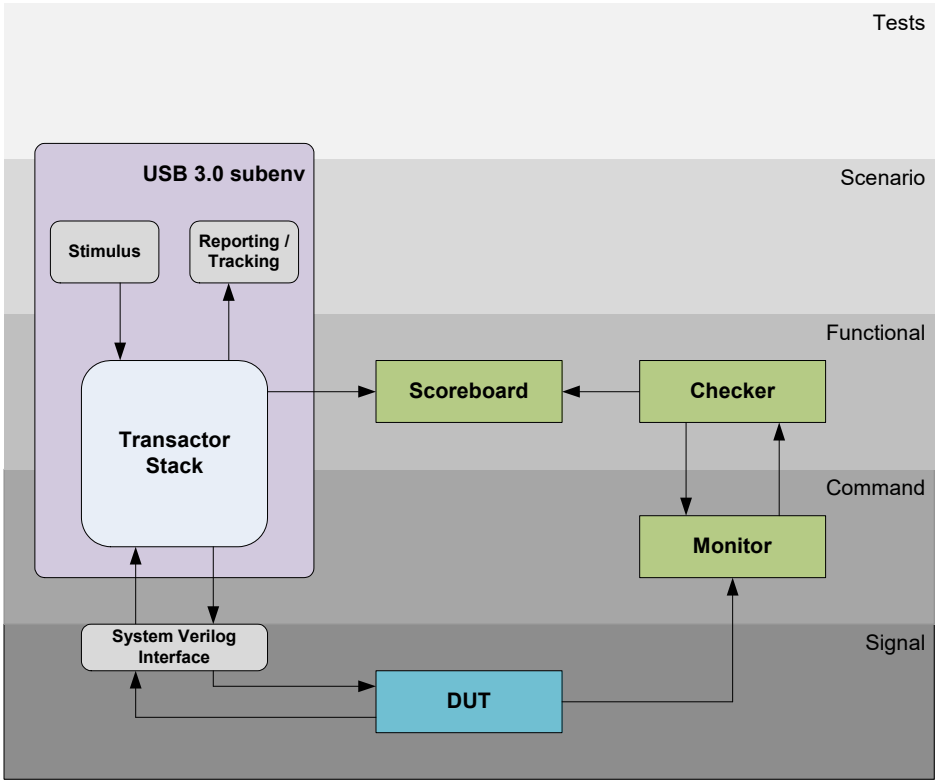
The USB VIP is a suite of VMM-based verification components that are compatible for use with SystemVerilog-Compliant testbenches. The USB VIP suite simulates transfers, transactions, and packets through active transactors as defined by the USB and USB 2.0 specifications. The VIP defines three transactors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

- 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.

The VIP provides a sub-environment that contains the transactors and generators. After instantiating the sub environment, you can select and combine active transactors to create an environment that verifies USB features in the DUT. Transactors support all functionality normally associated with active and passive VMM transactors, including the creation of transactions, exception injection to verify specification-defined error detection and recovery requirements, along with checking and reporting the response correctness.

The USB VIP can also be used for command-based Verilog testbenches. This usage supports directed testing. For information about command-based usage, see the USB Verification IP HDL User Guide.

Figure 1-1 USB VIP in VMM 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

1.3 USB Layer Stack Feature Support

1.3.1 Protocol Layer Features

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

USB VIP supports the following protocol layer functions:

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

1.3.2 Link Layer Features

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

- USB General Support
  - Host and Device support

- Speed fallback from SS to FS or HS
- Speed fall-forward from FS or HS to SS

- USB Support
  - LTSSM – Direct LTSSM state change support
  - SS power management
  - ITP support
  - Link advertisements
  - Link command and packet processing and response
- USB 2.0 Support
  - LPM
  - Suspend and Resume
  - USB2 power management

1.3.3 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
- UTMI+ support

- UTMI MAC and PHY support
- VIP configured as host and device can both act as MAC or PHY on the UTMI interface
- Reset, suspend and resume

1.3.4 SSIC Physical Layer Features

The USB VIP supports the following SSIC physical layer functions:

- Data scrambling/descrambling
- If there are > 1 PAIR or LANEs in the system, then model supports:
  - Spreading the Tx symbols across the pairs
  - Aggregating the Rx symbols from the pairs
  - 8b/10b encoding/decoding
- RMMI
  - 10, 20, or 40 bit parallel interface
  - SymbolClk generation based on Gear and Speed settings

1.4 VMM Support Provided by USB VIP

The following is a summary of the supported VMM features:

- Top level USB subenv that implements the following:
  - Configurable USB transactor stack that connects protocol, link, and physical transactors.
  - Random stimulus generators: atomic, scenario, and multi-stream
  - Consensus to support automatic end-of-test determination
- Callback support in all layers of the transactor stack
- Functional coverage in all layers of the transactor stack
- Trace support in all layers of the transactor stack
- Digital simulation of analog signaling required for attachment and detachment detection
- Configuration object support
- Support for input through transaction channels
  - 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 channel get actions using callbacks
  - Support for post randomization callbacks after transfer, transaction, or packet randomization by the transactor
  - Support for input transaction coverage through callbacks
  - Service input channels:
    - Available on all transactors 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 transactor channels
  - Performed in response to internal events
  - Generates channel transactions
  - Support for pre-channel put actions via callback registered with the transactor
  - Output transaction coverage via callbacks
  - Support for user provided transaction objects via user provided factory objects
  - Service output channels:
    - Send commands or command responses from protocol and link transaction layers
- Testbench scoreboarding, done using callbacks (STARTED and ENDED notifications on the VMM transaction handed out through callbacks).

2 Installation and Setup

2.1 Introduction

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.

2.1.1 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

2.1.2 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.

2.1.2.1 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:

DesignWare USB VIP Release Notes

2.1.2.2 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.

2.1.2.3 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)

2.1.3 Preparing for Installation

1. Set DESIGNWARE_HOME to the 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 $LM_LICENSE_FILE

2.1.4 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.

2.1.5 Setting Up a Testbench Design Directory

A *design directory* is where the DesignWare USB 3.0 VIP is set up for use in a testbench. A design directory is required for using Synopsys VIP and, for this, the dw_vip_setup utility is provided.

The dw_vip_setup utility allows you to:

- Create the design directory (design_dir), which contains the transactors, support files (include files), and examples (if any)

  Add a specific version of Synopsys USB VIP from DESIGNWARE_HOME to a design directory

For a full description of dw_vip_setup, refer to The dw_vip_setup Utility.

To create a design directory and add a model so it can be used in a testbench, use the following command:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a usb_protocol_svt -svtb
```

The models provided with Synopsys USB VIP include:

- usb_subenv_svt

2.1.6 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

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.3 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.

2.3.1 Simulator-Specific Settings

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

2.4 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
```

2.5 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"

2.5.1 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, refer to 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

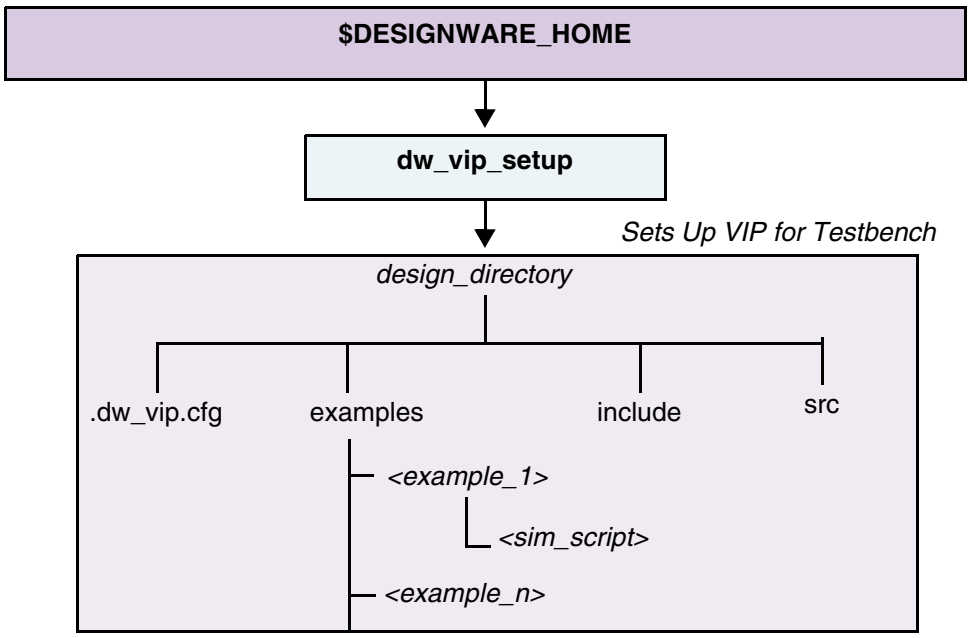


Table 2-1 Design Directory

Directory	Description
examples	Each 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 Discovery models. This directory is specified in simulator command lines.
src	Synopsys Discovery-specific include files (not used by all Synopsys Discovery). This directory may be specified in simulator command lines.
`.dw_vip.cfg`	A list of all Synopsys Discovery models being used in the testbench. This file is created and used by dw_vip_setup. When multiple different VIPs are added to the same design directory, the new model names are appended in this file.

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

2.5.1.1 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.

2.5.1.2 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.

2.5.1.3 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
```

2.5.2 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

2.5.2.1 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

2.5.2.2 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.

2.5.3 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.

2.5.4 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_vmm_basic_sys/

// To run the example using the generated run script with +incdir+
./run_usb_svt_vmm_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 vmm -full64 -sverilog +define+VMM_PACKER_MAX_BYTES=16000
+define+VMM_PACKER_MAX_BYTES=1500000 \
+define+VMM_DISABLE_AUTO_ITEM_RECORDING -timescale=1ns/1ps +define+SVT_VMM_TECHNOLOGY
+define+SYNOPSYS_SV \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_vmm_basic_sys/
. \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_vmm_basic_sys/
../../env \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_vmm_basic_sys/
../env \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_vmm_basic_sys/
env \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_vmm_basic_sys/
dut \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_vmm_basic_sys/
hdl_interconnect \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_vmm_basic_sys/
lib \
+incdir+<testbench_dir>/examples/sverilog/usb_svt/tb_usb_svt_vmm_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

3.1 Introduction to VMM

VMM 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.

This chapter assumes that you are familiar with SystemVerilog and VMM. For more information:

- For the IEEE SystemVerilog standard, see:
  - IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language

3.2 Available VMM/USB Examples

Synopsys has created an entire series of QuickStart examples and tutorials to help you to understand and build a VMM-USB testbench. The following table summarizes the available examples.

Table 3-1 SuperSpeed Examples

Name of Test	VIP Acting as Host	VIP Acting as a Device
BASIC EXAMPLES: usb_svt/examples/sverilog/tb_usb_svt_vmm_basic_sys		
usb_svt/tb_usb_svt_vmm_basic_sys:		
	Subenv for end of simulation control. In most 'basic' test cases, Built-in atomic generator, with customized transfer factory, is used to issue random transfers	In most 'basic' test cases, payload size (byte count) of VIP's response to Bulk IN transfers is modified using a Device VIP callback
usb_svt/tb_usb_svt_vmm_intermediate_sys		
	Subenv for end of simulation control. Built-in scenario generator, with customized transfer factory, is used to issue random transfers.	Has all the features of the basic example, but adds scoreboarding, coverage, and factories.

Table 3-1 SuperSpeed Examples (Continued)

Name of Test	VIP Acting as Host	VIP Acting as a Device
usb_svt/tb_usb_svt_vmm_advanced_sys		
	Subenv for end of simulation control. Built-in multi-stream scenario generator, with customized transfer factory, is used to issue random transfers.	Same as the Intermediate. Has all the features of the basic example, but adds scoreboarding, coverage, and factories
tests/ts.basic_ss_serial.sv		
	Initiating BULK IN, OUT transfers, both directed and random. ENDED' notification of transfers	Modification of response payload size for BULK IN transfers
tests/ts.basic_additional_ss_ltssm.sv		
	LTSSM: SS.Disabled to U0, with SS Serial Modification of LTSSM timers Notification of LTSSM state	LTSSM: SS.Disabled to U0, SS Serial Modification of LTSSM timers Notification of LTSSM state
tests/ts.basic_additional_ss_pipe3.sv		
	LTSSM: RX.Detect to U0, 16-bit PIPE3 Initiating Bulk OUTs with data that is random or custom or algorithm-based Initiating CONTROL IN, OUT transfers 'ENDED' notification of transfers	LTSSM: RX.Detect to U0, 16-bit PIPE3 Modification of response payload size for Bulk IN transfers
tests/ts.basic_additional_ss_isoc.sv		
	Initiating ISOC IN and OUT transfers to ISOC or ISOC_ONE or ISOC_TWO endpoints	Modification of response payload size for ISOC IN transfers
tests/ts.basic_additional_ss_stream.sv		
	Initiating 'Stream' Bulk OUT transfers to a device with multiple stream resources	Configuration of VIP's endpoints with multiple Stream resources
tests/ts.basic_additional_ss_aligned.sv		
	Completion of aligned Bulk IN and OUT transfers without zero-length Data Packet, but with testbench co-ordination Reading and modification of VIP configuration in middle of a test	Completion of aligned Bulk IN and OUT transfers without zero-length Data Packet, but with testbench co-ordination Reading and modification of VIP configuration in middle of a test
tests/ts.basic_additional_ss_compliance.sv		

Table 3-1 SuperSpeed Examples (Continued)

Name of Test	VIP Acting as Host	VIP Acting as a Device
	Compliance Mode master Modification of VIP's LTSSM timers to enter 'Compliance Mode' from 'Polling'. Compliance Patterns, CP0-8, transmission Driving 'Warm Reset' to exit 'Compliance Mode'. Modification of VIP's LTSSM timer in middle of test	'Compliance Mode' slave Modification of VIP's LTSSM timers to enter 'Compliance Mode' from 'Polling'. Reception of compliance patterns Reception of 'Warm reset' to exit 'Compliance Mode'. Modification of VIP's LTSSM timer in middle of test
tests/ts.basic_additional_ss_loopback.sv		
	Directed entry into 'Loopback' state from 'Rx.Detect' via 'Polling' Directed exit from 'Lookpack' state to 'Rx.Detect' and automatically reach 'U0'. Directed entry into 'Loopback' state from 'U0' via 'Recovery' Transmission of BERC and BRST patterns while in 'Loopback' state. Disabling of 8b/10b decoding checks	Modification of LFPS handshake timers for exiting 'Loopback' to reach 'Rx.Detect'. Disabling of 8b/10b decoding checks
tests/ts.basic_additional_ss_exceptions.sv		
	Directed CRC32 error injection into specific Host-transmitted Data packets and re-transmission of Data Packet upon receiving 'retry' Transaction packet. Reporting of CRC32 error in received Data packets Modification of expected VIP's error messages to less severity	Directed CRC32 error injection into specific Device-transmitted Data packets -Reporting of CRC32 error in received Data packets -Modification of expected VIP's error messages to less severity
INTERMEDIATE EXAMPLES : usb_svt/examples/sverilog/tb_usb_svt_vmm_intermediate_sys		
usb_svt/tb_usb_svt_vmm_intermediate_sys		
	In most 'intermediate' test cases, Built-in scenario generator, along with a collection of built-in scenarios, is used to issue random and specific scenarios of transfers.	In most 'intermediate' test cases, payload bytes (data values) of VIP's response to Bulk IN transfers are modified using Device VIP callbacks.
tests/ts.intermediate_ss_serial.sv		
	Built-in random scenario Built-in Bulk IN scenario Built-in 'transfer' coverage User-defined 'transfer' coverage Insert 'ended' transfers into Scoreboard	Modification of payload bytes (data values) for Bulk IN transfers Comparison of 'ended' transfers with entries in scoreboard

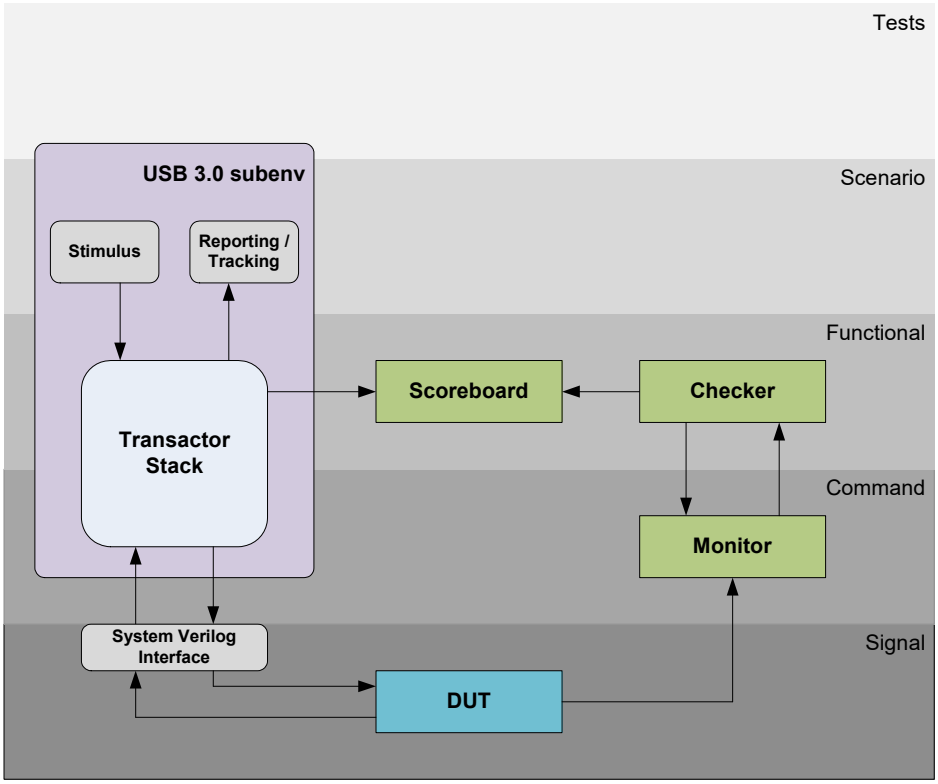
Table 3-1 SuperSpeed Examples (Continued)

Name of Test	VIP Acting as Host	VIP Acting as a Device
ADVANCED EXAMPLES : usb_svt/examples/sverilog/tb_usb_svt_vmm_advanced_sys		
tb_usb_svt_vmm_advanced_sys		
	Subenv for end of simulation control VMM data stream scoreboard In most 'advanced' test cases, Built-in multi-stream scenario generator is used to issue transfers and service commands.	In most 'advanced' test cases, payload bytes (data values) of VIP's response to Bulk IN transfers are modified using Device VIP callbacks
tests/ts.advanced_ss_serial.sv		
	Directed U1 entry - multiple times with transfers interleaved in between. Automatic U1 exit (timer based) Insert 'ended' transfers into Scoreboard	Downstream initiated U1 entry Automatic U1 exit (timer based) Built-in functional coverage User-defined functional coverage Comparison of 'ended' transfers with entries in scoreboard

3.3 USB VIP in a VMM Environment

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

Figure 3-1 USB VIP in VMM Architecture



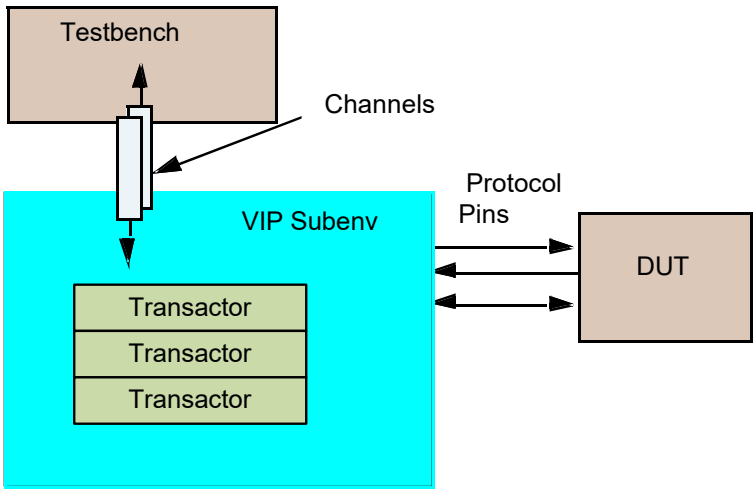
VMM system-level verification environments are constructed with sub-environments, transactors, self-checking structures, and other components that are designed for flexible reuse. Data objects are provided to configure VMM components and represent data that flows through the VIP.

3.4 VMM Support in USB Verification IP

In USB VIP, VMM-compliant classes and attributes (members) are provided to represent protocol activity and the characteristics of that activity. For example, a transaction object has members that might define a requestor ID, the payload size, and routing. The definitions of these transaction objects, along with the channels that handle them, form the interface between testbench and VIP. To create traffic, a user generates an object of the appropriate type and calls the put() task of the corresponding input channel.

USB VIP includes transactor models that can be controlled by an VMM testbench.

Figure 3-2 Verification IP for VMM



- **Testbench:** User created; puts objects into and gets objects out of an VMM channel. Typical objects define the test configuration, transactions, and exceptions (error injection).
- **Channels:** Provide a conduit for passing data between the testbench and the VIP transactor. For more information about channels, see "Channels".
- **VIP Sub-environment:** Provides a facility for combining VIP components into a reusable container and provides a conduit between a transactor stack and a testbench. The VIP sub-environment is VMM compliant to fit into your test environment.
- **VIP transactor:** Puts transaction objects into and gets transaction objects out of channels. Internally, it translates objects into protocol activity. The VIP transactor model is fully VMM compliant to fit into your test environment.

3.4.1 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 VMM approach to verification. They provide common functionality and structure needed for simulation (such as logging) 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 VMM is not simply a set of guidelines and recommendations. So, instead of writing your own logging routine, you can reuse the vmm_log class. Inheritance, extension, and polymorphism facilitate customization opportunities.

Important VMM base classes used by the USB VIP include:

- vmm_channel – object-based interface; connects elements in a verification environment
- vmm_data – base class for all data objects (such as transactions and configuration)
- vmm_xactor – base class for transactor models
- vmm_log – standard logging object
- vmm_env – base class for the verification environment that is built in the testbench

3.4.2 Sub-environments

A sub environment is a verification environment subset that is reusable in different verification environments. Sub-environments are composed of one or more transactors that are linked to elements such as a scoreboard or a response generator.

The USB VIP defines a sub-environment that extends from the vmm_subenv base class. The USB Sub-environment describes the USB sub-environment.

3.4.3 Transactor Components

Transactors are objects in a VMM compliant verification environment. The testbench and transactors exchange transactions through three distinct types of transactor interfaces:

- **Channels**
  - Input channels provide transactions to transactors which, in turn, transform them into lower level transactions, or at the lowest level into signal level bus activity for transmission to the testbench.
  - Transactors place transactions that they create in output channels.

    Transactions created by a transactor are based on the signal activity sensed by the transactor at the lowest level, or based on lower-level transaction data objects received from a lower-level transactor (if the transactor is not at the lowest layer of the protocol stack).
- **Callbacks**
  - Channel 'get' and 'put' operations are matched with 'get' and 'put' callbacks. These callbacks are called after the 'get' and prior to the 'put'. The 'get' and 'put' callbacks use 'drop_it' flags to filter out input, output, or activity transactions.
  - Callbacks are defined in a callback facade class (associated with each transactor), and accessed by registering (with the associated transactor) an instance of a class extended from that facade class.
  - Each transactor supports additional callbacks to access to data at internal dataflow points. Refer to the HTML documentation for a complete callback list.
- **Notifications**
  - Some transactors signal “significant events” through notifications and include transactions as vmm_data objects with these notifications. Testbenches can be configured to wait for notifications and then to retrieve the associated data object by making a call to the transactors VMM notification service instance.

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

“USB Verification IP Transactors” describes the three USB VIP transactors.

3.4.4 USB VIP Objects

The USB VIP defines several classes designed for a VMM environment. This section introduces the major USB VIP objects, including channels, configurations, transactions, exceptions, and callbacks.

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. Generators are VMM transactors that use factories to create streams of randomized objects and are useful for generating transactions. Generators are discussed in Generators; factories are discussed in Factories.

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

- "Configuration Objects"
- "Channels"
- "Transaction Objects"
- "Status Objects"
- "Exception and Exception List Objects"
- "Callbacks"

3.4.4.1 Configuration Objects

Predefined configuration objects, extended from the vmm_data base class, are provided for configuring the USB VIP to fit specific testbench applications. The configuration objects specify sub-environment attributes and support testbench capabilities, such as randomization and constraints. Configuration objects apply to all appropriate transactors in the stack.

Configuration data objects convey the testbench 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 transactors 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.

Configuration objects are used by sub-environments and transactors. Configuration objects sent through the transactor constructor must not be null and must be valid. If the object is not null, the constructor calls the is_valid method of the configuration object. If this method returns true, the transactor continues the construction; otherwise, an error message is displayed and the simulation is halted.

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. A static configuration parameter is frozen by setting its rand_mode OFF.

The first opportunity for submitting a configuration is in the constructor of the sub-environment component. The configuration object is a parameter to the new() method of the transactor. 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:

- **Sub-environment configuration** (*svt_usb_subenv_configuration*): This class provides settings for the basic testbench capabilities such as whether generators are present, which channels are present, whether exceptions are enabled. These basic testbench capabilities are not randomized, and are controlled via basic 'enable' flags and more complex enumerated choices.
- **Device** (*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** (*svt_usb_host_configuration*): The ‘prot’ transactor uses the information to support its breaking of transfers in to individual transactions.
- **Endpoint** (*svt_usb_endpoint_configuration*): This vmm_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** (*svt_usb_ustream_resource_configuration*): This vmm_data class contains information regarding a 'USB SS stream resource'.

For more information, see the Class Reference.

3.4.4.2 Channels

Channels provide a standard interface for passing data objects between components. Channels natively handle objects of type vmm_data. To define a channel, the VIP extends the vmm_channel base class to support a data object that was derived from vmm_data. The data objects, which typically represent protocol transactions, can be pushed into or pulled out of a channel; to connect two components, one component puts objects into the channel and the other pulls them out. Channels are unidirectional and specific to a data object class.

Channels that form the Interface between contiguous transactors in the sub-environment are automatically connected. Channels can accept stimulus objects or provide response objects. Refer to the following sections for a list of channels associated with each transactor:

- Protocol transactor channels: see Protocol Transactor Channels
- Link transactor channels: see Link Transactor Channels
- Physical transactor channels: see Physical Transactor Channels

3.4.4.3 Transaction Objects

Transaction objects, which are extended from the vmm_data base class, define a unit of bus protocol information that is passed across the bus. The attributes of transaction objects are public and are accessed directly for setting and getting values. Most transaction attributes can be randomized. The transaction 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 transaction objects, such as for different protocol layers.

Transaction data objects store data content and protocol execution information for USB connection transactions in terms of bit-level and timing details of the transactions. These data objects extend from the vmm_data base class and implement all methods specified by VMM for that class.

USB transaction data objects are used to:

- Generate random scenario stimulus, when used with VMM scenario, atomic, or multi-stream scenario generator macros
- Report observed transactions from receiver transactors
- Generate random responses to transaction requests
- Collect functional coverage statistics
- Support error injection

Class properties are public and accessed directly to set and read values. Transaction 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 transactors. 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 transaction 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 transaction 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 notification.
- **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 transactor.
- **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 transactor receives Protocol Service object containing power management transaction properties from the testbench.

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

- **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 channel. The Protocol transactor controls USB bus activity using the list of transactions. Alternatively, the testbench can leave the list of transaction objects empty and the protocol transactor will create the transactions needed to implement the transfer.

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

3.4.4.4 Status Objects

Status classes define status data descriptor objects. The USB VIP defines the following status classes:

- **Transactor** (*svt_usb_status*): This class provides a common location for status information coming from the different transactors in the USB transactor stack. This status information comes in two forms – data and notifies. The data members represent the current status as defined by the transactor implementing and updating the status information. The notifies are used to indicate a change in status, as defined by the transactor responsible for indicating the notification.
- **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 transactor there will be on 'sub-transactor' for each endpoint defined in the svt_usb_configuration object supplied to the USB VIP. This class makes available the endpoint specific status information.
- **Ustream Resource** (*svt_usb_ustream_resource_status*): This class contains information regarding a USB SS stream resource. In the USB protocol transactor there will be one sub-transactor for each instance of svt_usb_ustream_resource_status if the endpoint supports streams. That sub-transactor indicates the USB stream IDs it is assigned to support.

3.4.4.5 Exception and Exception List Objects

Exception objects, which extend from the vmm_data base class, represent injected errors or protocol variations. Each transaction 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.

3.4.4.6 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 transactor 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.

- 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 transactor puts a transaction object into an output channel 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 transaction 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. Callback must be registered using the append_callback() method of the transactor.

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-channel get callbacks:** called after a transaction is pulled from the input channel; provided with a handle to the transaction gotten from the channel
- **pre-channel put callbacks:** called prior to putting a transaction out on output channel, 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.

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

USB VIP Physical transactors accept the necessary modports through their constructors. Transactors 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 transactor, 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 transactor, 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 transactor, 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.
- **USB On-The-Go (OTG) Interface:** This interface (svt_usb_otg_if) includes signals that support the modeling of the OTG functionality. It includes modport definitions needed to provide the logical connection for various interface connections (such as the MAC to PHY connection).

See the USB SVT - Interfaces Reference page in the HTML Class Reference for more information.

"Interface Options" describes USB interface options.

3.4.6 Constraints

3.4.6.1 Description

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

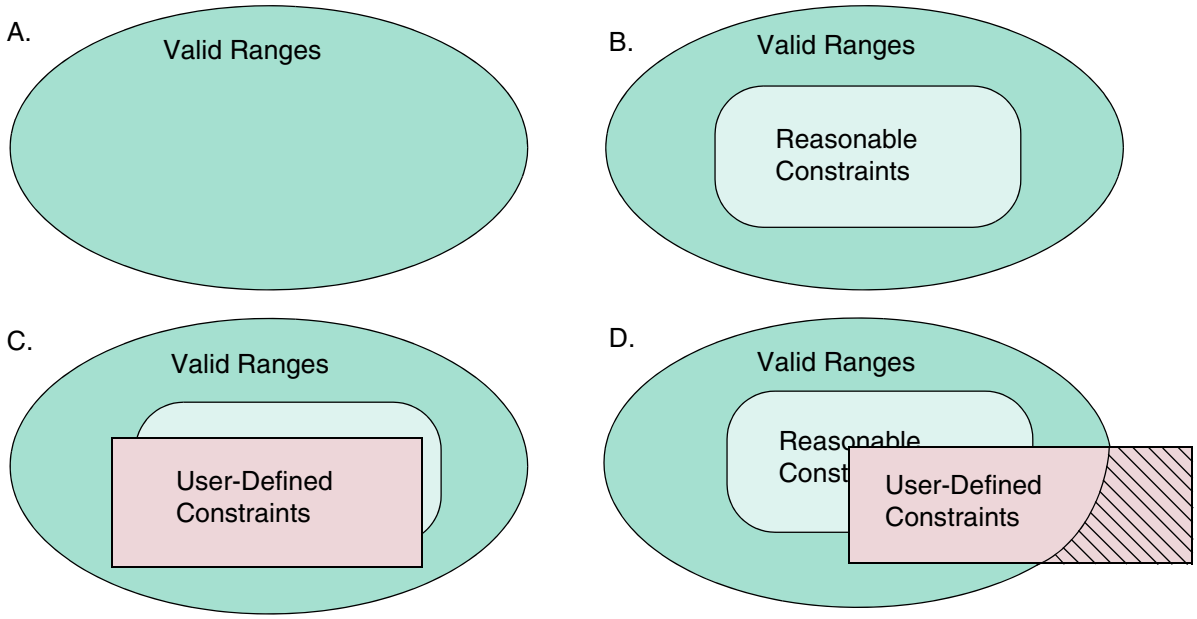
Classes that provide random attributes allow you to constrain the contents of the resulting object. When you call the randomize() method, which is a built-in method, all random attributes are randomized using all constraints that are enabled.

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 3-3 Constraints: Valid Ranges, Reasonable, and User-Defined



- Valid range constraints:
  - Provided with USB VIP
  - Keep values within a range that the transactors 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.

3.4.6.2 Implementation

The following two methods implement constraints:

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

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

Example:

svt_usb_transfer::test_constraints1 { payload_intended_byte_count <= 4096; }

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

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

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

  Example (this code might be in a testcase)

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

dev_subenv.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 subenv'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.

3.4.7 Generators

You use generators to create constrained random objects, typically transactions. Generators make use of the built-in *randomize()* method and rely on constraints to limit the scope of the randomizations. A generator must be declared to handle a specific data type (the factory object).

The simplest form of generator is an *atomic* generator, which produces individual objects randomly with no particular relationship between them. The following generic code snippet illustrates an atomic generator that operates on a transfer factory object:

```
while (gen_cnt < tb_cfg.test_len)
  begin
      // Randomize but don't generate completions
      int success = randomized_tr.randomize() with {
         m_enType !inside {
            svt_usb_transfer::member_1,
            svt_usb_transfer::member_2,
            svt_usb_transfer::member_n,
         }
      }
      gen_cnt++;
      // Make a copy of the transaction object, assign an ID, and push it into the
channel
      $cast(anyXact, randomized_tr.copy());
      anyXact.object_id = gen_cnt;
```

```
        gen_out_chan.put (anyXact);
        // Display the transaction
        msg = $psprintf("Copy of Transaction #%0d has been put in the channel", gen_cnt);
        `vmm_note(log, msg);
        anyXact.display("tbd TX In CH: ");
        // The ENDED notification indicates the transaction has completed on the protocol
        anyXact.notify.wait_for(vmm_data::ENDED);
        msg = $psprintf("Transaction #%0d has ended", gen_cnt);
        `vmm_note(log, msg);
end
```

This example shows the essential parts of a generator, which include a while loop to control how many objects are generated. Inside the loop, randomize() is called. Then, a copy of the randomized transaction is created and pushed into the channel interface. As part of the generator, you might also display each transaction object that gets generated, as shown, and use the ENDED notification to confirm that the transaction is completed.

The ‘randomize with’ construct used above is a convenient way of applying constraints to members “on-the-spot”. In terms of reuse, the generator does not need to know anything about the factory object that it is randomizing (except for the class name). In the code above, the definition of randomized_tr does not affect the generator code. The constraints are the only object information included, and they could be included elsewhere (in an extended class). As a result, the generator code can be independent from the testbench code--it simply needs to be given a factory object to randomize. By simply providing another factory object, the generator produces objects based on the new template. This is one of the important opportunities that VMM provides for reducing test-specific code and increasing reuse.

For every transaction there are a corresponding set of scenario, atomic, and multi-stream generator classes.

3.4.8 Factories

The object that is provided to a generator 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 generator uses the factory to create streams of randomized transactions. Also, USB VIP transactors create factory objects for output channels so user-defined extensions to a transaction class can be handled for scoreboarding.

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

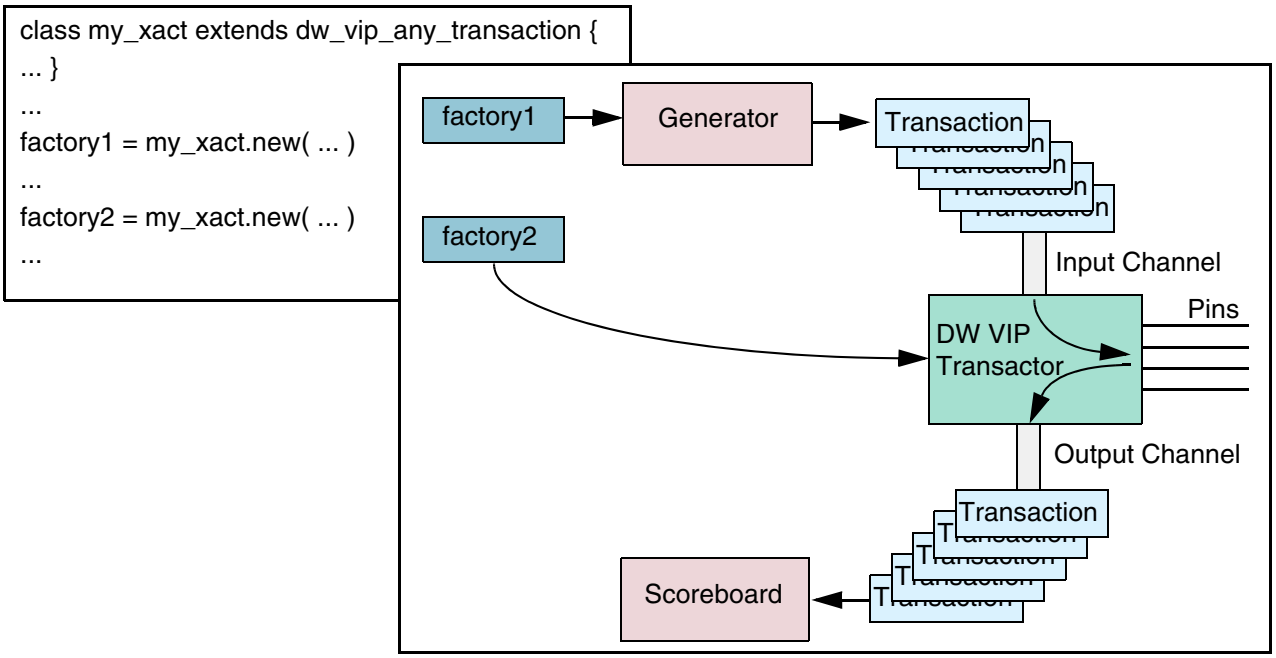
  The nomenclature for this is now 'randomized object'; these variables are prefixed with the 'randomized_' string.

- 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_chan_factory.

Figure 3-4 illustrates how a factory object works with a generator and a USB VIP transactor.

Figure 3-4 Factories with USB VIP



When using USB VIP, the factory object is typically a transaction. In Figure 3-4, the code excerpt extends a USB VIP transaction class and then establishes two instances to use as factory objects--one for the generator and one for the transactor. Typically, extensions to a transaction class are user constraints that scope the randomization to the desired test conditions. Based on the factory object and the extended constraints, the generator creates transactions and puts them into the input channel of the transactor. The transactor generates the protocol activity, handles any response, and optionally passes scoreboard information through the output channel to the scoreboard.

When a transactor creates an object to be output on an activity or output channel, 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.

Constructors for VIP transactors include optional factory arguments for the creation of user defined transaction, exception list, and exception objects. Each output channel has one factory associated with it in the constructor. Default transaction factories are created if the user fails to provide a factory in the constructor, as long as the corresponding channel exists.

3.4.9 Messages

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

Messages generated by VIP transactors are compatible with the vmm_log base class. 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. The possible types are listed in the *Verification Methodology Manual For SystemVerilog* in the description of the vmm_log class.
- **Severity:** Severity is similar to the urgency of the message or how serious it is. The possible values for severity are listed in the *Verification Methodology Manual For SystemVerilog* in the description of the vmm_log class.
- **Text:** This is the text of the message. Since VMM supports and promotes identifying messages by string matching against a regular expression, the text is especially important for messages that do not have a unique ID.

4 Integrating the VIP into a User Testbench

The VC VIP for USB provides a suite of advanced SystemVerilog verification components and data objects that are compliant to VMM. Integrating these components and objects into any VMM compliant testbench is straightforward.

For a complete list of VIP components and data objects, see the main page of the VC VIP USB Class Reference (only in HTML format) at the following location:

$DESIGNWARE_HOME/vip/svt/usb_svt/latest/doc/usb_svt_vmm_class_reference/html/index.html

4.1 VIP Testbench Integration Flow

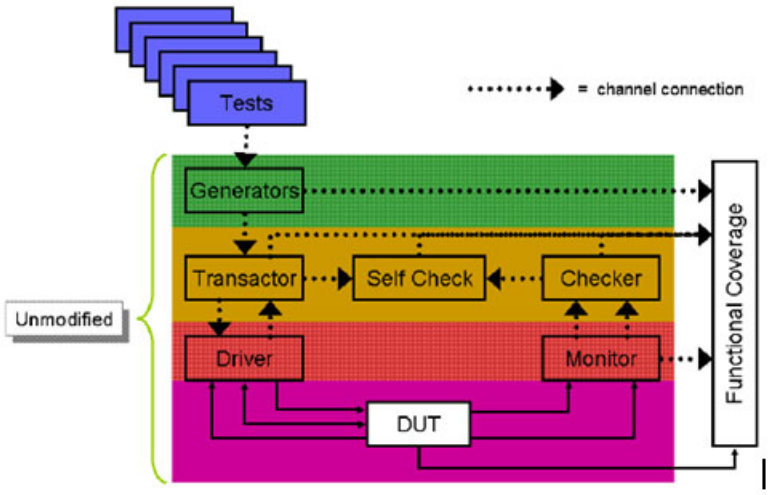
The USB subenv (`svt_usb_subenv`) is the top-level component provided by the VIP for modeling USB hosts, devices, and hubs. The VIP provides a sub-environment that contains the transactors and generators. After instantiating the subenv, you can select and combine active transactors to create an environment that verifies USB features in the DUT. Transactors support all functionality normally associated with active and passive error detection and recovery requirements, along with checking and reporting the response correctness.

The VIP defines the folllowing transactors that implement the USB Specification:

- `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

A USB environment is a user-defined VMM environment that encapsulates all of the VIP components, and implicitly constructs the required number of USB host and USB device subenvs as specified by its system configuration object. You can instantiate and construct the USB environment in the top-level environment of your VMM testbench.

Figure 4-1 Top-level Architecture of a USB VIP Testbench



Examples 4-1 is a top-level architecture of a simple VC VIP for USB testbench. The testbench includes an instance of a USB host subenv connecting through the USB SuperSpeed (SS) serial interface to an instance of a device controller that represents the DUT.

The steps for integrating the VIP into a VMM testbench are described in the following sections:

These steps are documented for a VIP configured as a USB host verifying a device controller (DUT) connected through a SuperSpeed serial interface. Similar steps can be followed for other serial interfaces. The code snippets presented in this chapter are generic and can be applied to any VMM compliant testbench.

For more information on the code usage, see the following example:

$DESIGNWARE_HOME/vip/svt/usb_svt/latest/examples/sverilog/tb_usb_svt_vmm_basic_sys

4.1.1 Connecting the VIP to the DUT

The following are the steps to establish a connection between the VIP to the DUT in your top-level testbench:

Include the standard VMM and VIP files and packages.

```
`include "svt_usb_if.svi"
`ifdef USB_IN_PACKAGE
// Pull in the VIP package
`include "svt_usb.vmm.pkg"
`endif

`ifdef SVT_MULTI_SIM_AUTOMATIC_DESIGN_UNIT
```

```
program basic_system_test ();
`else
program automatic basic_system_test ();
`endif

`ifdef USB_IN_PACKAGE
// Import the SVT library
import svt_vmm_pkg::*;

// Import the USB VIP
import svt_usb_vmm_pkg::*;

`else
`include "svt_usb_subenv_source.svi"
`endif
```

Note You must compile all VIP files with the timescale 1ps/1ps. You can use the timescale option at compile-time or add the `timescale 1ps/1ps statement before including the VIP files.

- Instantiate the top-level USB interface (`svt_usb_if`) and connect the serial signals to the DUT.

```
svt_usb_if usb_host_ss_serial_if();
.ssrxp_device (usb_host_ss_serial_if.usb_ss_serial_if.sstxp),
.ssrxm_device (usb_host_ss_serial_if.usb_ss_serial_if.sstxm),
.sstxp_device (usb_host_ss_serial_if.usb_ss_serial_if.ssrxp),
.sstxm_device (usb_host_ss_serial_if.usb_ss_serial_if.ssrxm),
.ssvbus_device (usb_host_ss_serial_if.usb_ss_serial_if.vbus),

//Bi-directionals Connected by Transmission Gates
inout ssvbus_device;
tran usb_ss_vbus_xmit(usb_host_ss_serial_if.usb_ss_serial_if.vbus, ssvbus_device);

//OR
dut dut_inst(tx_p,tx_m,rx_p,rx_m);
assign usb_host_ss_serial_if.usb_ss_serial_if.ssrxp = tx_p;
assign usb_host_ss_serial_if.usb_ss_serial_if.ssrxm = tx_m;
assign rx_p = usb_host_ss_serial_if.usb_ss_serial_if.sstxp;
assign rx_m = usb_host_ss_serial_if.usb_ss_serial_if.sstxm;
```

- Connect the top-level USB interface to a system clock.

```
//USB SS 5.0GT/s clock period is 200 ps and the unit of the
//testbench timescale is 1ps.
parameter  usb_ss_simulation_cycle = 200;
bit        ss_serial_clock_for_VIP;
```

```
initial begin
   ss_serial_clock_for_VIP = 0;
   #(usb_ss_simulation_cycle);
   //No clock edge at T=0 forever begin
   ss_serial_clock_for_VIP = ~ss_serial_clock_for_VIP;
   #(usb_ss_simulation_cycle/8);
   ss_serial_clock_for_VIP = ~ss_serial_clock_for_VIP;
   #(usb_ss_simulation_cycle/8);
   ss_serial_clock_for_VIP = ~ss_serial_clock_for_VIP;
   #(usb_ss_simulation_cycle/8);
   ss_serial_clock_for_VIP = ~ss_serial_clock_for_VIP;
   #((usb_ss_simulation_cycle/2) -
   (3*(usb_ss_simulation_cycle/8)));
   ss_serial_clock_for_VIP = ~ss_serial_clock_for_VIP;
   #(usb_ss_simulation_cycle/8);
   ss_serial_clock_for_VIP = ~ss_serial_clock_for_VIP;
   #(usb_ss_simulation_cycle/8);


   ss_serial_clock_for_VIP = ~ss_serial_clock_for_VIP;
   #(usb_ss_simulation_cycle/8);
   ss_serial_clock_for_VIP = ~ss_serial_clock_for_VIP;
   #(((usb_ss_simulation_cycle)-(usb_ss_simulation_cycle/2))-
   (3*(usb_ss_simulation_cycle/8)));
   end
end
```

```
//Assign generated clock to the SS clock input of the VIP SS serial
//interface (ssclk)
   assign usb_host_ss_serial_if.usb_ss_serial_if.ssclk = ss_serial_clock_for_VIP;
```

Note For VIP operations with serial interfaces, the VIP requires a 4x USB speed since clock recovery is enabled by default. This requirement applies to all VIP operations regardless of the VIP's configuration as a USB host, device or hub.

For SuperSpeed (SS) operations, the input clock of the VIP (usb_ss_serial_if.ssclk) requires to be 20 GT/s (for example, 4x 5 GT/s).

For USB 2.0 (HS or FS or LS) operations, the input clock of the VIP (usb_20_serial_if.clk) requires to be 1.92 GT/s (i.e. 4x 480 MT/s).

- Connect the top-level USB interface to the virtual interface of the USB environment.

  Code snippet in environment file:

```
class usb_svt_basic_serial_env extends vmm_env;
/** Definition for virtual (VIP) USB interface */
  virtual svt_usb_if  host_usb_if;
  virtual svt_usb_if  dev_usb_if;
function usb_svt_basic_serial_env::new(vmm_log             log,
                               virtual svt_usb_if   host_usb_if,
                               virtual svt_usb_if   dev_usb_if
                              );
  begin
```

```
        super.new();
    if (log == null)
    begin
          this.log = new("usb_svt_basic_serial_env", "env");
        end else
    begin
          this.log = log;
     end
     log_format = new();
     void'(this.log.set_format(log_format));
     this.host_usb_if = host_usb_if;
     this.dev_usb_if  = dev_usb_if;
     this.cfg = new(log);
```

/*** Setup tracking utilities*/

```
     xact_report = new(svt_usb_suite_spec_override::get_suite_spec());
     consensus = new("vmm_consensus", {log.get_instance(), ".consensus"});
    end
endfunction
```

Code snippet in test file:

```
class example_env extends usb_svt_basic_serial_env;
   function new(vmm_log log);
super.new(log,test_top.usb_host_ss_serial_if,test_top.usb_dev_ss_serial_if);
   endfunction
endclass


initial begin
   vmm_log log;
   example_env env;
```

Connect the top-level USB interface to the virtual interface of the USB environment using the constructor function (new) of test env. The `test_top` represents the top-level module in the VMM environment. The env is an instance of your testbench environment.

4.1.2 Instantiating and Configuring the VIP

The following are steps to instantiate and configure the USB subenv (`svt_usb_subenv`) in your testbench environment.

- Instantiate the USB environment in the initial block of your test program block.

