Verification Continuum™

VC Verification IP
PCIe
OVM User Guide

Version S-2021.06, June 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 https://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 OVM users of the VC PCIe, and is for design or verification engineers who want to verify PCIe operation using a OVM testbench written in SystemVerilog. Readers are assumed to be familiar with PCIe, object oriented programming (OOP), SystemVerilog, and Open Verification Methodology (OVM) techniques.

Manual Organization

The chapters of this databook are organized as follows:

- Chapter 1, "Introduction", introduces the VC PCIe and its features.
- Chapter 2, "Installation and Setup", describes system requirements and provides instructions on how to install, configure, and begin using the VC PCIe.
- Chapter 3, "Creating the PLI Object in 32-bit and 64-bit Simulation Modes", describes how to build and compile the message log and PLI files.
- Chapter 4, "General Concepts", introduces the PCIe VIP within the OVM environment and describes the data objects and components that comprise the VIP.
- Chapter 5, "Verification Features", describes the verification features supported by PCIe VIP such as, Verification Planner and Protocol Analyzer.
- Chapter 6, "PCIe Verification Topologies", describes various ways the PCIe VIP can be connected with other components.
- Chapter 7, "Using the PCIe Verification IP", shows how to install and run a getting started example.
- Chapter 8, "PCIe Device Agent", provides an overview on PCIe Device Agent and how to use them.
- Chapter 9, "PCIe Agent", provides an overview on PCIe Agent and how to use them.
- Chapter 10, "Using the Transaction Layer", describes features of the Transaction Layer and how to use them.
- Chapter 11, "Data Link Layer Features and Classes", describes features of the Data Link Layer and how to use them.
- Chapter 12, "PHY Layer Features and Classes", describes features of the PHY Link Layer and how to use them.

- Chapter 13, "Using the Driver Application", describes the procedure for using the driver application.
- Chapter 14, "Functional Coverage", describes how to enable and use the functional coverage features.
- Chapter 15, "M-PHY Adapter Layer", describes the PCIe SVT implementation of the M-PHY adapter.
- Chapter 16, "Using Callbacks", describes the basic usage of Callbacks, provides some examples, and gives tips for debugging them.
- Chapter 17, "Partition Compile and Precompiled IP", describes the PC and PIP features in Verification Compiler to optimize the compilation performance.
- Chapter 18, "Integrated Planning for VC VIP Coverage Analysis", describes how to enable and use the functional coverage features.
- Appendix A, "Protocol Checks", outlines the process for working through and reporting VC PCIe issues.
- Appendix B, "PCIe PIE-8 Interface", describes the PIE-8 specification in PCIe VIP.
- Appendix C, "PCIe Compile-time Parameters", contains a table of parameters that can only be set before compilation, not at runtime.
- Appendix D, "Verilog Task/Parameter to SVT Class Mapping", contains tables that show the correspondence between SVT classes and Verilog tasks or parameters.
- Appendix E, "ECN Support", lists the ECN that have been implemented in the PCIe VIP.
- Appendix F, "SolvNetPlus PCIe VIP Articles", provides the lists and links to all the technical articles published on the PCIe VIP.
- Appendix G, "Reporting Problems", outlines the process for working through and reporting VC PCIe issues.

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:

- Go to https://solvnetplus.synopsys.com and open a case.
  - Enter the information according to your environment and your issue.
  - If applicable, provide the information noted in Appendix G, "Reporting Problems".
- 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.
  - If applicable, provide the information noted in Appendix G, "Reporting Problems".
- 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

This chapter gives a basic introduction, overview and features of the PCIe OVM Verification IP.

This chapter discusses the following topics:

1.1 Introduction

The VC PCIe Verification IP supports verification of SoC designs that include interfaces implementing the PCIe Specification. This document describes the use of this VIP in testbenches that comply with the SystemVerilog Open Verification Methodology (OVM). This approach leverages advanced verification technologies and tools that provide:

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

1.2 Prerequisites

Familiarity with PCIe, object oriented programming, SystemVerilog, and the current version of OVM.

1.3 Online Class Reference HTML Help

For more information on PCIe Verification IP, refer to the Class Reference for Synopsys Verification IP for PCIe, which you can access by opening the following file in a browser:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/index.html

Please note that the search available in the PCIe Class Reference does not support multiple words, Boolean expressions, or wild card searches. Use only single word searches.

1.4 Product Overview

The PCIe OVM VIP is a suite of OVM-based verification components that are compatible for use with SystemVerilog-Compliant testbenches. The Synopsys PCIe VIP suite simulates PCIe transactions using an active agent as defined by the PCIe specification.