```
`ifdef SVT_MULTI_SIM_AUTOMATIC_DESIGN_UNIT
program basic_system_test ();
`else
program automatic basic_system_test ();
`endif
initial begin
   vmm_log log;
   example_env env;
```

- Instantiate the USB subenv (svt_usb_subenv) in the testbench environment class.

  A result of the overarching nature of `vmm_env` helps in organizing the discussion of VMM and VC VIP techniques and methods.

Table 4-1 lists the vmm_env class associated with methods that correspond to the basic steps.

Table 4-1 Testing Step

Testing Step	vmm_env method
Decide on a test configuration	gen_cfg()
Construct, configure and connect the elements in the environment	build()
Configure the DUT	cfg_dut ()
Start the test	start ()
Determine when the test is done	wait_for_end ()
Stop the test	stop ()
Perform any post-processing	cleanup ()
Report results	report()

Most of these methods are declared as virtual in the base class and should be extended by the user. To create a user environment, define a new class extended from vmm_env and extend the methods above to add test-specific code. To retain the core functionality of the base class methods, each extended method must call super.<method> as the first line of code. The code snippet below shows the user extension of vmm_env, creating the class usb_svt_basic_serial_env. The usb_svt_basic_serial_env class instantiates all the components of the verification environment which may include VC VIP models, user designs, generators, scoreboards. At the end of the extended class is a list of the methods that are extended.

Construct the subenv in the build phase of the testbench environment and bind the environment virtual interface to the subenv virtual interface.

```
class usb_svt_basic_serial_env extends vmm_env;
…
  virtual svt_usb_if  host_usb_if;
  svt_usb_subenv      host_subenv;
  usb_svt_shared_cfg  cfg;
…
function void usb_svt_basic_serial_env::build() ;
  begin
  /**
   * Construct ports that can be used to connect to the interfaces.
   */
          svt_usb_subenv_port_ss_serial  host_usb_ss_port = new(this.log, this.host_usb_if,
      this.host_usb_if);
          svt_usb_subenv_port_20_hsic  host_usb_20_hsic_port = new(this.log, this.host_usb_if,
      this.host_usb_if);
```

```
        svt_usb_subenv_port_20_serial  host_usb_20_serial_port = new(this.log,
    this.host_usb_if, this.host_usb_if);
    super.build();
/**
* Construct host_subenv
*/
   host_subenv    = new("svt_usb_subenv",
                        "host_subenv",
                        this.consensus,
                        this.cfg.host_cfg,
                        null,
                        this.xact_report,
                        host_usb_ss_port,
                        host_usb_20_serial_port);
```

- Create a basic configuration class by extending the vmm_data class.

  This configuration class specifies the USB protocol layer configuration for the USB host and device.

  For example:

```
class usb_svt_shared_cfg extends vmm_data ;
 /** usb_svt_env_timeout Holds the simulation timeout . */
  real usb_svt_env_timeout = `USB_SVT_ENV_TIMEOUT;
 /**
  * Configuration object used by the HOST transactor objects.
  */
  svt_usb_subenv_configuration host_cfg;
 /**
  * Configuration object used by the Device transactor objects.
  */
  svt_usb_subenv_configuration dev_cfg;
function new(vmm_log log = null);
    begin
      super.new((log == null) ? this.log : log);
   /**
    * Create new instance of HOST and DEVICE configuration objects.
    */
      host_cfg = new();
      dev_cfg  = new();
```

```
    host_cfg.component_type = svt_usb_types::HOST;
    dev_cfg.component_type  = svt_usb_types::DEVICE;


  /**
   * Enforce making the top level in the stack the PROTOCOL layer
   */
    host_cfg.top_xactor     = svt_usb_subenv_configuration::PROTOCOL;
    dev_cfg.top_xactor      = svt_usb_subenv_configuration::PROTOCOL;


  /**
   * Host VIP configured to have USB host function
   * Host VIP won't have device function
   */
    host_cfg.local_host_cfg        = new();
    host_cfg.local_device_cfg_size = 0;
/*** Device VIP configured to have one USB device function
   * Device VIP won't have host function*/
    dev_cfg.local_device_cfg_size = 1;
    dev_cfg.local_device_cfg[0]   = new();


    dev_cfg.local_device_cfg[0].endpoint_cfg[0] = new();
// Default num_endpoints is 1, so must populate one EP cfg
    dev_cfg.local_host_cfg = null;


  /**
   * Host VIP's configuration must include the configuration of the
   * targeted Device that would be communicating with Host VIP
   * over standard USB interface.
   * Host VIP uses the targeted Device's configuration information such as
   * endpoint information (how many and their types etc), capability so
   * that the transactions/packets issued to Device are consistent with
   * device's configuration and capability.
   * Since Device is a VIP in these examples, assign the pointer of the
   * Device VIP config's local device configuration to the Host VIP's
   * remote device configuration. So, from now onwards, any changes
   * made to the Device VIP's local_device_cfg will also be reflected
   * in the Host VIP's remote_device_cfg and vice versa.
   */
```

```
      host_cfg.remote_device_cfg_size = dev_cfg.local_device_cfg_size;
      host_cfg.remote_device_cfg = dev_cfg.local_device_cfg;
      host_cfg.remote_host_cfg = dev_cfg.local_host_cfg;


      dev_cfg.remote_host_cfg = host_cfg.local_host_cfg;
      dev_cfg.remote_device_cfg_size  = host_cfg.local_device_cfg_size;


    end
  endfunction


  function void setup_usb_ss_defaults();
    begin
      host_cfg.capability        = svt_usb_configuration::PLAIN;
      dev_cfg.capability         = svt_usb_configuration::PLAIN;
      host_cfg.speed             = svt_usb_types::SS;
      dev_cfg.speed              = svt_usb_types::SS;

      host_cfg.usb_ss_signal_interface = svt_usb_configuration::USB_SS_SERIAL_IF;
      dev_cfg.usb_ss_signal_interface  = svt_usb_configuration::USB_SS_SERIAL_IF;

      dev_cfg.local_device_cfg[0].device_address        = 0;
      dev_cfg.local_device_cfg[0].connected_bus_speed   = svt_usb_types::SS;

      dev_cfg.local_device_cfg[0].connected_hub_device_address       = 0;
      dev_cfg.local_device_cfg[0].functionality_support     = svt_usb_types::SS;
      dev_cfg.local_device_cfg[0].num_endpoints             = max_usb_ss_endpoints;


      if (max_usb_ss_endpoints <= 0) begin
        `vmm_fatal(log, $psprintf("setup_usb_ss_defaults() - The max_usb_ss_endpoints
property is set to %0d. This property must be set to a value >=1. Unable to continue
!!",
                max_usb_ss_endpoints));
      end


      for (int ep_num = 0; ep_num < max_usb_ss_endpoints; ep_num++)
begin
        dev_cfg.local_device_cfg[0].endpoint_cfg[ep_num] = new();
end


      if (max_usb_ss_endpoints >= 1)
begin
/*** Configuring End point '0' to CONTROL
```

Note 'direction' is irrelevant for a CONTROL endpoint. Though direction is chosen as 'IN' here, this endpoint will support both IN and OUT transfers.

```
*/
        dev_cfg.local_device_cfg[0].endpoint_cfg[0].ep_number  = 0;
        dev_cfg.local_device_cfg[0].endpoint_cfg[0].direction  = svt_usb_types::IN;
        dev_cfg.local_device_cfg[0].endpoint_cfg[0].ep_type = svt_usb_types::CONTROL;
```

```
        dev_cfg.local_device_cfg[0].endpoint_cfg[0].max_burst_size     = 0;
        dev_cfg.local_device_cfg[0].endpoint_cfg[0].max_packet_size    =
`SVT_USB_SS_CONTROL_MAX_PACKET_SIZE;
        dev_cfg.local_device_cfg[0].endpoint_cfg[0].supports_ustreams  = 1'b0;
      end
if (max_usb_ss_endpoints >= 8)
begin
`vmm_note(log, $psprintf("setup_usb_ss_defaults() - The max_usb_ss_endpoints property is
set to %0d. Configuring up to endpoint number 6. Other endpoints must be configured from
the testcase.", max_usb_ss_endpoints));
end


    host_cfg.usb_capability = svt_usb_configuration::USB_SS_ONLY;
    dev_cfg.usb_capability = svt_usb_configuration::USB_SS_ONLY;
    host_cfg.usb_ss_initial_ltssm_state = svt_usb_types::U0;
    dev_cfg.usb_ss_initial_ltssm_state = svt_usb_types::U0;
    host_cfg.usb_20_signal_interface = svt_usb_configuration::NO_20_IF;


    dev_cfg.usb_20_signal_interface     = svt_usb_configuration::NO_20_IF;
    host_cfg.usb_ss_rx_buffer_mode      = svt_usb_configuration::NOMINAL_EMPTY;
    dev_cfg.usb_ss_rx_buffer_mode       = svt_usb_configuration::NOMINAL_EMPTY;
    host_cfg.usb_ss_rx_buffer_latency   = 0;
    dev_cfg.usb_ss_rx_buffer_latency    = 0;
  end
endfunction
```

Note The USB host configuration must include the configuration of a USB device that the host communicates with. Hence, the remote_device_cfg is constructed.

For more information on the configuration class, refer to the `svt_usb_configuration` and `svt_usb_endpoint_configuration` Class References at the following locations:

- $DESIGNWARE_HOME/vip/svt/usb_svt/latest/doc/usb_svt_vmm_class_reference/html/class_svt_usb_configuration.html
- $DESIGNWARE_HOME/vip/svt/usb_svt/latest/doc/usb_svt_vmm_class_reference/html/class_svt_usb_endpoint_configuration.html
- Construct the customized USB configuration and pass the configuration to the USB environment (`usb_svt_basic_serial_env`) of your testbench environment.
  - Construction of the USB configuration object cfg is done in the constructor function of the env.
  - Configuration required for the test is done in `gen_cfg()` function of the env.
  - Configuration is assigned to the subenv in `build()` function of the env.

```
class usb_svt_basic_serial_env extends vmm_env;
…
usb_svt_shared_cfg cfg;
function usb_svt_basic_serial_env::new(vmm_log log, virtual svt_usb_if   host_usb_if);
  begin
    super.new();
```

```
    this.host_usb_if = host_usb_if;
    this.cfg = new(log);
      …
   end
endfunction


function void usb_svt_basic_serial_env::gen_cfg() ;
  begin
    super.gen_cfg();
   /**
    * Setup the default configuration required for this test
    */
     cfg.setup_usb_ss_defaults();


    `vmm_trace(log, "ENV RUN-FLOW: gen_cfg() - Starting...");
    this.cfg.host_cfg.base_chan_src = svt_usb_subenv_configuration::ATOMIC_GEN;
    this.cfg.host_cfg.stream_id = 0;
    this.cfg.dev_cfg.stream_id = 1;
      …
  end
endfunction
```

The `usb_svt_shared_cfg` is the customized USB configuration as defined in the previous step. The `cfg` is an instance of this configuration.

- Assign configuration to the subenv object in the build phase of your testbench environment.

```
class usb_svt_basic_serial_env extends vmm_env;
…
usb_svt_shared_cfg cfg;
…
function void usb_svt_basic_serial_env::build() ;
begin
…
   super.build();
…
         host_subenv    = new("svt_usb_subenv",
                          "host_subenv",
                          this.consensus,
                          this.cfg.host_cfg,
                          null,
                          this.xact_report,
                          host_usb_ss_port,
                          host_usb_20_serial_port);
…
endfunction
```

For detailed procedure of instantiating and configuring the VIP, refer to the files at the following locations:

- $DESIGNWARE_HOME/vip/svt/usb_svt/latest/examples/sverilog/tb_usb_svt_vmm_basic_sys
  - env/usb_svt_shared_cfg.sv
  - env/usb_svt_basic_serial_env.sv
  - tests/ts.basic_ss_serial.sv

4.1.3 Generate Constrained Random Stimulus

Constrained random generation makes use of the built-in randomize() method that all objects support. The most common use for random generation is to produce a series of random transfers which follow a set of applied constraints. The example shows a simple transfer class whose instance can be passed as `randomized_obj` of transfer atomic generator which randomly produces a string of individual transfers randomly with no particular relationship between them. This string of transfers is a scenario. These transfers may be constrained to have a relationship with one another.

```
/**
 * This class represents a customized USB transfer. This class extends the
 * USB VIP's svt_usb_transfer class, and as such may be used by the VIP
 * USB PROTOCOL (svt_usb_protocol class). This class adds pre-defined distribution
 * constraints for transfer types, and adds integer weighting variables that may
 * be set by testcases to control the distribution of generated transfer types.
 */
class usb_svt_basic_cust_transfer extends svt_usb_transfer;


  /** Weight for selecting control in transfers. */
  int CONTROL_IN_TRANSFER_wt = 0;
  /** Weight for selecting control out transfers. */
  int CONTROL_OUT_TRANSFER_wt = 0;
  /** Weight for selecting bulk out transfers. */
  int BULK_OUT_TRANSFER_wt = 0;
/** Weight for selecting intr in transfers. */
  int INTR_IN_TRANSFER_wt = 0;


  /**
   * Creates distribution of generated SuperSpeed transfer types, according to the
   * weight control values (which are also adjusted for the configuration).
   */
  constraint xfer_type_distribution {
    xfer_type dist {
              svt_usb_transfer::CONTROL_TRANSFER         := (CONTROL_IN_TRANSFER_wt +
CONTROL_OUT_TRANSFER_wt),
              svt_usb_transfer::BULK_OUT_TRANSFER        := BULK_OUT_TRANSFER_wt,
              svt_usb_transfer::BULK_IN_TRANSFER         := BULK_IN_TRANSFER_wt,
              svt_usb_transfer::ISOCHRONOUS_OUT_TRANSFER := ISOC_OUT_TRANSFER_wt,
              svt_usb_transfer::ISOCHRONOUS_IN_TRANSFER  := ISOC_IN_TRANSFER_wt,
              svt_usb_transfer::INTERRUPT_OUT_TRANSFER   := INTR_OUT_TRANSFER_wt,
              svt_usb_transfer::INTERRUPT_IN_TRANSFER    := INTR_IN_TRANSFER_wt
    };
  }
```

```
        /**
        * Creates distribution of generated CONTROL transfer direction, according to the
        * weight control values (which are also adjusted for the configuration).
        */
        constraint setup_data_bmrequesttype_dir_distribution {
          ((CONTROL_IN_TRANSFER_wt + CONTROL_OUT_TRANSFER_wt) > 0) ->
          {
            setup_data_bmrequesttype_dir dist {
                   svt_usb_types::DEVICE_TO_HOST := CONTROL_IN_TRANSFER_wt,
                   svt_usb_types::HOST_TO_DEVICE := CONTROL_OUT_TRANSFER_wt
                  };
          }
        }
/**
 * For Bulk OUTs initiated by Host, constrain payload size
 * Using a smaller value as Maximum payload size, to reduce simulation time
```

Note Value used must be good for both SS and 20, since this transfer class is used in both SS and 20 tests.

```
 */
        constraint payload_intended_byte_count_bulk_out
      {
          (xfer_type == svt_usb_transfer::BULK_OUT_TRANSFER) ->
            payload_intended_byte_count <= 1024;
      }

  /** CONSTRUCTOR: Create a new instance of this class. */
        function new(svt_usb_configuration cfg = null);
          begin
            super.new(cfg);
          end
        endfunction

  /** Extends the method to report this class' name. */
        virtual function string get_class_name();
          begin
            get_class_name = "usb_svt_basic_cust_transfer";
          end
        endfunction

  /** Extends the method to report this class' data values. */
  virtual function string psdisplay(string prefix  = "");
      begin
      end
```

```
  endfunction
```

/** Extends the method to allocate a new instance of this class. */

```
  virtual function vmm_data allocate();
    begin
      usb_svt_basic_cust_transfer  new_xact = new(cfg);
      allocate = new_xact;
      `vmm_verbose(log, $psprintf("usb_svt_basic_cust_transfer::allocate() - Completed,
results:\n%s", new_xact.psdisplay("ALLOC_OBJ::")));
    end
  endfunction
```

/** Extends the method to create a copy of an object of this class. */

```
  virtual function vmm_data copy(vmm_data to = null);
    begin
      usb_svt_basic_cust_transfer  new_inst ;
      vmm_data  new_data = (to == null) ? allocate() : to;

      if(!$cast(new_inst, new_data))
       `vmm_fatal(log,"copy()- Cannot copy to non vmm_data instance");

      void'(super.copy(new_inst));


      new_inst.CONTROL_IN_TRANSFER_wt = this.CONTROL_IN_TRANSFER_wt;
      new_inst.CONTROL_OUT_TRANSFER_wt = this.CONTROL_OUT_TRANSFER_wt;
      new_inst.BULK_OUT_TRANSFER_wt = this.BULK_OUT_TRANSFER_wt;
      new_inst.BULK_IN_TRANSFER_wt = this.BULK_IN_TRANSFER_wt;
      new_inst.ISOC_OUT_TRANSFER_wt = this.ISOC_OUT_TRANSFER_wt;
      new_inst.ISOC_IN_TRANSFER_wt = this.ISOC_IN_TRANSFER_wt;
      new_inst.INTR_OUT_TRANSFER_wt = this.INTR_OUT_TRANSFER_wt;
      new_inst.INTR_IN_TRANSFER_wt = this.INTR_IN_TRANSFER_wt;

      copy = new_inst;


      `vmm_verbose( log, "copy() - Completed");
      `vmm_verbose(log,this.psdisplay("ORIG::"));
      `vmm_verbose(log,new_inst.psdisplay("COPY::"));
    end
  endfunction


  function void pre_randomize();
    super.pre_randomize();
    payload.USER_DEFINED_ALGORITHM_wt   = 1;
    payload.TWO_SEED_BASED_ALGORITHM_wt = 0;
  endfunction
endclass
```

The constraints are the only object information that is required. The atomic generator does not require information specific about the transfers. So the generator code itself can be independent from the testbench code. It simply needs to be given an object to randomize. This is an example of reducing test-specific code and increasing reuse.

The object-based interface provided by VC VIP for USB and VMM creates a subtle yet important shift in thinking. Building on the fact that the models abstract protocols into transfer objects, specifying test conditions is a matter of generating constrained random objects. The shift here is that defining the transfers occurs in protocol terms and uses native language syntax (constraints and assignments), so the verification engineer can think in protocol terms, not model details. This abstraction allows a more natural and efficient thought process.

The object provided to a generator is referred to as a factory object or, simply, a factory. It is a blueprint for randomization and the template for the generated objects. When using VC VIP for USB models, factories are created by extending the provided transfer objects and applying user-defined constraints. When the factory is randomized, the user constraints will be used and, through inheritance rules, the extended objects can be put into a channel. The VIP models also support the addition of user data members in factories, allowing user customization of the transfers themselves.

In addition, a list of random and directed scenarios are available in the VIP examples. For more information on the example scenarios, refer to the example directories at the following location:

$DESIGNWARE_HOME/vip/svt/usb_svt/latest/examples/sverilog/usb_svt/tb_usb_svt_vmm_basic_sys/

- `env/usb_svt_basic_cust_transfer.sv`
- `tests/ts.basic_ss_serial.sv`
- `tests/ts.basic_additional_20_enumeration.sv`

Note
You must set a device response sequence for active devices in the run phase.

4.1.4 Control the Test

The VC VIP models share the same features regarding test control. Although there are many features that support test control, this guide highlights two basic features: starting/stopping and logging. The VC VIP models are all extended from the base class `vmm_xactor` so they all have the same control interface for starting and stopping. Each model implements `start_xactor` and stop_xactor methods that handle all steps needed to either start or stop a model. Encapsulation of function is one of the key elements in the architecture of VC VIP and VMM. The testbench does not have to know how to start a model; it simply has to call the `start_xactor` method.

The start task in the environment class is where the test is launched. This generally means that any models or components in the simulation are started. The code snippet below is from the example. This simulation consists of VC VIP models with the start code (simply calls the start_xactor method of each model).

```
`ifdef SVT_MULTI_SIM_AUTOMATIC_DESIGN_UNIT
program basic_system_test ();
`else
program automatic basic_system_test ();
`endif
initial begin
   vmm_log log;
   example_env env;
   usb_svt_basic_cust_transfer host_xfer_factory;
log = new("basic_system_test", "program");
   env = new(log);
   env.build();
```

```
    if (env.host_subenv.xfer_atomic_gen != null) begin
      host_xfer_factory = new(env.cfg.host_cfg);
      host_xfer_factory.CONTROL_IN_TRANSFER_wt   = 1;
      host_xfer_factory.CONTROL_OUT_TRANSFER_wt  = 1;
      host_xfer_factory.BULK_OUT_TRANSFER_wt     = 2;
      host_xfer_factory.BULK_IN_TRANSFER_wt      = 2;
      env.host_subenv.xfer_atomic_gen.randomized_obj = host_xfer_factory;
    end

    env.start();
    fork
      begin
```

/** * Start the directed test */

```
        do_directed_test(env.host_subenv.xfer_atomic_gen, env.cfg.host_cfg,env.log);
```

/*** Starting atomic generator in Host Subenv*/

```
        env.host_subenv.xfer_atomic_gen.start_xactor();
      end
    join_none

    env.wait_for_end();
    env.cleanup();

    env.intermediate_report = 0;
    env.final_report = 1;
    env.report();
  end
```

/**This task provides directed (manual) stimulus to VIP.

* Atomic generator is passed so that manual stimlus is injected via VIP subenv's atomic generator.

* Cfg is passed so that stimus objects (transfer type) are created with current cfg transfer count is passed by reference such that if additional manual stimulus tasks were to be added, the aggregate count is avaialble globally in test case */

```
task do_directed_test(svt_usb_transfer_atomic_gen xfer_atomic_gen, svt_usb_configuration
cfg, vmm_log log);
   begin
   end
endtask : do_directed_test
endprogram
```

Another standard function that the VC VIP models provide for controlling tests is logging and messaging. There are many ways to control and configure logging. We examine a few basic ones that will get you going. Messaging is performed by vmm_log objects (logs), which are included in every model. By the way, the example_env that we created also contains a log that it inherits from vmm_env. Inheritance and reuse go hand in hand.

VMM defines message types and severities. This allows two degrees of freedom for sorting and operating on messages. Severities are defined as levels ranging from FATAL to DEBUG. These levels allow the messages to be processed distinctly.

Each log object has a threshold level and will log any messages with a severity equal to or higher than the threshold. The level of logging detail is set with the `set_verbosity` method, as the following example code:

```
// Configure how much messaging to display
    this.log.set_verbosity(vmm_log::NORMAL_SEV);
```

The `vmm_log` class has many powerful and flexible features. For example, several of the methods can operate globally on all log objects. A global format can also be specified so that all messages use the same format. These testbench-wide controls are especially useful when integrating multiple models into a single simulation.

4.2 Compiling and Simulating a Test with the VIP

The steps for compiling and simulating a test with the VIP are described in the following sections:

- "Directory Paths for VIP Compilation"
- "VIP Compile-time Options"
- "VIP Runtime Option"

4.2.1 Directory Paths for VIP Compilation

You need to specify the following directory paths in the compilation commands for the compiler to load the VIP files.

```
+incdir+project_directory_path/include/sverilog
+incdir+project_directory_path/src/sverilog/simulator
+incdir+project_directory_path/include/verilog
+incdir+project_directory_path/src/verilog/simulator
```

Where, `project_directory_path` is your project directory and simulator is vcs, ncv or mti. For example:

```
+incdir+/home/project1/testbench/vip/include/sverilog
+incdir+/home/project1/testbench/vip/src/sverilog/vcs
+incdir+/home/project1/testbench/vip/include/verilog
+incdir+/home/project1/testbench/vip/src/verilog/vcs
```

4.2.2 VIP Compile-time Options

The following are the required compile-time options for compiling a testbench with the VC VIP for USB:

```
+define+SVT_VMM_TECHNOLOGY
-ntb_opts rvm      (To analyze SV files with VMM)
+define+SYNOPSYS_SV
```

Table 4-2 Macro

Macro	Description
`SVT_VMM_TECHNOLOGY`	Specifies SystemVerilog based VIPs that are compliant with VMM.
`SYNOPSYS_SV`	Specifies SystemVerilog based VIPs that are compliant with VMM.

4.2.3 VIP Runtime Option

No VIP specific runtime option is required to run simulations with the VIP. Only relevant VMM runtime options are required.

For example,

```
./output/simvcssvlog +vmm_log_nowarn_at_200 -l ./logs/simulate.log run
```

5 The USB Sub-environment

VMM system-level verification environments are constructed with transactors, self-checking structures, and other components that are designed for flexible reuse. This facilitates the capability of creating custom environments for testing a variety of designs through the a common component set. When a VIP defines multiple interacting transactors, implementing an environment subset that comprises basic components can minimize maintenance, simplify instantiation, and reduce the complexity of VIP as viewed from the highest abstraction level.

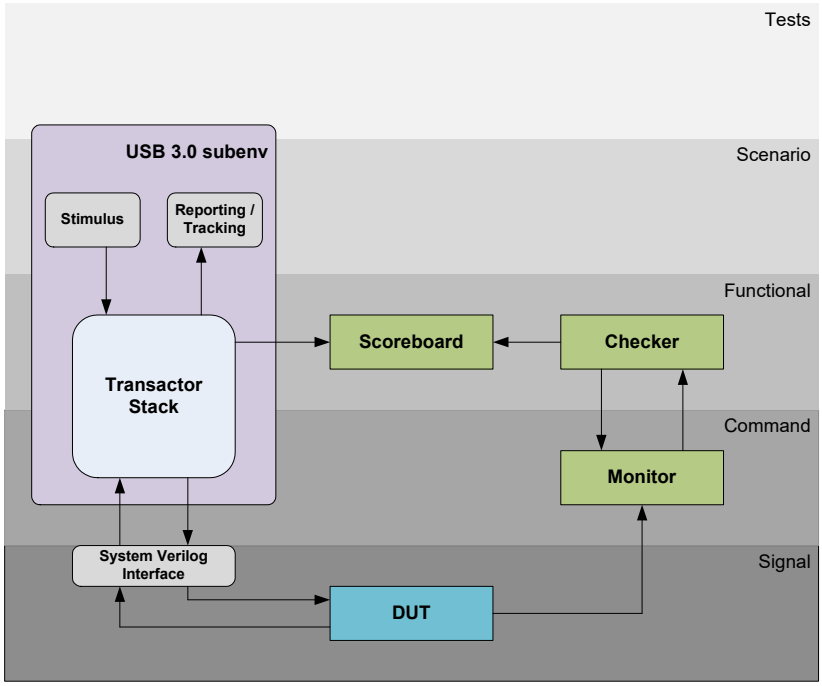
Sub-environments are VMM components that are composed of basic components such as transactors and stimulus generators. The USB VIP defines a sub-environment that can be customized to support a variety of verification scenarios.

This chapter describes the USB VIP sub-environment object. Refer to the Class Reference HTML for a description of objects, classes, and attributes mentioned in this chapter.

5.1 Description

USB sub-environments are component objects in a VMM-compliant verification environment. The sub-environment object, *svt_usb_ subenv*, extends from *svt_subenv,* which extends from *vmm_subenv*. Figure 5-1 depicts the implementation of a USB sub-environment within a typical VMM model.

Figure 5-1 The USB Sub-environment within a VMM Model



As shown in Figure 5-1, the USB subenv consists of the following functional components:

- **Transactor Stack:** This component specifies the active USB protocol transactors that the sub-environment contains.
- **Stimulus:** This component specifies the objects that inject test objects into the transactor stack.
- **Reporting / Tracking:** This component contains the objects that maintain reports, coverage, and consensus objects created and utilized by the sub-environment.

The following sections describe the components that comprise the USB sub-environment.

5.2 Sub-environment Components

5.2.1 Transactor Stack

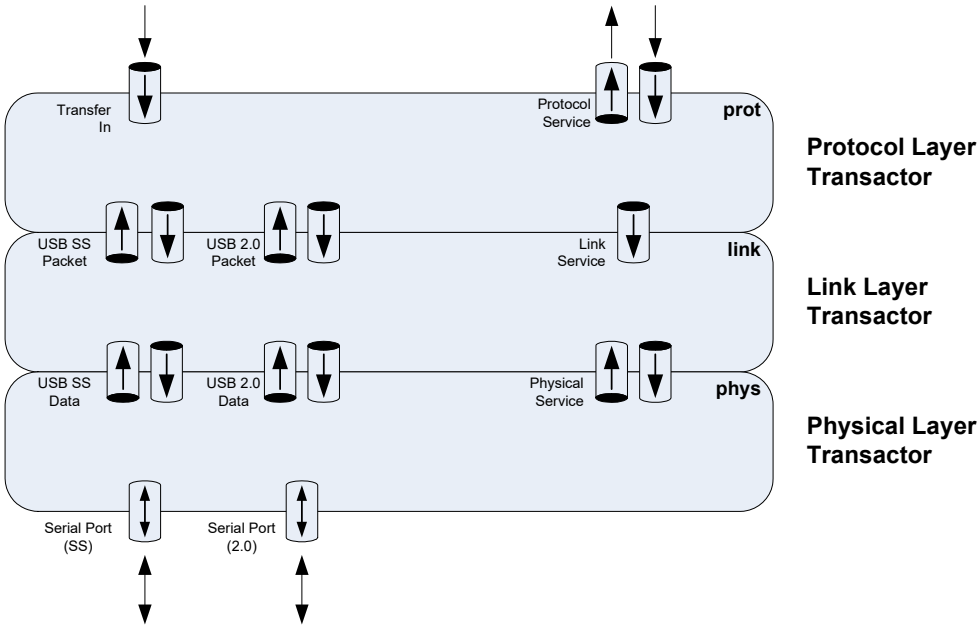
USB transactors are component objects in a VMM-compliant environment that implement a specific layer of the USB protocol. The USB VIP defines four transactors, each of which extends from the *svt_xactor* class:

- Protocol transactor
- Link transactor
- Physical transactor
- SSIC Physical transactor

USB Verification IP Transactors describes each USB transactor.

The transactor stack is a logical sub-environment element that consists of instantiated USB transactors. A valid stack consists of up to four transactors. The transactor set contained in the subenv is configurable and specified by sub-environment constructor parameters. Figure 5-2 displays an example protocol stack.

Figure 5-2 USB Protocol Stack Example



The set of transactors within a stack are contiguous. The subenv class defines four transactor attributes that correspond to the four transactors configurable within the sub-environment. The following lists the four transactor 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 transactors reside on the same side of the USB bus, relative to the serial bus connecting the two USB entities (host-device). The remote_phys transactor resides on the opposite side of the serial bus from the other transactors.

The sub-environment configuration specifies that top and bottom transactors in the sub-environment through bottom_xactor and top_xactor attributes. The top layer of the transactor stack receives stimulus objects that drive the verification through a channel interface. While the most typical stacks install the Protocol transactor at the top of the stack, the link or physical transactors may also be defined as the top transactor. Stimulus data that is input into the stack must be consistent with that accepted by the top layer of the defined stack. Refer to the Data Objects section of the respective transactor descriptions in USB Verification IP Transactors for detailed information.

Test environments can provide the sub-environment with input channels to provide stimulus input. Although the sub-environment includes generic vmm_channel parameters to support this, any channel that the testbench supplies must be compatible top transactor. For example, if the Protocol transactor is at the top of the stack, then the channel must be an svt_usb_transfer_channel.

When input channels are not provided by the test environment, the sub-environment instantiates scenario generators to provide the stimulus. In this case the sub-environment automatically enables the scenario generator compatible with the top transactor. For example, if link transactor is at the top of the stack, the packet scenario generator is enabled while the transfer and data scenario generators are not.

The svt_usb_subenv does not create default output channels. If no output channels are provided by the test environment, then the output channels are not created.

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

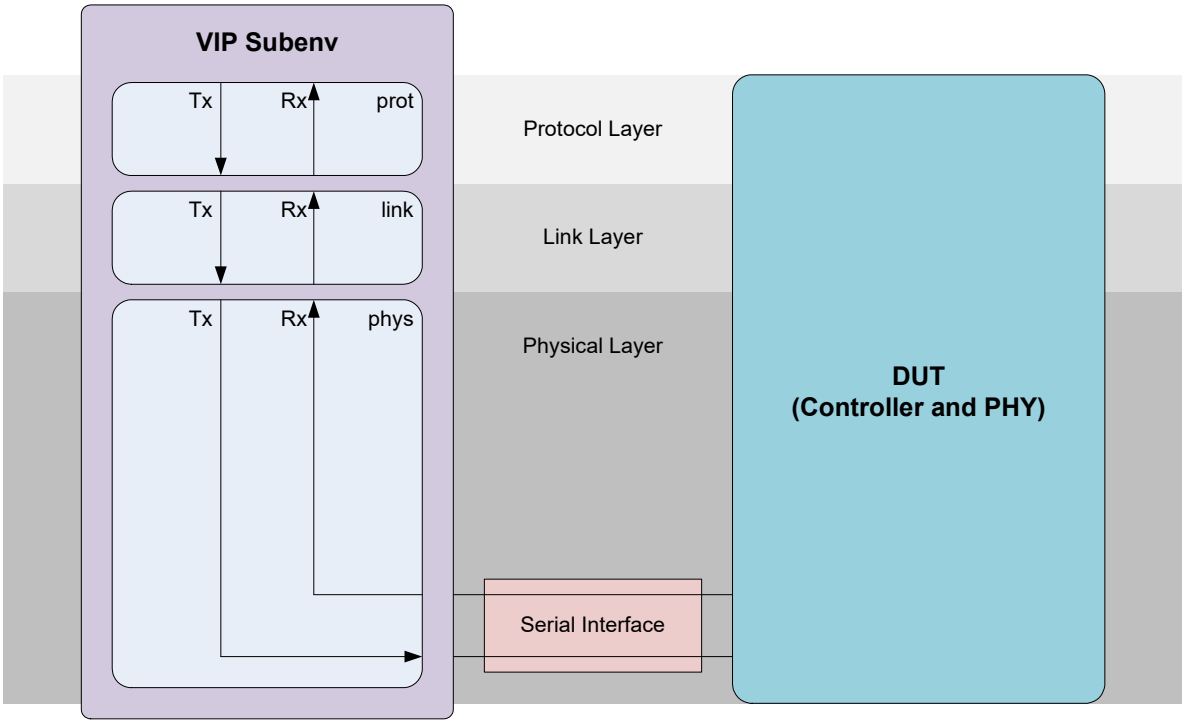
The following examples display transactor stack formations in common sub-environment configurations. Refer toVerification Topologies for additional examples.

Example 1: Verifying a Controller-PHY (Figure 5-3)

The VIP sub-environment in Figure 5-3 utilizes a three layer protocol stack, containing protocol, link, and physical transactors. If the VIP is configured as a host, the protocol transactor receives stimulus data through a transfer channel; otherwise, the protocol transactor receives transfers through an input channel in response to the Rx data stream.

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

Figure 5-3 Verifying a USB Controller and PHY

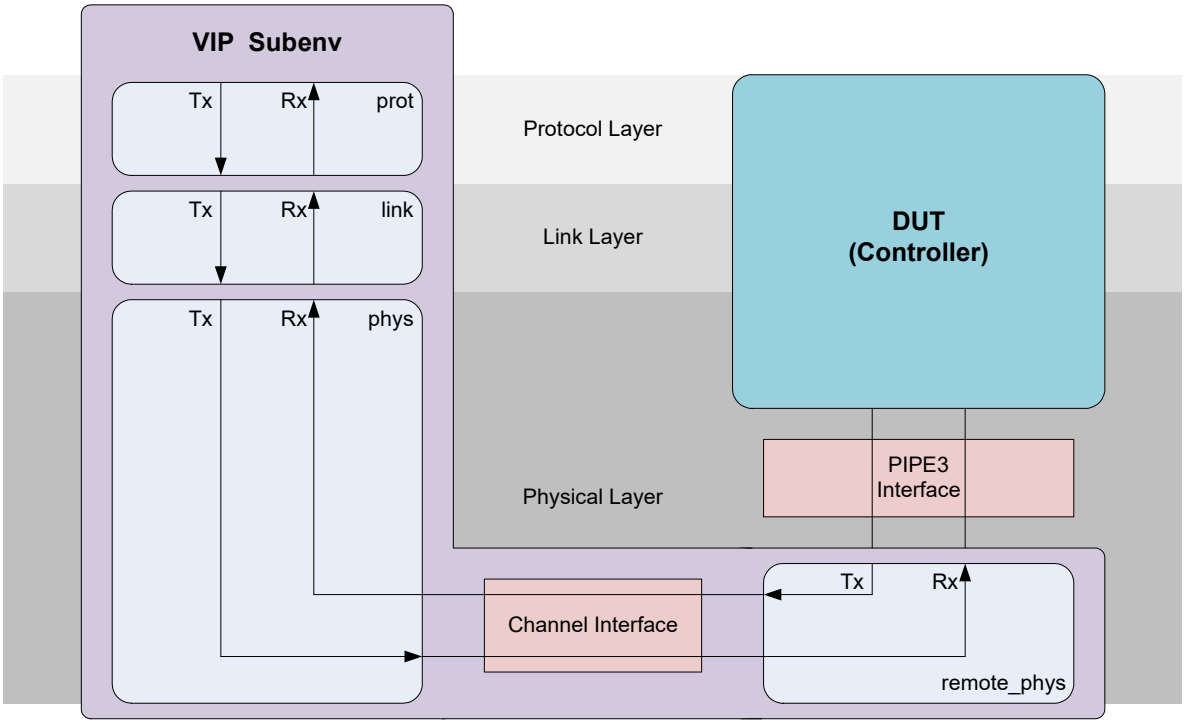


Example 2: Verifying a Controller (Figure 5-4)

The VIP sub-environment in Figure 5-4 utilizes a four layer protocol stack, containing protocol, link, and two physical transactors. If the VIP is configured as a host, the protocol transactor receives stimulus data through a transfer channel; otherwise, the protocol transactor receives transfers through an input channel in response to the Rx data stream.

The two physical transactors connect through a channel interface. The remote physical transactor 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 transactor interfaces.

Figure 5-4 Verifying a USB Controller



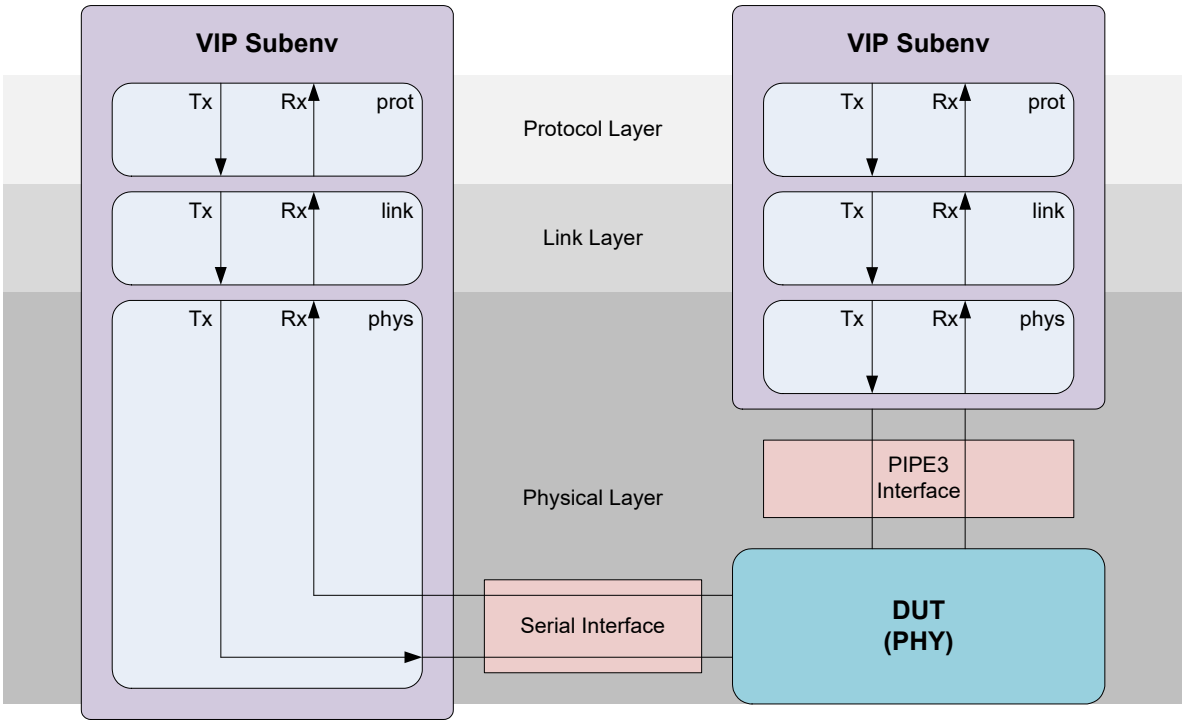
Example 3: Verifying a PHY (Figure 5-5)

The environment Figure 5-5 utilizes two sub-environments. The sub-environment on the left side of Figure 5-5 is remote from the DUT, whereas the sub-environment on the right side is local to the DUT.

The sub-environment that is remote from the DUT contains a three layer transactor stack, similar to the sub-environment in Figure 5-3. The Physical transactor connects to the DUT through a serial interface.

The sub-environment that is local to the DUT also contains a three layer transactor stack. Unlike the Physical transactor on the remote side, which simulates the MAC and PHY portions of the USB physical layer, the Physical transactor 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 sub-environment and the PHY side of the interface must connect to the DUT.

Figure 5-5 Verifying a USB Controller



5.2.2 Stimulus Objects

The USB VIP support four methods of providing stimulus objects to the sub-environment:

5.2.2.1 Direct channel

In direct channel injection, the user places transaction objects (see Transaction Objects) into channels located on the top transactor.

5.2.2.2 Atomic Generator

Atomic generators are transactors derived from vmm_xactor. These generators produce individual objects randomly without regard for any relationship between the objects.

The following macros accept arguments that define atomic generator classes.

- usb_20_data_atomic_gen: Populate and use this generator if you want to generate and send (constrained) randomized USB 2.0 data, unrelated to USB Packets, and with no interdependence.
- usb_ss_data_atomic_gen: Populate and use this generator if you want to generate and send (constrained) randomized SS data, unrelated to USB Packets, and with no interdependence.
- link_service_atomic_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the link layer, with no interdependence between such requests.
- usb_20_pkt_atomic_gen: Populate and use this generator if you want to generate and send (constrained) randomized USB 2.0 packets, unrelated to USB Transactions or Transfers, and with no interdependence.

- usb_ss_pkt_atomic_gen: Populate and use this generator if you want to generate and send (constrained) randomized SS packets, unrelated to USB Transactions/Transfers, and with no interdependence.
- usb_20_phys_service_atomic_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the USB 2.0 physical layer, with no interdependence between such requests.
- usb_ss_phys_service_atomic_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the SS physical layer, with no interdependence between such requests.
- prot_service_atomic_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the protocol layer, with no interdependence between such requests.
- xfer_atomic_gen: Populate and use this generator if it is sufficient to generate transfer (svt_usb_transfer) instances with randomized properties, but no interdependence.

5.2.2.3 Scenario Generator

Scenario generators are transactors derived from vmm_xactor. These generators create sequences of instances of a specified factory class. The sequence is represented as an array of objects. Constraints define the rules governing the sequence of objects that the generator creates. When the array of transactions is randomized, an entire sequence is generated. Scenarios can even generate sequences of sequences.

The following macros accept arguments that define scenario generator classes.

- usb_20_data_scenario_gen: Populate and use this generator if your test needs to create sequences of USB 2.0 data, unrelated to USB Packets, but with interdependence between their randomized properties.
- usb_ss_data_scenario_gen: Populate and use this generator if your test needs to create sequences of SS data, unrelated to USB Packets, but with interdependence between their randomized properties.
- link_service_scenario_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the link layer, with interdependence between such requests.
- usb_20_pkt_scenario_gen: Populate and use this generator if your test needs to create sequences of USB 2.0 packets, unrelated to USB Transactions or Transfers, but with interdependence between their randomized properties.
- usb_ss_pkt_scenario_gen: Populate and use this generator if your test needs to create sequences of SS packets, unrelated to USB Transactions or Transfers, but with interdependence between their randomized properties.
- usb_20_phys_service_scenario_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the USB 2.0 physical layer, with interdependence between such requests.
- usb_ss_phys_service_scenario_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the SS physical layer, with interdependence between such requests.
- prot_service_scenario_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the protocol layer, with interdependence between such requests.

- xfer_scenario_gen: Populate and use this generator if your test needs to create sequences of sets of transfers that have interdependence in their randomized properties.

For detailed information about these macros, see the Class Reference HTML.

5.2.2.4 Multi-stream Scenario Generator

These generators generate and coordinate stimulus across multiple interfaces. Multi-stream scenario generators also allows hierarchical layering of scenarios and the reuse of block level scenarios at system level. Multi-stream scenarios are described by extending the vmm_ms_scenario class.

The following macros accept arguments that define multi-stream scenario generator classes:

- usb_20_data_ms_scenario_gen: Populate and use this generator if your test needs to create sequences of USB 2.0 data, unrelated to USB Packets, but with interdependence between their randomized properties.
- usb_ss_data_ms_scenario_gen: Populate and use this generator if your test needs to create sequences of SS data, unrelated to USB Packets, but with interdependence between their randomized properties.
- link_service_ms_scenario_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the link layer, with interdependence between such requests.
- usb_20_pkt_ms_scenario_gen: Populate and use this generator if your test needs to create sequences of USB 2.0 packets, unrelated to USB Transactions or Transfers, but with interdependence between their randomized properties.
- usb_ss_pkt_ms_scenario_gen: Populate and use this generator if your test needs to create sequences of SS packets, unrelated to USB Transactions or Transfers, but with interdependence between their randomized properties.
- usb_20_phys_service_ms_scenario_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the USB 2.0 physical layer, with interdependence between such requests.
- usb_ss_phys_service_ms_scenario_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the USB 2.0 physical layer, with interdependence between such requests.
- prot_service_ms_scenario_gen: Populate and use this generator if you want your test to issue (constrained) randomized service requests to the protocol layer, with interdependence between such requests.
- xfer_ms_scenario_gen: Populate and use this generator if your test needs to create sequences of sets of transfers that have interdependence in their randomized properties.

For detailed information about these macros, see the Class Reference HTML.

5.2.3 Reporting and Tracking Objects

The sub-environment support the following reporting and tracking objects:

5.2.3.1 Checks

VIP transactors use an object-based mechanism for defining and encapsulating checks dynamically performed by the transactors. Such an object oriented approach is useful for control of 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 transactors) 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 transactors. 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 a list of checks provided by these classes, see Class Reference HTML.

Checks are enabled by default. The following sub-environment configuration attributes enable checks:

- enable_prot_chk
- enable_link_chk
- enable_phys_chk

The VIP provides messages directly through vmm_log, including note, trace, debug, verbose, warning, error and fatal vmm_log messages. User set display severity through standard vmm_log methods.

5.2.3.2 Reporting

The test environment can selectively enable and disable transaction reporting. Enabling this results in displaying all top level transactor input transactions in the log. The following sub-environment configuration attributes enable reporting:

- enable_link_reporting
- enable_phys_reporting
- enable_prot_reporting

The sub-environment also permits the generation of trace files by each transactor. The following sub-environment configuration attributes enables trace generation:

- enable_link_tracing
- enable_phys_tracing
- enable_prot_tracing

5.2.4 Notifications

Sub-environments denote significant events through notifications and include transactions as data objects with these notifications. Testbenches can be configured to wait for notifications and receive transactions as part of a notification.

The following are the notification attributes defined by the sub-environment.

- 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

These notifications are VMM notification IDs associated with the sub-environment's 'notify' instance.

For more information, see Class Reference HTML.

6
USB Verification IP Transactors

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

6.1 SuperSpeed Packet Scenarios

The class svt_usb_packets represents a USB packet VMM/VIP transaction. The packet includes the packet fields, but also contains a list of the physical data objects that make up the packet.

The #speed attribute is the entry point to this object. If #speed attribute is assigned to svt_usb_types::SS then USB Super-speed attributes apply; otherwise, USB 2.0 attributes apply.

The following table describes various super-speed link scenarios, and the resulting settings for the noted status-type attributes for each of those scenarios:

Table 6-1 SuperSpeed Packet Scenarios

Packet Atrribute	Direction N: set by	Packet with payload good header -LGOOD	Packet without payload good header -LGOOD	Packet with payload bad header -LBAD	Packet without payload bad header -LBAD	Packet with payload good header payload aborted (EDB) -LGOOD	Replayed packet good header -LGOOD	Packet with payload dropped via callback	HDP only dropped via callback
status	TX (link_ss_tx)	ACCEPT	ACCEPT	PARTIAL_ACCEPT (not ENDED)	PARTIAL_ACCEPT (not ENDED)	ACCEPT	ACCEPT	ABORTED	ABORTED
	RX (link_ss_rx)	ACCEPT	ACCEPT	ACCEPT (not ENDED)	ACCEPT (not ENDED)	ACCEPT	ACCEPT	ABORTED	ABORTED
header_status	TX (link_ss_tx)	ACCEPT	ACCEPT	PARTIAL_ACCEPT	PARTIAL_ACCEPT	ACCEPT	ACCEPT	ABORTED	ABORTED
	RX (object_detect)	ACCEPT	ACCEPT	ACCEPT	ACCEPT	ACCEPT	ACCEPT	ACCEPT	ACCEPT

Table 6-1 SuperSpeed Packet Scenarios (Continued)

Packet Atrribute	Direction N: set by	Packet with payload good header -LGOOD	Packet without payload good header -LGOOD	Packet with payload bad header -LBAD	Packet without payload bad header -LBAD	Packet with payload good header payload aborted (EDB) -LGOOD	Replayed packet good header -LGOOD	Packet with payload dropped via callback	HDP only dropped via callback
payload_ststatus	TX (link_ss_tx)	ACCEPT	DISABLED	PARTIAL_ ACCEPT	DISABLED	ACCEPT	CANCELL ED	INITIAL	DISABLED
	RX (object_ detect)	ACCEPT	ACCEPT	PARTIAL_ ACCEPT (not ENDED)	PARTIAL_ ACCEPT (not ENDED)	ACCEPT	ACCEPT	ABORTED	ABORTED
payload_presence	TX (protocol / generator)	ACCEPT	DISABLED	ACCEPT	DISABLED	ACCEPT	DISABLED	ACCEPT	DISABLED
	RX (object_ detect)	PAYLOAD_ PRESENT	PAYLOAD_ NOT_PRE SENT	PAYLOAD_ PRESEN	PAYLOAD_ NOT_ PRESENT	PAYLOAD_ PRESENT _BUT_ ABORTED (set by Link Transmitter)	PAYLOAD_ NOT_ PRESENT (set by Link Transmitter)	PAYLOAD_ PRESENT	PAYLOAD_ NOT_ PRESENT

6.2 Protocol Transactor

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

The USB Protocol transactor object extends from the *svt_xactor* class, which extends from the *vmm_xactor* base class. This object implements all methods specified by VMM for the *vmm_xactor* class.

The Class Reference HTML describes Protocol transactor functions and attributes.

6.2.1 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:

- Channel support
  - Transfer In, Out and Response
  - Service In, Out (ITP, SOF, LMP, LPM)
- Configurable Stimulus to input channel
  - Direct channel
  - VMM atomic generator
  - VMM scenario generator

- VMM multi-stream scenario generator

- Error injection
  - USB Transfer
  - USB Transaction
  - USB Packets
- Testbench visibility and control through callbacks
- Randomization factories
  - USB Transfer
  - USB Transaction
  - USB Packet
  - Exception lists

6.2.2 Protocol Transactor Channels

The following list describes objects that move information between the Protocol Transactor and other VIP objects. Figure 6-1 displays Protocol transactor information flow and the following objects:

- **Transfer channels:** These channels convey USB transfer objects between the testbench and the Protocol transactor. The transactor supports the following Transfer channels.
  - ***Transfer Input channel:*** This channel receives transfer objects that represent the USB transfers that will be initiated by the protocol transactor when the VIP is acting as USB Host. This channel is mandatory on host transactors and not used on device transactors.

    Protocol transactor attribute name: *transfer_in_chan*.
  - ***Transfer Output channel:*** When the VIP is acting as USB Device (Peripheral), it creates a new transfer data object when it receives a packet that represents the start of a new transfer. Each time this happens, the new transfer data object may (optionally) be an output to the testbench through this channel. The testbench can then fill in the transfer details that control how the VIP Device responds to this new transfer. This channel is optional on device transactors and not used by host transactors.

    Protocol transactor attribute name: *transfer_out_chan*.
  - ***Transfer Response channel:*** This channel receives transfer objects that respond to objects previously sent through the Transfer Output channel. Transactors should only place objects on this channel that were previously received on the Transfer Output channel, with appropriately modified internal details. This channel is optional on device transactors, not used by host transactors.

    Protocol transactor attribute name: *transfer_response_chan*.
- **Packet channels:** These channels convey USB packet objects between the Protocol and Link transactors. The transactor supports the following Packet channels:
  - ***Packet Input channels:*** These channels receive USB packets from Link transactors. Each channel supports either USB SS or USB 2.0 traffic.

    Protocol transactor attribute name: *usb_ss_packet_in_chan, usb_20_packet_in_chan*.
  - ***Packet Output channels:*** These channels transmit USB packets to Link transactors. Each channel supports either USB SS or USB 2.0 traffic.

Protocol transactor attribute name: *usb_ss_packet_out_chan, usb_20_packet_out_chan*.

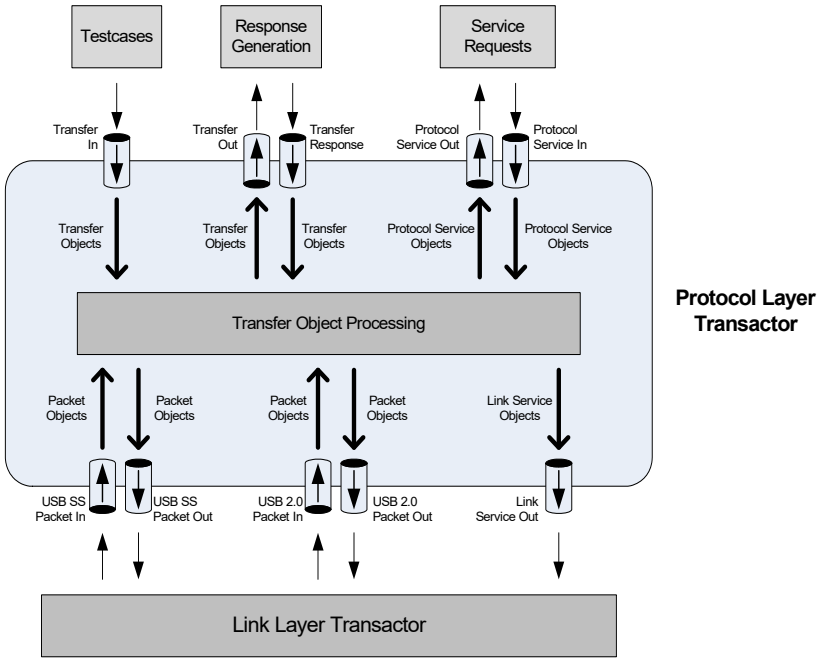
- **Protocol Service channels:** These channels convey USB Protocol Service objects between the testbench and the Protocol transactor. The transactor supports the following channels:
  - ***Protocol Service Input channels:*** This channel receives Protocol Service objects from the testbench.

    Protocol transactor attribute name: *protocol_service_in_chan*.
  - ***Protocol Service Output channels:*** This channel transmits Protocol Service objects to the testbench.

    Protocol transactor attribute name: *protocol_service_out_chan*.
- **Link Service channel:** This channel transmits Link Service objects to the Link transactor.

  Protocol transactor attribute name: *link_service_out_chan*.

Figure 6-1 Protocol Transactor Data Flow



6.2.3 Data Objects

The following is a list of objects that represent information the Protocol Transactor receives, sends, or processes. Figure 6-1 displays the flow of information objects within the Protocol transactor.

Transaction 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, as described in section 4.4 "Generalized Transfer Description" of the USB specification.
- **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 transactor.

- **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.

6.2.4 Protocol Transactor Modes

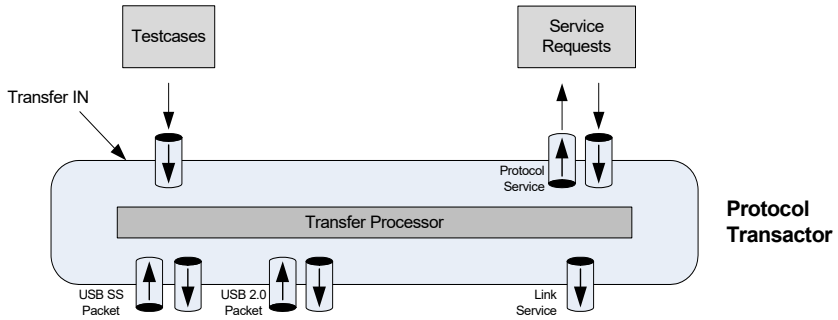
6.2.4.1 Host Mode

When configured as a USB Host, the Protocol transactor transfers data to and from USB device endpoints. The transactor has a single transfer in channel, through which it receives objects. Figure 6-2 displays the structure of the USB VIP in Host mode.

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 channel and does not use Transfer Out or Transfer Response channels.

Figure 6-2 Protocol Transactor – Host Mode

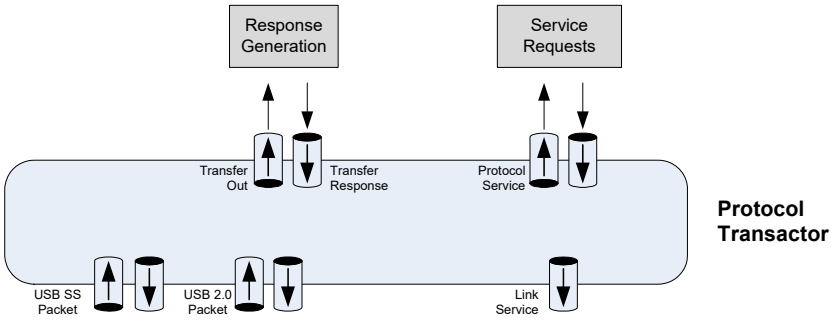


6.2.4.2 Device Mode

When configured as a USB Device, the Protocol transactor receives data and responds to data requests from USB Hosts. The Protocol transactor constructs transfer objects from inbound traffic received from USB interfaces, then sends these objects to the testbench through the Transfer Out channel. The testbench responds to transfer objects sent by the Protocol transactor through the Transfer Response channel. If the testbench does not provide any response transfer object, the Protocol transactor randomizes a transfer response.

Figure 6-3 displays the structure of the USB VIP in Device mode. Transfer Out and Transfer Response channels are optional in Device mode. If these channels are not present, the testbench can send response transfer objects through callback methods. Device mode does not use the Transfer In channel.

Figure 6-3 Protocol Transactor – Device Mode



6.2.5 Data Transformation Objects

The following list describes Protocol Transactor objects that manipulate data objects. Figure 6-4 displays the objects that affect Protocol transactor data flow objects.

6.2.5.1 Transfer Processing

The transactor accepts transfers from the Transfer In channel until the channel 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 transactor 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 transactor by the testbench. Transfers for different endpoints or streams are mixed as specified by the scheduling technique defined in the transactor.

The transactor 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 transactor 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.

6.2.5.2 Factory Objects

The Protocol transactor supports the following factories:

- **Transfer factory:** These factories create transfer objects.

  Protocol transactor attribute name: *randomized_usb_20_transfer_response, randomized_usb_ss_ep_mgr_transfer, randomized_usb_ss_transfer_response, usb_ss_ep_mgr_transfer_factory*

- **Transaction factory:** These factories create transaction objects.

  Protocol transactor attribute name: *randomized_usb_20_transaction, randomized_usb_ss_transaction, usb_20_transaction_factory, usb_ss_transaction_factory.*

- **Packet factory:** These factories create packets for transmission to the Link Transactor through the Packet Output channels.

  Protocol transactor attribute names: *randomized_usb_20_packet, randomized_usb_ss_packet, usb_20_packet_factory, usb_ss_packet_factory.*

- **Protocol service factory:** These factories create Protocol Service transfers.

  Protocol transactor attribute name: *randomized_usb_protocol_service_response, protocol_service_factory.*

Figure 6-4 displays a functional representation of the Protocol transactor factory objects

6.2.5.3 Exception List Factories

USB Protocol transactors 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 6-4 displays a functional representation of the Protocol transactor exception list factories.

The Protocol transactor supports the following exception list factories:

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

  Protocol transactor 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.*

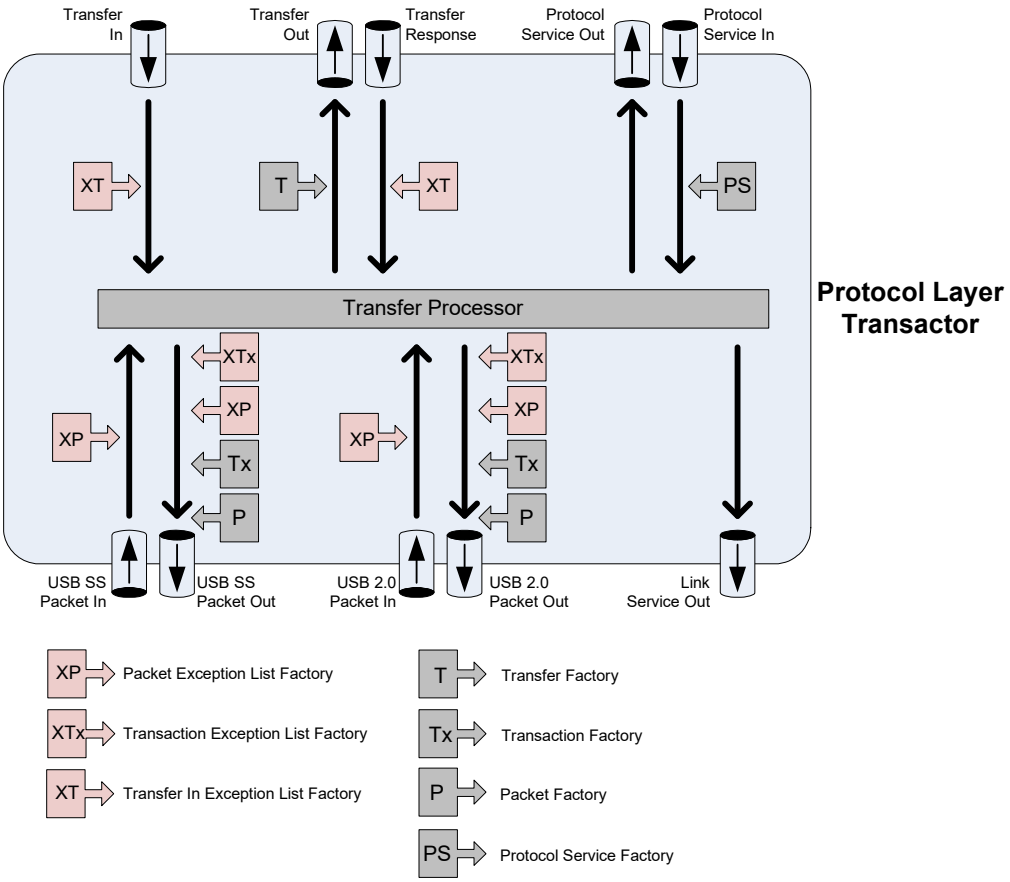
- **Transaction exception list factory:** These randomization factories create exceptions that are injected into USB transactions.

  Protocol transactor 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 transactor attribute name: *randomized_transfer_in_exception_list, randomized_transfer_response_exception_list.*

Figure 6-4 Protocol Transactor Factories – Objects and Exception List



6.2.5.4 Callbacks

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

The Protocol transactor supports the following callbacks:

- **Packet object callbacks:** These callbacks are triggered by or report status on packets.

  Protocol transactor callback method names: `discarded_20_packet, discarded_ss_packet, invalid_packet, new_20_response_transfer, new_ss_response_transfer, post_usb_20_packet_in_chan_get, post_usb_ss_packet_in_chan_get, pre_rx_packet, pre_usb_20_packet_out_chan_put, pre_usb_ss_packet_out_chan_put, randomized_packet, received_data_packet, usb_20_packet_in_chan_cov, usb_ss_packet_in_chan_cov, usb_20_packet_out_chan_cov, usb_ss_packet_out_chan_cov`.

- **Transaction object callbacks:** These callbacks are triggered by or report status on transactions.

  Protocol transactor callback method names: `invalid_transaction, link_service_out_chan_cov, pre_transaction, protocol_service_in_chan_cov, protocol_service_out_chan_cov, randomized_transaction, transaction_ended`

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

Protocol transactor callback method names: `invalid_transfer, new_ss_ep_mgr_transfer, post_transfer_in_chan_get, post_transfer_response_chan_get, pre_transfer_out_chan_put, protocol_service_ended, randomized_ep_mgr_transfer, randomized_transfer_basic_response, randomized_transfer_complete_response, transfer_ended, transfer_in_chan_cov, transfer_out_chan_cov, transfer_response_chan_cov`

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

  Protocol transactor callback method names: `invalid_protocol_service, post_protocol_service_in_chan_get, pre_link_service_out_chan_put, pre_protocol_service_out_chan_put, randomized_protocol_service_response`

6.2.5.5 Notifications

The USB Protocol transactor supports VMM and USB VIP specific notifications. Notifications for which the current transaction is pertinent include a handle to the current data entity (transfer, transaction, or packet) with the notification data. Notifications cannot cause an immediate change to the transactor behavior because they are non-blocking. Callbacks are available to support test bench modification of behavior.

The Protocol transactor supports the following Notifications

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

  Protocol transactor 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 notifications:** These notifications are triggered by or report status on transactions.

  Protocol transactor attribute names: `NOTIFY_USB_TRANSACTION_ENDED`

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

  Protocol transactor 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 notifications:** These notifications are triggered by or report status on service objects.

  Protocol transactor attribute names: `NOTIFY_LINK_ENTERED_RECOVERY, NOTIFY_LINK_ENTERED_U0, NOTIFY_PORT_CONFIG_LMP_REQUIRED`

The `svt_usb_protocol_callbacks` class defines Protocol transactor callbacks.

6.2.6 Protocol Transactor Status

The USB VIP Protocol Transactor provides the following status information

- **Bus Interval/frame/µframe:** The protocol transactor updates a set of shared status objects to provide:
  - The number of bus intervals since the transactor started or reset.
  - The current bus interval start time (realtime).
  - The number of µframes since the transactor started or reset.

- The current µframe start time (realtime).

- **Bus speed:** The protocol transactor retrieves bus speed out from the shared status object. The transactor retrieves SS and 2.0 speeds if both are active.
- **Link state:** The protocol transactor retrieves the link state from stored in the shared status object.

6.2.7 Protocol Transactor SuperSpeed Link Callback, Factory, and Notification Flows

This section provides detailed information about the sequence and content of callbacks provided by the VIP. The flows are based on whether the model is actiona as a Host or a Device.

6.2.7.1 Common Flows

This section shows how the most common flows are implemented.

6.2.7.1.1 Creating and Preparing a New Device Transfer

The following steps outline how to create a and prepare a new device transfer:

- svt_usb_protocol::randomized_usb_ss_transfer_response

  This factory object is used to create a new transfer object for the device VIP. The factory's allocate() method is called to obtain the new object.

- svt_usb_protocol_callbacks::new_ss_response_transfer(svt_usb_protocol xactor, ref svt_usb_transfer transfer)

  Allows the user to modify the new transfer object prior to doing the basic first time randomization.

- transfer.randomize()

  The transfer object that is randomized is the new transfer object returned by the previous callback. This is unlike objects from other factories which randomize the factory object and then take a copy after the randomization.

- svt_usb_protocol_callbacks::randomized_transfer_basic_response(svt_usb_protocol xactor, svt_usb_transfer transfer)

  Allows the user to examine or modify the new transfer object after the basic randomization.

- svt_usb_protocol_callbacks::pre_transfer_out_chan_put(svt_usb_protocol xactor, int chan_id, ref svt_usb_transfer transfer, ref bit drop)

  This callback gives the user another access to the new transfer object prior to putting it in the transfer_out_chan channel. Also, there is the opportunity to drop the transfer and not put it in the channel.

- Following steps if the drop bit was returned false in the previous callback to pre_transfer_out_chan_put().

  a. svt_usb_protocol_callbacks::transfer_out_chan_cov(svt_usb_protocol xactor, int chan_id, svt_usb_transfer transfer)

     This callback is only made if the transfer was not dropped by the previous pre_transfer_out_chan_put() callback.

  b. svt_usb_protocol::transfer_out_chan.sneak(transfer)

     Transfer is put into the transfer_out_chan

  c. if (!protocol_block.transfer_response_chan.notify.is_on(vmm_channel::EMPTY)

If the user wants to supply response transfer, the transfer_response_chan must be given an object before time advances in order to enable the VIP to do these steps:

- svt_usb_protocol::transfer_response_chan.get(transfer)
- svt_usb_protocol::randomized_transfer_response_exception_list

  This factory object will be randomized and any exceptions it produces will be added to the transfer.exception_list.
- svt_usb_protocol_callbacks::post_transfer_response_chan_get(svt_usb_protocol xactor, int chan_id, svt_usb_transfer transfer, ref bit drop)

  After the response transfer is obtained and exception_list is added to it, the user is provided with this callback to make changes if desired or drop the transfer completely.
- "svt_usb_protocol_callbacks::transfer_response_chan_cov(svt_usb_protocol xactor, int chan_id, svt_usb_transfer transfer)

  This callback is only made if the transfer was not dropped by the previous post_transfer_response_chan_get() callback.

d. else (vmm_channel::EMPTY)

   In this case the transfer_response_chan did not receive a transfer object in zero time, and the VIP will do the following steps:

   i. transfer.randomize()

      This is the transfer that was put in the transfer_out_chan channel & is being randomized.

   ii. svt_usb_protocol_callbacks::randomized_transfer_complete_response(svt_usb_protocol xactor, svt_usb_transfer transfer)

      This callback is made after the transfer object is randomized which allows the user to modify the transfer object before it is processed by the VIP.

6.2.7.1.2 Creating and Preparing a New Transaction

The following lists how to create and prepare a new transaction:

- VIP uses the *svt_usb_protocol::usb_ss_transaction_factory* factory to create new USB transactions.
- This transaction is subsequently randomized using USB randomization factory *svt_usb_protocol::randomized_usb_ss_transaction*. VIP issues *svt_usb_protocol_callbacks::randomized_transaction(this, transfer, transaction_ix, rand_point);* callback to modify the transaction.
- VIP uses the *svt_usb_protocol::randomized_usb_ss_transaction_exception_list* randomization factory to create exceptions to be injected into USB SS transaction.
- "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.

6.2.7.1.3 Randomizing Transactions for Responses

The transaction object contains an attribute that may need to be randomized by a Host or Device i to help determine how to respond. The Host depends on the host_response attribute, and the Device depends on the device_response attribute. When the transaction object is randomized for this purpose, only the host_response or device_response attribute is randomized and all other transaction object attributes remain unchanged.

- *svt_usb_protocol::randomized_usb_ss_transaction*. This factory object is used to randomize the transaction object. First the transaction object created in the previous step is copied into the randomized_usb_ss_transaction object. Then randomized_usb_ss_transaction.randomize() is called. And finally, the randomized factory object is copied back into the new transaction object.
- *svt_usb_protocol_callbacks::randomized_transaction(svt_usb_protocol xactor, svt_usb_transfer transfer, int transaction_ix, svt_usb_types::protocol_randomization_point_enum rand_point)* This is called after the transaction is randomized but before the transaction's exception_list is randomized.

6.2.7.1.4 Preparing a Packet

The following lists how to prepare a packet.

- VIP uses *svt_usb_protocol::usb_ss_packet_factory* to create new USB SS packets.
- Subsequently the packet is randomized using a randomization factory *svt_usb_protocol::randomized_usb_ss_packet*. VIP issues *svt_usb_protocol_callbacks::randomized_packet(svt_usb_protocol xactor, svt_usb_transfer transfer, int transaction_ix, int packet_ix, svt_usb_types::protocol_randomization_point_enum rand_point)* callback if randomization is successful.