The VIP provides a system environment that contains an active device and MAC agent. The agent supports all the functionality normally associated with active OVM components, including the creation of transactions, checking and reporting the protocol correctness, transaction logging and functional coverage.

1.5 Key Features

Following are the main key architectural features of the PCIe VIP.

- Emulates Root Complex and Endpoint
- Full protocol stack:

  Application, Transaction, Link, and Phy layers
- Checkers verify handshakes and functional accuracy at each layer
- Interfaces to capture sent and received packets for external scoreboard
- Error Injection at all levels
- Protocol checks integrated w/ OVM report object
- Extensive debug Aids
  - All states and Primitives available as ascii strings in waveform viewer
  - All signals named as close to the standard as possible
  - Log file similar to common trace formats from Bus Analyzers
  - Integrated with Protocol Analyzer
  - Symbol logger
- Verilog and OVM APIs
- Pre-configured instances
- Few parameters required to initiate transactions
- Full controllability for complex configurations
- Software Applications

- Built-in Scoreboard
- Shadow Memory for self-checking
- Driver
- Standard PCIe Bus Transactions
- Requester
- Background traffic to various memory ranges
- Completer
  - Automatically handles completions to mem/cfg/io writes & read

- Error Injections
  - Built in, Exceptions provide automated injection, checking, and recovery
  - User defined injections
    - Per transaction
    - Per symbol
- Debug
  - Grey box SystemVerilog Model
  - Key internal signals can be viewed in waveform viewer
  - ASCII string values for internal states
  - Multiple log options
  - OVM reporter
  - Transaction log
  - Symbol log

1.6 Other Supported Features

1.6.1 Requester, Driver, and Completer Applications

PCIe VIP currently supports the following verification functions:

- Functional coverage
- Protocol checking
- Control on delays and timeouts
- Built-in completer memory
- Verification Planner
- Protocol Analyzer
- Shadow memory with application-level scoreboard.

  Note: Shadow memory is limited to default behavior; settings are not changeable.
- Requester and driver applications

1.6.2 Methodology Features

PCIe VIP currently supports the following methodology functions:

- VIP organized as a set of agents and applications
- Analysis ports for connecting the agent to the scoreboard, or any other component
- Error injections via Factory, callback, or transaction item

2 Installation and Setup

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

The checklist consists of the following major steps:

1. Verifying the Hardware Requirements
2. Verifying Software Requirements
3. Preparing for Installation
4. Downloading and Installing
5. What's Next?

This chapter contains the following additional topics:

- Licensing Information
- Environment Variable and Path Settings
- Determining Your Model Version
- Integrating a VC VIP into Your Testbench
- Including and Importing Model Files into Your Testbench
- Compile-time and Runtime Options

If you encounter any problems with installing the VC PCIe, see *Customer Support*.

2.1 Verifying the Hardware Requirements

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

- 1200 MB available disk space for installation
- 16 GB Virtual memory (recommended)
- FTP anonymous access to ftp.synopsys.com (optional)

2.2 Verifying Software Requirements

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

2.2.1 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 PCIe VIP, check the support matrix for "SVT-based" VIP in the following document:

  Support Matrix for SVT-Based Synopsys PCIe VIP is in:

  Synopsys PCIe Release Notes

2.2.2 Synopsys Common Licensing (SCL) Software

- The SCL software provides the licensing function for the VC PCIe. Acquiring the SCL software is covered here in the installation instructions in Licensing Information.

2.2.3 Other Third Party Software

- **Adobe Acrobat**: VC PCIe documents are available in Acrobat PDF files. You can get Adobe Acrobat Reader for free from http://www.adobe.com.
- **HTML browser:** VC PCIe 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)

2.3 Preparing for Installation

1. Set DESIGNWARE_HOME to the absolute path where Synopsys PCIe 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

2.4 Downloading and Installing

Attention The Electronic Software Transfer (EST) system only displays products your site is entitled to download. If the product you are looking for is not available, contact est-ext@synopsys.com.

Follow the instructions below for downloading the software from Synopsys. You can download from the Download Center using either HTTPS or FTP, or with a command-line FTP session. If your Synopsys SolvNet password is unknown or forgotten, go to http://solvnet.synopsys.com.