- VIP uses the *svt_usb_protocol::randomized_usb_ss_tx_packet_exception_list* to create exceptions to be injected into outgoing USB SS packets.
- VIP issues *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.

6.2.7.1.5 Sending a Packet

The following list the actions in sending a packet:

- VIP calls *svt_usb_protocol_callbacks::pre_usb_ss_packet_out_chan_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 channel.
- If the packet is not dropped by the above call VIP will perform following actions
  - VIP issues s*vt_usb_protocol_callbacks::usb_ss_packet_out_chan_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 channel.
  - If the transaction is not yet started then transaction.start_time = $realtime, and transaction.notify.indicate(vmm_data::STARTED);
  - If the transfer is not yet started then transfer.start_time = $realtime, and transfer.notify.indicate(vmm_data::STARTED);
  - VIP then puts the packet into ss_packet_out_chan using *svt_usb_protocol::packet_out_chan.put(packet)*. This sends the packet from the protocol layer to the link layer for further processing.
  - *svt_usb_protocol::notify.indicate(protocol.NOTIFY_SS_HOST_TX_PACKET, packet)*. This notification is issued only if VIP is a Host. Packet here refers to actual packet object being sent.
  - *svt_usb_protocol::notify.indicate(protocol.NOTIFY_SS_DEVICE_TX_PACKET, packet)* This notification is issued only if VIP is a Device. Packet here refers to actual packet object being sent.

6.2.7.1.6 Getting a Packet

The flow for getting a packet:

- The VIP pulls a USB packet out of the USB SuperSpeed input channel using *svt_usb_protocol::packet_in_chan.get(packet)*
- The VIP calls *post_usb_ss_packet_in_chan_get(svt_usb_protocol xactor, int chan_id, svt_usb_packet packet, ref bit drop)*, and then acts on the descriptor.
- If the packet was not dropped in the previous call, then the VIP performs following actions:
  - VIP issues *svt_usb_protocol_callbacks::usb_ss_packet_in_chan_cov(svt_usb_packet)*
  - *svt_usb_protocol::notify.indicate(protocol.NOTIFY_SS_HOST_RX_PACKET, packet)*. This notification is issued only if VIP is a Host. Packet here refers to actual packet object being received.
  - *svt_usb_protocol::notify.indicate(protocol.NOTIFY_SS_DEVICE_RX_PACKET, packet)*. This notification is issued only if VIP is a Device. Packet here refers to actual packet object being received.
  - (Applicable for device) If the received packet needs to be deferred, deferral steps are detailed in the section "Defer Received Packet".
  - Received packet is randomized in order to create a random *svt_usb_packet::rx_pkt_pre_processing_delay* value. VIP then issues *randomized_packet_rx_pkt_pre_processing_delay(svt_usb_protocol xactor, svt_usb_packet packet)*; callback with *rand_point = PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY.*
  - If the received packet is unclaimed by the device or any endpoint(i.e because of invalid device_address or invalid endpoint number), the the VIP calls svt_usb_protocol_callbacks::unclaimed_ss_packet(svt_usb_protocol xactor, svt_usb_packet packet, ref bit drop);
  - If drop is not set to 0 in previous call by the user, the the VIP issues an error message stating that an unclaimed USB packet came through the USB SuperSpeed Packet input channel (from the link layer). The VIP then calls *svt_usb_protocol_callbacks::discarded_ss_packet(svt_usb_protocol xactor, svt_usb_packet packet);* and discards the USB packet descriptor that came through the USB SuperSpeed Packet input channel (from the link layer)

6.2.7.1.7 Preparing a Packet for Processing

The following two steps show the flow of preparing a packet.

- The VIP uses *svt_usb_protocol::randomized_usb_ss_rx_packet_exception_list* randomization factory to create exceptions to be added to received USB SuperSpeed packet. The randomized_usb_ss_rx_packet_exception_list factory object is randomized using the received packet and the resulting exceptions are added to the exception_list of the received packet.
- The VIP issues *svt_usb_protocol_callbacks::pre_rx_packet(svt_usb_protocol xactor, svt_usb_transfer transfer, int transaction_ix, int packet_ix)*. This method allows the user to examine or modify the incoming packet or exceptions before the VIP begins processing the received packet.

6.2.7.1.8 Defering a Received Packet

If deferral is required, then the VIP does the following:

- Creates a copy of the Rx packet (using the Rx packet's allocate() function), assigning the copy to ds_pkt.
- Sets ds_pkt.was_deferred = 1.

- Calls svt_usb_protocol_callbacks::post_deferral_determination_ss_pkt (svt_usb_protocol xactor , svt_usb_packet rx_pkt , svt_usb_packet ds_pkt)
- Calls *svt_usb_protocol_callbacks::randomized_packet_rx_pkt_pre_processing_delay(svt_usb_protocol xactor, svt_usb_packet packet);* callback with *rand_point = PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY*. Randomizes the ds_pkt using the PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY randomization point, randomizing both rx_pkt_pre_processing_delay and deferred_pkt_reflection_delay.
- Uses ds_pkt as a factory to create a copy and assigns it to us_pkt.
- Places us_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.
- Places ds_pkt in rx_pkt_ppd_queue.

If deferral is not required, then the VIP does the following:

- Sets ds_pkt equal to Rx packet.
- Calls *svt_usb_protocol_callbacks::post_deferral_determination_ss_pkt (svt_usb_protocol xactor , svt_usb_packet rx_pkt , svt_usb_packet ds_pkt)*
- Calls *svt_usb_protocol_callbacks::randomized_packet_rx_pkt_pre_processing_delay(svt_usb_protocol xactor, svt_usb_packet packet);* callback with *rand_point = PROTOCOL_RAND_SS_RX_PKT_PRE_PROCESSING_DELAY*.
- Places ds_pkt in the rx_pkt_ppd_queue.

6.2.7.2 Protocol SuperSpeed Callback Flow when the VIP is Acting as a Host (Non-isochronous & Isochronous Transfers)

The following sections describe the flow when the model is acting as a Host for both non-isochronous and isochronous tranfers.

6.2.7.2.1 Beginning a Transfer

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

- VIP gets the transfer from transfer input channel *svt_usb_protocol::transfer_in_chan.get(svt_usb_transfer transfer)*

  VIP uses svt_usb_protocol::randomized_transfer_in_exception_list
- Randomization factory to create exceptions to be injected into USB transfer. The *svt_usb_protocol::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.
- After pulling a USB transfer descriptor out of the input channel, VIP calls *svt_usb_protocol_callbacks::post_transfer_in_chan_get(svt_usb_protocol xactor, int chan_id, svt_usb_transfer transfer, ref bit drop);* and then acts on the descriptor.
- VIP calls *svt_usb_protocol_callbacks::transfer_in_chan_cov(svt_usb_protocol xactor, int chan_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 channel. This method is called only if the drop bit from the previous *post_transfer_in_chan_get()* call returned a false value.

6.2.7.2.2 Processing a Transfer

The transfer may be completed as a single transaction, or it may require multiple transactions to complete the transfer. Processing of a transfer is broken into transmit phase and receive phase.

Transmit Phase

A loop is used to process the transfer until all of its transactions are completed (or until the transfer is aborted).

- Create and Prepare New Transaction

  The first step is to create a new transaction object that represents the next transaction needed to implement the transfer. The detailed callback sequence and notifications can be seen in the "Create and Prepare New Transaction" section of the "Common Flows" chapter.

  Prepare Packet

  Packet factory is used to create packet & the transaction created is used to set packet attributes. Packet is then randomized and its exception_list is populated.

  Refer to the "Common Flows" chapter of this document to see the callback flow for "Prepare Packet".

- Send Packet

  Packet created needs to be sent to link layer to be transmitted on the bus.

  Refer to the "Common Flows" chapter of this document to see the callback flow for "Send Packet".

- VIP issues *svt_usb_protocol_callbacks::transaction_ended(svt_usb_protocol xactor, svt_usb_transfer transfer, int transaction_ix)*; callback when a USB transaction is complete(when the vmm_data::ENDED notification is sent for that USB transaction). This callback can be extended to collect functional coverage data, or to check the transaction against a scoreboard or other such operations.
- VIP issues *svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol xactor, svt_usb_transfer transfer)* when a USB transfer is complete (when the *vmm_data::ENDED* notification is sent for that USB transfer).

Receive Phase

A loop is used to process the transfer until all of its transactions are completed (or until the transfer is aborted). In the receive phase, the VIP protocol transactor layer performs the following actions:

1. If the transaction requires the Host to receive a DATA or HANDSHAKE packet from the device, then that action will use the "Get Received Packet" callback flow described in the "Common Flows" chapter of this document.
2. If the received packet is for IN endpoint processor, VIP issues *received_data_packet(svt_usb_protocol xactor, 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.
3. VIP issues *svt_usb_protocol_callbacks::transaction_ended(svt_usb_protocol xactor, svt_usb_transfer transfer, int transaction_ix)*; callback when a USB transaction is completed (when the vmm_data::ENDED notification is sent for that USB transaction). This callback can be extended to collect functional coverage data, or to check the transaction against a scoreboard, or other such operations.

4. VIP issues *svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol xactor, svt_usb_transfer transfer*) when a USB transfer is complete (when the vmm_data::ENDED notification is sent for that USB transfer).

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

High Level Sequence of Operations

1. VIP receives the packet from packet_in_chan.
2. VIP retrieves packets from the link layer through the appropriate packet input channel (see Receive Phase).
3. VIP locates device object for packet's device_address and determines if deferral is required based on the value of USB_SEND_DEFERRED_PACKETS protocol service.
4. If deferral is required, VIP does the following:
   a. Creates a copy of the Rx packet (using the Rx packet's allocate() function), assigning the copy to downstream_pkt.
   b. Sets downstream_pkt.was_deferred = 1.
   c. 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 channel.

Detailed Sequences

Receive Phase

1. VIP pulls a USB packet descriptor out of the SS packet input channel (from the link layer) and calls *post_usb_ss_packet_in_chan_get(svt_usb_protocol xactor, int chan_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 xactor, svt_usb_transfer transfer, int transaction_ix)* after the USB Endpoint Manager transaction is complete (when the vmm_data::ENDED notification is sent for the USB Endpoint Manager transaction). 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 xactor, 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 xactor, 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 xactor, 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 xactor, svt_usb_transfer transfer, int transaction_ix)* callback immediately after a USB Endpoint Manager transaction is complete (when the vmm_data::ENDED notification is sent for that Endpoint Manager transaction). 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 xactor, 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 xactor, svt_usb_transfer transfer, int transaction_ix, int packet_ix)* callback when a protocol processor thread has completed processing a USB packet received and associated with a USB transfer. VIP can optionally extend this callback to collect functional coverage data, or modify the data in the packet so that it is reflected in the transfer.

Send Packet of IN Transfer

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

1. VIP prepares the transaction.
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 xactor, 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 xactor, 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 xactor, 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 xactor, 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 xactor, svt_usb_transfer transfer, int transaction_ix)* callback when a USB Endpoint Manager transaction is complete (when the vmm_data::ENDED notification is sent for that USB Endpoint Manager transaction). 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 xactor, 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.

6.2.7.4 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.

6.2.7.4.1 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.

6.2.7.4.2 Stream Endpoint Manager Flow

Figures 6-5 and 6-6 illustrate the stream endpoint manager flow for host and device respectively.

Figure 6-5 Host Stream Endpoint Manager Flow

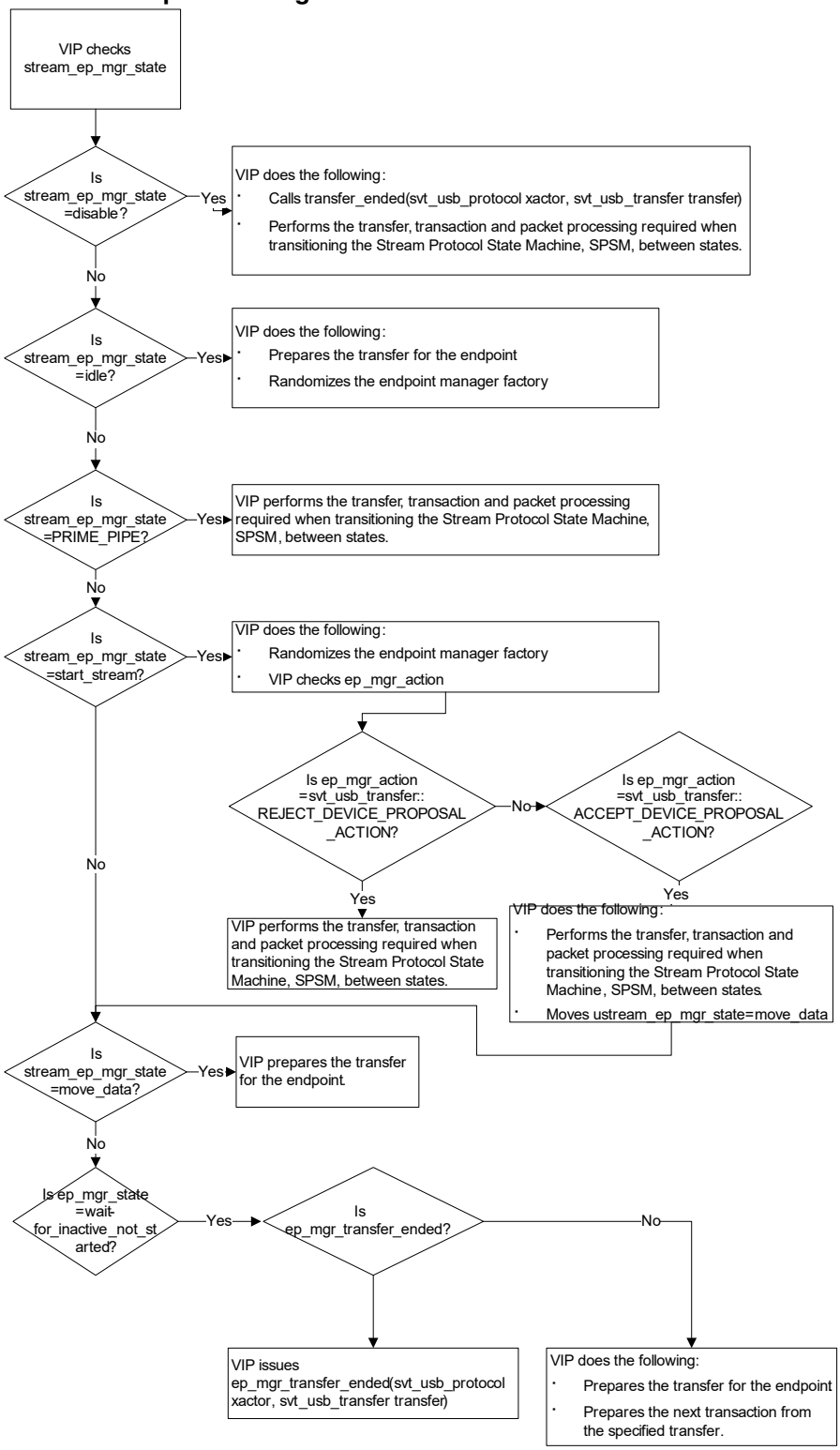
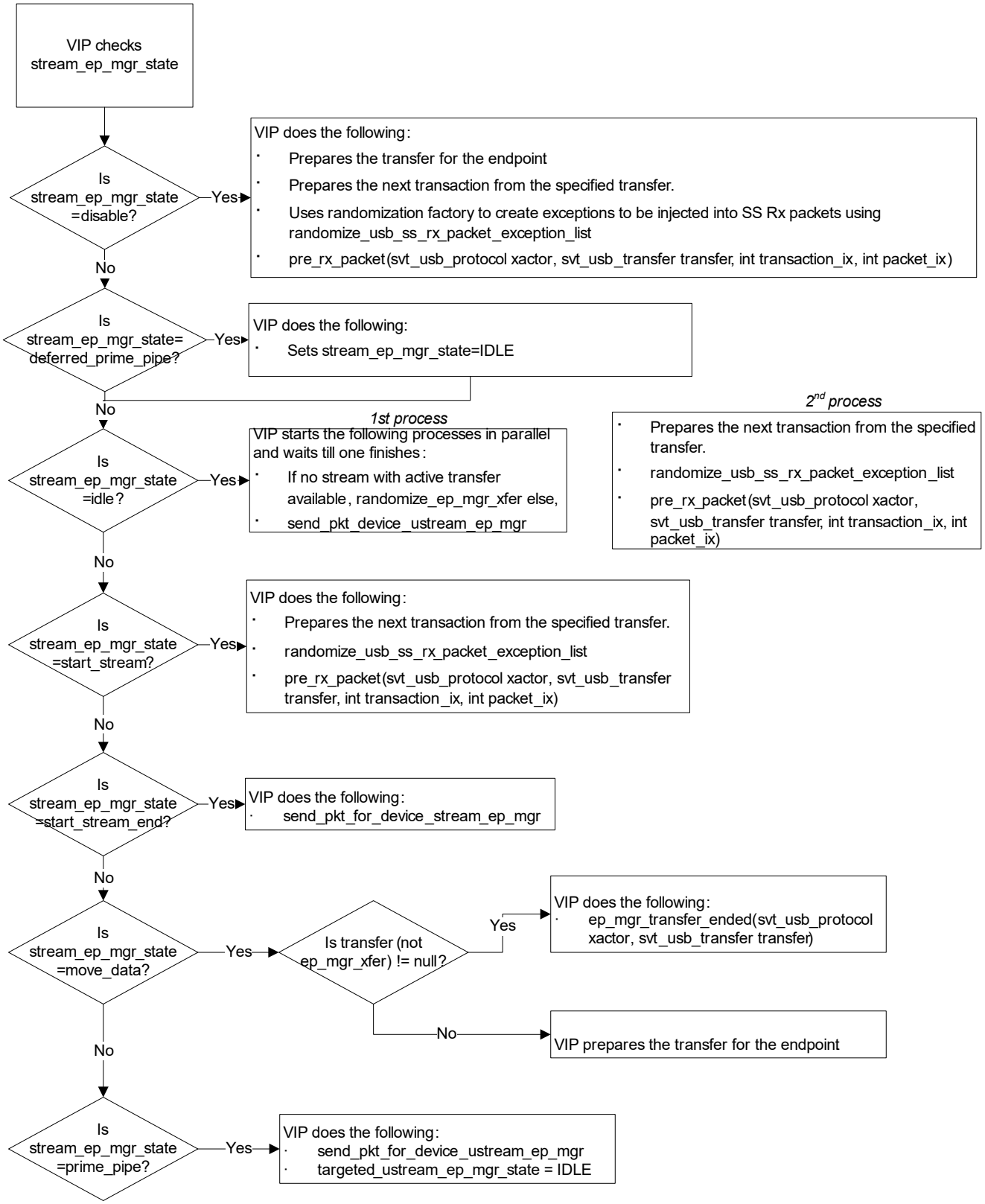


Figure 6-6 Device Stream Endpoint Manager Flow



6.2.8 Protocol Transactor 2.0 Link Callback, Factory, and Notification Flows

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

6.2.8.1 Protocol 2.0 Link, Callback, and Notification Flow when the VIP is Acting as a Host

6.2.8.1.1 Non-isochronous Transfers

Transfer Start Phase

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

Sequence of operations:

1. VIP gets transfer from transfer_in input channel.
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 channel, VIP calls *svt_usb_protocol_callbacks::post_transfer_in_chan_get(svt_usb_protocol xactor, int chan_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_chan_get()* returns a false value. VIP calls *svt_usb_protocol_callbacks::transfer_in_chan_cov(svt_usb_protocol xactor, int chan_id, svt_usb_transfer transfer);* to allow the testbench to collect functional coverage information from a USB transfer received in the transfer_in input channel.

Transfer Processing Phase

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

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

Transfer Loop for Host non-isochronous

Sequence of operations:

1. Create and prepare new transactions.

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

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

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

4. If the transfer loop does not have to be repeated, VIP issues *transfer.notify.indicate(vmm_data::ENDED),* and then issues *svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol xactor, 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. 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 xactor, 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. transaction.notify.indicate(vmm_data::ENDED)* *3. svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*

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

1. 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 xactor, 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.
4. If the host response is ACK, with no injection errors, then VIP does the following:
   - *transaction.status = svt_transaction::ACCEPT*
   - *transaction.notify.indicate(vmm_data::ENDED)*
   - *svt_usb_protocol::notify.indicate(protocol_block.protocol.NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*

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

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

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

Host Response to Device Handshake for OUT transactions

If the device HANDSHAKE is an ACK, does not have any packet exception errors, and if the transaction is either a Complete SPLIT or not a SPLIT, complete the following steps:

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

If the device HANDSHAKE is a NYET, does not have any packet exception errors, and if the transaction is not a SPLIT, complete the following steps:

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

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

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

4. To repeat the transaction loop, go back to the flow at the top of the transaction loop. Otherwise, the transaction has already been marked ENDED and the callback to transaction_ended is made using *svt_usb_protocol_callbacks::transaction_ended (svt_usb_protocol xactor, svt_usb_transfer transfer, int transaction_ix).*

6.2.8.1.2 Isochronous Transfers

Transfer Start Phase

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

Sequence of operations:

1. VIP gets transfer from transfer_in input channel.
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 channel, VIP calls *svt_usb_protocol_callbacks::post_transfer_in_chan_get(svt_usb_protocol xactor, int chan_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_chan_get()* returns a false value. VIP calls *svt_usb_protocol_callbacks::transfer_in_chan_cov(svt_usb_protocol xactor, int chan_id, svt_usb_transfer transfer);* to allow the testbench to collect functional coverage information from a USB transfer received in the transfer_in input channel.

Transfer Processing Phase

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

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

Transfer Loop for Host Isochronous

Sequence of operations:

1. Create and prepare new transactions.

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

2. Process the transaction.

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

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

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

4. If the transfer loop does not have to be repeated, VIP issues *transfer.notify.indicate(vmm_data::ENDED),* and then issues *svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol xactor, 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. 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 xactor, 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 transaction.notify.indicate(vmm_data::ENDED) • VIP calls svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())
If the transaction exceeds the maximum allowed number of errors, VIP aborts the transfer.

Host response to isochronous OUT transactions
If the transaction is not a SPLIT or is the last Start SPLIT, then VIP does the following: 1. Call *transaction.status = svt_transaction::ACCEPT*. 2. Trigger the notification that the transaction has ENDED using *transaction.notify.indicate(vmm_data::ENDED).* 3. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*.
If the device handshake is a NYET and the transaction is not a SPLIT, then VIP does the following: 1. Subtract the transaction.payload.byte_count value from the transfer.payload_bytes_remaining attribute. 2. Mark the transaction status as ACCEPT using *transaction.status = svt_transaction::ACCEPT.* 3. Trigger the transaction notify that the transaction has ENDED using *transaction.notify.indicate(vmm_data::ENDED).* 4. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy()).*

4. To repeat the transaction loop, go back to the flow at the top of the transaction loop. Otherwise, the transaction has already been marked ENDED and the callback to transaction_ended is now made using *svt_usb_protocol_callbacks::transaction_ended (svt_usb_protocol xactor, svt_usb_transfer transfer, int transaction_ix).*

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

6.2.8.2.1 Non-isochronous Transfers

Transfer Start Phase

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

Sequence of operations:

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

Transfer Processing Phase

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

The following loops are discussed here:

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

Transfer Loop for Device Non-isochronous

Sequence of operations:

1. Create and prepare new transactions.

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

2. Process the transaction.
3. The transaction is processed according to the transaction loop described in Transaction Loop for Device Non-Isochronous.
4. (Optional) Repeat the transfer loop if more transactions are required.

   This step depends on the specific transfer. If there is more payload to send and the transfer is not aborted, then continue with the transfer loop.
5. If the transfer loop does not have to be repeated, VIP issues *transfer.notify.indicate(vmm_data::ENDED),* and then issues *svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol xactor, 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 *transfer.notify.indicate(vmm_data::STARTED).*
2. If the transaction requires the Device to receive a TOKEN, DATA or HANDSHAKE packet from the Host, use the Get Received Packet callback flow.

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

Device response if the transaction is not a SPLIT OUT or a SETUP transaction
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 xactor, svt_usb_transfer transfer, svt_usb_packet packet)* - transaction.status = svt_transaction::ACCEPT - transaction.notify.indicate(vmm_data::ENDED) - svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy()) 3. If the transaction is not a SETUP transaction, VIP calls *transfer.payload.append_payload(transaction.payload))*
If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.
Device response if the transaction is a START SPLIT transaction
If the transaction is not an Interrupt, complete the following steps: 1. 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 xactor, svt_usb_transfer transfer, svt_usb_packet packet)
   - transfer.payload.append_payload(transaction.payload))
   - If the transaction is a SETUP transaction, VIP calls the following:
   - *svt_usb_protocol_callbacks::received_data_packet(svt_usb_protocol xactor, svt_usb_transfer transfer, svt_usb_packet packet)*
2. VIP then calls the following:
   - *transaction.status = svt_transaction::ACCEPT*
   - *transaction.notify.indicate(vmm_data::ENDED)*
   - *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*
3. If the transaction is not a SETUP transaction, VIP calls *transfer.payload.append_payload(transaction.payload)).*

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

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

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. *transaction.notify.indicate(vmm_data::ENDED)*
3. *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*

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

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

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. *transaction.notify.indicate(vmm_data::ENDED)*
3. *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*

If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.
Device response if the transaction is a Complete SPLIT IN (BULK or CONTROL) transaction
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. *transaction.notify.indicate(vmm_data::ENDED)* 3. *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy())*
If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.

4. To repeat the transaction loop, go back to the flow at the top of the transaction loop. Otherwise, the transaction has already been marked ENDED and the callback to transaction_ended is now made using *svt_usb_protocol_callbacks::transaction_ended (svt_usb_protocol xactor, svt_usb_transfer transfer, int transaction_ix).*

6.2.8.2.2 Isochronous Transfers

Transfer Start Phase

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

Sequence of operations:

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

Transfer Processing Phase

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

The following loops are discussed here:

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

Transfer Loop for Device Isochronous

Sequence of operations:

1. Create and prepare new transactions.

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

3. The transaction is processed according to the transaction loop described in Transaction Loop for Device Non-Isochronous.
4. (Optional) Repeat the transfer loop if more transactions are required.

   This step depends on the specific transfer. If there is more payload to send and the transfer is not aborted, then continue with the transfer loop.
5. If the transfer loop does not have to be repeated, VIP issues *transfer.notify.indicate(vmm_data::ENDED),* and then issues *svt_usb_protocol_callbacks::transfer_ended(svt_usb_protocol xactor, 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 *transfer.notify.indicate(vmm_data::STARTED).*
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 xactor, 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 *transaction.notify.indicate(vmm_data::ENDED).* 3. Provide a copy of the ENDED transaction with the top protocol notification that the transaction has ENDED using *svt_usb_protocol::notify.indicate(svt_usb_protocol::NOTIFY_USB_TRANSACTION_ENDED, transaction.copy()).*
If the transaction has exceeded the maximum number of allowed errors, VIP aborts the transfer.

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

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 xactor, svt_usb_transfer transfer, int transaction_ix).*

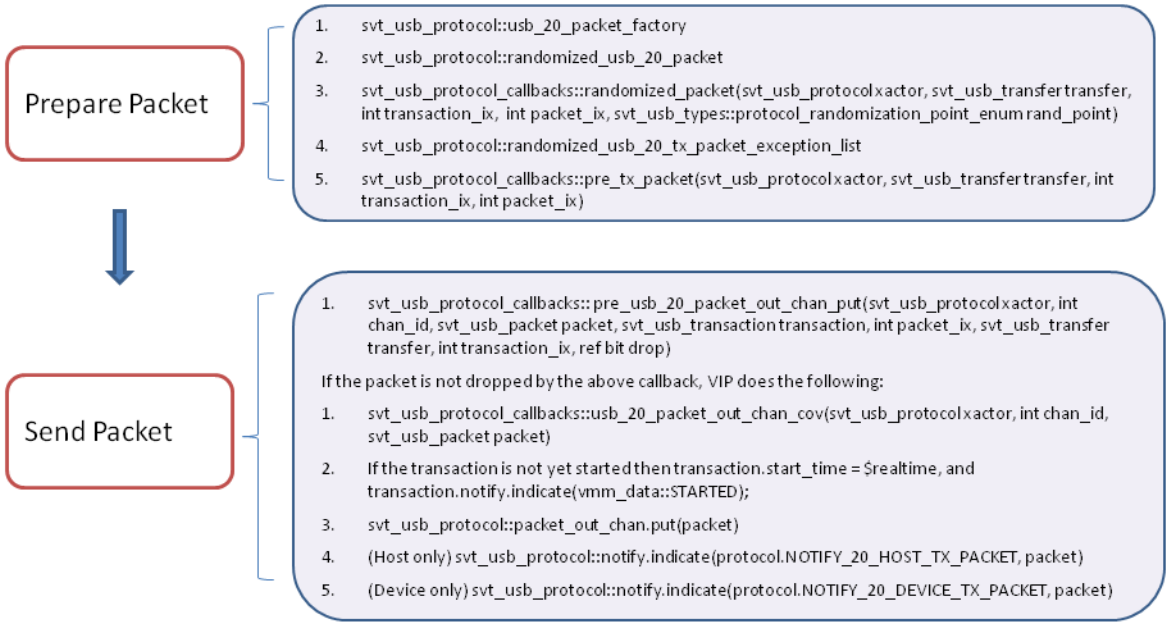
6.2.8.3 Common Protocol 2.0 Callback and Notification Flows

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

Prepare and Send Packet

Figure 6-7 illustrates the callback and notification flow.

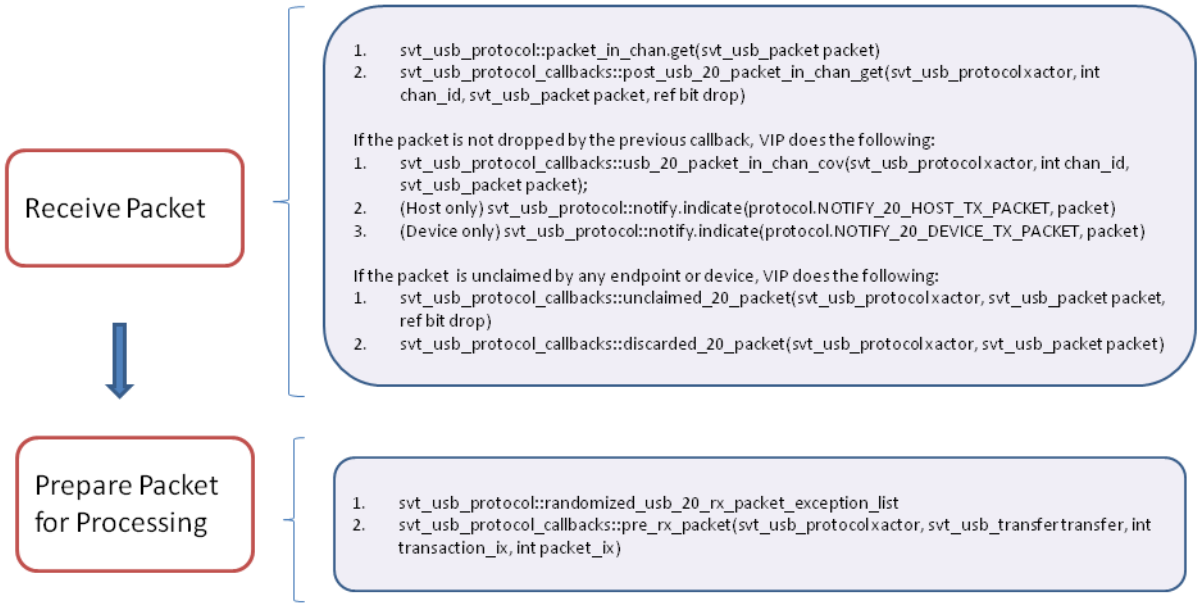
Figure 6-7 Sequence of Callback and Notifications when Preparing and Sending a Packet



Get Received Packet

Figure 6-8 illustrates the callback and notification flow.

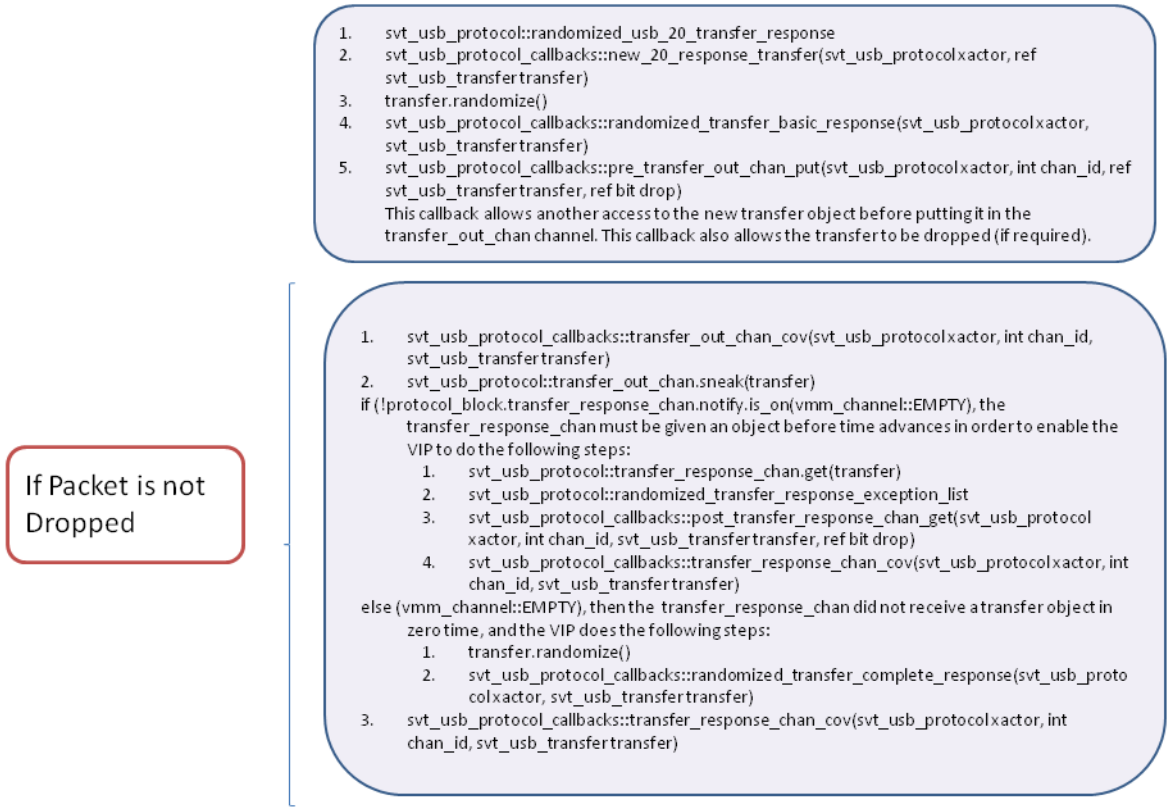
Figure 6-8 Sequence of Callback and Notifications when Receiving and Preparing a Packet for Processing



Create and Prepare New Device Transfer

Figure 6-9 illustrates the callback and notification flow.

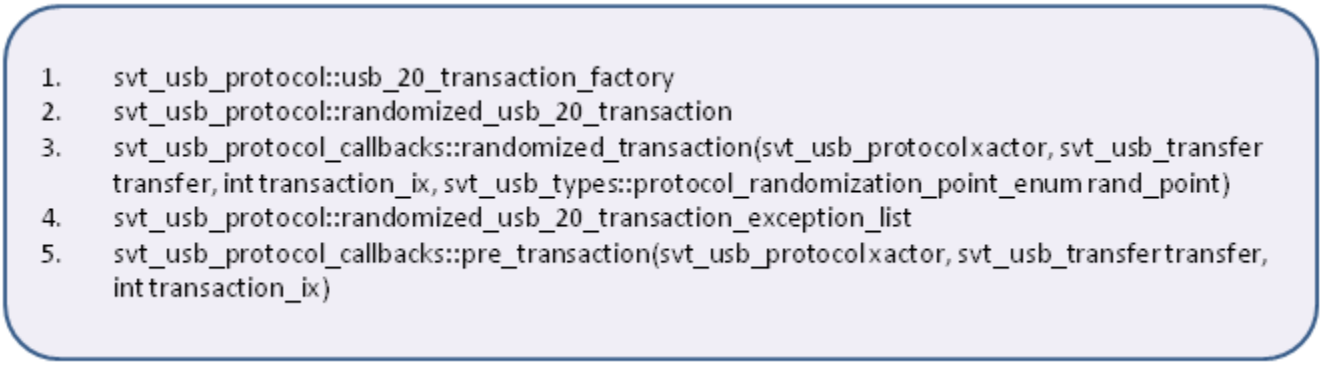
Figure 6-9 Callback and Notification Flow when Creating and Preparing New Device Transfer



Create and Prepare New Transaction

Figure 6-10 illustrates the callback and notification flow.

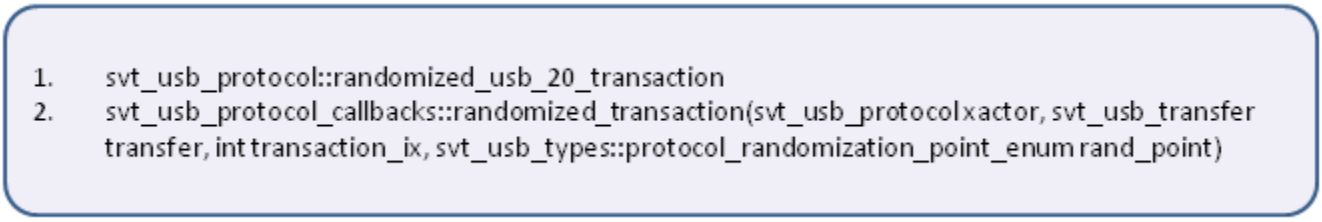
Figure 6-10 Sequence of Callback and Notifications when Creating and Preparing a new Transaction



Randomize Transaction for Response

The transaction object contains an attribute that may need to be randomized by a Host or Device in order to help determine how to respond. The Host depends on the *host_response* attribute and the Device depends on the *device_response* attribute. When the transaction object is randomized for this purpose, only the *host_response* or *device_response* attribute is randomized and all other transaction object attributes remain unchanged. Figure 6-11 illustrates the callback flow.

Figure 6-11 Sequence of Callbacks when Randomizing Transactions for Response



6.2.9 Related Topics About Protocol Transactor

FOR INFO ABOUT	SEE
Protocol service commands	Protocol Service commands in the HTML class reference.
Factories, callbacks, and channels	Complete list of VMM factories, callbacks, and channels used by the Protocol Transactor in the HTML class reference.

6.3 Link Transactor

USB Link transactors are component objects in a VMM-compliant verification environment. The USB Link transactor object extends from the *svt_xactor* class, which extends from the *vmm_xactor* base class. This object implements all methods specified by VMM for the *vmm_xactor* class.

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

The Class Reference HTML describes Link transactor functions and attributes.

6.3.1 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 channels – SS and 2.0
- Data input and output channels – SS and 2.0
- Configurable input channel stimulus
  - Auto connect to protocol layer
  - Direct channel
  - VMM atomic generator
  - VMM scenario generator
  - VMM multi-stream scenario generator
- Error injection
  - USB Packet
  - USB Link Command
  - USB Symbol
- Callbacks providing testbench visibility and control

6.3.2 SuperSpeed Packet Scenarios

The class svt_usb_packets represents a USB packet VMM/VIP transaction. The packet includes the packet fields, but also contains a list of the physical data objects that make up the packet.

The #speed attribute is the entry point to this object. If #speed attribute is assigned to svt_usb_types::SS then USB Super-speed attributes apply; otherwise, USB 2.0 attributes apply.

The following table describes various super-speed link scenarios, and the resulting settings for the noted status-type attributes for each of those scenarios.

Table 6-2 SuperSpeed Packet Link Scenarios

Packet Atrribute	Direction N: set by	Packet with payload good header -LGOOD	Packet without payload good header -LGOOD	Packet with payload bad header -LBAD	Packet without payload bad header -LBAD	Packet with payload good header payload aborted (EDB) -LGOOD	Replayed packet good header -LGOOD	Packet with payload dropped via callback	HDP only dropped via callback
status	TX (link_ss_tx)	ACCEPT	ACCEPT	PARTIAL_ACCEPT (not ENDED)	PARTIAL_ACCEPT (not ENDED)	ACCEPT	ACCEPT	ABORTED	ABORTED
	RX (link_ss_rx)	ACCEPT	ACCEPT	ACCEPT (not ENDED)	ACCEPT (not ENDED)	ACCEPT	ACCEPT	ABORTED	ABORTED
header_status	TX (link_ss_tx)	ACCEPT	ACCEPT	PARTIAL_ACCEPT	PARTIAL_ACCEPT	ACCEPT	ACCEPT	ABORTED	ABORTED
	RX (object_detect)	ACCEPT	ACCEPT	ACCEPT	ACCEPT	ACCEPT	ACCEPT	ACCEPT	ACCEPT
payload_ststatus	TX (link_ss_tx)	ACCEPT	DISABLED	PARTIAL_ACCEPT	DISABLED	ACCEPT	CANCELLED	INITIAL	DISABLED
	RX (object_detect)	ACCEPT	DISABLED	ACCEPT	DISABLED	ACCEPT	DISABLED	ACCEPT	DISABLED
payload_presence	TX (protocol / generator)	PAYLOAD_PRESENT	PAYLOAD_NOT_PRESENT	PAYLOAD_PRESENT	PAYLOAD_NOT_PRESENT	PAYLOAD_PRESENT_BUT_ABORTED (set by Link Transmitter)	PAYLOAD_NOT_PRESENT (set by Link Transmitter)	PAYLOAD_PRESENT	PAYLOAD_NOT_PRESENT
	RX (object_detect)	PAYLOAD_PRESENT	PAYLOAD_NOT_PRESENT	PAYLOAD_PRESEN	PAYLOAD_NOT_PRESENT	PAYLOAD_PRESENT_BUT_ABORTED	PAYLOAD_NOT_PRESENT	PAYLOAD_PRESENT	PAYLOAD_NOT_PRESENT

6.3.3 Link Transactor Channels

The following list describes objects that move information through the Link transactor. Figure 6-12 displays Link protocol information flow and the following objects:

- **Packet channels:** These objects convey USB packet objects between the Protocol and Link transactors. The transactor supports the following Packet channels:
  - ***Packet Input channels:*** These objects receive USB packets from Protocol layer transactors for outbound transactions. Separate channels are provided to support USB SS and USB 2.0 traffic.

    Minimum SS input channel depth is four packets. Objects are removed from SS channels only after receiving an LGOOD from the transaction destination.

    Link transactor attribute names: *usb_ss_packet_in_chan, usb_20_packet_in_chan*.
  - ***Packet Output channels:*** These objects transmit USB packets to Protocol layer transactors for inbound transactions. Separate channels are provided to support USB SS and USB 2.0 traffic.

    The Protocol transactor removes objects from the channel arbitrarily. The channel does not provide a blocking function, allowing Link transactors to continue processing data from the Physical layer even if the Protocol transaction does not remove packets from the channel.

    Link transactor attribute names: *usb_ss_packet_out_chan, usb_20_packet_out_chan*.
- **Data channels:** These objects convey USB data objects between the Physical and Link transactors. The transactor supports the following Data channels:
  - ***Data Input channels:*** These objects receive USB data from Physical layer transactors for inbound transactions. Separate channels are provided to support USB SS and USB 2.0 traffic.

    The required channel depth is one object, as Link transactors accept one object per clock period and Physical transactors provide one object per clock period.

    Link transactor attribute names: *usb_ss_data_in_chan, usb_20_data_in_chan*.
  - ***Data Output channels:*** These objects transmit USB data to Physical layer transactors for outbound transactions. Separate channels are provided to support USB SS and USB 2.0 traffic.

    The required channel depth is one object, as Link transactors provide one object per clock period and Physical transactors accept one object per clock period.

    Link transactor attribute names: *usb_ss_data_out_chan, usb_20_data_out_chan*.
- **Link Service channel:** This object receives Link Service objects from the Protocol transactor.

  Link transactor attribute name: *link_service_in_chan*.
- **Physical Service Output channels:** These objects convey USB Physical Service objects from the Link transactor to the Physical transactor. Separate channels are provided to support USB SS and USB 2.0 traffic. The transactor supports the following channels:

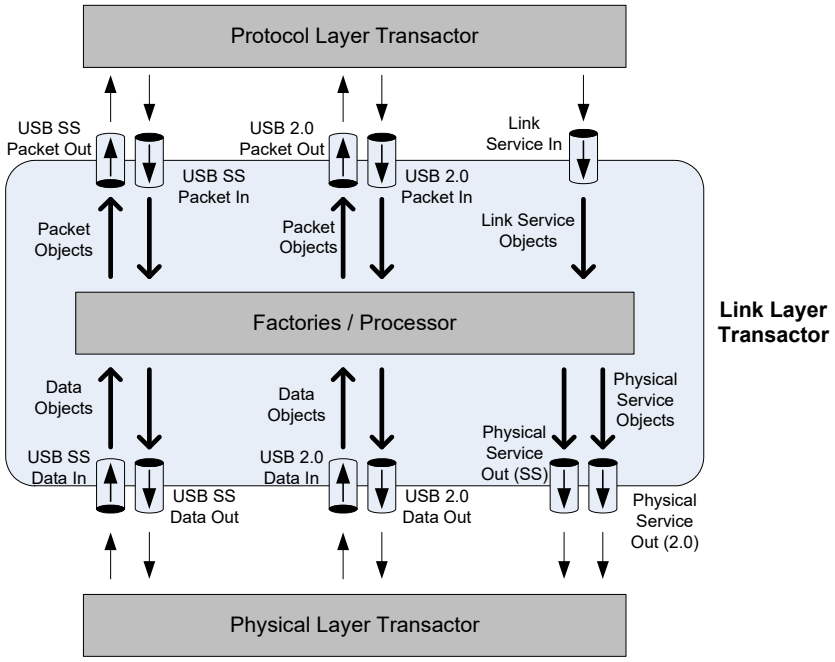
  Link transactor attribute names: *usb_ss_physical_service_out_chan*, *usb_20_physical_service_out_chan*.

6.3.4 Data Objects

The following is a list of objects that represent information the Link Transactor receives, sends, or processes. Figure 6-12 displays the flow of information objects within the Link transactor.

Transaction Objects describes USB data objects

Figure 6-12 Link Transactor 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.

6.3.5 Data Transformation Objects

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

6.3.5.1 Factory Objects

The Link transactor supports the following factories:

- **Rx object factories:** These factories allocate objects for injecting into an Rx data stream.

  Link transactor 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 transactor 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.*

6.3.5.2 Exception List Factories

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

The Link transactor supports the following exception list factories:

- **Rx object factories:** These factories generate exceptions for injecting into an Rx data stream.

  Link transactor 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 transactor 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.*

6.3.5.3 Callbacks

USB Link Transactor Callback objects extend from the *svt_xactor_callbacks* class, which extend from the *vmm_xactor_callbacks* base class. USB Link Transactor Callback objects implement all methods specified by VMM for the vmm_xactor_callbacks class.

To create unique implementation of Protocol transactor callbacks, extend the *svt_usb_link_callbacks* class. To register an instance of the callback object with the USB Link transactor, use the *append_callback* class. This registration is typically performed at the end of the build() implementation if the testbench is derived from the *vmm_env* class.

The Link transactor supports the following callbacks

- **Channel input:** These callbacks indicate that the transactor collected data from an input channel:

  Link transactor callback method names: *post_usb_20_data_in_chan_get, post_usb_ss_data_in_chan_get, usb_20_data_in_chan_cov, usb_ss_data_in_chan_cov, post_link_service_in_chan_get, usb_link_service_in_chan_cov, post_usb_20_packet_in_chan_get, post_usb_ss_packet_in_chan_get, pre_usb_ss_rx_symbol_set_detected, usb_20_packet_in_chan_cov, usb_20_packet_in_ended, usb_ss_packet_in_chan_cov*

- **Channel output:** These callbacks indicate that the transactor placed data on an output channel:

  Link transactor callback method names: *pre_usb_20_data_out_chan_put, pre_usb_ss_data_out_chan_put, pre_usb_20_packet_out_chan_put, pre_usb_ss_packet_out_chan_put, pre_usb_20_physical_service_out_chan_put, pre_usb_ss_physical_service_out_chan_put, usb_20_data_out_chan_cov, usb_20_packet_out_chan_cov, usb_20_physical_service_out_chan_cov, usb_ss_data_out_chan_cov, usb_ss_packet_out_chan_cov, usb_ss_physical_service_out_chan_cov*

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

  Link transactor 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 Transactor:

  Link transactor 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 transactor callback method names: *object_observed, service_in_ended*

6.3.5.4 Notifications

USB Link transactors support general VMM notifications and USB VIP specific notifications. For notifications where the current transaction is pertinent, the notification data includes a handle to the current transaction.

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