Passive mode FTP is required. The passive command toggles between passive and active mode. If your FTP utility does not support passive mode, use http. For additional information, refer to the following web page:

https://www.synopsys.com/apps/protected/support/EST-FTP_Accelerator_Help_Page.html

2.4.1 Downloading From the Electronic Software Transfer (EST) System (Download Center)

1. Point your web browser to "https://solvnet.synopsys.com/DownloadCenter".
2. Enter your Synopsys SolvNetPlus Username and Password.
3. Click "Sign In" button.
4. Choose "Verification Compiler Verification IP" from the list of available products under "My Product Releases"
5. Select the Product Version from the list of available versions.
6. Click the "Download Here" button for HTTPS download.
7. After reading the legal page, click on "Yes, I agree to the above terms".
8. Click the download button(s) next to the file name(s) of the file(s) you wish to download.
9. Follow browser prompts to select a destination location.
10. You may download multiple files simultaneously.

Note The Protocol Analyzer is not included in the PCIe VIP download. It is a separate download, which you can get using the procedure above and selecting the Protocol Analyzer file, vip_pa_*version*_run, in step 8.

2.4.2 Downloading Using FTP with a Web Browser

1. Follow the above instructions through the product version selection step.
2. Click the "Download via FTP" link instead of the "Download Here" button.
3. Click the "Click Here To Download" button.
4. Select the file(s) that you want to download.
5. Follow browser prompts to select a destination location.

If you are unable to download the Verification IP using above instructions, see *Customer Support* section to obtain support for download and installation.

2.5 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 VC VIP into Your Testbench

2.6 Licensing Information

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

http://www.synopsys.com/keys

Attention Licensing is required if the VIP component classes are instantiated in the design. This includes envs, agents, drivers, monitors, sequencers, and components in UVM and OVM. This includes groups, subenvs, and transactors in VMM.

The Synopsys PCIe VIP uses a licensing mechanism that is enabled by the following license features which includes Gen3 and Gen4.

- VIP-PCIE-SVT
- VIP-PCIE-G3-OPT-SVT (required only for Gen3 support)
- VIP-PCIE-G4-SVT (required only for Gen4 support)

Only one license is consumed per simulation, regardless of how many PCIe VIP models are instantiated in the design. Each of the above features can also be enabled by VIP Library license. For more details, see *VC VIP Library Release Notes*.

Note the following:

- When G3 is being used, the VIP will consume the VIP-PCIE-G3-OPT-SVT license key
- When G4 is being used, the VIP will consume the VIP-PCIE-G3-OPT-SVT and the VIP-PCIE-G4-SVT license keys.

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.

2.6.1 Controlling License Usage

Using the DW_LICENSE_OVERRIDE environment variable, you can control which license is used as follows.

To use only DesignWare-Regression and VIP-LIBRARY-SVT licenses, set DW_LICENSE_OVERRIDE to:

DesignWare-Regression VIP-LIBRARY-SVT

To use only a VIP-PCIE-SVT license, set DW_LICENSE_OVERRIDE to:

VIP-PCIE-SVT

If DW_LICENSE_OVERRIDE is set to any value and the corresponding feature is not available, a license error message is issued.

2.6.1.1 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.

2.6.1.2 Simulation License Suspension

All Synopsys 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.

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

2.7 Environment Variable and Path Settings

The following are environment variables and path settings required by the PCIe verification models:

- DESIGNWARE_HOME – The absolute path to where the 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.

2.7.1 Simulator-Specific Settings

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

2.8 Determining Your Model Version

The following steps tell you how to check the version of the models you are using.

Note: 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 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 VIP models in your design directory, use the setup utility as follows:

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

2.9 Integrating a VC VIP into Your Testbench

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

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

- pcie_device_agent_svt: This is the name used for the entire set of sub-models.
- pcie_global_shadow_svt
- pcie_cfg_database_svt
- pcie_driver_app_svt
- pcie_io_target_svt
- pcie_mac_agent_svt
- pcie_mem_target_svt
- pcie_requester_app_svt
- pcie_target_app_svt
- pcie_tl_svt
- pcie_dl_svt
- pcie_pl_svt

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

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

Notes for OVM Users

1. OVM users must set the value of the OVM_PACKER_MAX_BYTES macro to 1500000 to ensure that a large enough internal buffer is set up to pack and unpack the data structure, since the default buffer size is only 4Kb. This is can be done in a run script or on the command line, as follows:

   ```
   +define+OVM_PACKER_MAX_BYTES=1500000
   ```

   This sets the internal buffer size to the minimum size needed for the PCIe VIP suite. If other suites are being used which require a larger buffer, then a larger size must be specified.

2. If you are using OVM version 1.1b, 1.1c, or 1.1d, then you must define the OVM_DISABLE_AUTO_ITEM_RECORDING macro. Since PCIe is a pipelined protocol (that is, the previous transaction does not necessarily complete before another transaction is started), the PCIe VIP handles triggering of the begin and end events and transaction recording. The PCIe VIP does not use the OVM automatic transaction begin and end event triggering feature. If OVM_DISABLE_AUTO_ITEM_RECORDING is not defined the VIP issues a fatal error.

   If you are using OVM version 1.2 or newer, you do not need to set the macro.

   When the OVM_DISABLE_AUTO_ITEM_RECORDING macro is set, recording is disabled for all VIPs in the design.

The following command adds the full model to the design_dir directory.

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

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

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

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

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

2.9.1 Installing and Setting Up More than One VIP Protocol Suite

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

The examples in this chapter call that directory design_dir but you can use any name. In this example, assume you will use the pcie_svt and AXI VIP suites in the design. Your $DESIGNWARE_HOME contains both pcie_svt and AXI VIPs.

First, install a pcie_svt example into the design_dir directory. After the pcie_svt example has been installed, the VIP suite must be set up in and located in the same design_dir directory as the pcie_svt VIP. Use the following commands to perform those steps:

```
// First install the pcie_svt unified OVM example:
% $DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -e
      pcie_svt/tb_pcie_svt_ovm_unified_vip_sys -svtb

// Add AXI to the same design_dir as the pcie_svt:
% $DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -add axi_system_env_svt -svlog
```

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

2.9.1.1 Getting Help on Example Run/make Scripts

You can get help on the generated make/run scripts in the following ways:

1. Invoke the run script with no switches, as in:

```
run_pcie_svt_ovm_unified_vip_sys
  usage:  run_pcie_svt_ovm_unified_vip_sys [-32] [-verbose] [-debug] [-waves] [-clean]
[-nobuild] [-norun] [-pa] <scenario> <simulator>
  where   <scenario> is one of: base_multi_link_test
base_pipe5_gen6_full_eq_in_serdes_arch_mode_test
```

```
base_pipe5_gen6_in_serdes_arch_mode_test base_pipe5_in_serdes_arch_mode_test
base_pipe5_test base_pipe_test base_pma_test base_serdes5_test base_serdes_test
<simulator> is one of:  vcsvlog vcsmxvlog mtivlog vcsmxpcvlog vcsmxpipvlog ncvlog
vcspcvlog
          -32       forces 32-bit mode on 64-bit machines
          -verbose  enable verbose mode during compilation
          -debug    enable debug mode for SVT simulations
         -waves    [fsdb|verdi|dump] enables waves dump and optionally opens viewer (VCS
only)
          -clean    clean simulator generated files
          -nobuild  skip simulator compilation
          -norun    exit after simulator compilation
          -pa       invoke PA after execution
```

2. Invoke the make file with help switch as in:

```
gmake help
Usage: gmake USE_SIMULATOR=<simulator> [VERBOSE=1] [DEBUG=1] [FORCE_32BIT=1]
[WAVES=fsdb|verdi|dump] [NOBUILD=1] [PA=1] [<scenario> ...]
Valid simulators are: vcsvlog vcsmxvlog mtivlog vcsmxpcvlog vcsmxpipvlog ncvlog
vcspcvlog
Valid scenarios are: base_multi_link_test
base_pipe5_gen6_full_eq_in_serdes_arch_mode_test
base_pipe5_gen6_in_serdes_arch_mode_test base_pipe5_in_serdes_arch_mode_test
base_pipe5_test base_pipe_test base_pma_test base_serdes5_test base_serdes_test
```

Note: You must have PA installed if you use the -pa or PA=1 switches.

2.9.2 Updating an Existing Model

To add an update to an existing model, perform the following steps:

1. Set the $DESIGNWARE_HOME environment variable to the latest version.
2. Navigate to the directory where the existing *design_dir* is present (and not into the *design_dir*).
3. Update the directory with the latest release by passing the model name, "pcie_device_agent_svt" to -add option while running the dw_vip_setup application.

   Example:

   ```
   $DESIGNWARE_HOME/bin/dw_vip_setup -path <existing_design_dir_name> -add
   pcie_device_agent_svt
   ```

Note In case of any backward incompatible changes in the VIP, you need to update the *design_dir* by reinstalling the example in same location and/or update the testbench files accordingly by referring to the backward incompatible changes listed in the *Release Notes*.