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

The Link transactor supports the following notifications:

- **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`

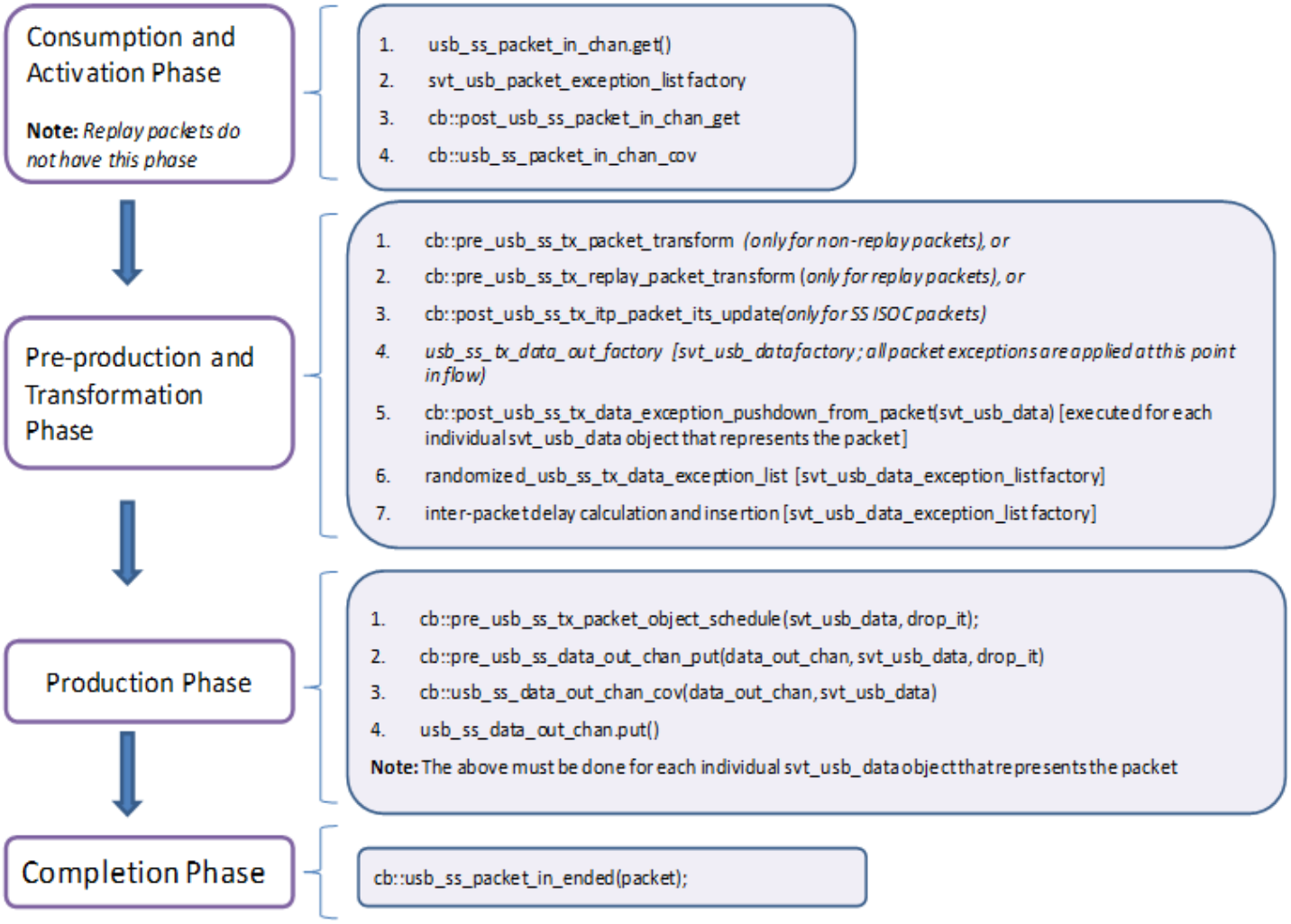
6.3.6 Link Transactor SuperSpeed Link Callback, Factory, and Notification Flows

This section explains the various link-layer flows through the use of flowcharts. Figures 6-13 to 6-23 illustrate the various SS link transactor flows.

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

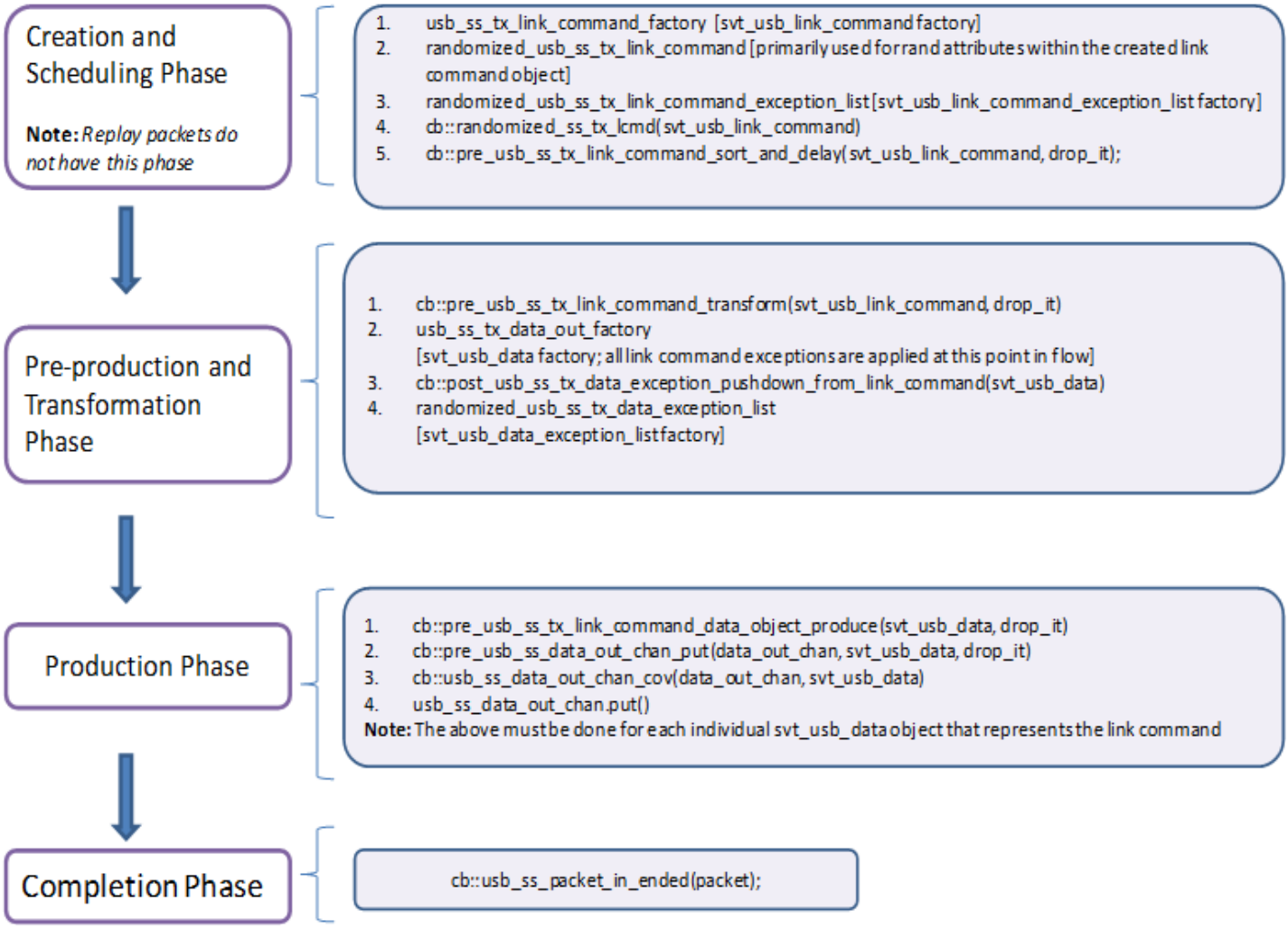
SuperSpeed Packet: Transmit Flow

Figure 6-13 Super Speed Transmit Packet Flow



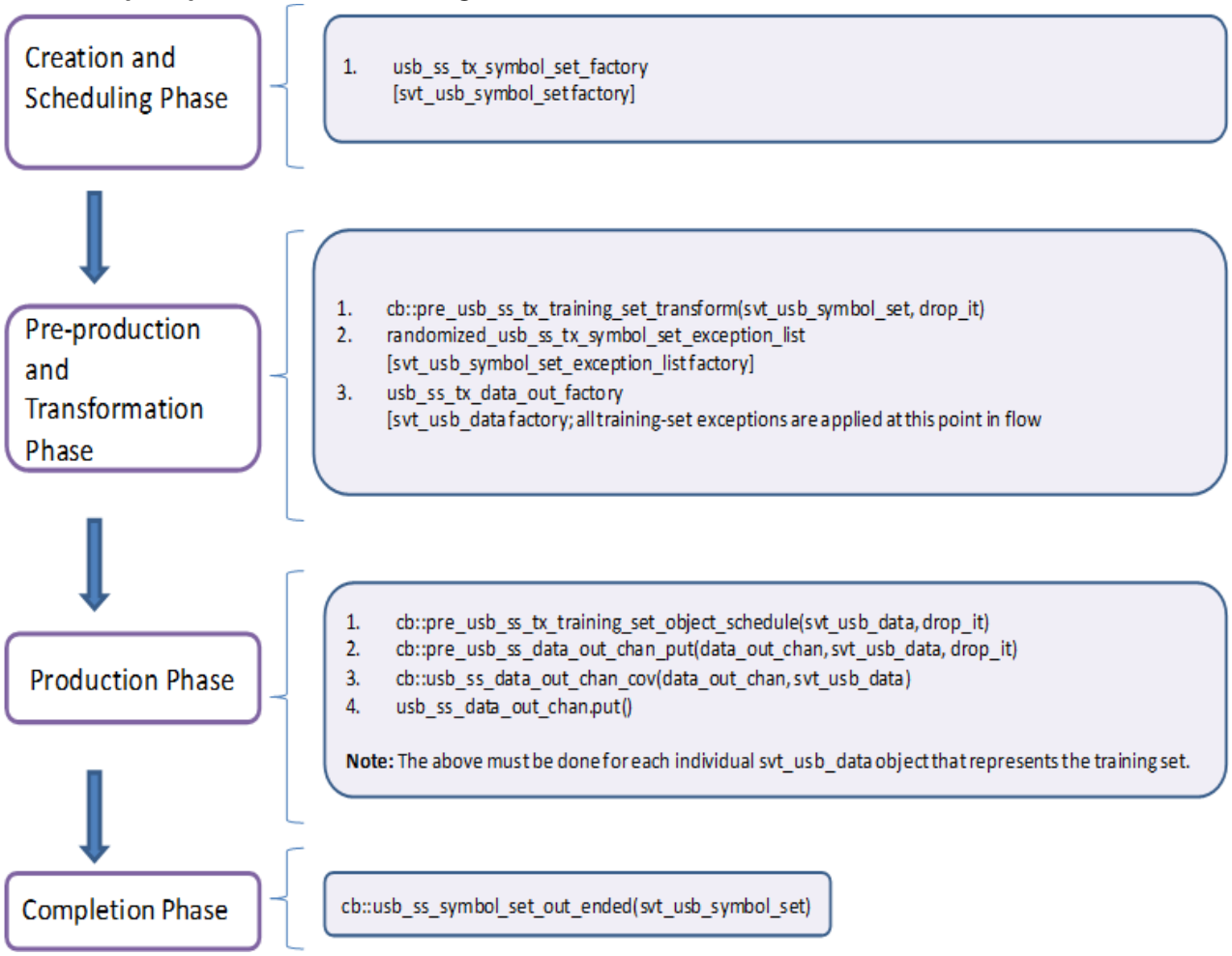
SuperSpeed Packet: Transmit Link Command Flow

Figure 6-14 SuperSpeed Transmit Link Command Flow



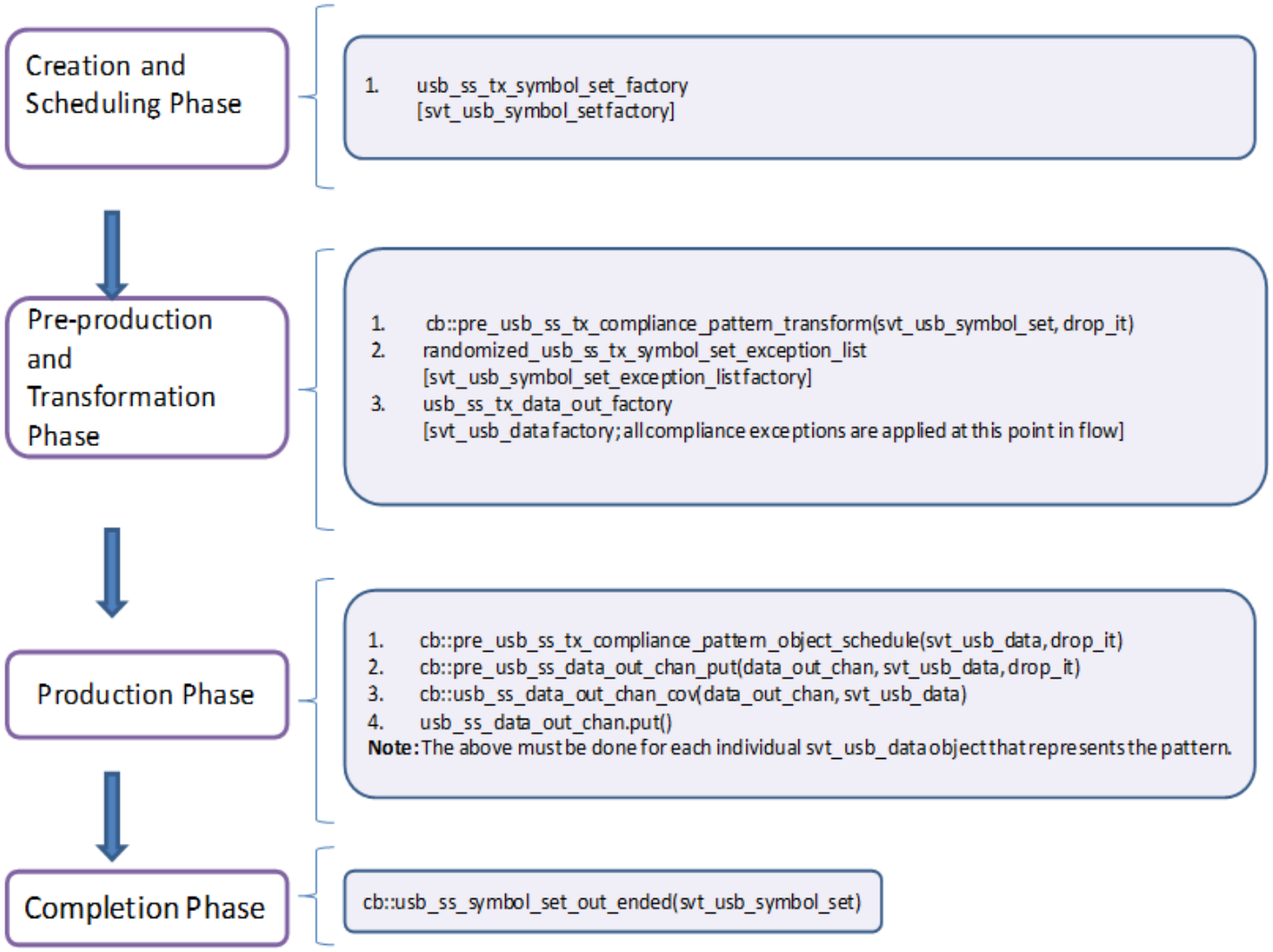
SuperSpeed Packet: Transmit Training Set Flow

Figure 6-15 Super-speed Transmit Training-Set Flow



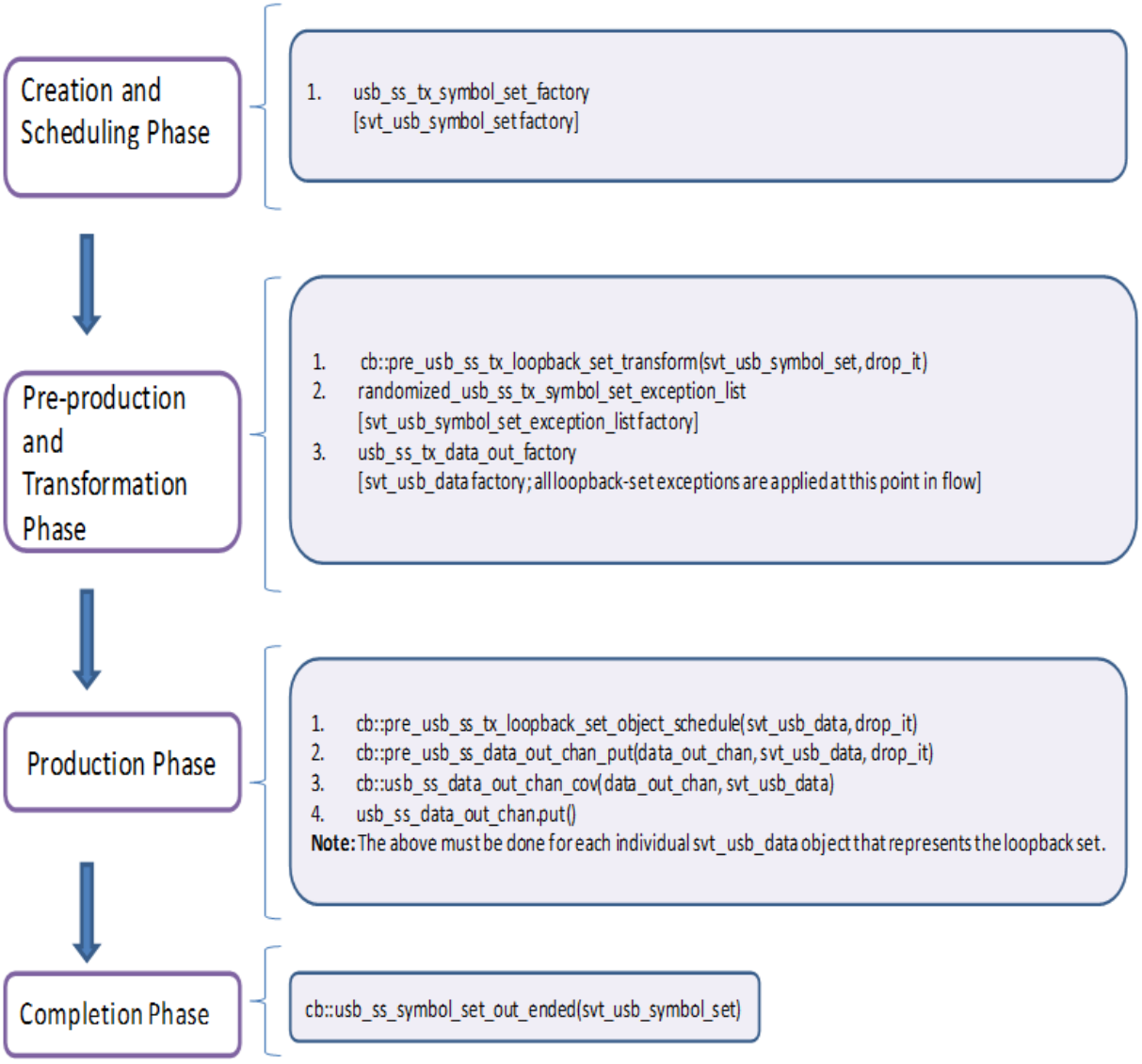
SuperSpeed Packet: Transmit Compliance Pattern Flow

Figure 6-16 SuperSpeed Transmit and Compliance Pattern Flow



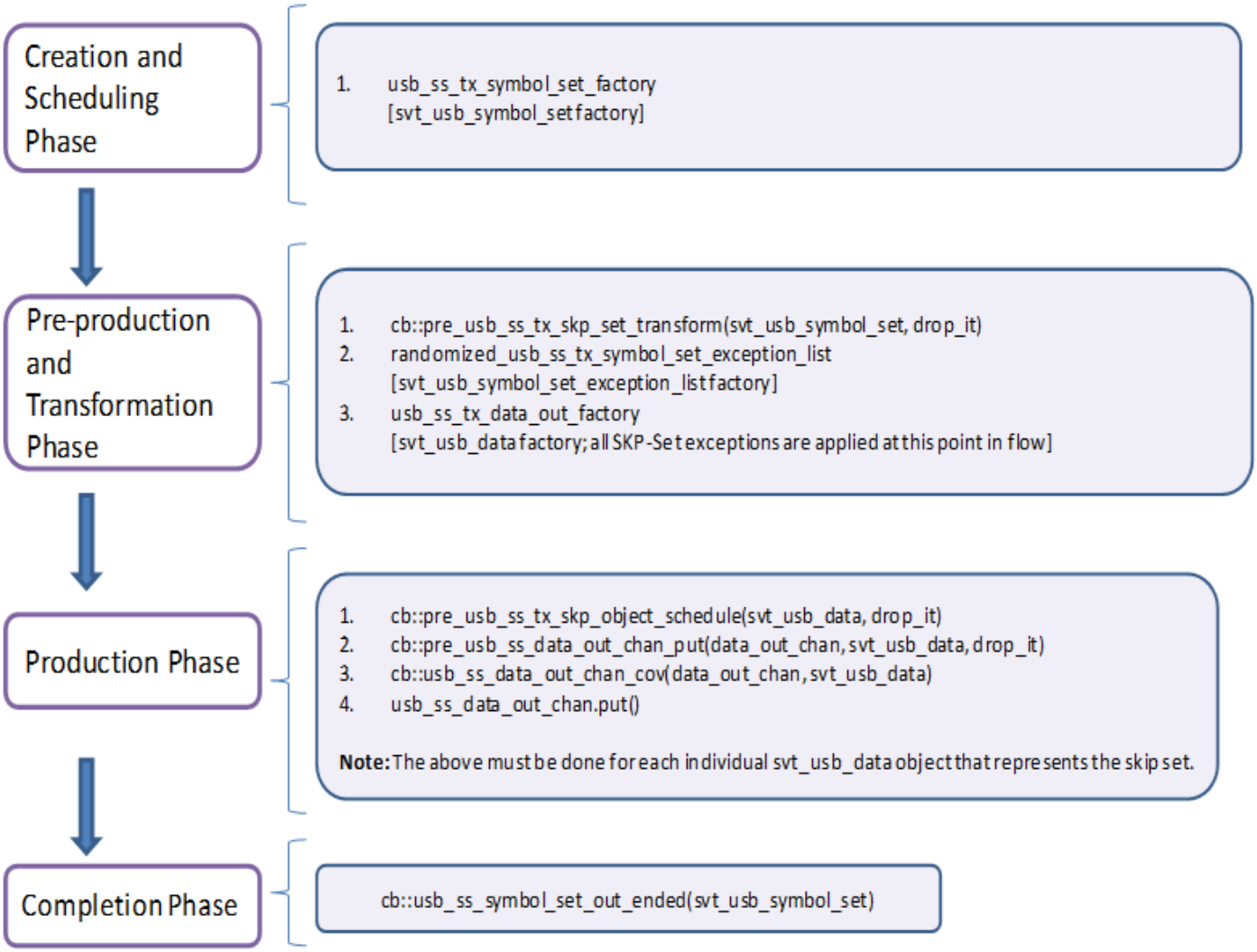
SuperSpeed Packet: Transmit Loopback Set Flow

Figure 6-17 Super-speed Transmit Loopback Set Flow



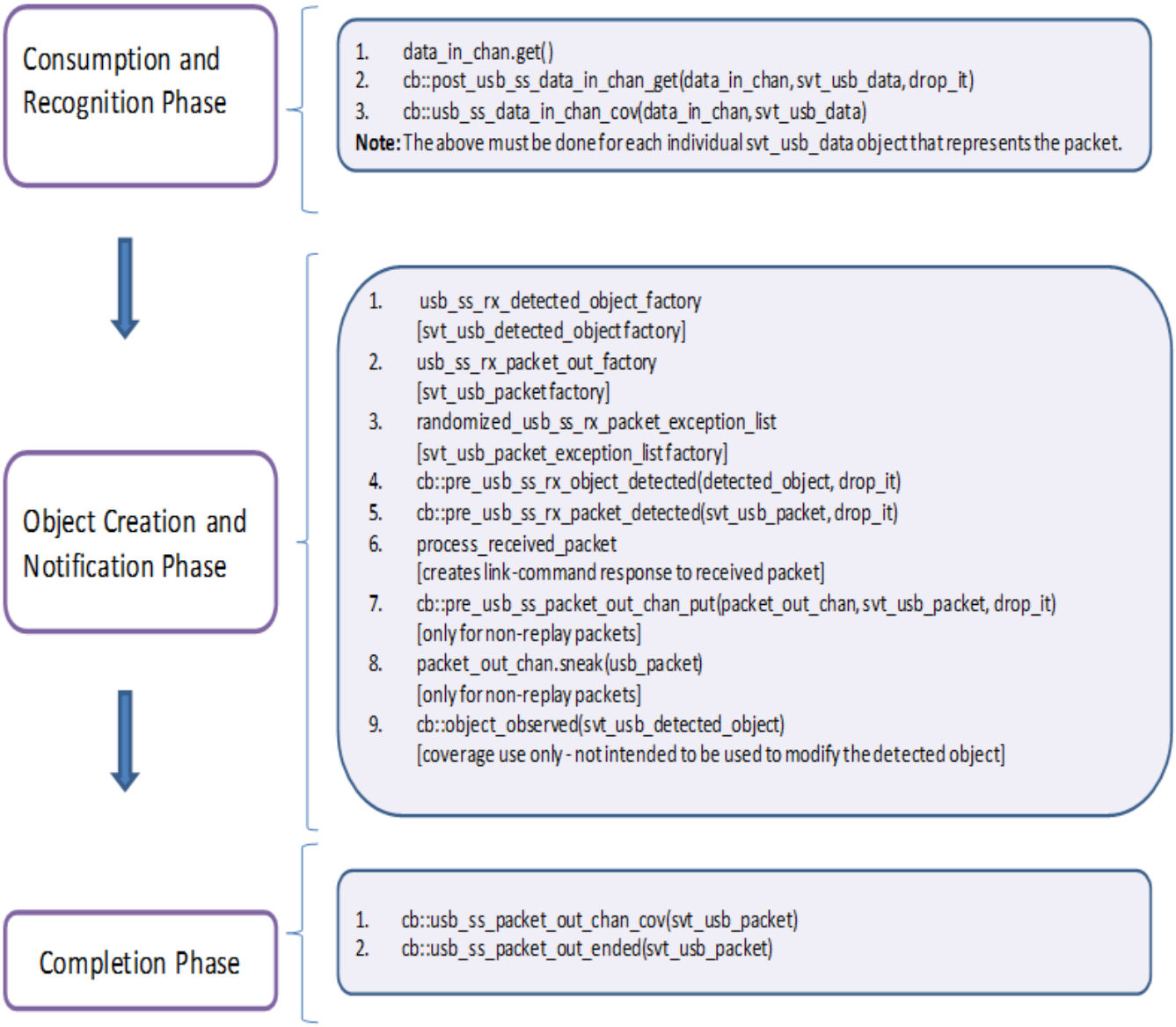
SuperSpeed Packet: Skip-Ordered Set Flow

Figure 6-18 SuperSpeed Transmit Skip Ordered Set Flow



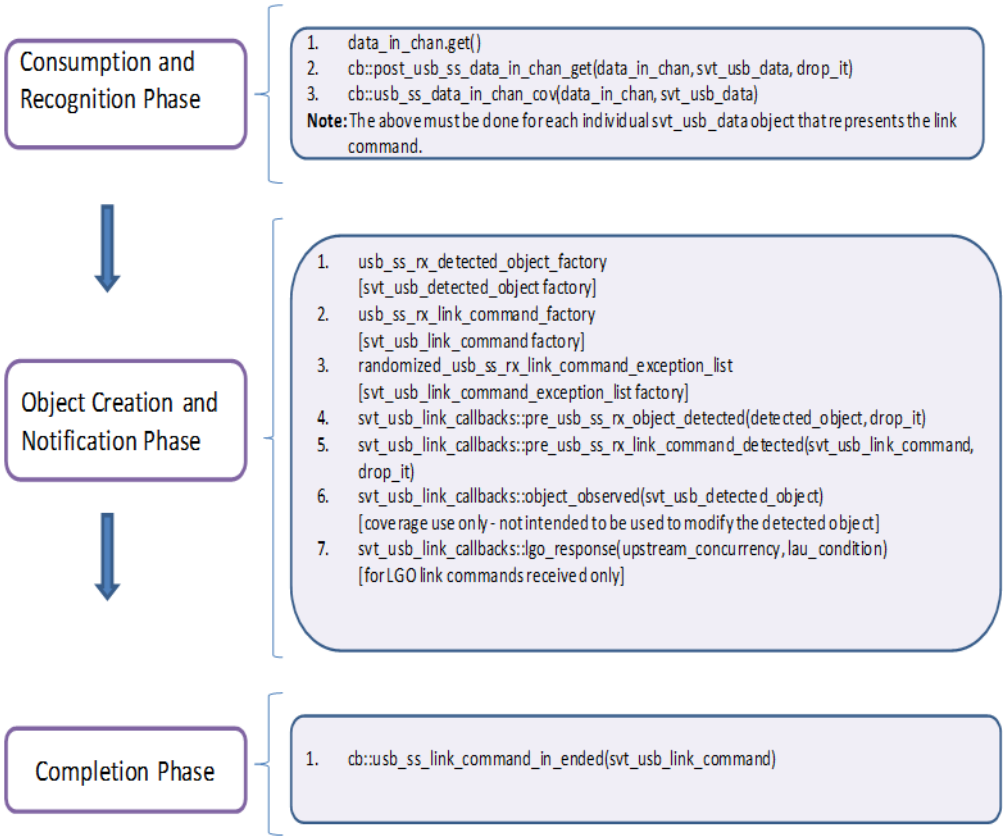
SuperSpeed Packet: Receive Flow

Figure 6-19 SuperSpeed Receive Packet Flow



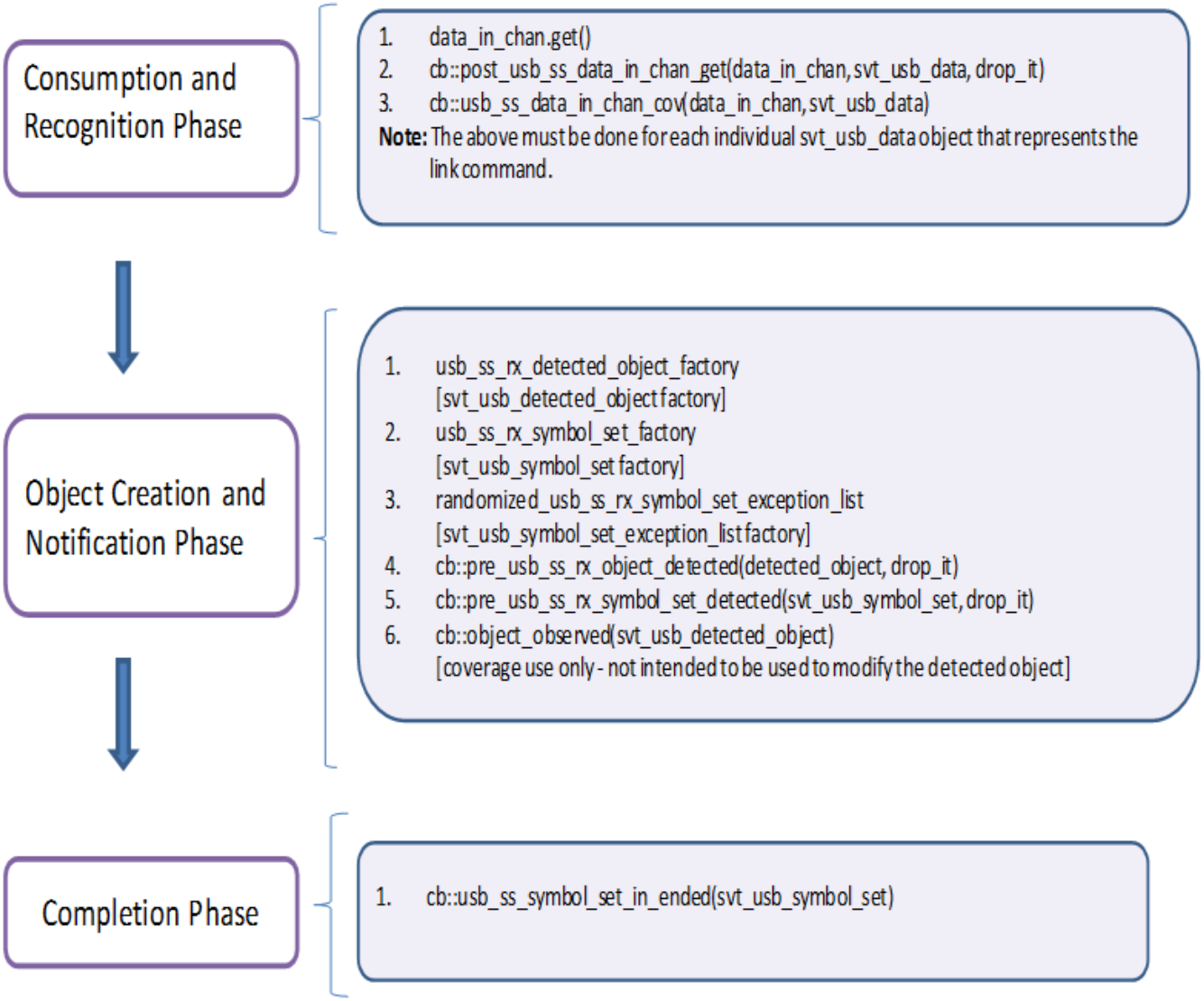
SuperSpeed Packet Receive Link Command Flow

Figure 6-20 SuperSpeed Receive Link Command Flow



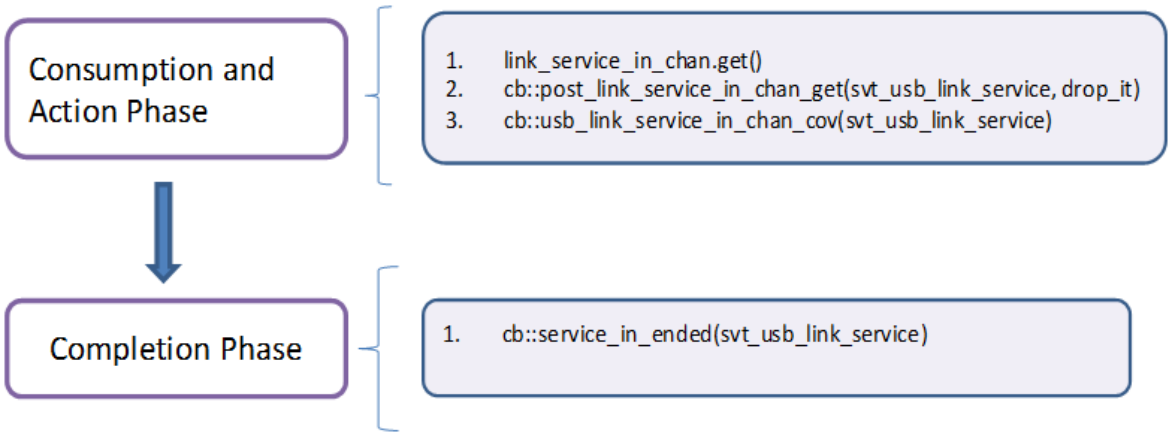
SuperSpeed Packet: Receive Symbol Set Flow

Figure 6-21 SuperSpeed Receive Symbol Set Flow



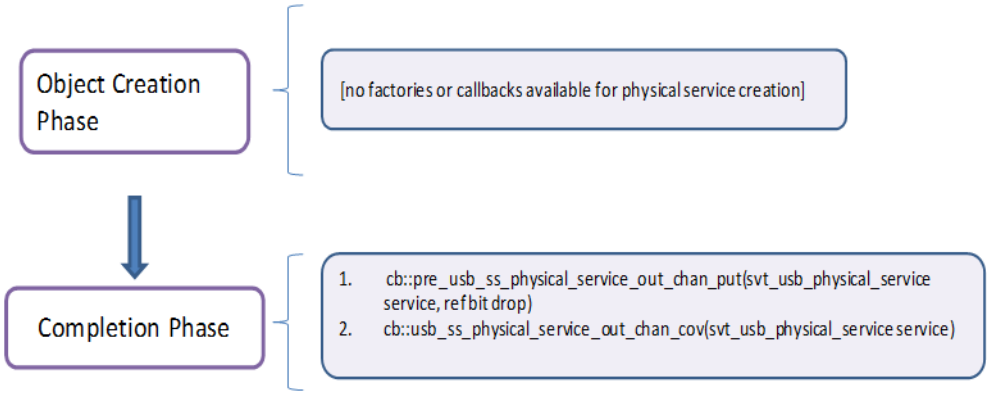
SuperSpeed Packet: Link Service Request Flow (Incoming)

Figure 6-22 Link Service Request Flow (Incoming)



SuperSpeed Packet: Physical Request Flow (Outgoing)

Figure 6-23 SuperSpeed Physical Request Flow (Outgoing)



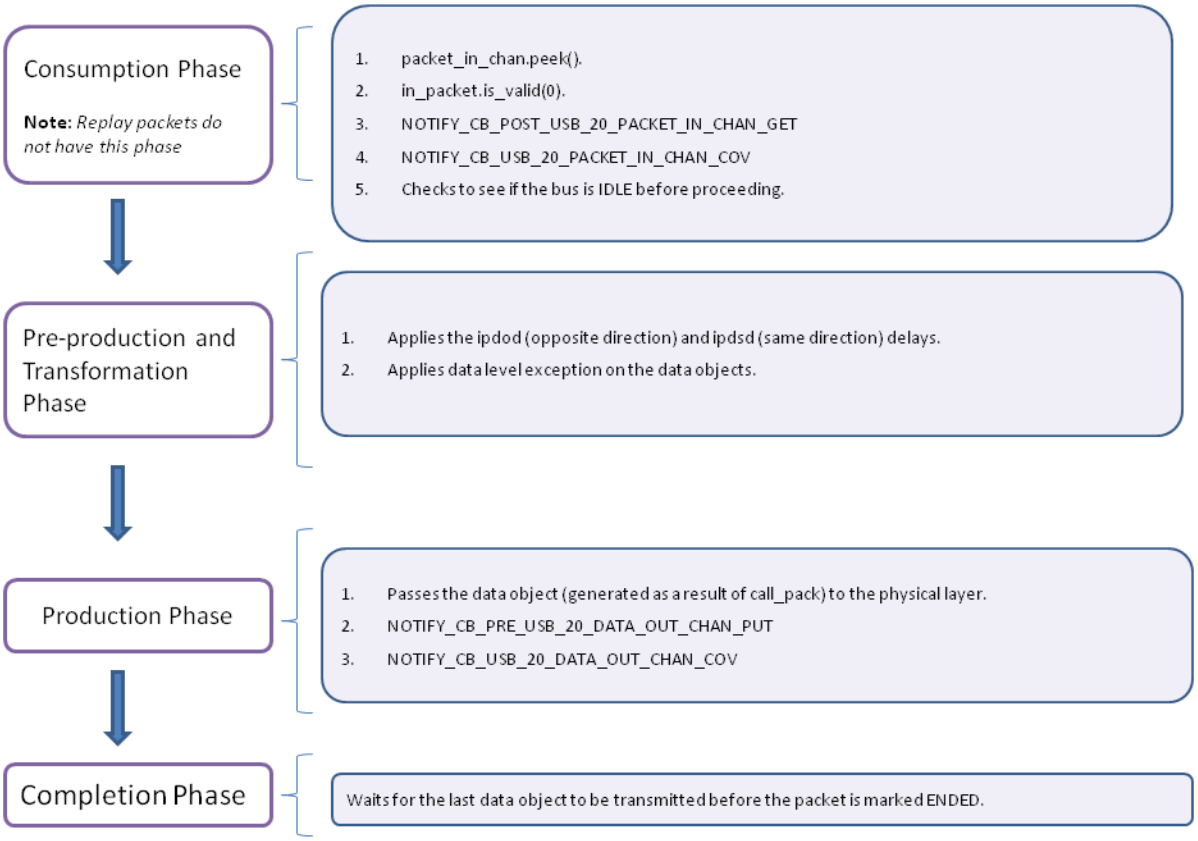
6.3.7 Link Transactor 2.0 Link Callback, Factory, and Notification Flows

This section explains the various link-layer flows through the use of flowcharts. Figures 6-24 and 6-25 illustrate the various 2.0 link transactor flows.

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

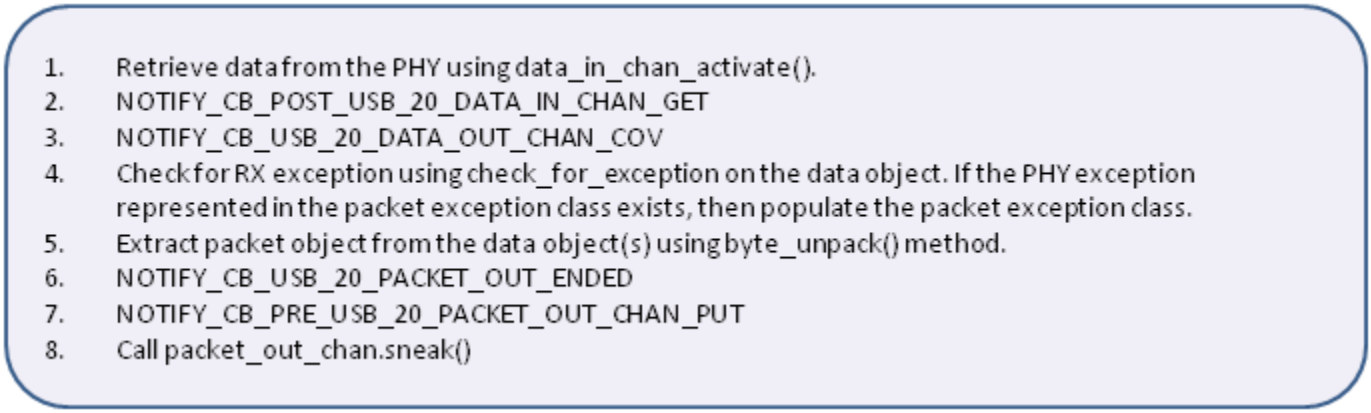
USB 2.0 Link Transmit Packet Flow

Figure 6-24 USB 2.0 Link Transmit Packet Flow



USB 2.0 Receive Packet Flow

Figure 6-25 USB 2.0 Receive Packet Flow



USB 2.0 Service Command Handling

Table 6-3 lists the various USB 2.0 link service commands that are available.

Table 6-3 USB 2.0 Service Command Usage

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

6.3.8 SuperSpeed Packet Chronology

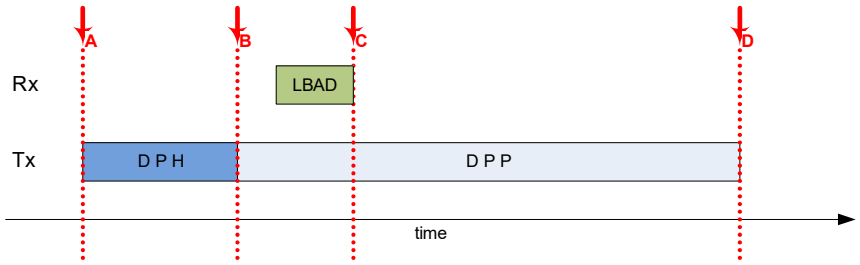
This section describes the timing of packets on the bus. Possible outcomes of low-level packet exchange on the bus are represented by the events and status values of the packet data objects within the VIP.

6.3.8.1 Receiving LBAD

6.3.8.1.1 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 6-26 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 6-4 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

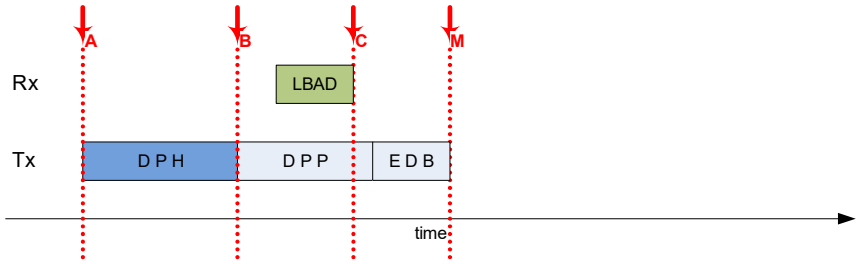
Table 6-4 Status Attribute Chronology: Receiving LBAD During DPP Tx

	Prior to Point "A"	at Point "A"	at Point "B"	at Point "C"	at Point "D"
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

6.3.8.1.2 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 6-27 Packet Chronology: Receiving LBAD During DPP Tx, EDB Result



At points "A" through "C" the events are identical to Figure 6-26 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 6-5 Status Attribute Chronology: Receiving LBAD During DPP Tx, EDB Result

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

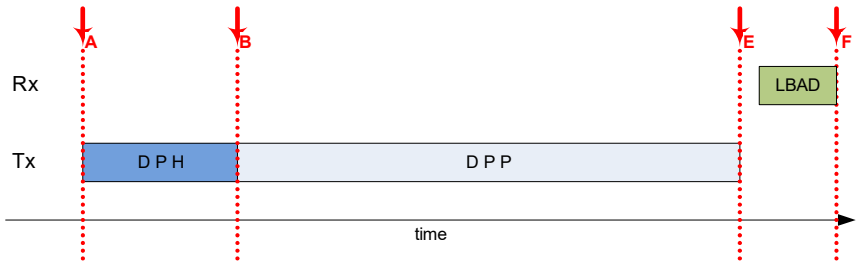
Table 6-5 Status Attribute Chronology: Receiving LBAD During DPP Tx, EDB Result

	Prior to Point "A"	at Point "A"	at Point "B"	at Point "C"	at Point "M"
payload_presence	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT_ BUT_ABORTED

6.3.8.1.3 After DPP Tx Complete

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

Figure 6-28 Packet Chronology: Receiving LBAD After DPP Tx Complete



At points "A" and "B" the events are identical to Figure 6-26 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

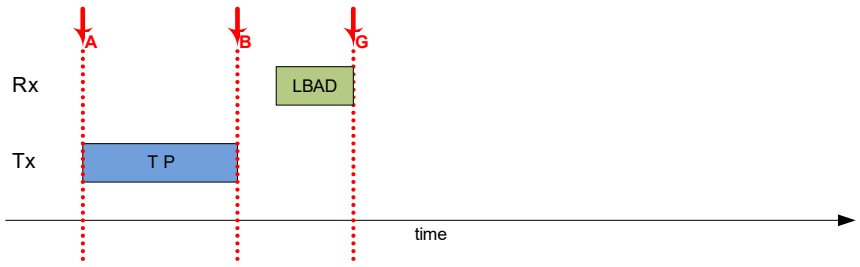
Table 6-6 Status Attribute Chronology: Receiving LBAD After DPP Tx Complete

	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

6.3.8.1.4 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 6-29 Packet Chronology: Receiving LBAD After HP, No DPP



At point "A" the events are identical to Figure 6-26 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

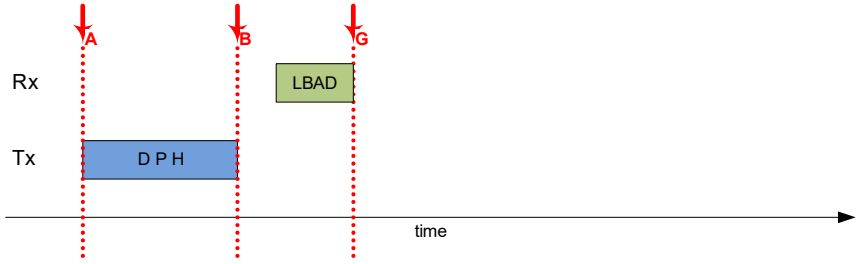
Table 6-7 Status Attribute Chronology: Receiving LBAD After HP, No DPP

	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

6.3.8.1.5 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 6-30 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

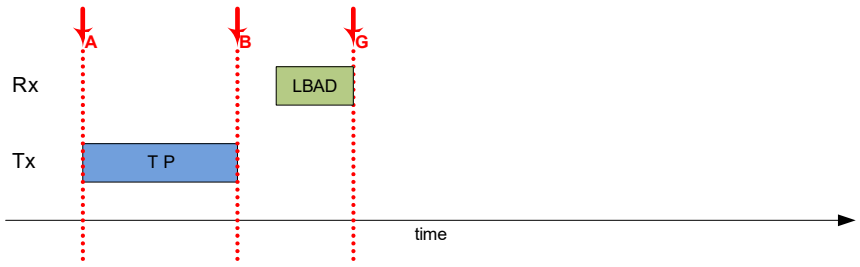
Table 6-8 Status Attribute Chronology: Receiving LBAD After Data Packet replay

	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

6.3.8.1.6 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 6-31 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 6-9 Status Attribute Chronology: Receiving LBAD After Non-Data Packet Replay

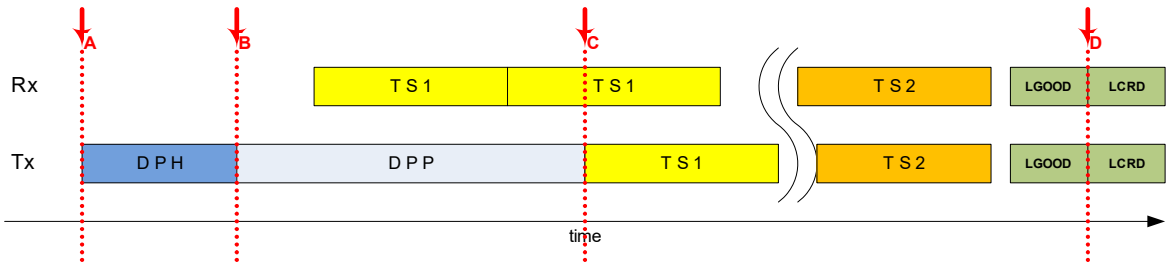
	Prior to Point "A"	at Point "A"	at Point "B"	at Point "G"
status	RETRY	ACTIVE	PARTIAL_ACCEPT	RETRY
header_status	RETRY	ACTIVE	ACCEPT	ACCEPT
payload_status	DISABLED	DISABLED	DISABLED	DISABLED
payload_presence	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT

6.3.8.2 Receiving TS1 Ordered Sets

6.3.8.2.1 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 6-32 Packet Chronology: Receiving TS1 ordered sets During DPP Tx



At points “A” and “B” the events are identical to Figure 6-26 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 6-10 Status Attribute Chronology: Receiving TS1 Ordered Sets During DPP Tx

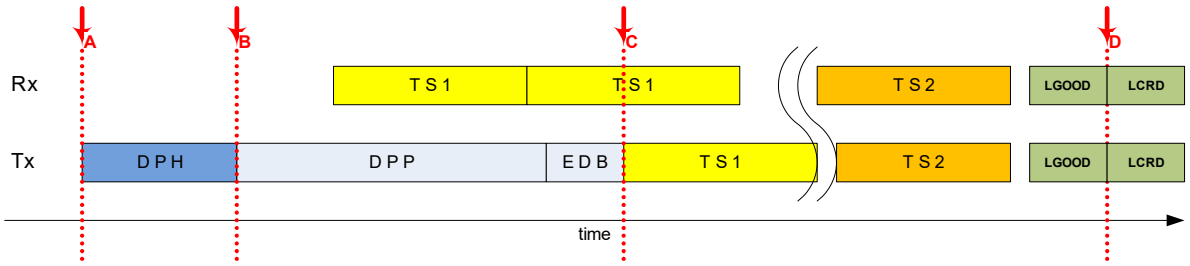
	Prior to Point “A”	at Point “A”	at Point “B”	at Point “C”	at Point “D”
status	INITIAL	ACTIVE	ACTIVE	PARTIAL_ ACCEPT	RETRY
header_status	INITIAL	ACTIVE	ACCEPT	ACCEPT	ACCEPT
payload_status	INITIAL	INITIAL	ACTIVE	ACCEPT	ACCEPT
payload_presence	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_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

6.3.8.2.2 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 6-33 Packet Chronology: Receiving TS1 Ordered Sets During DPP Tx, EDB Result



At points “A” and “B” the events are identical to Figure 6-26 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

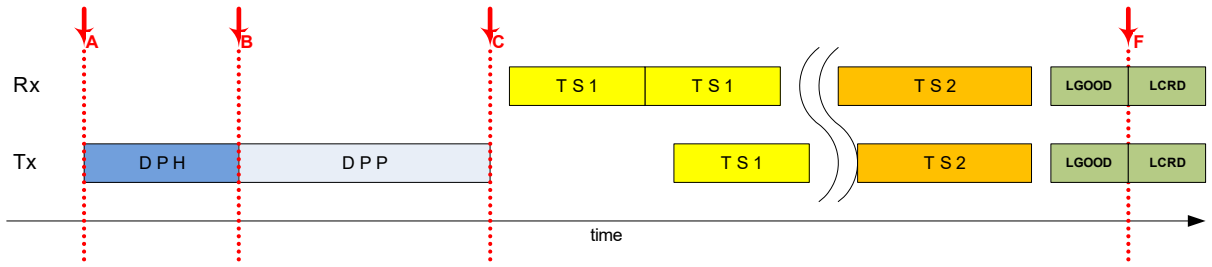
Table 6-11 Status Attribute Chronology: Receiving TS1 Ordered Sets During DPP Tx, EDB Result

	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

6.3.8.2.3 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 6-34 Packet Chronology: Receiving TS1 Ordered Sets After DPP Tx Complete



At points “A” through “C” the events are identical to Figure 6-28 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

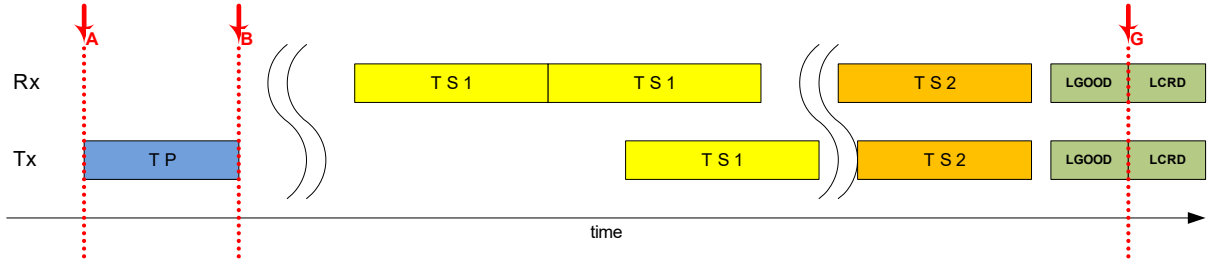
Table 6-12 Status Attribute Chronology: Receiving TS1 Ordered Sets After DPP Tx Complete

	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
payload_presence	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT

6.3.8.2.4 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 6-35 Packet Chronology: Receiving TS1 Ordered Sets After HP, No DPP



At points “A” and “B” the events are identical to Figure 6-29 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

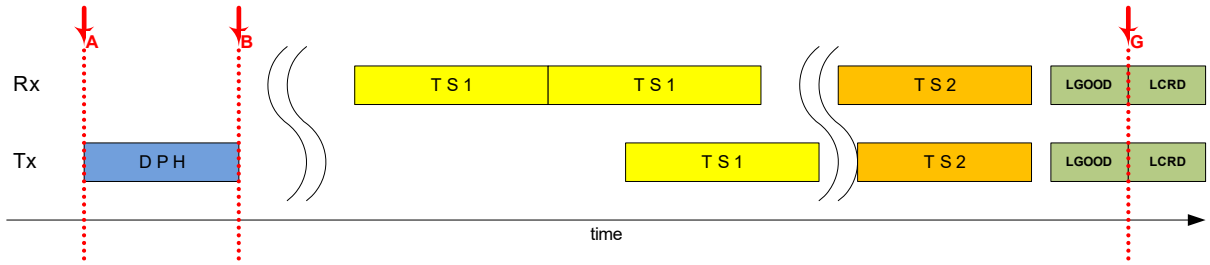
Table 6-13 Status Attribute Chronology: Receiving TS1 Ordered Sets After HP, No DPP

	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

6.3.8.2.5 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 6-36 Packet Chronology: Receiving TS1 Ordered Sets After Data Packet replay



At points “A” and “B” the events are identical to Figure 6-30 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

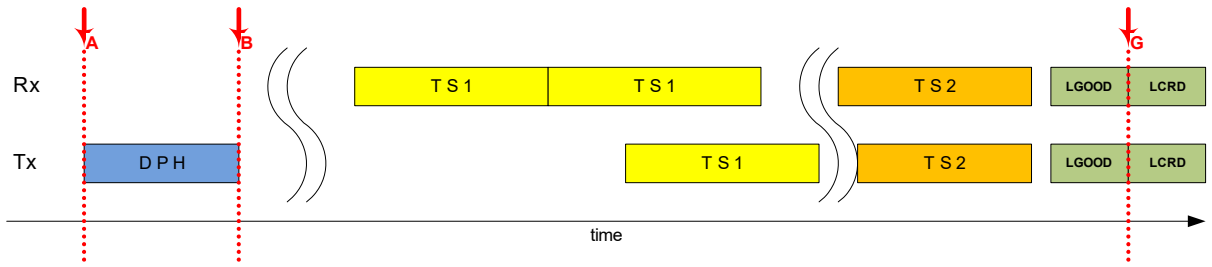
Table 6-14 Status Attribute Chronology: Receiving TS1 Ordered Sets After Data Packet replay

	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

6.3.8.2.6 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 6-37 Packet Chronology: Receiving TS1 Ordered Sets After Data Packet Replay



At points “A” and “B” the events are identical to Figure 6-29 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 6-15 Status Attribute Chronology: Receiving TS1 Ordered Sets After Data Packet Replay

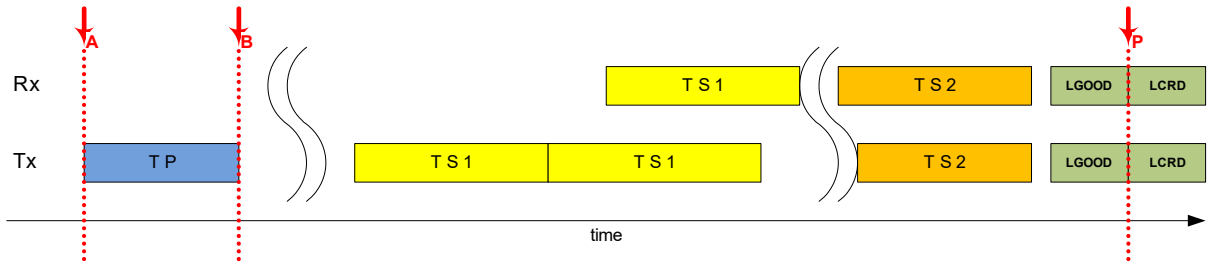
	Prior to Point "A"	at Point "A"	at Point "B"	at Point "G"
status	RETRY	ACTIVE	PARTIAL_ACCEPT	RETRY
header_status	RETRY	ACTIVE	ACCEPT	ACCEPT
payload_status	DISABLED	DISABLED	DISABLED	DISABLED
payload_presence	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT	PAYLOAD_ NOT_PRESENT

6.3.8.3 Transmitting TS1 Ordered Sets

6.3.8.3.1 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 6-38 Packet Chronology: Transmitting TS1 Ordered Sets After HP Tx



At points "A" and "B" the events are identical to Figure 6-29 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

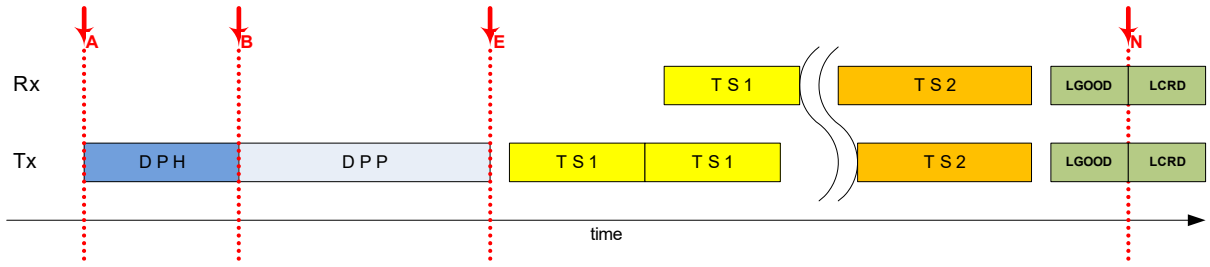
Table 6-16 Status Attribute Chronology: Transmitting TS1 Ordered Sets After HP Tx

	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

6.3.8.3.2 After DPP Tx

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

Figure 6-39 Packet Chronology: Transmitting TS1 Ordered Sets After DPP Tx



At points “A” through “E” the events are identical to Figure 6-28 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

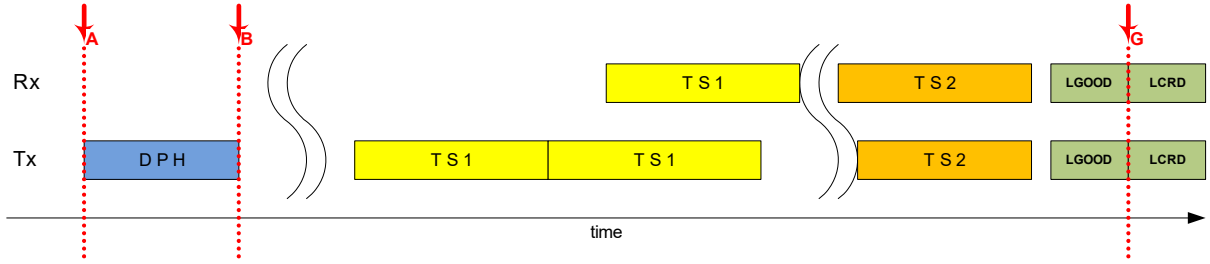
Table 6-17 Status Attribute Chronology: Transmitting TS1 Ordered Sets After DPP Tx

	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

6.3.8.3.3 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 6-40 Packet Chronology: Transmitting TS1 Ordered Sets After Data Packet Replay



Events at points "A" through "G" are identical to Figure 6-30 events at points "A" through "G"

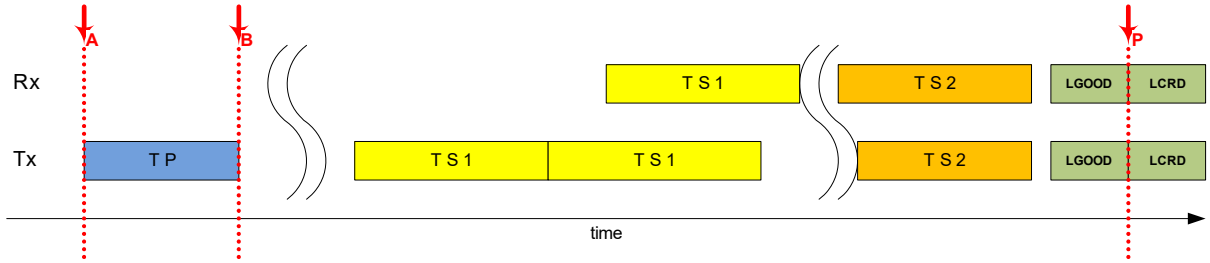
Table 6-18 Status Attribute Chronology: Transmitting TS1 Ordered Sets After Data Packet Replay

	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

6.3.8.3.4 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 6-41 Packet Chronology: Transmitting TS1 Ordered Sets After Non-Data Packet Replay



At points "A" and "B" the events are identical to Figure 6-29 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 6-19 Status Attribute Chronology: Transmitting TS1 Ordered Sets After Non-Data Packet Replay

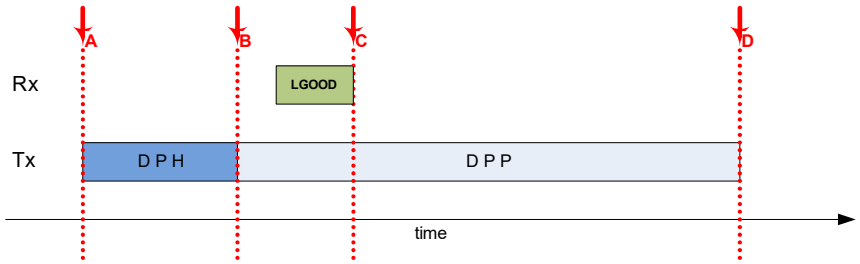
	Prior to Point "A"	at Point "A"	at Point "B"	at Point "G"
status	RETRY	ACTIVE	PARTIAL_ACCEPT	RETRY
header_status	RETRY	ACTIVE	ACCEPT	ACCEPT
payload_status	DISABLED	DISABLED	DISABLED	DISABLED
payload_presence	PAYLOAD_NOT_PRESENT	PAYLOAD_NOT_PRESENT	PAYLOAD_NOT_PRESENT	PAYLOAD_NOT_PRESENT

6.3.8.4 Receiving LGOOD

6.3.8.4.1 During DPP Tx

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

Figure 6-42 Packet Chronology: Receiving LGOOD During DPP Tx



At points "A" through "C" the events are identical to Figure 6-26 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 vmm_data::ENDED on packet

Table 6-20 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

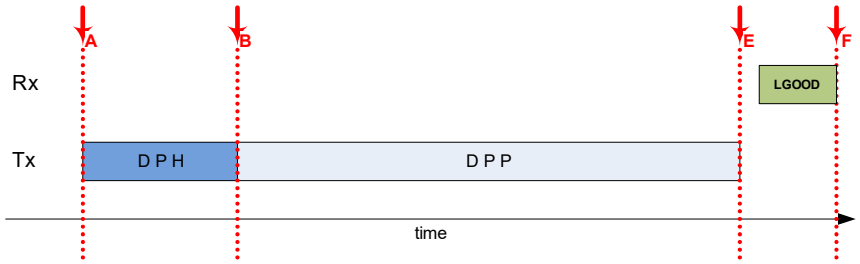
Table 6-20 Status Attribute Chronology: Receiving LGOOD During DPP Tx

	Prior to Point "A"	at Point "A"	at Point "B"	at Point "C"	at Point "D"
payload_presence	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT

6.3.8.4.2 After DPP Tx Complete

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

Figure 6-43 Packet Chronology: Receiving LGOOD After DPP Tx Complete



At points "A" and "B" the events are identical to Figure 6-26 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 indicated vmm_data::ENDED on packet

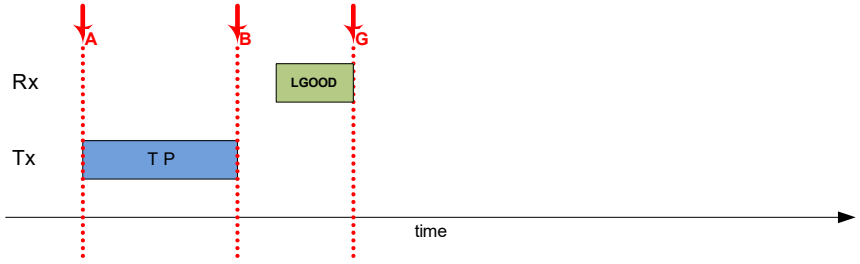
Table 6-21 Status Attribute Chronology: Receiving LGOOD After DPP Tx Complete

	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
payload_presence	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESENT	PAYLOAD_ PRESEnt

6.3.8.4.3 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 6-44 Receiving LGOOD After HP, No DPP



At points “A” and “B” the events are identical to Figure 6-29 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 vmm_data::ENDED on packet

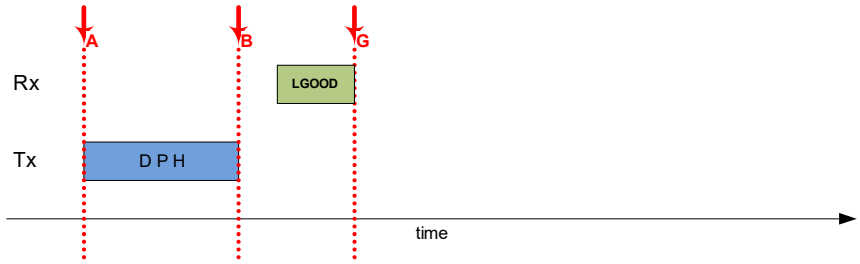
Table 6-22 Status Attribute Chronology: Receiving LGOOD After HP, No DPP

	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

6.3.8.4.4 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 6-45 Packet Chronology: Receiving LGOOD After Data Packet Replay



Events at points “A” and “B” are identical to Figure 6-30 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 vmm_data::ENDED on packet

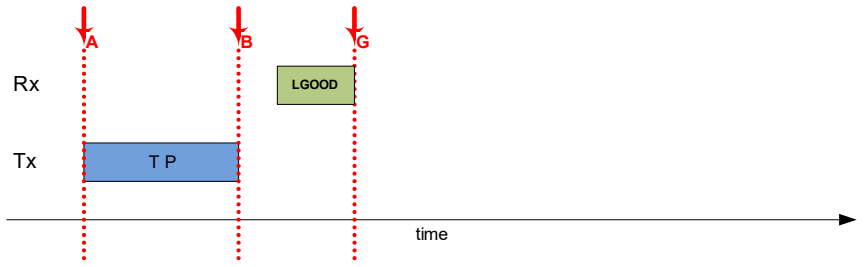
Table 6-23 Status Attribute Chronology: Receiving LGOOD After Data Packet Replay

	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

6.3.8.4.5 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 6-46 Packet Chronology: Receiving LGOOD After Non-Data Packet Replay



At point "A" and "B" the events are identical to Figure 6-26 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 vmm_data::ENDED on packet

Table 6-24 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

6.3.9 Related Topics About Link Transactor

FOR INFO ABOUT:	SEE:
Link service commands	The HTML class reference.
Factories, callbacks, and channels	Complete list of VMM factories, callbacks, and channels used by the Link Transactor in the HTML class reference.

6.4 Physical Transactor

USB VIP Physical transactors are component objects in a VMM-compliant verification environment. The USB VIP Physical transactor object extends from the svt_xactor class, which extends from the vmm_xactor base class. This object implements all methods specified by VMM for the vmm_xactor class.

The USB VIP Physical transactor is the element in a layered USB protocol stack that models the physical layer of the USB protocol. In this capacity, the USB VIP Physical transactor provides two features:

- Physical layer processing: VIP Physical transactors 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 transactors connect to simulations through a signal interface. Physical transactors 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 transactor functions and attributes.

6.4.1 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 channels
- In/Out Service

  reset, Vbus ON/OFF, Attach/Detach, Power State, Scrambling ON/OFF, Rx/Tx - Polarity, Rx/Tx LFPS ON/OFF
- Configurable input channel stimulus
  - Auto connect to link layer
  - Direct channel
  - VMM atomic generator
  - VMM scenario generator
- Error injection
- Callbacks providing testbench visibility and control

6.4.2 Data Flow Support

6.4.2.1 Physical Transactor Channels

The following list describes objects that the move information between the Protocol Transactor and other VIP objects. Figure 6-47 represents Physical Layer information flow and displays the following objects in context of that flow.

- **Data channels:** These objects convey USB data objects between the Physical and Link transactors. The transactor supports the following data channels:
  - ***Physical Data Input channels:*** These objects receive USB data objects from other Physical layer transactors. Separate channels are provided to support USB SS and USB 2.0 traffic.

    VIP object name: *usb_ss_physcial_data_in_chan, usb_20_physical_data_in_chan*.
  - ***Physical Data Output channels:*** These objects transmit USB data objects to other Physical layer transactors. Separate channels are provided to support USB SS and USB 2.0 traffic.

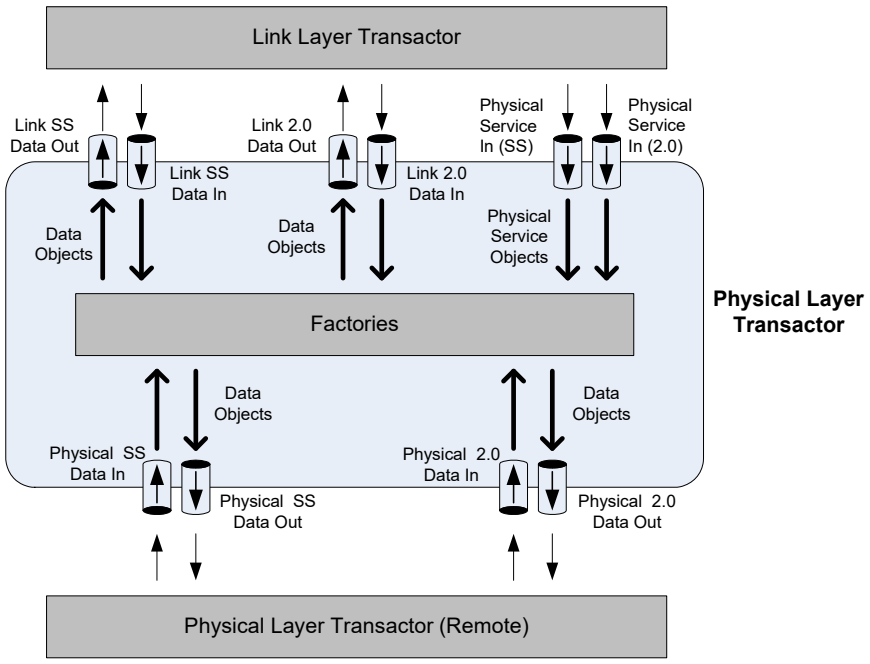
    VIP object name: *usb_ss_physcial_data_out_chan, usb_20_physical_data_out_chan*.
  - ***Link Data Input channels:*** These objects receive USB data objects from Link Layer transactors. Separate channels are provided to support USB SS and USB 2.0 traffic.

    VIP object name: *usb_ss_link_data_in_chan, usb_20_link_data_in_chan*.
  - ***Link Data Output channels:*** These objects transmit USB data to Link Layer transactors. Separate channels are provided to support USB SS and USB 2.0 traffic.

    VIP object name: *usb_ss_link_data_out_chan, usb_20_link_data_out_chan*.
- **Physical Service channels:** These objects receive USB Physical Service objects from the Link transactor. Separate channels are provided to support USB SS and USB 2.0 traffic.

  VIP object name: *usb_20_physical_service_in_chan, usb_20_physical_service_in_chan*.

Figure 6-47 Physical Transactor Data Flow through Channel Objects

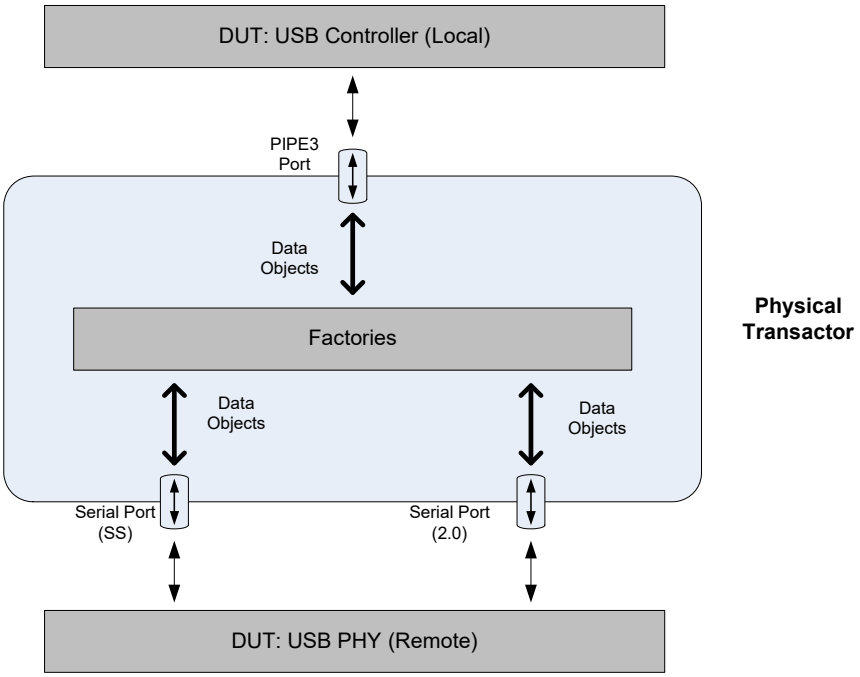


6.4.2.2 Signal Interfaces

Unlike other USB transactors, USB Physical transactors communicate through signal interfaces as well as channels. The following is a list of the ports that access Physical transactor signal interfaces. Figure 6-48 represents Physical Layer information flow and displays the following objects in context of that flow.

- **MAC and PHY Interfaces:** Physical transactors communicate with link transactors through the following ports:
  - **PIPE3 Bus:** pipe3_up_port, pipe3_down_port
- **Serial Interfaces:** Physical transactors communicate with the physical layer of the testbench or DUT through the following ports:
  - **USB 2.0 Serial Bus:** usb_20_serial_up_port, usb_20_serial_down_port
  - **USB SS Serial Bus:** usb_ss_serial_up_port, usb_ss_serial_down_port

Figure 6-48 Physical Transactor 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 transactors do not support the simultaneous configuration of a signal interface on link and physical interfaces. Signal interfaces are only supported on one transactor side at a time.

For a description of valid Physical transaction configurations that utilize signal interfaces, see the Interface Options

6.4.2.3 Data Objects

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

Transaction Objects describes USB data objects

- **Data Objects:** These objects that represent USB transfer data units 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.

6.4.2.4 Transaction Support

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

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

Physical transactors 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

6.4.3 Interface Options

The VIP Physical Layer transactor supports the following verification topologies:

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

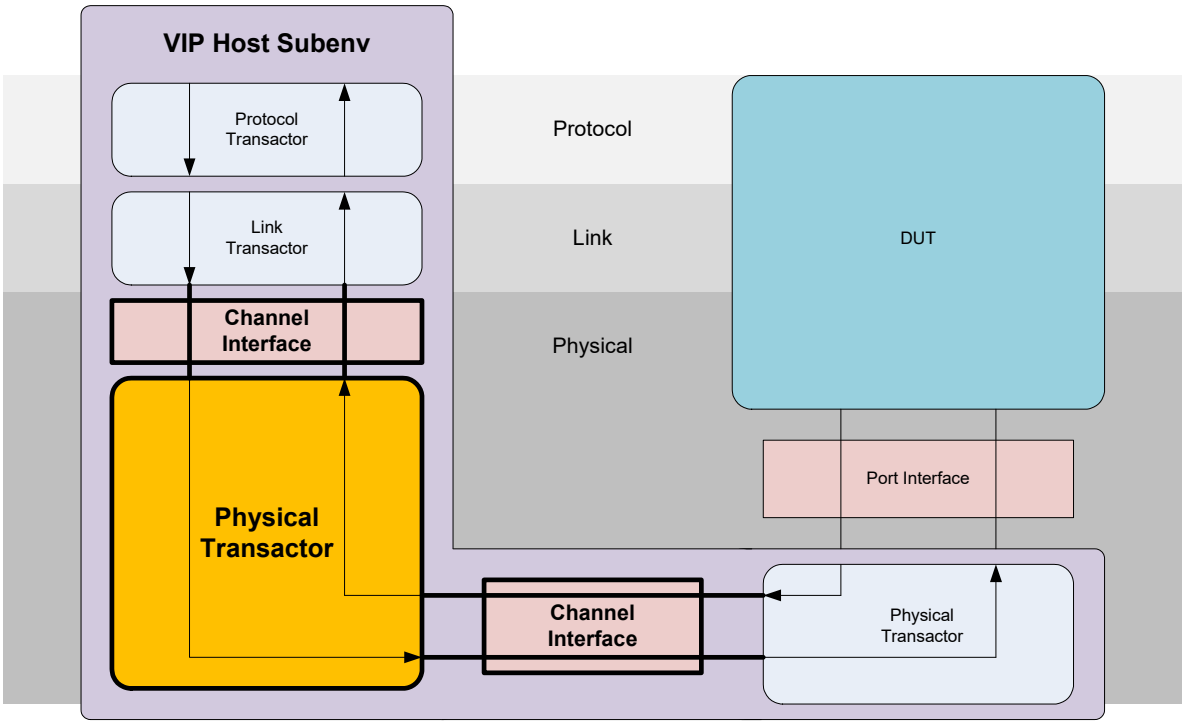
The following sections describe these scenarios. Channels, ports, and data objects listed in these descriptions are described in Data Flow Support.

6.4.3.1 Connecting the Physical transactor to a Link transactor and a remote Physical transactor

This Physical transactor 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 Transactor; the core is connected to the other side of the remote Physical transactor, as described in Connecting the Physical transactor to a Link transactor and a remote DUT. When connecting Link layer and a remote Physical Layer transactors, channels typically exchange information between the transactors.

Figure 6-49 displays the data flow for this Physical Transactor implementation.

Figure 6-49 Physical Transactor connected to a Link Transactor and a remote Physical transactor

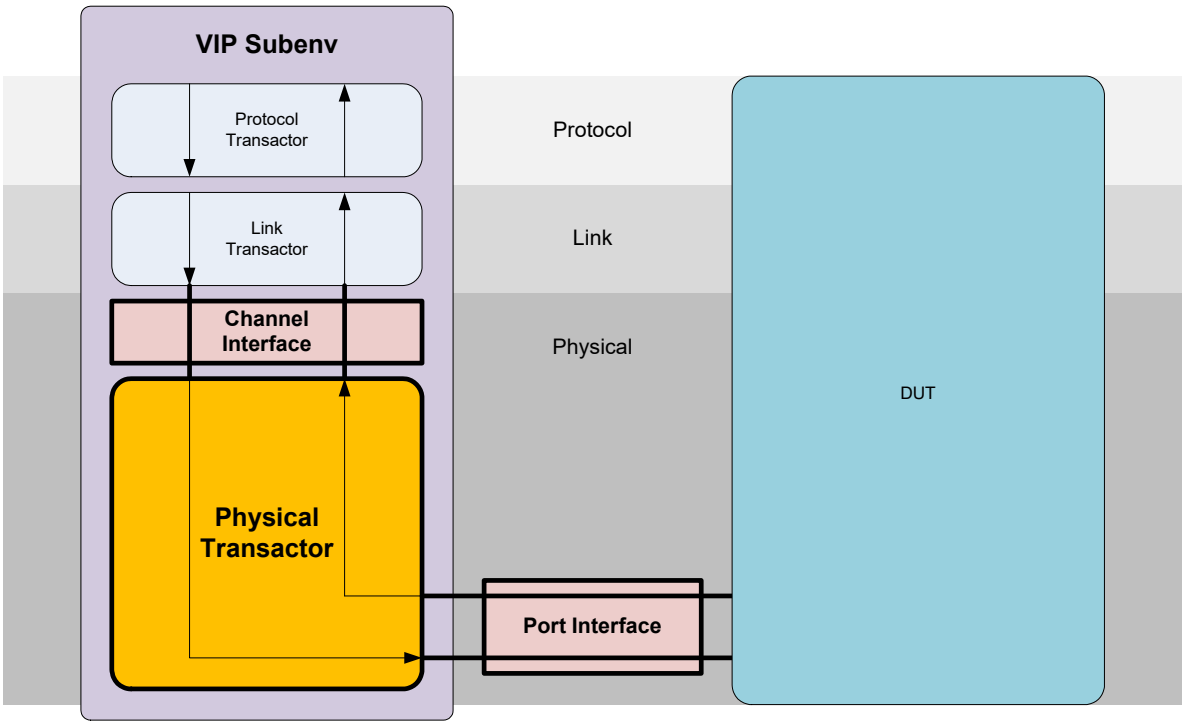


6.4.3.2 Connecting the Physical transactor to a Link transactor and a remote DUT PHY

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

Figure 6-50 displays the data flow for this Physical Transactor implementation.

Figure 6-50 Physical Transactor connected to a Link transactor and a remote DUT PHY

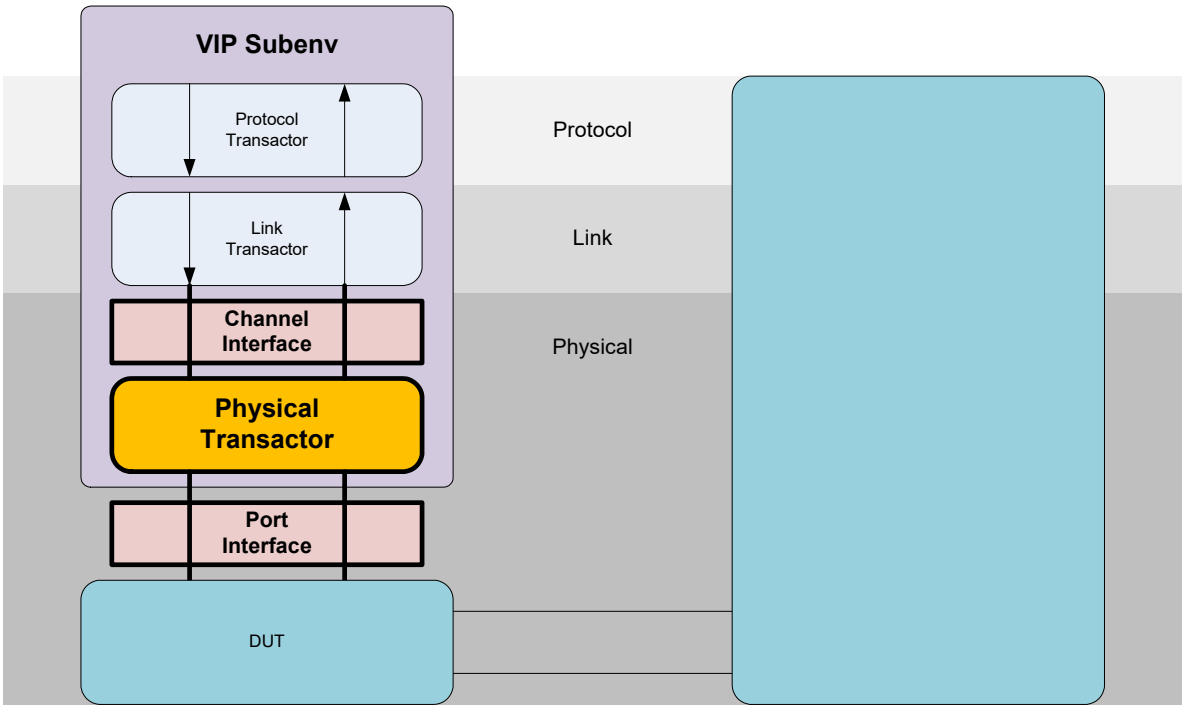


6.4.3.3 Connecting the Physical transactor to a Link transactor and a remote DUT

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

Figure 6-51 displays the data flow for this Physical Transactor implementation.

Figure 6-51 Physical Transactor connected to a Link transactor and a remote DUT

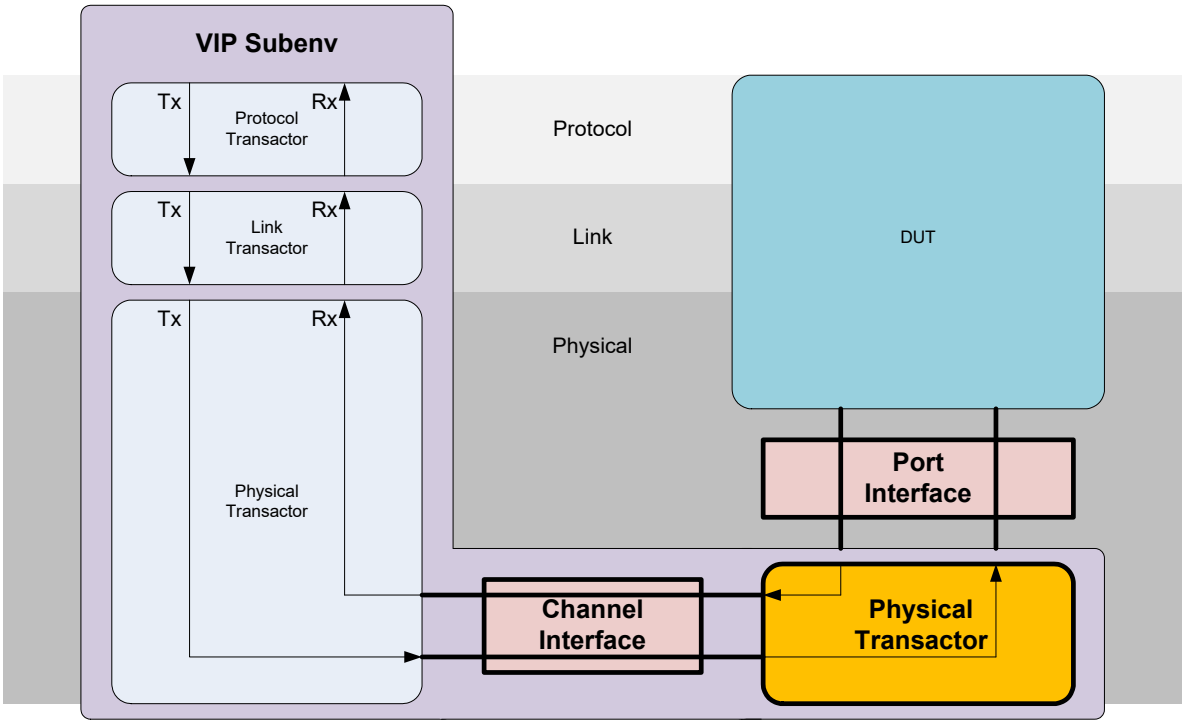


6.4.3.4 Connecting the Physical transactor to a Link Layer transactor and a local PHY (DUT)

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

Figure 6-52 displays the data flow for this Physical Transactor implementation.

Figure 6-52 Physical Transactor connected to a Link Layer transactor and a local PHY (DUT)



6.4.4 Interface File Features

Interface files implement features that VMM 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 – transactor 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

6.4.5 Information Transformation Objects

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

6.4.5.1 Data Output Factory Objects

USB Physical transactors support a factory for each output channel:

- **link data out factories:** These factories allocate new data descriptor instances to represent data placed in link data out channels. This factory is always present. Replace this factory with the extended version to cause the Physical transactor to use an extended data descriptor.

  Protocol transactor attribute names: *usb_ss_link_data_out_factory, usb_20_link_data_out_factory.*

- **physical data out factories:** These factories allocate new data descriptor instances to represent transmitted data placed in physical data out channels. This factory is always present. Replace this factory with the extended version to cause the Physical transactor to use an extended data descriptor.

  Protocol transactor attribute names: `usb_ss_physical_data_out_factory`, `usb_20_physical_data_out_factory`.

6.4.5.2 Exception List Factories

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

Assign factories to USB Physical transactors to generate exception lists. Replacement factories are either assigned by the USB Physical transactor'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 transactors. Exception lists resulting from randomization are applied to the data descriptor.

  Protocol transactor attribute names: `randomized_usb_ss_link_data_exception_list_factory`, `randomized_usb_20_link_data_exception_list_factory`.

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

  Protocol transactor attribute names:
  `randomized_usb_ss_physical_data_exception_list_factory`, `randomized_usb_20_physical_data_exception_list_factory`.

6.4.6 Exception Support

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

6.4.6.1 Error Injection

USB Physical transactors 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

6.4.6.2 Error Detection

The USB Physical transactor 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

6.4.7 Notification Support

Physical transactors use notifications to facilitate the communication of state and event control/status between other USB stack transactors and the testbench. Notifications are defined in the shared notify object

The following are examples for using physical–layer notification

- 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.

6.4.8 Physical Transactor Callbacks

Physical Transactor Callbacks objects extend from the `svt_xactor_callbacks` class, which extends from the `vmm_xactor_callbacks` base class. This object implements all methods specified by VMM for `vmm_xactor_callbacks`.

To create unique implementations of `svt_usb_physical callbacks`, extend the `svt_usb_physical_callbacks` class. To register an instance of the callback object with the USB Physical

transactor, use `append_callback`. This registration is typically after the `build()` implementation if the testbench is derived from the `vmm_env` class.

The Physical transactor supports the following callbacks

- **Channel input:** These callbacks indicate that the transactor collected data from an input channel.

  Physical transactor callback method names: `post_usb_ss_link_data_in_chan_get, post_usb_20_physical_service_in_chan_get, post_usb_20_link_data_in_chan_get, post_usb_20_physical_data_in_chan_get, usb_20_link_data_in_ended, post_usb_ss_physical_data_in_chan_get, post_usb_ss_physical_service_in_chan_get, usb_20_link_data_in_chan_cov, usb_20_physical_data_in_chan_cov, usb_20_physical_service_in_chan_cov, usb_20_service_in_ended, usb_ss_link_data_in_chan_cov, usb_ss_link_data_in_ended, usb_ss_physical_data_in_chan_cov, usb_ss_physical_service_in_chan_cov, usb_ss_service_in_ended`

- **Channel output:** These callbacks indicate that the transactor placed data on an output channel.

  Physical transactor callback method names: `pre_usb_ss_link_data_out_chan_put, pre_usb_ss_physical_data_out_chan_put, pre_usb_20_link_data_out_chan_put, pre_usb_20_physical_data_out_chan_put, usb_20_link_data_out_chan_cov, usb_20_link_data_out_ended, usb_20_physical_data_out_chan_cov, usb_ss_link_data_out_chan_cov, usb_ss_link_data_out_ended, usb_ss_physical_data_out_chan_cov`

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

  Physical transactor 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 Transactor.

  Physical transactor 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 transactor callback method names: `errors_driven`

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

  Physical transactor callback method names: `transaction_invalid_traffic`

6.4.9 Related Topics About Physical Transactor

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

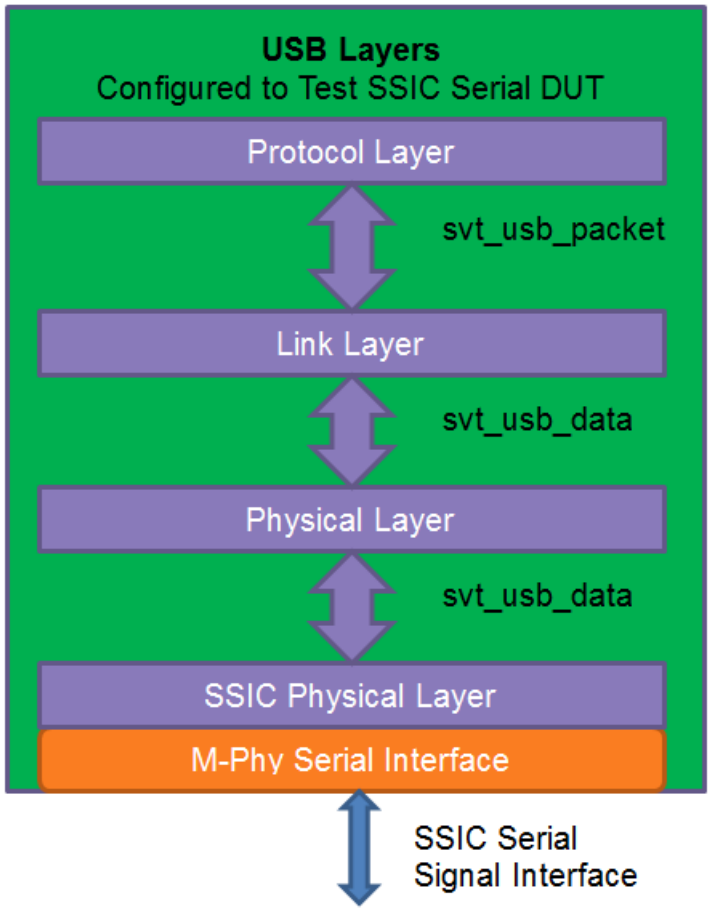
6.5 SSIC Physical Layer

The USB VIP implements the SSIC interface through a VMM transactor called **svt_usb_ssic_physical.** The USB VIP SSIC Physical transactor object extends from the svt_xactor class, which extends from the vmm_xactor base class. This object implements all methods specified by VMM for the vmm_xactor class.

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

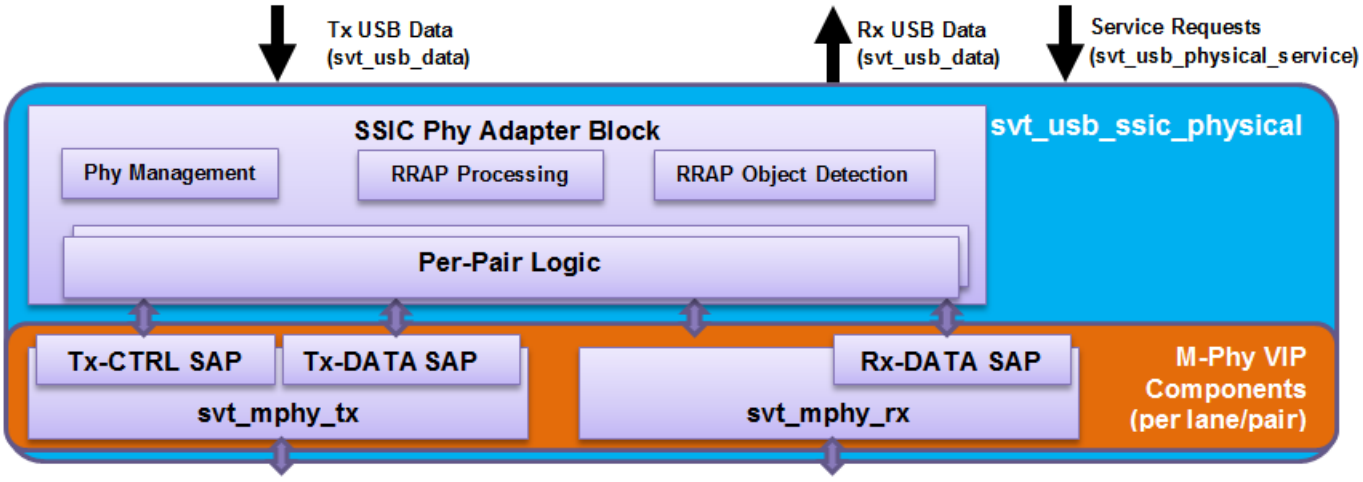
The following figure shows the releationship of the SSIC Physical Layer to the other stacked layers of the VIP.

Figure 6-53 VIP Layered Architecture with SSIC/M-Phy Serial Interface



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

Figure 6-54 Functionality of SSIC Physical Layer



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

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

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

Each lane also supports simultaneous CRTL and DATA accesses

6.5.1 Configuration Classes

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

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

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

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

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

6.5.2 SSIC Physical Service Channel

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

- Providing svt_usb_physical_service objects directly to the SSIC Physical transactor is not supported.
- Uses physical_service_in_chan.

6.5.3 SSIC Physical Transaction Channels

The SSIC Physical transactor uses the following channels.

- svt_usb_data_channels. While the SSIC Physical transactor has IN and OUT channels for this data type test bench interaction directly with these channels is not expected. Instead interact with Physical transactor's channels.
  - data_in_chan
    - The svt_usb_data objects from the Physical transactor (from the Link) to be transmitted on the USB bus.
  - data_out_chan
    - The svt_usb_data objects to the Physical transactor (to the Link) which were received on the USB bus.
  - usb_data_from_per_rx_chan[]
    - The internal use only channel which holds the svt_usb_data objects which were received on a single LANE of the SSIC implementation in use.
  - usb_data_to_per_tx_chan[]
    - The internal use only channel which holds the svt_usb_data objects to be transmitted on a single LANE of the SSIC implementation in use.
- svt_mphy_transaction channels
  - sap_ctrl_to_mphy_rx_chan[] & sap_ctrl_to_mphy_tx_chan[]
    - The svt_mphy_transaction SAP CONTROL object to the M-Phy instance for a single lane.
  - sap_data_from_mphy_rx_chan[] &sap_data_to_mphy_tx_chan[]
    - The svt_mphy_transaction SAP DATA object from/to the M-Phy instance for a single lane.

6.5.4 Signal Interface

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

6.5.5 Connection Options

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

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

6.5.6 Data Factory Objects

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

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

6.5.7 Exception List Factories

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

- randomized_ssic_rrap_packet_tx_exception_list
- randomized_ssic_rrap_transaction_exception_list

6.5.8 Error injection

The SSIC physical layer provides the following error injection methods.

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

6.5.9 Error Detection

The SSIC layer provides the following error detection methods.

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

- Reserved field error

6.5.10 Notifications

Following are notifications supported byte SSIC Physical transactor:

- NOTIFY_SSIC_BURSTEND_PADDING
- NOTIFY_SSIC_RECEIVED_INVALID_RRAP_PACKET
- NOTIFY_MPHY_RX_AGGREGATE_STATE_CHANGE

6.5.11 Transactions

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

6.5.11.1 Data Transactions

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

- **svt_usb_data –** Transfers data symbols between the Link, Physical, and SSIC Physical layers.
- **svt_mphy_transaction –** Produced and/or consumed by the SSIC Physical Layer to communicate CTRL and DATA SAPs to/from the M-Phy interface components.
- **svt_usb_ssic_rrap_transaction –** Represents an SSIC *Remote Register Access Protocol* command / response pair. Its **implementation** array will consist of 2 **svt_usb_ssic_rrap_packet** data objects.
- **svt_usb_ssic_rrap_packet** - Represents a single SSIC *Remote Register Access Protocol* command or response.

6.5.11.2 Service Transactions

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

- SSIC_RRAP
  - Requests execution of the SSIC RRAP transaction defined by **rrap_transaction** (USB SSIC only)
- SSIC_MTX_EXIT_HIBERN8
  - Requests SSIC Tx Phy(s) to exit Hibernate state (USB SSIC only)
- SSIC_MTXRX_CFG_FOR_HIBERN8
  - Requests SSIC Tx & Rx Phy(s) to be configured to allow entry to Hibernate state. If issued while in HS_BURST, upon exiting HS_BURST CFGRDY CTRL SAP objects are issued to all local M-Phy instances. (USB SSIC only)
- SSIC_MTX_EXIT_BURST
  - Requests SSIC Tx Phy(s) to exit BURST state. Use of this in LS burst will not cause the SSIC specification defined LS BURST CLOSURE sequence. To follow the defined LS BURST CLOSURE sequence, use a physical service with **physical_command** set to **SSIC_RRAP** and have the **rrap_transaction** setup to write to the BURST_CLOSURE RRAP register. (USB SSIC only)
  - If a **SSIC_MTXRX_CFG_FOR_HIBERN8** occurred while in HS_BURST mode when this command is processed, a CFGRDY SAP CTRL is issued to all M-Phys once the mphy_tx_aggregate_fsm_state and mphy_rx_aggregate_fsm_state in svt_usb_status indicate

STALL has been reached. The command is not ENDED until both Phy states indicate HIBERN8 was reached.

- SSIC_MTX_ENTER_BURST
  - Requests SSIC Tx Phy(s) to enter BURST state (USB SSIC only)
- SSIC_MTX_LINE_RESET
  - Requests SSIC Tx Phy(s) to perform LINE Reset (USB SSIC only)
- SSIC_MPHY_RESET
  - Requests SSIC Tx Phy(s) to perform the reset operation selected by mphy_reset_kind (USB SSIC only)
- SSIC_DSP_DISCONNECT
  - Requests a DSP disconnect (USB SSIC only)
- SSIC_TEST_MODE_HS_BURST
  - Used in Test mode. Moves Tx Phy of pair under test from STALL to HS_BURST, Transmit the payload in the object attached to this service request (USB SSIC only)

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

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

6.5.12 SSIC Interface

The **svt_usb_if()** has six arrays of sub-interfaces to support SSIC:

- Intefaces used by VIP if DUT subsystem includes both local and remote phys, and VIP connects at its own MAC/Phy interface (i.e. VIP does not model Phy at all):
  - **svt_mphy_rmmi_mtx_dut_phy_if**
  - **svt_mphy_rmmi_mrx_dut_phy_if**
- Interfaces used if DUT subsystem does not include a phy at all (i.e. VIP models both local and remote phys) and DUT connects at remote MAC/Phy RMMI Interface:
  - **svt_mphy_rmmi_mtx_dut_controller_if**
  - **svt_mphy_rmmi_mrx_dut_controller_if**
- Interfaces used if both VIP and DUT contain/model phys, and both connect to the serial signal interfaces:
  - **svt_mphy_serial_tx_if**
  - **svt_mphy_serial_rx_if**

Each of the above is an array, and in the applicable arrays one entry in each for every SSIC Lane to be used must be instantiated.

Attention The signal level interfaces (whether Serial or RMMI) are as defined by the M-Phy interface. Consutl M-Phy documentation for more details.

6.5.13 SSIC Physical Transactor Callbacks

The SSIC Physical transactor contains the following callbacks:

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

6.5.14 Shared Status

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

- mphy_tx_status[$] – Array of objects (one per configured M-Phy lane) of type svt_mphy_status. These instances become the shared status objects associated with each M-Phy Tx component in use in the svt_usb_ssic_physical layer component.
- mphy_rx_status[$] – Array of objects (one per configured M-Phy lane) of type svt_mphy_status. These instances become the shared status objects associated with each M-Phy Rx component in use in the svt_usb_ssic_physical layer component.
- mphy_tx_aggregate_fsm_state – Represents the effective state of the M-Phy Tx components, as combined from all active lanes.
- mphy_rx_aggregate_fsm_state – Represents the effective state of the M-Phy Rx components, as combined from all active lanes.
- ssic_mphy_tx_aggregate_state_stable – Bit which is high when all the M-Phy Tx components have the same FSM state, and low otherwise.
- ssic_mphy_rx_aggregate_state_stable – Bit which is high when all the M-Phy Rx components have the same FSM state, and low otherwise.

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

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

7 Using the USB Verification IP

7.1 Introduction

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

7.2 SystemVerilog VMM Example Testbenches

This section describes SystemVerilog VMM example testbenches that show general usage for various applications. A summary of the examples is listed in Table 7-1.

Table 7-1 SystemVerilog Example Summary

Name	Source in design_dir/Description
Basic Example--Demonstrates how to implement a VMM testbench using USB VIP with SS Serial interface. This example consists of a top-level testbench, a Verilog hdl_interconnect, a VMM verification environment, a host and device sub-environment components, one test file with an atomic generator, and two directed tests.	
tb_usb_svt_vmm_basic_sys	examples/sverilog/usb_svt/tb_usb_svt_vmm_basic_sys/
Basic Additional Examples	
ts.basic_ss_serial.sv	
ts.basic_additional_20_serial.sv	
ts.basic_additional_20_serial_split.sv	
ts.basic_additional_ss_aligned.sv	
ts.basic_additional_ss_compliance.sv	
ts.basic_additional_ss_isoc.sv	
ts.basic_additional_ss_ltssm.sv	
ts.basic_additional_ss_pipe3.sv	
ts.basic_additional_ss_stream.sv	

Name	Source in design_dir/Description
Intermediate Example--Demonstrates how to implement a VMM testbench using USB VIP with SS Serial interface. This example consists of a top-level testbench, a Verilog hdl_interconnect, a VMM verification environment, a host and device sub-environment components, and one test file with a scenario generator.	
tb_usb_svt_vmm_intermediate_sys	examples/sverilog/usb_svt/tb_usb_svt_vmm_intermediate_sys/
Intermediate Additional Examples	
ts.intermediate_ss_serial.sv	examples/sverilog/usb_svt/tb_usb_svt_vmm_intermediate_sys/tests
ts.intermediate_additional_20_serial.sv	

7.3 Special Notes on coreConsultant as of the 3.45a Release

The 3.45a release is the last release Synopsys will support the use of coreConsultant with VC VIP for USB. Configuration Creator will replace coreConsultant as the preferred interactive tool for configuring the USB VIP.

Attention Synopsys highly recommends you begin to use Configuration Creator in the 3.45a release in place of coreConsultant. Synopsys will remove coreConsultant in the next release.

You can launch Configuration Creator from the 3.45a installation tree using the following command line:

```
% $DESIGNWARE_HOME/bin/vipcc
```

Configuration Creator documentation can be found at:

```
$DESIGNWARE_HOME/vip/tools/vipcc/<version>/docs
```

Once you finish inputting your configuration setting using Configuration Creator, the tool can generate a <configuration_name>.cfg file which you then load into the VIP. The file contains your configuration settings.

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

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

```
    end
  end
```

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

7.4 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.
- 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.vmm.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.

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_vmm_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 log.

The <configuration_name>.cfg is the file generated that can be loaded into VIP. The <configuration_name>.log 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_subenv_configuration object instance. The following code snippet illustrates how to use the load_prop_vals() method to load a configuration.

```
svt_usb_subenv_configuration cfg = new();
if (cfg.load_prop_vals(filename)) begin
```

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

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

7.5 Creating a Test Environment

7.5.1 Base Class: vmm_env

The vmm_env base class provides a testbench template to organize both the flow and the code associated with a test. You create a test environment by defining a new class extended from vmm_env. Because it provides an overall structure, the environment class contains (or affects) the entire test environment. Typically, the majority of code in an VMM testbench is contained in the environment, where it can be reused by other tests or projects. Because of this, the vmm_env class has a large impact on user coding, so understanding the vmm_env class is central to effectively writing the environment code.

Another result of the overarching nature of vmm_env is that it is an excellent vehicle for organizing the discussion of VMM and USB VIP techniques and methods. Therefore the subjects presented here are organized and presented in the context of vmm_env.

The vmm_env base class contains several methods that correspond to the major steps all tests follow:

Table 7-2 Primary Test Steps

Testing Step	vmm_env method
1. Decide on a test configuration	gen_cfg()
2. Construct, configure and connect the elements in the environment	build()
3. Configure the DUT	cfg_dut()
4. Start the test	start()
5. Determine when the test is done	wait_for_end()
6. Stop the test	stop()
7. Perform any post-processing	cleanup()
8. Report results	report()

Most of these methods are declared in the base class as virtual and should be extended by the user. To create a user environment, define a new class extended from vmm_env and extend the methods above to add test-

specific code. To retain the core functionality of the base class methods, each extended method must call super.method as the first line of code.

The code snippet below shows the user extension of vmm_env, creating the class user_env. The user_env class instantiates all the components of the testing environment, which may include VIP products, user designs, generators, scoreboards, and so on. At the end of this extended class is a list of the methods that are extended.

```
//////////////////////////////////////////////////////////////////////
// Verification Environment
//////////////////////////////////////////////////////////////////////
class user_env extends vmm_env
   int status;            // general purpose var to hold return
   int test_failed = 1;
   int gen_cnt     = 0;   // number of transactions that have been generated
   int dev_cnt     = 0;   // number of transactions detected by device
   string  msg;

   TestControlPort  tcp = TestControlBind;

   // Instantiate the objects that compose the environment
   static vmm_log                          log;

   // USB VIP models
   svt_usb_subenv                        host;
   svt_usb_subenv                        dev;

   // Transfers
   svt_usb_transfer                      randomized_hs_xfer;

   // Configurations
   test_cfg                              tb_cfg;
   svt_usb_subenv_configuration          host_cfg;
   svt_usb_subenv_configuration          dev_cfg;
   svt_usb_endpoint_configuration        endpoint_cfg;

   // Interfaces
   svt_usb_if                            dev_usb_ss_if;
   svt_usb_if                            host_usb_ss_if;


   // Environment components
   event                               end_simulation;
   event                               gen_enough;
   event                               dev_enough;


   // Utility method which creates the properly typed USB Subenv object for the host
   extern virtual protected function svt_usb_subenv new_host_subenv(string inst);

   // Utility method which creates the properly typed USB Subenv object for the device
   extern virtual protected function svt_usb_subenv new_device_subenv(string inst);
```

```
   function svt_usb_subenv user_env::new_host_subenv(string inst);
      begin
         new_host_subenv = new("svt_usb_subenv", inst, this.consensus,
            this.host_cfg, this.host_usb_ss_if);
      end
   endfunction


   function svt_usb_subenv user_env::new_device_subenv(string inst);
      begin
         new_device_subenv = new("svt_usb_subenv", inst, this.consensus,
            this.dev_cfg, this.dev_usb_ss_if);

   // List of methods in this class
   extern function new(string name);
   extern virtual function gen_cfg();
   extern virtual function build();
   extern virtual task cfg_dut();
   extern virtual task start();
   extern virtual task wait_for_end();
   extern virtual task stop();
   extern virtual task report();
endclass: user_env
```

In addition to the methods that have already been mentioned, the vmm_env base class also contains a method called run, which does not require any user extension. This method binds the individual steps into a test sequence by calling the other methods in the following order:

gen_cfg -> build -> cfg_dut -> start -> wait_for_end -> stop -> cleanup -> report

Calling this one method (run) launches the entire test sequence inside a program block:

```
   initial
      begin
      // Call build and then insert verbosity control
         env.build();
      // User can choose levels of messages for more or less information
         env.host.log.set_verbosity(vmm_log::TRACE_SEV);
         env.device.log.set_verbosity(vmm_log::TRACE_SEV);
         env.log.set_verbosity(vmm_log::NORMAL_SEV);
      // The remaining steps in the testing sequence will be called by run()
         env.run();
      $finish;
   end
```

Notice how small the program is when you use VMM. Most of the code is in the environment, which is a reusable component. The test-specific code is minimized and as much code as possible is common so it is not replicated unnecessarily. This yields a smaller code base to maintain.

You may have noticed that there is a call to env.build in the main program. This shows the flexibility built into the VMM architecture in allowing the user to customize the standard test sequence. vmm_env maintains a list of the standard test sequence methods that have already been called. When run is executed, it skips the methods that have already been run. So the standard sequence can be modified to suit the needs of each test.

In the example code, calls to set_verbosity are inserted into the standard sequence by first calling build. After setting the verbosity for logging, run is called to continue with the remainder of the test sequence. You

can see how all the models have the same interface and usage. In fact, since the testbench itself uses VMM tasks for logging, it is also controlled in exactly the same manner.

The user inherits structure and base functionality from vmm_env and yet can still customize where needed. Furthermore, the customization is under the user's control. This is a common theme throughout VMM and the VIP products.

In the sections that follow, the individual steps in the test sequence are shown in detail.

7.6 Instantiating the VIP

Instantiate the VIP sub-environment, svt_usb_subenv, in the test bench and pass in the appropriate interface to the subenv's constructor.

```
svt_usb_subenv new_host_subenv;
new_host_subenv = new("svt_usb_subenv", "inst_name", this.consensus,  this.host_cfg,
this.host_usb_ss_if);
```

For more information, see the HTML class reference.

7.7 Configuring the VIP Models

VIP models are configured using configuration objects that are provided and ready for use. Configuration objects can be handled just like any other objects so they can be randomized and passed as an argument to a method. The objects come with constraints so that they adhere to protocol limits. These can be controlled or extended as desired to create specific test conditions, or used as is to produce a wide range of stimulus.

The test configuration is established in gen_cfg. The code below shows that the configuration objects are randomized and then some of the attributes of the vip_cfg object are manually assigned. This is one approach to controlling the values of attributes.

```
function user_env::gen_cfg ();
   begin
      super.gen_cfg();

      // Enforce making the top level in the stack the PROTOCOL layer
      this.cfg.host_cfg.top_xactor = svt_usb_subenv_configuration::PROTOCOL;
      this.cfg.dev_cfg.top_xactor = svt_usb_subenv_configuration::PROTOCOL;

      if (`DISPLAY_CFG)
         begin
            host_cfg.display("host_cfg: ");
            cfg_device.display("cfg_device: ");
            tb_cfg.display("tb_cfg: ");
         end
      end
   endfunction
```

The testbench then declares a test-level configuration object to determine testbench behavior.

The next method in the test sequence is build. This section constructs, configures and connects elements. The new() task for the USB VIP transactors has an argument specifying a configuration object. The code below shows the host_cfg and cfg_device objects that gen_cfg generated and supplied to the new task for the transactors. You can also configure Transactors after construction with the change_xactor_config method.

```
function user_env::build() ;
   begin
```

```
super.build();
   randomized_hs_xfer = new();
   setup_if ();
   setup_remote_if();

   host =   new_host_subenv ("host_subenv");
// The instName "USB HOST" appears in VIP messages and must be unique

   device = new_device_subenv ("dev_subenv");
// The instName "USB DEVICE" appears in VIP messages and must be unique

// Connect the subenv via channels at the bottom.
   host_subenv.phys.usb_ss_physical_data_out_chan =
               dev_subenv.phys.usb_ss_physical_data_in_chan;
   dev_subenv.phys.usb_ss_physical_data_out_chan =
               host_subenv.phys.usb_ss_physical_data_in_chan;

// Establish logger hierachy. Env's log is on top so recursive commands
// will include the models as well
   this.log.is_above(host.log);
   this.log.is_above(device.log);

endfunction

task user_env::cfg_dut() ;
   begin
      super.cfg_dut();
   end
endtask
```

7.8 Generating Constrained Random Stimulus

Constrained random generation uses the built-in randomize() method that all SystemVerilog objects possess. The most common use for random generation is to produce a series of random transactions which follow a set of applied constraints. The SV-VMM basic example shows a simple generator that produces individual (atomic) transactions randomly with no particular relationship between them.

VMM provides three generator types: an atomic generator, a scenario generator, and a multi-stream scenario generator. These provide a great deal of functionality for specifying and controlling the generated random sequences.

7.9 Controlling the Test

All VIP models share the same features regarding test control. Although there are many features that support test control, this section highlights two basic features: starting/stopping and logging. The VIP transactors are all extended from the base class vmm_xactor so they all have the same control interface for starting and stopping. Each transactor implements start_xactor and stop_xactor methods that handle all steps that are needed to either start or stop a transactor. Encapsulation of function is one of the key elements in the architecture of VIP and VMM. The testbench does not have to know how to start a transactor, it simply has to call the start_xactor method.

The start task in the environment class is where the test is 'launched'. This generally means that any transactors or components in the simulation are started. The code snippet below is from the example. This

simulation consists of VIP transactors and the start code simply calls the start_xactor method of each transactor.

```
task user_env::start() ;
   begin

      super.start();

      // Start the VIP models
      fork
         begin
            host.start();
         end

         begin
            device.start();
         end
      join

      sim_ctrl_subenv.start()
   end

endtask
```

VIP transactors provide logging and messaging for controlling tests. Messaging is performed by vmm_log objects (logs), which are included in every transactor. The user_env that was created earlier also contains a log that it inherits from vmm_env. Inheritance and reuse go hand in hand.

VMM defines message types and severities. This allows two degrees of freedom for sorting and operating on messages. Severities are defined as levels ranging from FATAL to DEBUG. These levels allow the messages to be processed distinctly.

Each log object has a threshold level and will log any messages with a severity equal to or higher than the threshold. The level of logging detail is set with the set_verbosity method, as this code from the top-level program shows:

```
      // Configure how much messaging to display
         env.host.log.set_verbosity(vmm_log::TRACE_SEV);
         env.device.log.set_verbosity(vmm_log::TRACE_SEV);
         env.log.set_verbosity(vmm_log::NORMAL_SEV);
```

The vmm_log class has many powerful and flexible features. For example, several methods operate globally on all log objects. A global format can also be specified so that all messages use the same format. These testbench-wide controls are especially useful when integrating multiple models into a single simulation.

7.10 SuperSpeed Low Power Entry Support

7.10.1 Overview

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

7.10.2 Automatic Low-Power Entry Attempts

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

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

7.10.2.1 Controlling the U1 Inactivity Timer

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

- u1_timeout specifies the U1 inactivity timer enabled status and the U1 inactivity timeout value.

  Set u1_timeout to 0x00 to disable the U1 inactivity timer. This is the default value.

  Set u1_timeout to 0xFF to disable the U1 inactivity timer and program the downstream port to reject U1 entry requests initiated by the connected upstream port.

  Set u1_timeout between 0x01 and 0xFE to enable the U1 inactivity timer.

- u1_timeout_factor specifies the period by which the U1 inactivity timeout value is multiplied to determine the timeout period. The timeout value is

  (u1_timeout_factor) 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.

7.10.2.2 Controlling the U2 Inactivity Timer

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

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

- u2_timeout specifies the U2 inactivity timer enabled status and the U2 inactivity timeout value.

  Set u2_timeout to 0x00 to disable the U2 inactivity timer. This is the default value.

Set u2_timeout to 0xFF to disable the U2 inactivity timer and program the downstream port to reject U2 entry requests initiated by the connected upstream port

Set u2_timeout between 0x01 and 0xFE to enable the U2 inactivity timer.

- u2_timeout_factor specifies the period by which the U2 inactivity timeout value is multiplied to determine the timeout period. The timeout value is

  (u2_timeout_factor) 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.

The following SystemVerilog code creates and submits a Link Service Command that sets the current U2 inactivity timeout value to 0x01:

```
`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_U2_TIMEOUT;
link_service.u2_inactivity_timeout = 8'h01;
host_subenv.link.link_service_in_chan.put(link_service);
```

7.10.3 Automatic Low-Power Entry for Upstream Ports

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

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

This feature is not supported by the USB Specification.

7.10.4 Testbench-Initiated Low-Power Entry Attempts

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

- USB_SS_ATTEMPT_U1_ENTRY
- USB_SS_ATTEMPT_U2_ENTRY

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

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

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

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

- low_power_entry_ignore_pending_protocol
- low_power_entry_ignore_pending_link

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

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

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

- low_power_entry_wait_until_permitted

  If this variable is set to 1, the request is queued.

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

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

7.11 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.

7.11.1 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 transactors via the transactor callback classes.

Table 7-3 lists the default functional coverage features provided with USB3.0 Verification IP.

Table 7-3 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_vmm_class_reference/html/index.html

7.11.2 Covergroup Organization

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

The coverpoints are divided into the following four categories:

- "Normal Behavior Sequences" on page 177
- "Error Condition Sequences" on page 177
- "Packet Field Range Coverpoints" on page 178
- "Data Collection Coverpoints" on page 178

7.11.2.1 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

7.11.2.2 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

7.11.2.3 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

7.11.2.4 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)

7.11.2.5 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.

7.11.3 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 };}
```

7.11.4 Default Functional Coverage Class Hierarchy

The basic classes are:

- Pattern sequences
- Transactor coverage data callbacks
- Transactor coverage callbacks

7.11.4.1 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.

7.11.4.2 Transactor Coverage Data Callbacks

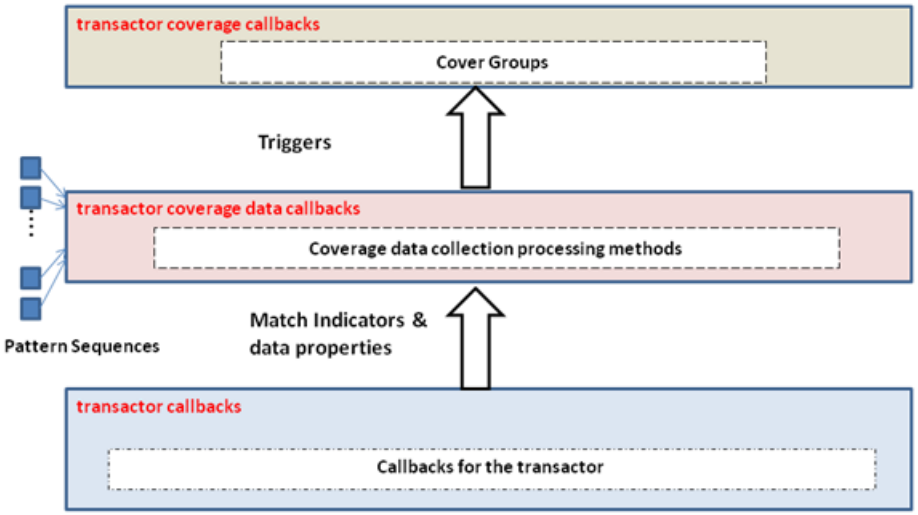
This callback class defines default data and event information that are used to implement the coverage groups. The naming convention uses “def_cov_data” in the class names for easy identification of these classes. This class also includes implementations of the coverage methods that respond to the coverage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

7.11.4.3 Transactor Coverage Callbacks

This class is extended from the transactor 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 7-1.

Figure 7-1 Default Functional Coverage Class Hierarchy



7.11.5 Coverage Callback Classes

The following naming convention is used for coverage callback classes:

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

Where:

<layer>: protocol, link, physical

<speed>: ss or 20

<VIP config>: host or device

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

Table 7-4 lists the coverage callback classes.

Table 7-4 Coverage Callback Classes

Class Name	Type	Layer	Configuration
svt_usb_protocol_ss_host_def_cov_data_callbacks	Data	Protocol	SuperSpeed Host
svt_usb_protocol_ss_host_def_cov_callbacks	Cover Groups	Protocol	SuperSpeed Host
svt_usb_protocol_ss_device_def_cov_data_callbacks	Data	Protocol	SuperSpeed Device
svt_usb_protocol_ss_device_def_cov_callbacks	Cover Groups	Protocol	SuperSpeed Device
svt_usb_protocol_20_host_def_cov_data_callbacks	Data	Protocol	USB20 Host

Table 7-4 Coverage Callback Classes

Class Name	Type	Layer	Configuration
svt_usb_protocol_20_host_def_cov_callbacks	Cover Groups	Protocol	USB20 Host
svt_usb_protocol_20_device_def_cov_data_callbacks	Data	Protocol	USB20 Device
svt_usb_protocol_20_device_def_cov_callbacks	Cover Groups	Protocol	USB20 Device
svt_usb_link_ss_def_cov_callbacks	Cover Groups	Link	SS Host or Device
svt_usb_link_ss_def_cov_data_callbacks	Data	Link	SS Host or Device
svt_usb_link_20_host_def_cov_callbacks	Cover Groups	Link	USB20 Host
svt_usb_link_20_host_def_cov_data_callbacks	Data	Link	USB20 Host

Note Information about coverage callbacks in USB3.0 Verification IP can be found in the class reference HTML located at:
$DESIGNWARE_HOME/vip/svt/usb_svt/latest/doc/usb_svt_vmm_class_reference/html/index.html

7.11.6 Using Functional Coverage

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

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

Refer to the following file for additional information:

$DESIGNWARE_HOME/vip/svt/usb_svt/latest/examples/sverilog/tb_usb_svt_vmm_intermediate_sys/env/ usb_svt_intermediate_env.sv

The example sets the attributes in gen_cfg() method. As the example uses host sub environment to set the attributes, all host related coverage is enabled.

7.11.6.1 Coverage Extensions

You can extend the coverage callback classes to specify user-defined coverage in addition to default coverage provided in the model.

Refer to the following file for additional information:

$DESIGNWARE_HOME/vip/svt/usb_svt/latest/examples/sverilog/tb_usb_svt_vmm_intermediate_sys/env/usb_svt_intermediate_coverage_callbacks.sv

Extended covergroup callback is enabled by new'ing the extended covergroup class and then appending the callback as shown here:

```
usb_svt_intermediate_coverage_callbacks usb_cov
usb_cov = new();
host_subenv.prot.append_callback(usb_cov)
```

7.11.7 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>
```

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

7.12 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.

7.12.1 VIP Acting as a Host

Table 7-5 describes the VIP behavior when it is acting as a host.

Table 7-5 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.	

7.12.2 VIP Acting as a Device

Table 7-6 describes the VIP behavior when it is acting as a host.

Table 7-6 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.	

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

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

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

LTSSM state is SS.Disabled

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

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

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

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

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

LTSSM state is Rx.Detect.Reset

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

LTSSM state is Rx.Detect.Active

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

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

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

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

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.

LTSSM state is Polling.LFPS

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

- polling_lfps_burst_time
- polling_lfps_repeat_time
- tx_lfps_duty_cycle
- tx_lfps_period

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

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

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

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

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

LTSSM state is Polling.RxEQ

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

LTSSM state is Polling.Active

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

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

LTSSM state is Polling.Configuration

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

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

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

LTSSM state is Polling.Idle

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

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

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

7.14 UTMI+ Support

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

7.14.1 Port Interface

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

- svt_usb_utmi_dut_mac_if
- svt_usb_utmi_dut_phy_if

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

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

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

$DESIGNWARE_HOME/vip/svt/usb_svt/latest/doc/usb_svt_vmm_class_reference/html/index.html

7.14.2 Configuring the UTMI+ Interface

Table 7-7 lists all the available configuration parameters you can use to set UTMI behavior.

Table 7-7 Configuration Paramaters to Control the UTMI Interface

Parameter	Legal values	Description
usb_20_signal_interface	Enum: UTMI_IF	Determines the interface to be used by the VIP
utmi_data_width	Integer: 8 or 16	This parameter determines the width of the UTMI+ interface.It is speed dependent: • HS/FS -- 8-bit interface • HS/FS -- 16-bit interface • FS Only -- 8-bit interface • LS Only -- 8-bit interface. This attribute is dependent on the speed attribute.
speed	Enum: svt_usb_types::HS, svt_usb_types::FS svt_usb_types::LS	This atttribute is used to control the UTMI+ interface speed. NOTE: For the VIP to support HS_FS functionality speed must to be set to HS.
component_subtype	ENUM MAC PHY	Represents the behavior of the VIP relative to the signal interface.

7.14.3 Error Injection

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

The VIP supports control over RxErrror by doing the following:

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

7.14.4 Attach / Detach

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

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

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

7.14.6 UTMI+ Messages

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

- Error

  A device tries to attach using the UTMI+ interface with an active high speed transceiver. The device is not attached.

- Notes
  - Transceiver reset completed. Current state of the reset pin is 1'b0.
  - Attach command called when OpMode does not equal 2'b01 (non-driving mode). Waiting for Opmode to return to 2'b01.
  - Attach requested when reset is being driven. Command ignored.
  - Data bus width is changed to16 (or 8) (the most recently sample value of databus16_8 at the transceiver reset).

7.15 USB 2.0 OMTG 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:

7.15.1 OTG Interface Signals

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

7.15.1.1 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.

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 7-8 lists and describes the OTG link interface signals.

Table 7-8 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

Table 7-8 OTG Link Interface Signals

Signal	Direction	Polarity	Default	Size	Description and Values
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
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

7.15.1.2 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.

7.15.1.2.1 ADP Interface Signals

Table 7-9 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 7-9 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

7.15.1.2.2 SRP Interface Signals

Table 7-10 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 7-10 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

7.15.2 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.

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

Figure 7-2 explains the SRP process flow when the VIP is configured as an A-device.

Figure 7-2 SRP Protocol Flow when the VIP is Configured as an A-Device

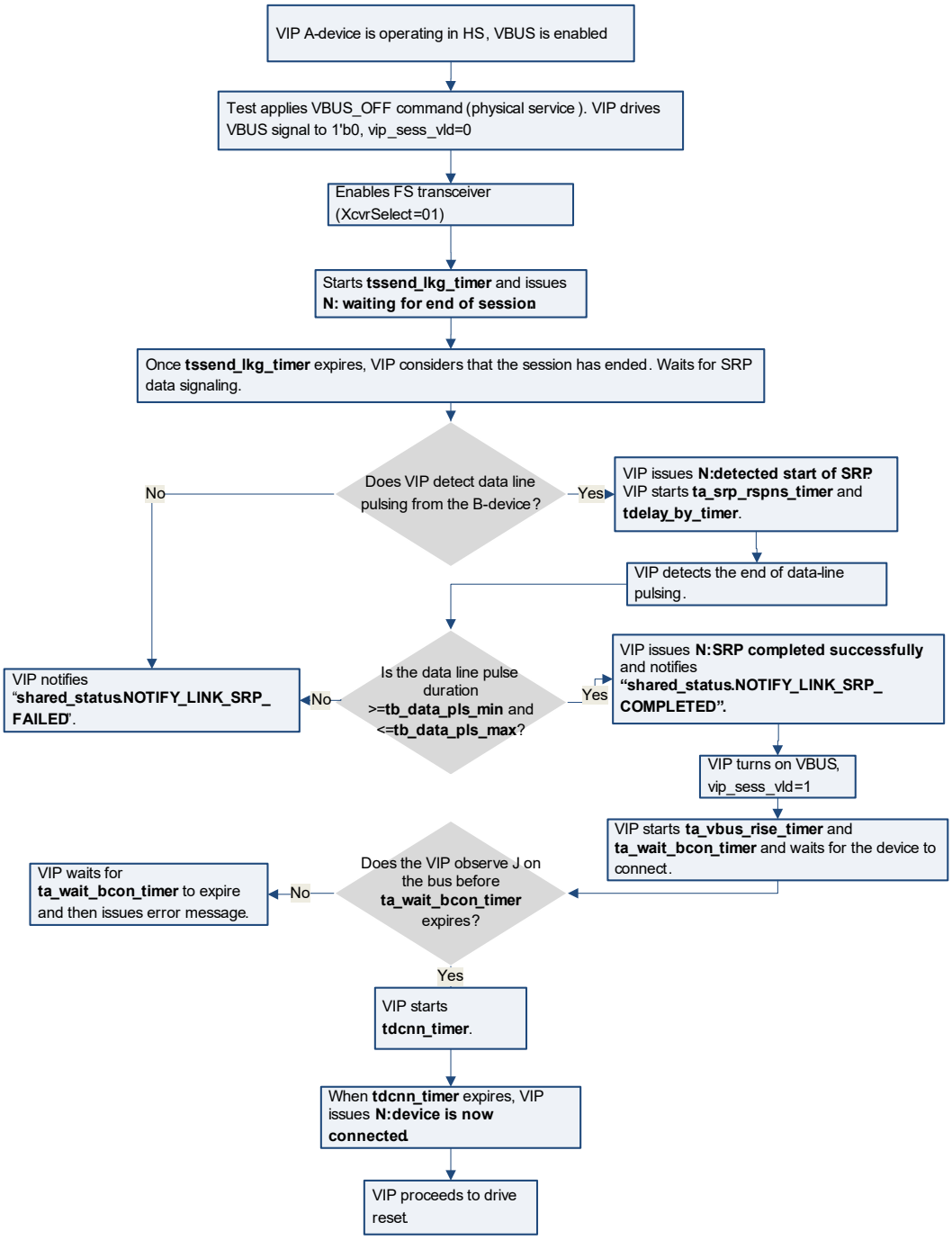


Table 7-11 expands on the abbreviated messages that are used in Figures 7-2 and 7-3.

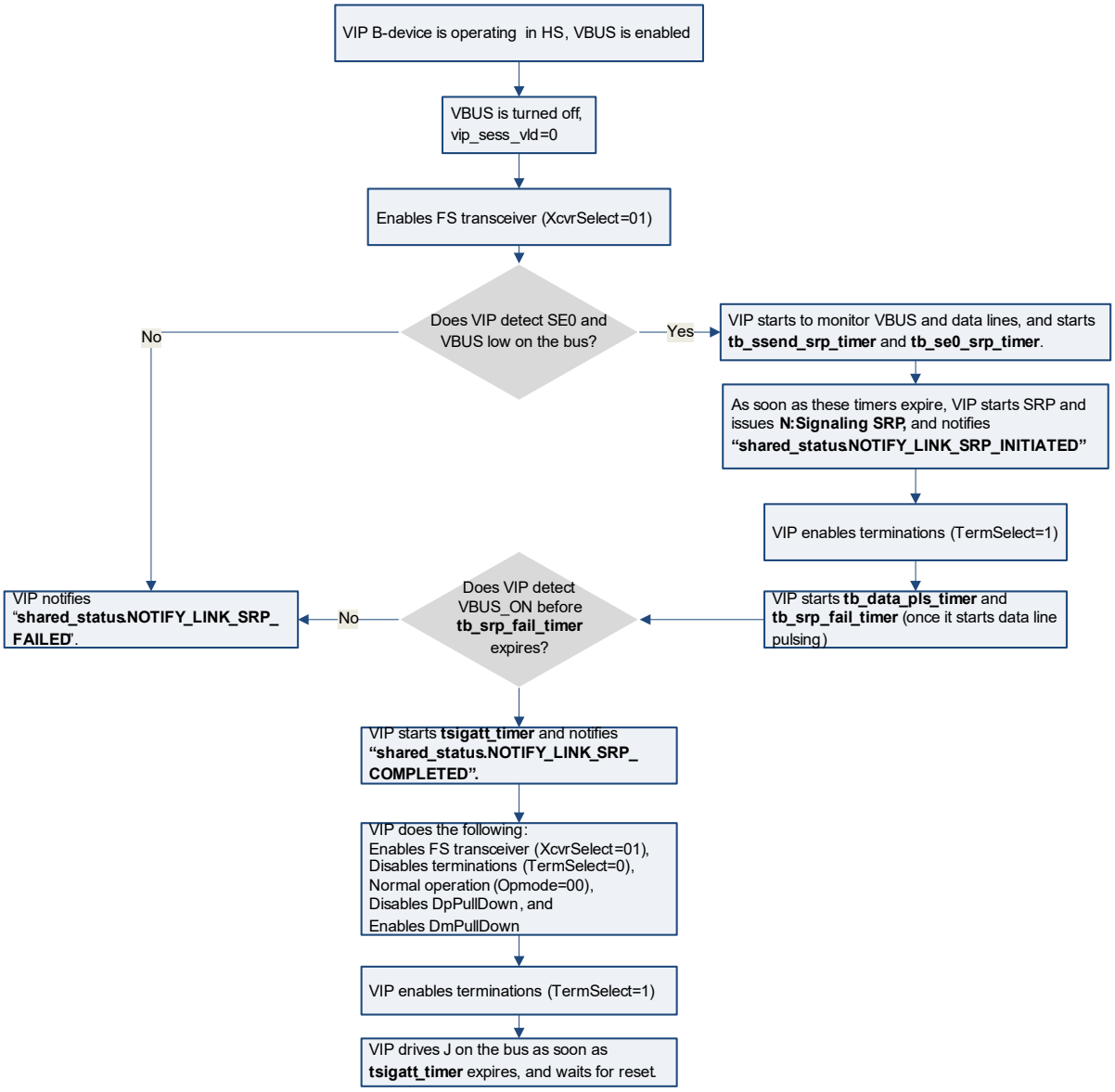
Table 7-11 Legend Explaining the Abbreviated Messages

Abbreviated Messages Used in Figure 7-2	Actual Messages Issued by VIP
N: waiting for end of session	Waiting for the end of a session by monitoring VBUS and data lines.
N: detected start of SRP	A-device detected start of SRP since D+ went high
N: SRP completed successfully	SRP completed successfully since D+ pull-up resistor remained ON for a period within the range specified by TB_DATA_PLS
N: device is now connected	tdcnn timer expired. Device is now connected.
Abbreviated Message Used in Figure 7-3	**Actual Message Issued by VIP**
N: signaling SRP	B-device signaling SRP by generating data line pulsing.

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

Figure 7-3 explains the SRP process flow when the VIP is configured as a B-device. The abbreviated messages listed in Figure 7-3 are expanded in detail in Table 7-11.

Figure 7-3 SRP Protocol Flow when the VIP is Configured as a B-Device



7.15.3 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.

7.15.3.1 Initializing and Storing Configurations for OTG Roles

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

The initial configuration is first stored when the VIP sub-environment is constructed. 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 sub-environment also notifies the testbench whether the role swap was successful or not through one of the following notifications:

- NOTIFY_OTG_ROLE_SWAP_SUCCEEDED or
- NOTIFY_OTG_ROLE_SWAP_FAILED

After successful role swap, the VIP sub-environment and the transactor stack is reconfigured using the reconfigure() command. When svt_usb_subenv::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_subenv::reconfigure() method directly by the user (testbench) are not retained by the subenv in the initial_cfg or swapped_cfg configurations. If a role swap occurs, the configuration used for the new role is the most recent configuration applied 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_subenv::set_otg_cfg() for the inactive role while the current active role is in effect.

7.15.3.2 Role Swapping Process Overview

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

1. Testbench creates two svt_usb_subenv_configuration instances, one each representing the configuration for the initial OTG role (initial_cfg) and the swapped OTG role (swapped_cfg).
2. Testbench constructs (using the new() method) the VIP subenv 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 sub-environment initially has the OTG Peripheral role, the testbench requests a role swap by calling the attempt_otg_role_swap() command.
   - If the role swap succeeds, the sub-environment reconfigures the transactor stack (and itself) using the swapped_cfg configuration data object (which represents the configuration of the VIP while in the OTG Host role), and notifies the Testbench through the NOTIFY_OTG_ROLE_SWAP_SUCCEEDED notification.

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

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

7.15.3.3 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.

7.15.3.3.1 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 subenv attempt_otg_role_swap() method, which will enable the HNP procedure.
- When GetStatus() is used to interrogate the value of the DUT's "Host request flag" status bit, if the flag is set a protocol service data object should be created by the testbench and sent to the VIP to initiate a role swap attempt.
- It is also up to the testbench to control the timing of the GetStatus() control transfers to adhere to the timing requirements specified for HNP Polling.

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

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

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

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

Figure 7-4 explains the HNP flow when the VIP is configured as an A-device. Figure 7-5 explains the role reversion sequence of events when the A-device reverts back to acting as a host.

Figure 7-4 HNP Flow When VIP is Configured as an A-Device

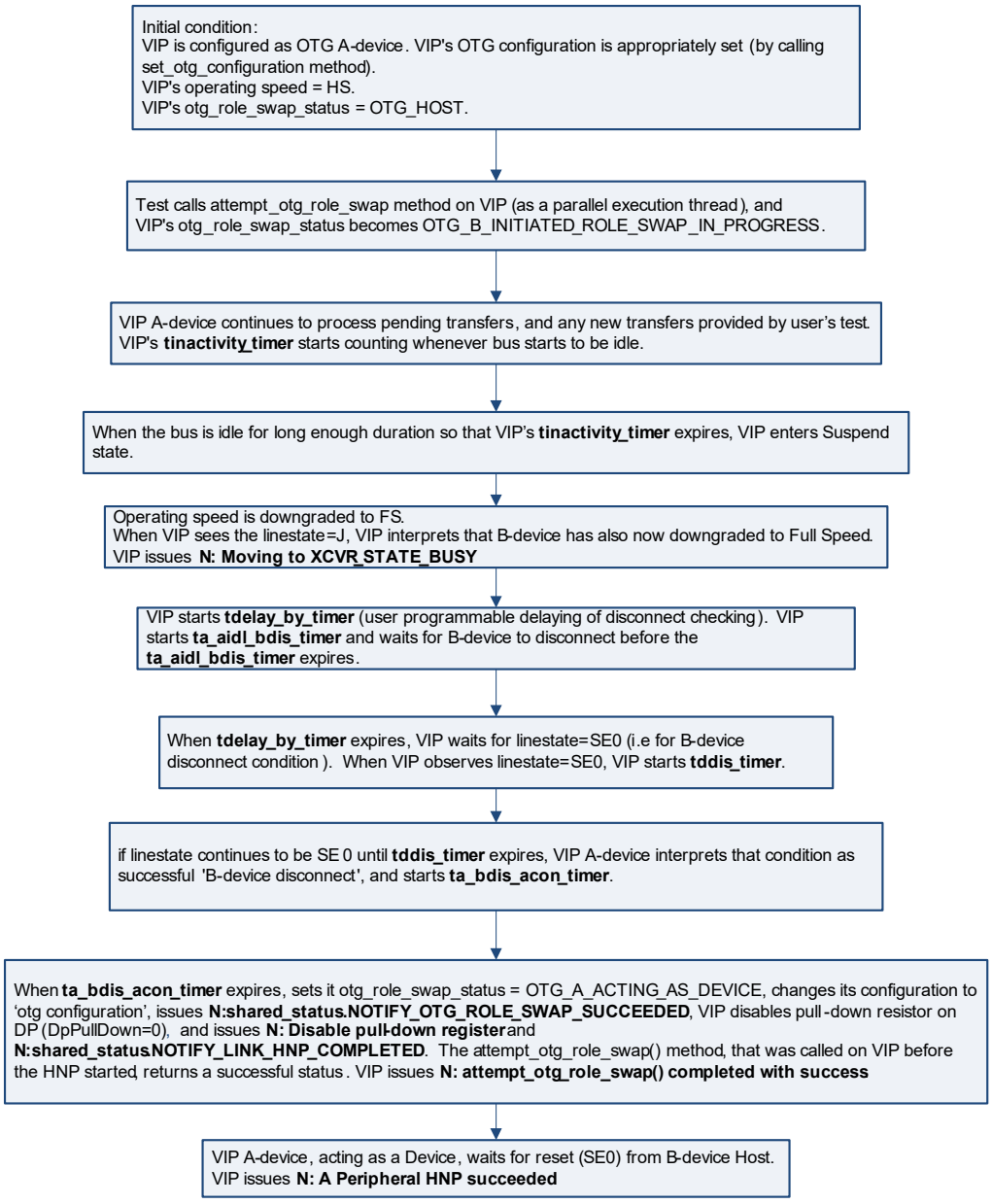


Figure 7-5 Reverse HNP Flow when the VIP A-Device Reverts Back to Acting as a Host

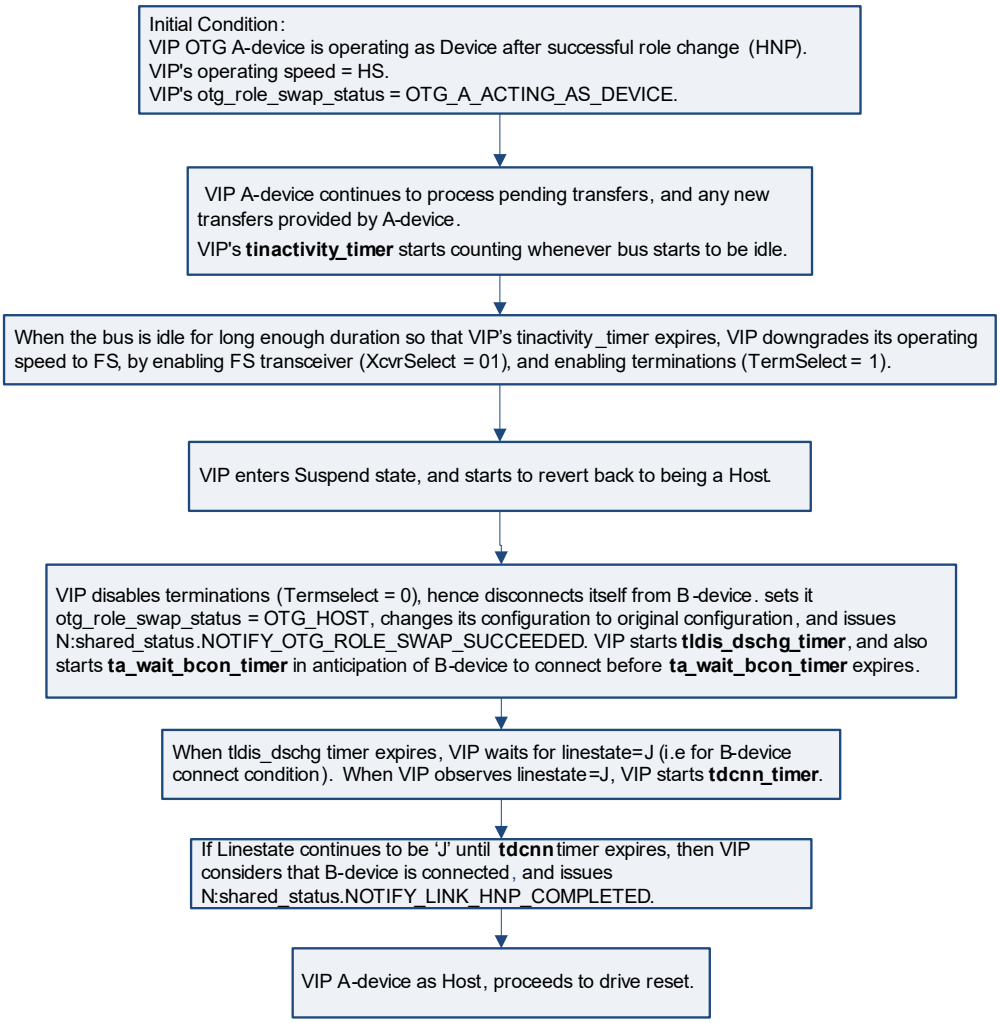


Table 7-12 expands on the abbreviated messages that are used in Figures 7-4 and 7-5.

Table 7-12 Legend Explaining the Abbreviated Messages Used in Figures 7-4 and 7-5

Abbreviated Messages Used in Figure 7-4	**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 7-5	**Actual Message Issued by VIP**
N: Enable FS transceiver	Enable FS transceiver (XcvrSelect = 01)
N: Enable terminations	Enable terminations (TermSelect = 1)

Table 7-12 Legend Explaining the Abbreviated Messages Used in Figures 7-4 and 7-5

Abbreviated Messages Used in Figure 7-4	Actual Messages Issued by VIP
N: Disable terminations	Disable terminations (TermSelect = 0)
N: Reverse HNP Complete	Reverse HNP Complete on A-Device

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

Figure 7-6 explains the HNP flow when the VIP is configured as a B-device.

Figure 7-6 HNP Flow When VIP is Configured as a B-Device

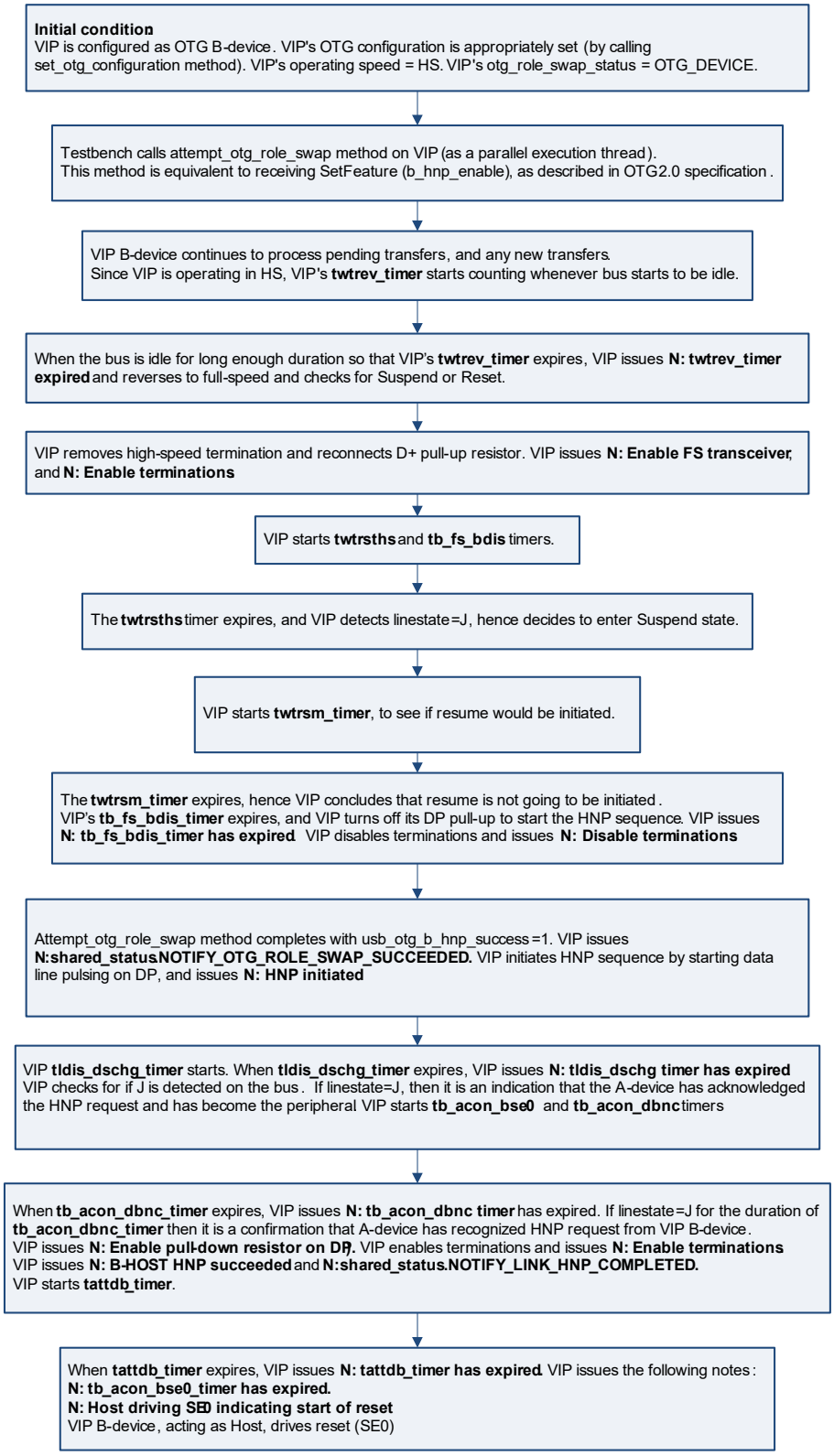


Figure 7-7 Reverse HNP Flow When the VIP B-Device Reverts Back to Acting as a Device

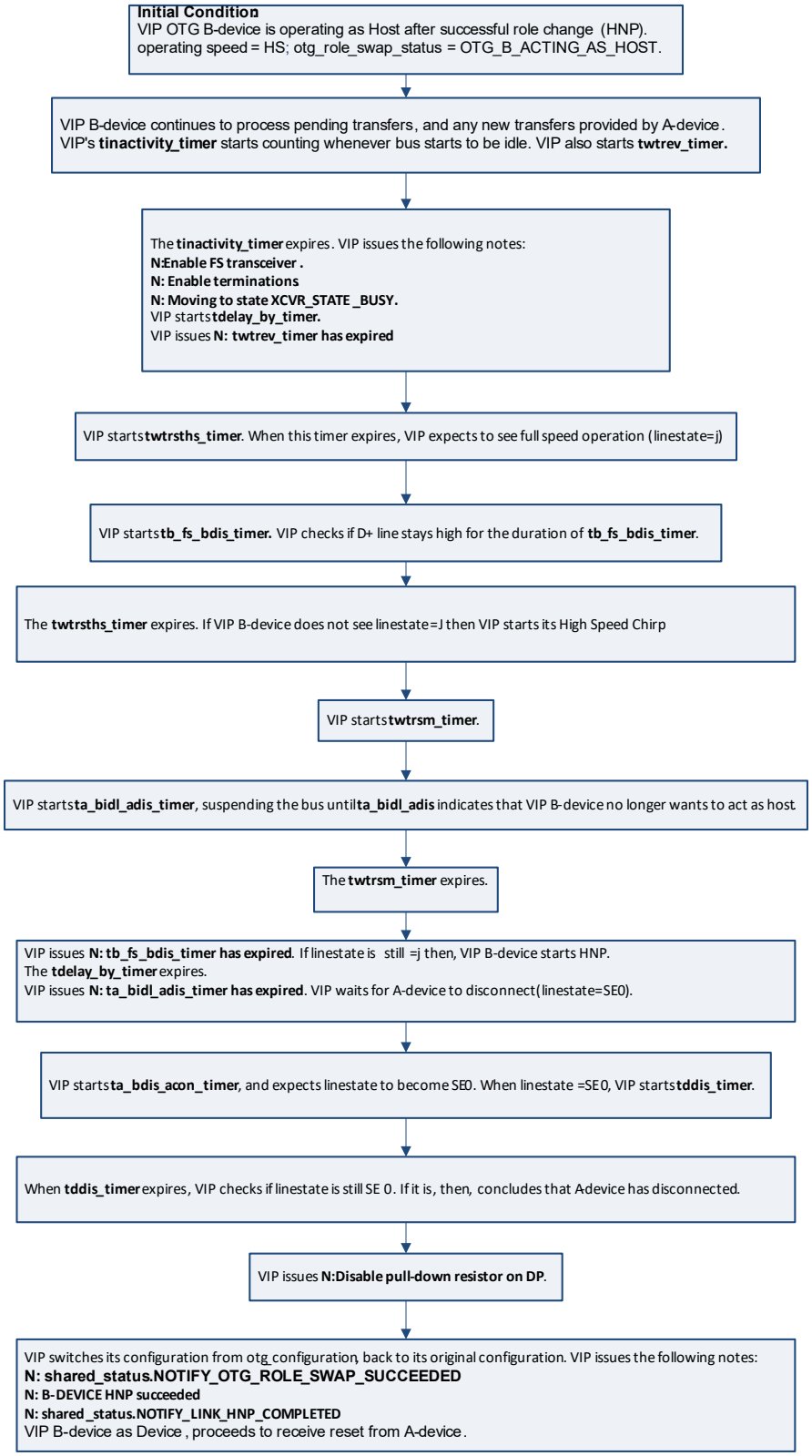


Table 7-13 expands on the abbreviated messages that are used in Figures 7-6 and 7-7.

Table 7-13 Legend Explaining the Abbreviated Messages Used in Figures 7-6 and 7-7

Abbreviated Messages Used in Figure 7-6	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 7-7	**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

7.15.4 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

7.15.4.1 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)

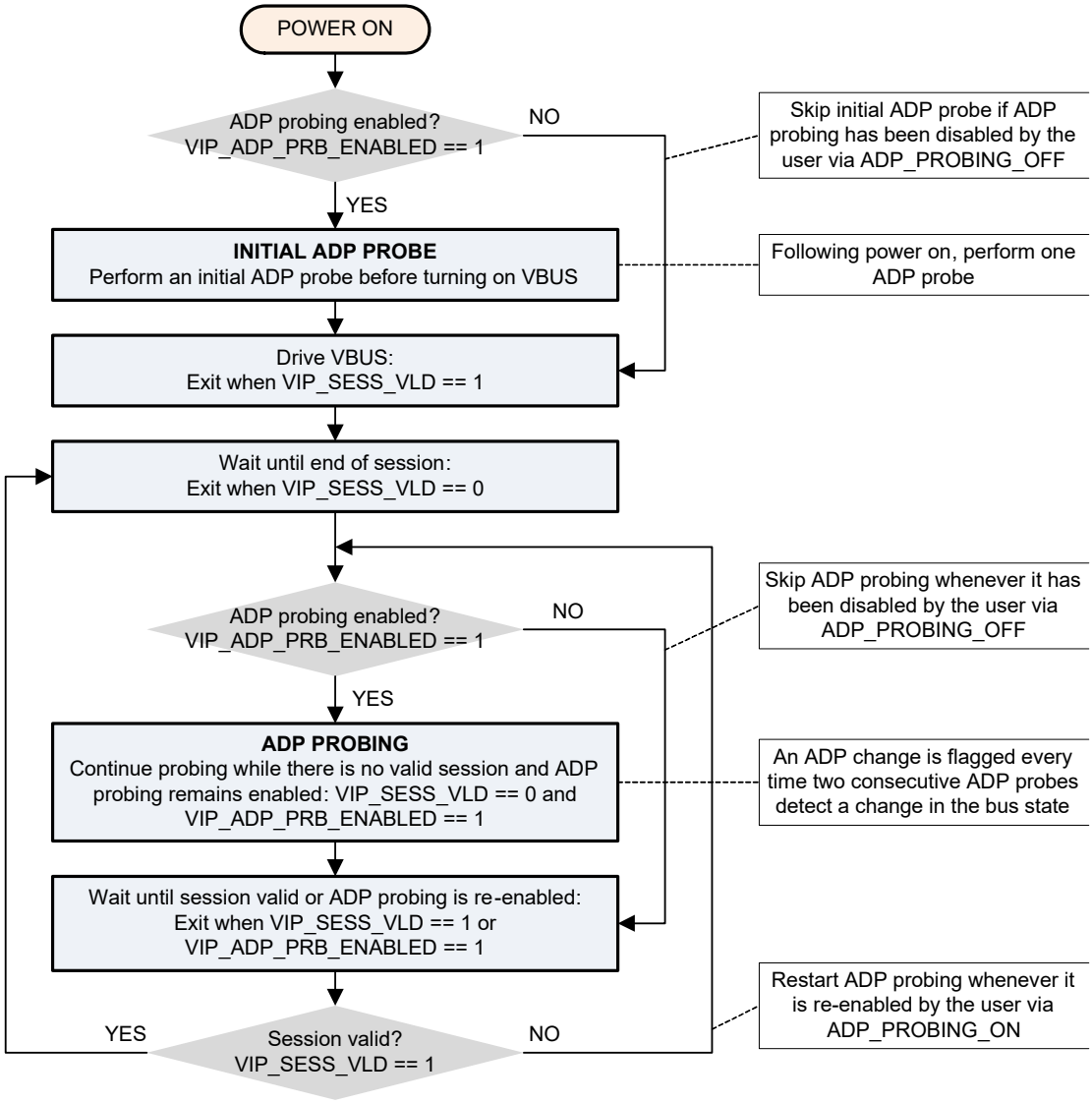
7.15.4.1.1 ADP Probing When VIP is Configured as an A-Device

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 7-8 illustrates a simple ADP Probing process for VIP as an A-device or an Embedded Host using a waveform.

Figure 7-8 ADP Probing Flow when VIP is Configured as an A-Device or Embedded Host



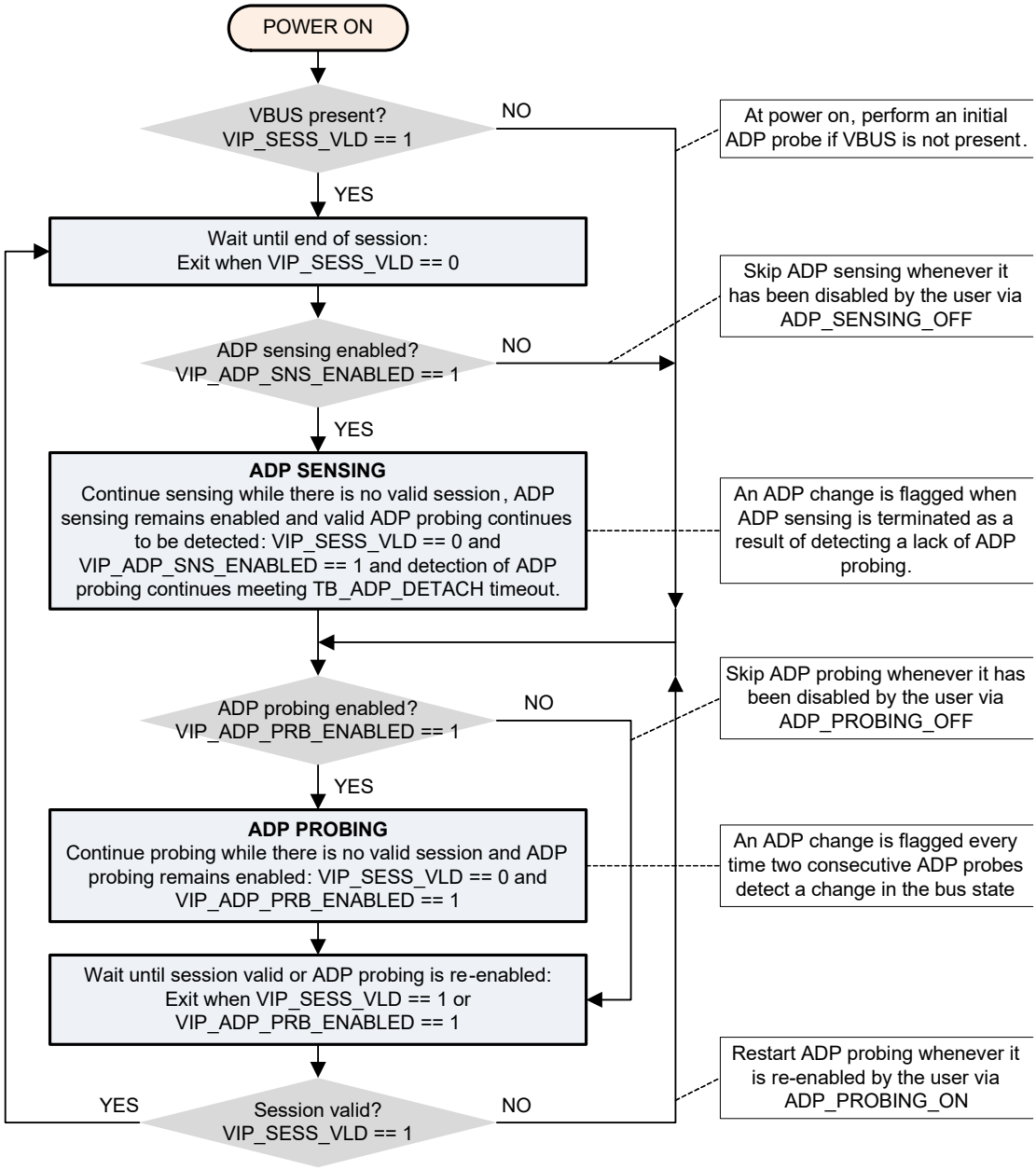
7.15.4.1.2 ADP Probing When VIP is Configured as a B-Device

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 7-9 explains the ADP control when VIP is configured as a B-device.

Figure 7-9 ADP Probing and Sensing Flow when the VIP is Configured as a B-Device



7.15.4.2 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

7.15.4.2.1 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 7-9.

7.15.4.3 ADP Notifications

VIP issues the following notifications to indicate the success or failure of an ADP probing or sensing operation:

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

7.15.4.4 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.

7.16 HSIC Overview

The this section covers HSIC Support.

7.16.1 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.

7.16.2 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)

7.16.3 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 7-14 HSIC Configuration Parameters

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

Table 7-14 HSIC Configuration Parameters (Continued)

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

7.16.4 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.

7.16.4.1 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.

7.16.4.2 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.

7.16.5 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.

7.16.5.1 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).

7.16.5.2 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.

7.16.6 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.

7.16.7 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 7-15 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.

7.16.8 Channels

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

7.16.9 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.

7.16.10 Notifications

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

Table 7-16

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

7.16.11 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.

7.16.12 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.

7.16.13 Usage

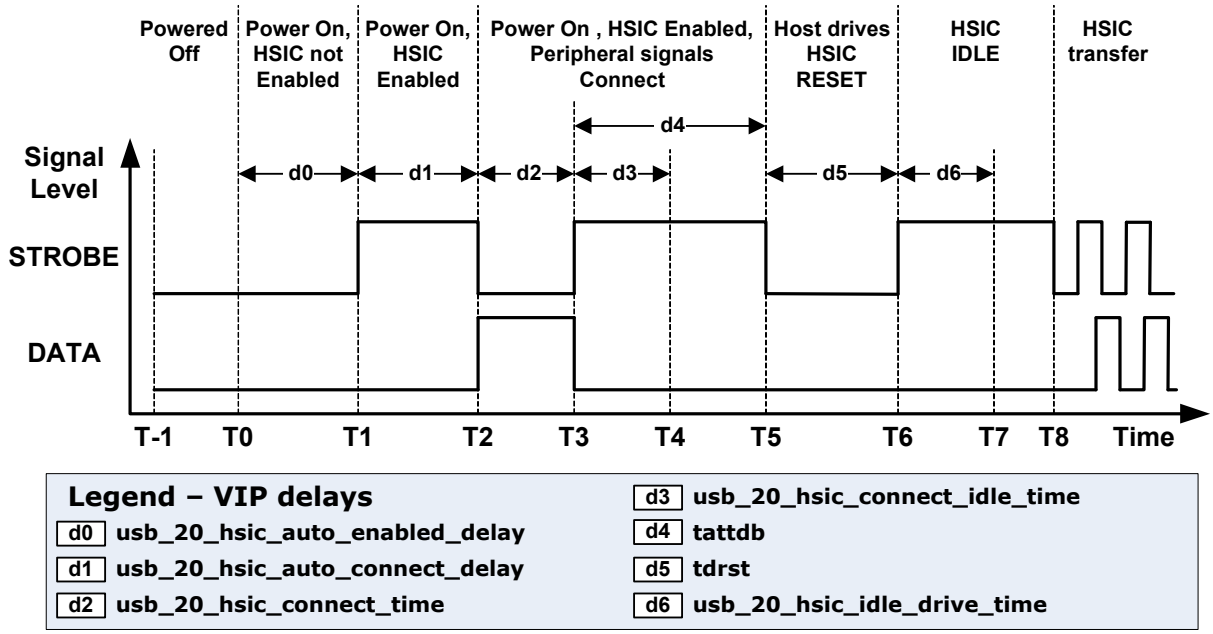
Following is a discussion of the HSIC-specific usage scenarios and how users may control and/or interact with them. The HSIC-specific usage scenarios are:

- 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.

7.16.13.1 Discovery Process

Discovery is the HSIC power up and connection sequence used to establish communication between two HSIC devices. Figure 7-10 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 7-10 Discovery Process with VIP Set Values



7.16.13.2 VIP as Host

Using Figure 7-10 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).

T1 - Powered Off state	
Action	In this state, the VIP's HSIC interface provides weak pull-downs to prevent the Strobe and Data signals from floating.
Exit	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.

7.16.13.3 VIP as Peripheral

Using Figure 7-10as 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.
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.
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.

7.16.13.4 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).

7.16.13.5 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.

7.16.13.6 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 7-11 and Figure 7-12. 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 7-11 End of signaling scenarios with DATA low

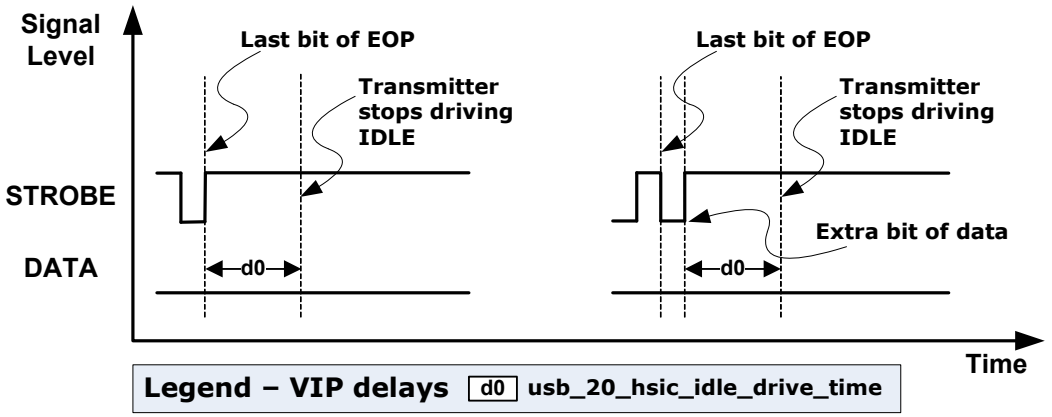
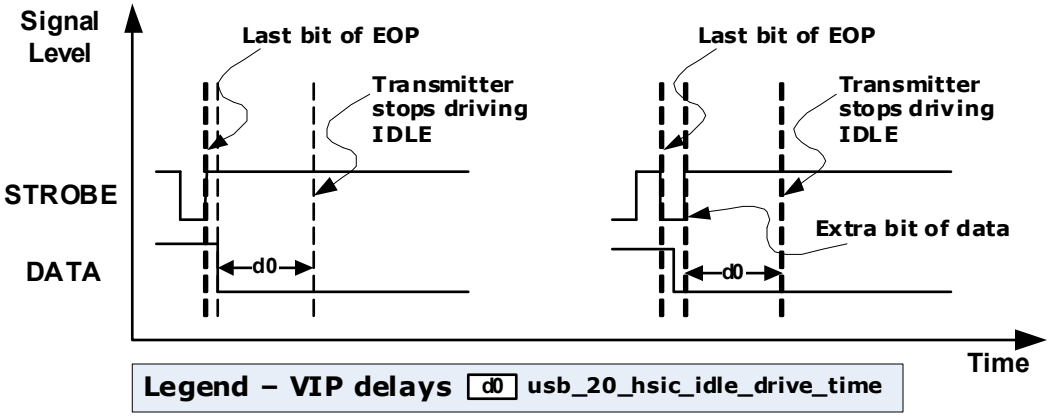
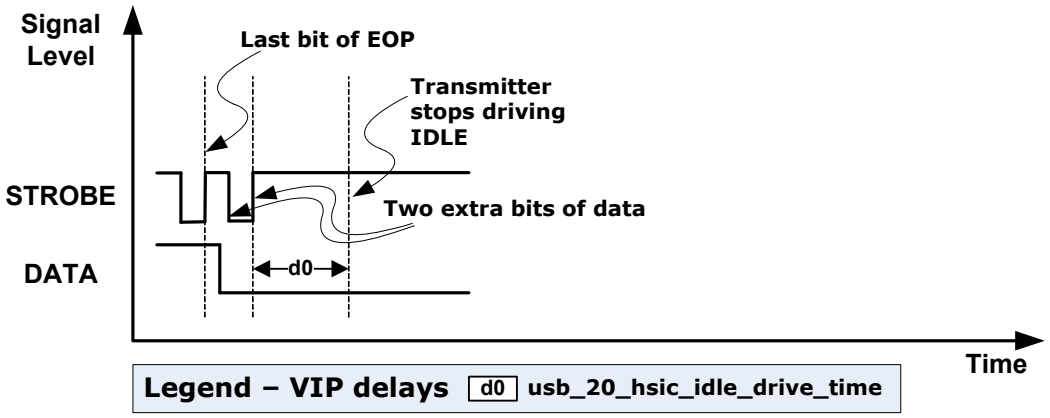


Figure 7-12 End of signaling scenarios with DATA high (fast EOP)



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 7-13.

Figure 7-13 End of signaling scenarios with DATA high (fast EOP disabled)



7.16.13.7 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.

7.16.13.8 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.

7.16.13.9 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.

7.16.13.10 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.

7.17 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.

7.17.1 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.

7.17.1.1 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.

7.17.1.2 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

7.17.2 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.

7.17.2.1 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.

7.17.2.2 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.

7.17.3 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.”

7.17.3.1 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.

7.17.3.2 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

7.17.4 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.”

7.17.4.1 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.

7.17.4.2 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

7.17.5 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.”

7.17.5.1 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.

7.17.5.2 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

7.17.6 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.

7.17.7 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.

7.17.8 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.

7.18 SuperSpeed InterChip Physical (SSIC) Usage

The USB Verification IP supports SSIC RMMI interfaces connected to an M-Phy supporting an RMMI interface. The following sections show general usage of the SSIC Physical transactor.

The **svt_usb_subenv::new()** constructor takes a **svt_usb_subenv_configuration** object as the **cfg** argument and a svt_usb_configuration object as the **remote_cfg** argument. These parameters represent

- The **cfg** object represents the VIP's view of its configuration with respect to its local side of the connection.
- The **remote_cfg** object represents the VIP's view of the remote Phy's configuration.

Note, the subsections focus on configuring parameter values **cfg** and r**emote_cfg** at instantiation time for several connection scenarios.

7.18.1 Configuring the VIP for a DUT Without a SSIC Physical Layer

If the VIP USB Host/Device is being used to test a DUT which does not contain an SSIC Phy (i.e. if the DUT's connection is from the MAC perspective of an SSIC MAC/Phy RMMI interface) then the VIP requires two svt_usb_subenv_configuration objects: **cfg** and **remote_cfg**. In this case, the VIP can be thought of as containing two *virtual* Physical layers (one local, and the other remote):

7.18.1.1 cfg Parameter Settings

The following settings would be typical for the cfg parameter discussed previously:

- **ssic_profile** = <desired profile to use>
- **ssic_pairs** = must equal the number of pairs required by **ssic_profile**.
- **bottom_xactor** = **SUB_PHYSICAL**
- **top_xactor** >= **PROTOCOL** (assuming VIP functionality up through protocol layer *transfers* is desired)
- **component_subtype** = **MAC**
- **usb_ss_signal_interface** = **USB_SSIC_CHANNEL**
- **usb_mphy_rx_cfg[$]** and **usb_mphy_tx_cfg[$]** have settings of:
  - Must size to have **cfg.ssic_pairs** entries in each array
  - **mphy_component_type** == **svt_mphy_configuration::PHY**
  - **mphy_interface_type** == **svt_mphy_configuration::RMMI_REMOTE**
  - **mphy_tlm_if_type** == **svt_mphy_configuration::TLM_LOCAL**
  - For **mphy_if**
    - Rx (need one per lane)

      Assign **mphy_rmmi_mrx_dut_controller_if** to the **remote_cfg** Tx M-Phy configuration's **rmmi_mtx_dut_controller_if**
    - Tx (need one per lane)

      Assign **mphy_rmmi_mtx_dut_controller_if** to the **remote_cfg** Rx M-Phy configuration's **rmmi_mrx_dut_controller_if**

7.18.1.2 remote_cfg Parameter Settings

The following settings would be typical for the cfg parameter discussed previously:

- **ssic_profile** = Must match the value in **cfg**
- **ssic_pairs** = Must match the value in **cfg**
- **component_subtype** = **PHY**
- **usb_ss_signal_interface** = **SSIC_RMMI_IF**
- **usb_mphy_rx_cfg[$]** and **usb_mphy_tx_cfg[$]** have settings of:
  - Must size to have **cfg.ssic_pairs** entries in each array
  - **mphy_interface_type** == **svt_mphy_configuration::RMMI_REMOTE**
  - **mphy_tlm_if_type** == **svt_mphy_configuration::TLM_REMOTE**
  - **mphy_component_type** == **PHY**
  - **mphy_if** == **<svt_mphy_if>** what is instanced in HDL
    - Rx (need one per lane)

      Assign **mphy_rmmi_mrx_dut_controller_if** to the instance of the **svt_mphy_rmmi_mrx_dut_controller_if** interface type associated with this lane (as instantiated in the testbench).
    - Tx (need one per lane)

      Assign **mphy_rmmi_mtx_dut_controller_if** to the instance of the **svt_mphy_rmmi_mtx_dut_controller_if** interface type associated with this lane (as instantiated in the testbench).

7.18.2 Configuring the VIP for a DUT With a SSIC Physical Layer

In this section, configuration options are shown for a DUT with a SSIC Physical Layer.

If the VIP USB Host/Device is being used to test a DUT which contains an SSIC Phy if the DUT's connection is directly to the SSIC Serial interface), then the VIP requires one svt_usb_subenv_configuration object cfg (and no remote_cfg). In this case, the VIP can be thought of as containing one virtual Phy (its local phy). The cfg object represents the VIP's view of its configuration with respect to its local side of the connection, while the remote_cfg object represents the VIP's view of the remote Phy's configuration.

7.18.2.1 cfg Parameter Settings

- **ssic_profile** = <desired profile to use>
- **ssic_pairs** = must equal the number of pairs required by **ssic_profile**.
- **bottom_xactor** (VMM) **/ bottom_layer** (VMM) = **SUB_PHYSICAL**
- **top_xactor** (VMM) **/ top_layer** (VMM) >= **PROTOCOL** (assuming VIP functionality up through protocol layer *transfers* is desired)
- **component_subtype** = **MAC**
- **usb_ss_signal_interface** = **SSIC_SERIAL_IF**
- **usb_mphy_rx_cfg[$]** and **usb_mphy_tx_cfg[$]** have settings of:
  - Must size to have **cfg.ssic_pairs** entries in each array
  - **mphy_component_type** == **svt_mphy_configuration::PHY**

- **mphy_interface_type == svt_mphy_configuration::SERIAL**
- **mphy_tlm_if_type == svt_mphy_configuration::TLM_NONE**
- Rx (need one per lane)

  Assign **mphy_serial_rx_if** to the instance of the **svt_mphy_serial_rx_if** type associated with this lane (as instantiated in the testbench).
- Tx (need one per lane)

  Assign **mphy_serial_tx_if** to the instance of the **svt_mphy_serial_tx_if** type associated with this lane (as instantiated in the testbench).

7.18.2.2 remote_cfg Parameter Settings

For this type of connection (an SSIC Serial connection), the parameter remote_cfg remains null.

8 Verification Topologies

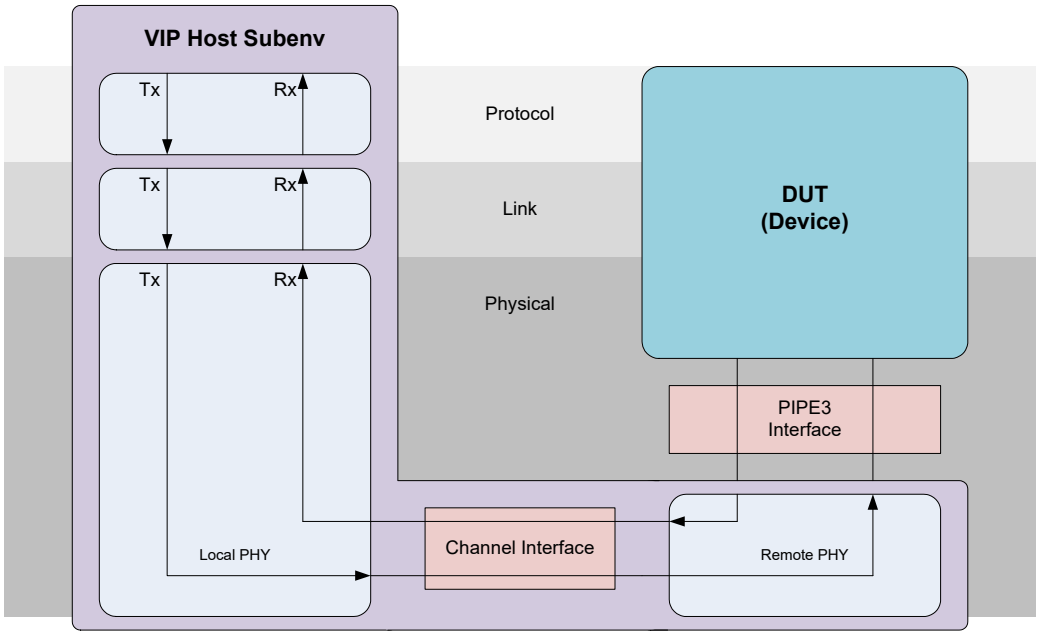
This chapter presents several sub-environment testbench topologies. In addition to a brief description of the testbench topologies, each section lists configuration parameters required to instantiate the VIP sub-environment.

8.1 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_subenv_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 8-1 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_CHANNEL;
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**

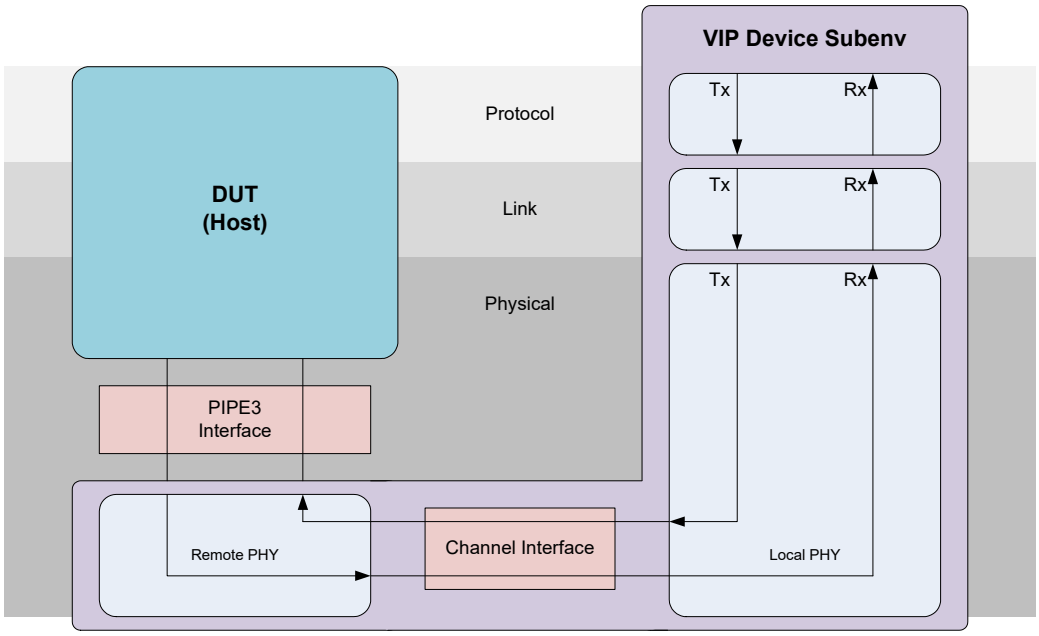
```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS;
component_subtype_enum component_subtype = PHY;
usb_ss_signal_interface_enum usb_ss_signal_interface = PIPE_IF;
usb_capability_enum usb_capability = USB_SS_ONLY;
```

8.2 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_subenv_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 8-2 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_CHANNEL;
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)**

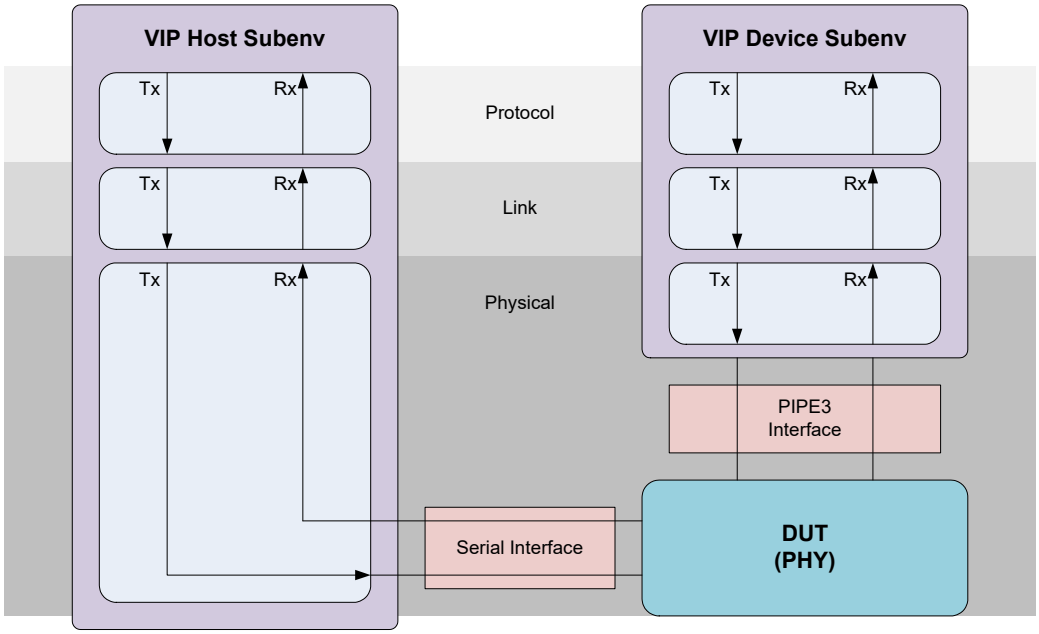
```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::SS;
component_subtype_enum component_subtype = PHY;
usb_ss_signal_interface_enum usb_ss_signal_interface = PIPE3_IF;
usb_capability_enum usb_capability = USB_SS_ONLY;
```

8.3 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_subenv_configuration.

Figure 8-3 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_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)
```

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

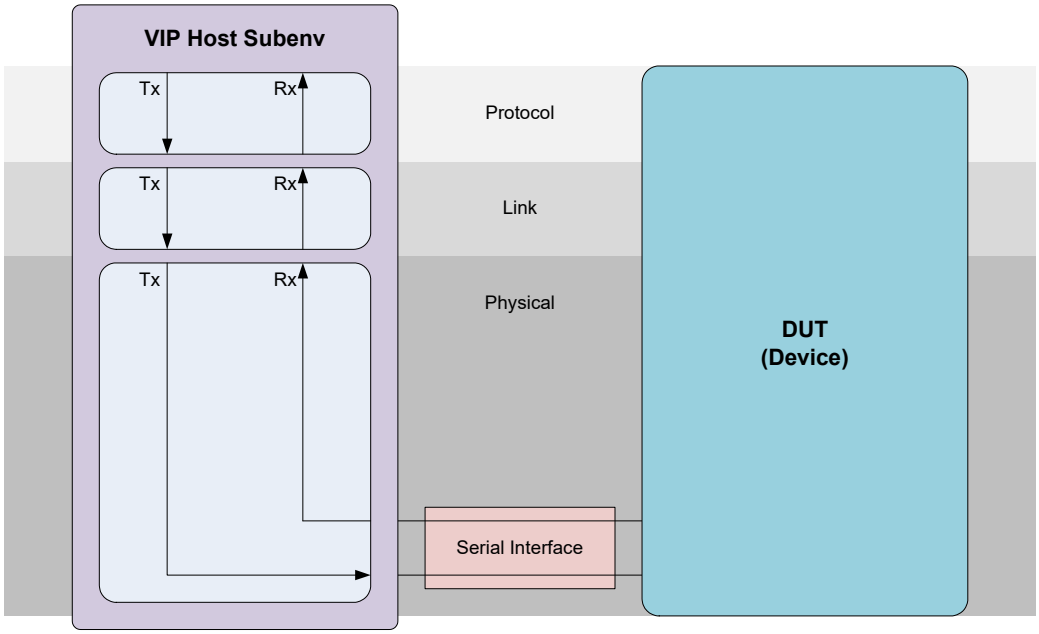
```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = PIPE_IF;
usb_capability_enum usb_capability = USB_SS_ONLY;
svt_usb_device_configuration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

8.4 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_subenv_configuration. This section provides configuration objects for SS and 2.0 serial interfaces.

Figure 8-4 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**

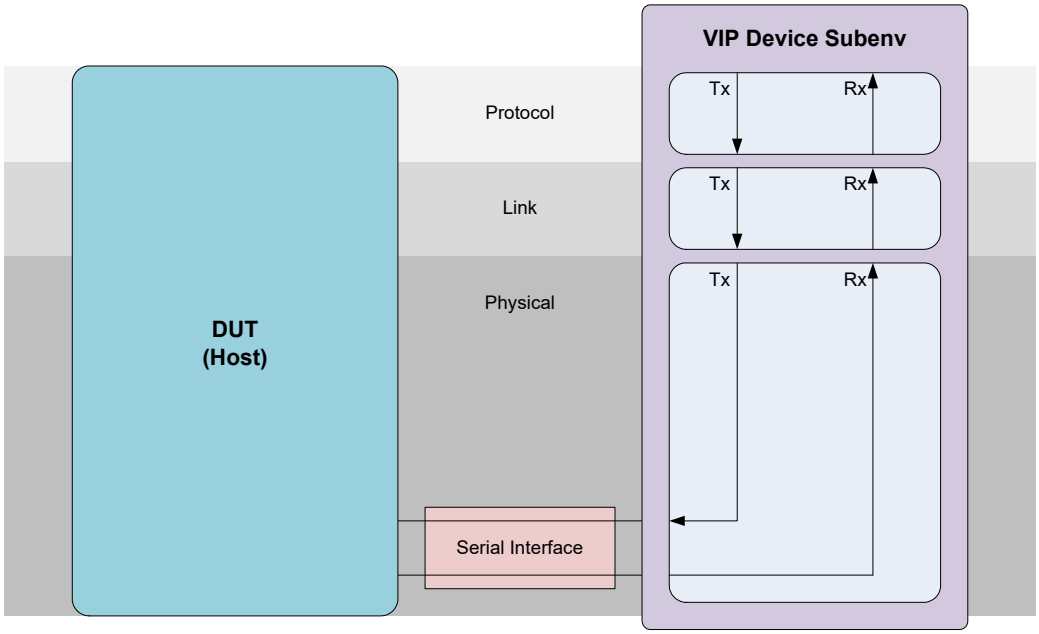
```
svt_usb_types::component_type_enum component_type = svt_usb_types::HOST;
svt_usb_types::speed_enum speed = svt_usb_types::HS;  (other speeds FS/LS)
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_20_SERIAL_IF;
usb_capability_enum usb_capability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

8.5 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_subenv_configuration. This section provides configuration objects for SS and 2.0 serial interfaces.

Figure 8-5 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**

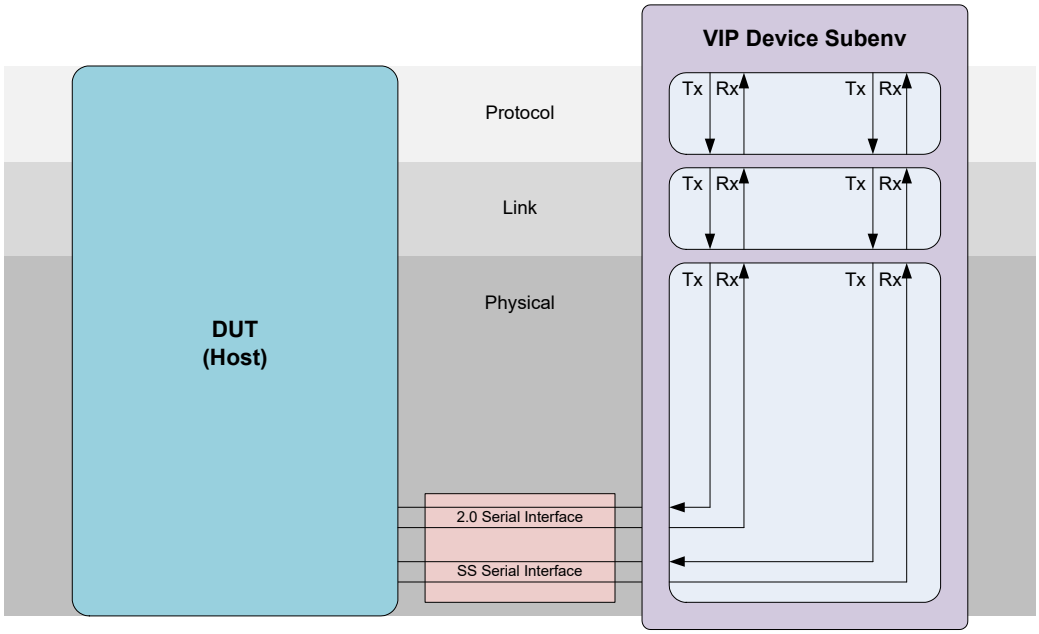
```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::HS;  (other speeds FS/LS)
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_20_SERIAL_IF;
usb_capability_enum usb_capability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

8.6 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_subenv_configuration.

Figure 8-6 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**

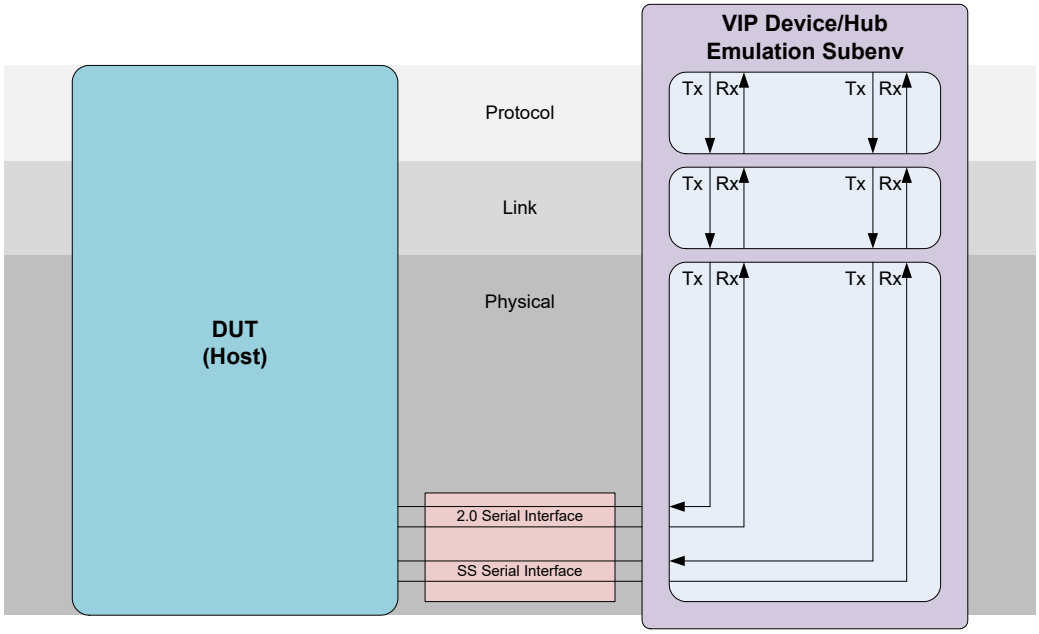
```
svt_usb_types::component_type_enum component_type = svt_usb_types::DEVICE;
svt_usb_types::speed_enum speed = svt_usb_types::SS;
component_subtype_enum component_subtype = MAC
usb_ss_signal_interface_enum usb_ss_signal_interface = USB_SS_SERIAL_IF;
usb_ss_signal_interface_enum usb_20_signal_interface = 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)
```

8.7 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_subenv_configuration.

Figure 8-7 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;
```

9 VIP Tools

9.1 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.

9.1.1 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 XML files.

For Example:

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

Similarly for USB receiver:

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

9.1.2 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 and select the Protocol Analyzer option in the main window to invoke Protocol Analyzer.

Interactive Mode

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

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

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

9.1.3 Documentation

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

VERDI_HOME/doc/Verdi_Transaction_and_Protocol_Debug.pdf

9.1.4 Limitations

Interactive support is available only for VCS.

10 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:

10.1 Using Trace Files for Debugging

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

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

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

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

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

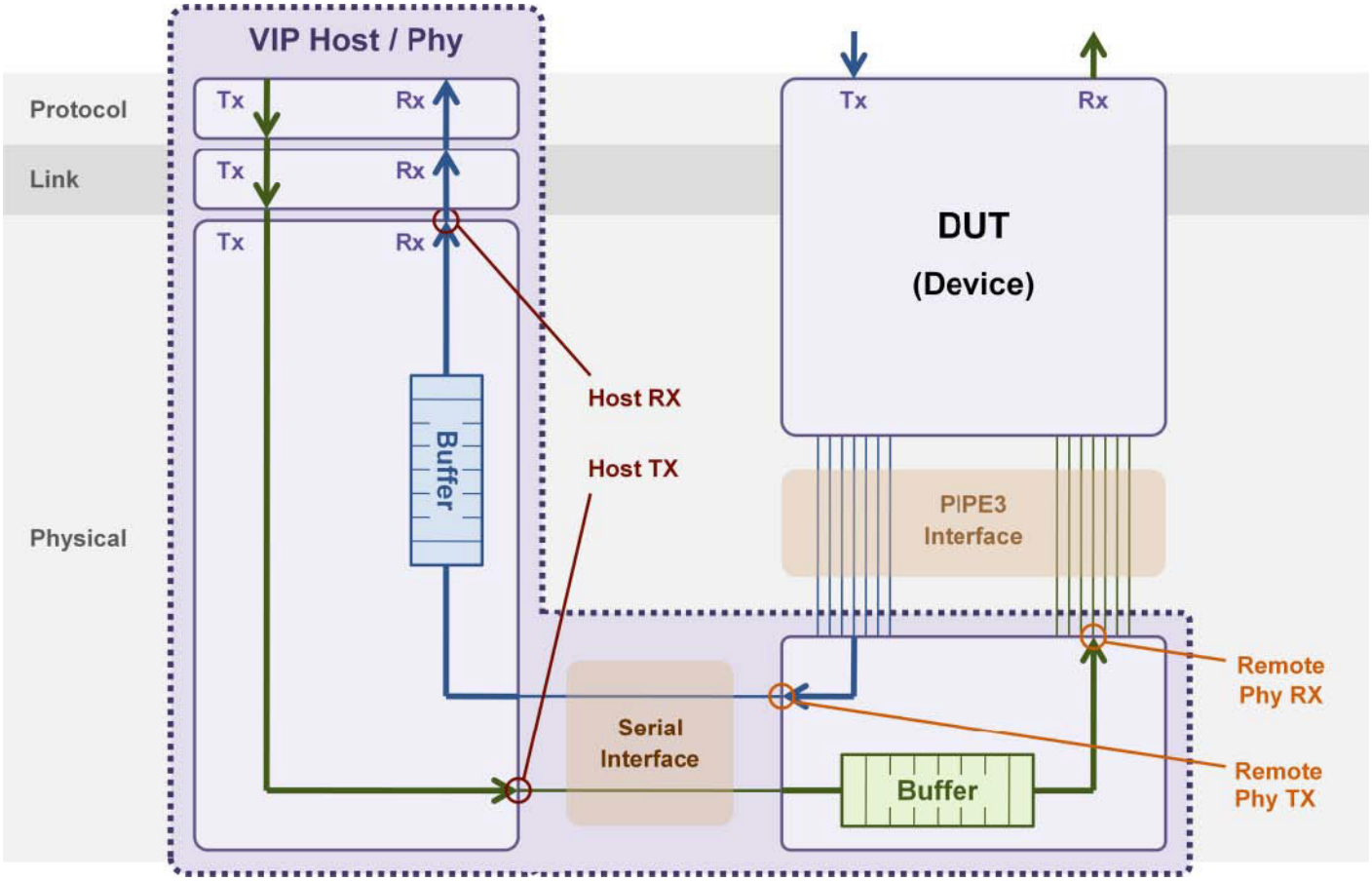
```
vip_subenv.prot.SS.transfer_trace
```

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

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

Figure 10-1 shows the output channels of each transactor, where objects are captured for corresponding trace files content.

Figure 10-1 An Environment with a Device DUT Connected to a Host VIP and Remote PHY



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

Data Trace Files

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

where:

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

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

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

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

Packet Trace Files

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

where:

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

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

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

Transaction Trace Files

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

where:

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

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

Transfer Trace Files

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

where:

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

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

For example, if the instance name of the VIP sub-environment is “vip_subenv”. then the simulation of this environment produces the files listed in Table 10-1:

Table 10-1 Generated Trace Files

Trace File Type	Trace File Name

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

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

10.2 Enabling Tracing

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

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

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

Code Sample 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_subenv_instrance>.prot.SS.TX.transfer_trace and <vip_subenv_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-VMM test examples provided along with the VIP.

10.3 Setting Verbosity Levels

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

To set the verbosity level in the testbench, use the VMM-specified log-levels in the code as shown in the following code snippet:

```
/**
     * Here the verbosity level, of vip_usb_host instance, is set to the DEBUG_SEV.
     */
    vip_usb_host.log.set_verbosity (vmm_log::DEBUG_SEV);

    /**
     * Below code sets the verbosity to all the sub-instances of vip_usb_host,
recursively
     * down the hierarchy indicated by the last argument ('1')
     */
```

```
vip_usb_host.log.set_verbosity (vmm_log::DEBUG_SEV, "/./", "/./", 1);
```

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

10.3.1 Method 1: To enable the specified severity in the VIP, DUT, and testbench

Use Example: VCS

```
vcs <other run time options> +vmm_log_default=DEBUG
```

10.3.2 Method 2: To enable the specified severity to specific sub-classes of VIP

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

Use Example: VCS

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

Use Example: simv

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

10.3.2.1 Valid VIP Sub-Units

Table 10-2 lists the valid VIP sub-units that you can use.

Table 10-2 Valid VIP Sub-Units

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

Table 10-2 Valid VIP Sub-Units

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

10.4 Elevating or Demoting Messages

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

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

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

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

Sometimes, for example in tests with error injection, you may want to demote specific VIP messages in order to continue simulation without immediate stoppage. The following code snippet illustrates how you can demote an error message reported by VIP to a warning type message, allowing the simulation to continue without termination.

```
/**
     * Demonstrate 'how to demote the message from an ERROR severity to a WARNING
severity':
     * Error messages can only be demoted into warning messages.
     * Before demoting a message you need to check severity level of the message.
     * The following demoted message is an ERROR message issued, if the Device
     * sees invalid token packet size.
     * You can demote a specific error message by replacing "Received token packet size
is invalid"
     * text with a sub-string from other error messages.
     */
    vip_usb_device.log.modify(,,,,vmm_log::ERROR_SEV,
                              "/Received token packet size is invalid/",,
                              vmm_log::WARNING_SEV, vmm_log::CONTINUE
                             );
```

10.5 Disabling Specific In-line Checking

By default, all VIP error checking is enabled by default. For valid checks defines, see the chk_cov_mgr in the Class Reference HTML.

The following code snippet illustrates the enabled default VIP error checking:

```
svt_err_check_stats temp_check;
temp_check = usb_host.link.chk_cov_mgr.find("skp_symbol_ratio_check");
temp_check.set_is_enabled(0);
```

A
Reporting Problems

A.1 Introduction

This chapter outlines the process for working through and reporting 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.2 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.3 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
```

Note

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.4 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.5 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.5.1 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.5.2 Questa

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

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

A.5.3 Incisive

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

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

A.6 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" on page 253.

A.7 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.8 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.