2.10 Including and Importing Model Files into Your Testbench

After you set up the models, you must include and import various files into your top testbench files to use the VIP. You must include the file, import_pcie_svt_ovm_pkgs.svi that internally imports the existing PCIe OVM top package (svt_pcie_ovm_pkg) and individual sub-packages for partition compile at user testbench. Following is a code list of the includes and imports for PCIe:

```
/* include ovm package before VIP includes,  If not included elsewhere*/
```

```
`include "svt_pcie.ovm_pkg"

/** Include the PCIE SVT OVM package */
`include "svt_pcie.ovm.pkg"

/** Defines required for PCIE SVC */
`define PCIESVC_MEM_PATH                      test_top.mem0
`define EXPERTIO_PCIESVC_GLOBAL_SHADOW_PATH  test_top.global_shadow0
`define SVC_RANDOM_SEED_SCOPE                 test_top.global_random_seed

/** Import OVM Package */
import ovm_pkg::*;
`include "ovm_macros.svh"

/** Imports SVT PCIe packages */
`include "import_pcie_svt_ovm_pkgs.svi"

/** Include Util parms */
`include "svc_util_parms.v"

/** Include specific pcie_svt files
`include "svt_pcie_defines.svi"
`include "svt_pcie_device_configuration.sv"
`include "svt_pcie_device_agent.sv"
```

You must also include various VIP directories on the simulator command line. Add the following switches and directories to all compile scripts:

- +incdir+<design_dir>/include/verilog
- +incdir+<design_dir>/include/sverilog
- +incdir+<design_dir>/src/verilog/<vendor>
- +incdir+<design_dir>/src/sverilog/<vendor>

Supported vendors are vcs, mti and ncv. For example:

```
+incdir+<design_dir>/src/sverilog/vcs
```

Using the previous examples, the directory `<design_dir>` would be /tmp/design_dir.

2.11 Compile-time and Runtime Options

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

$DESIGNWARE_HOME/vip/svt/pcie/latest/examples/sverilog/<*test_name*>

The files containing the options are:

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

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

vcs_build_options

Required: +define+OVM_PACKER_MAX_BYTES=1500000
Optional: -timescale=1ns/1ps
Required: +define+SYNOPSYS_SV

Required (For partition compile): +define+SVT_PCIE_OPTIMIZED_COMPILE

vcs_run_options

Required: +OVM_TESTNAME=$scenario
Note: “scenario” is the ovm test name you pass to VCS

3 Creating the PLI Object in 32-bit and 64-bit Simulation Modes

This chapter contains the following topics:

3.1 Compiling the Messaging PLI for the PCIe Model

On most simulators you can run in either a native 64-bit mode, or a legacy-emulated 32-bit mode.

Running in emulated 32-bit mode uses the same PLIs as those used on 32-bit machines. Generally they do not even need to be recompiled. This can be useful on a a multi-machine heterogeneous grid with both 32 and 64-bit processors, since a single PLI can be shared across machines. The downside is that simulations that require very large amounts of memory (greater than 3GB) will be memory-constrained by the 32-bit library.

Compilation of the 32-bit PLI typically requires special command-line switches to inform the compiler/linker to create 32-bit object files (see “Running in 32-bit mode” and “Run a 64-bit executable using 32-bit PLI object”).

In the native 64-bit mode, you must use a 64-bit PLI. If a 64-bit mode compiler is available, there generally is no need for special methods to build the PLI objects.

The purpose of the msglog PLI is to provide a message logging facility in the underlying Verification Compiler SVT model and to provide a mechanism for all components in the model to use a common logging facility. The msglog PLI tasks and functions are written in C and they are for logging various levels of messages. These tasks and functions should be either included in a build by dynamically linking a library that contains the compiled object file msglog.o, or explicitly statically linked to msglog.o.

Before you can use the pcie_svt model, you must build the msglog.o object file and the PLI files. The PLI generation is needed for legacy support of the msglog feature and for time 0 messages generated by the underlying Verilog model.

If you run the example scripts included with the installation, those files are generated automatically. You can use either the run script or the Makefile to run the example scripts. Information about building the msglog.o and PLI files is provided in the prescript file.

If you do not run the example scripts, you can use the following procedure to build the msglog.o and PLI files:

Convert the Verilog parameters and the defines that msglog uses from the ${*design-dir*}/src/verilog/vcs/svc_util_parms.v file.You can use the param2def.sh script to do that. The param2def.sh script converts Verilog-style parameters to C-style #defines. Use the following command to run the param2def.sh script:

```
& ${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/bin/param2def.sh < ${design-
dir}/src/verilog/vcs/svc_util_parms.v > svc_util_parms.h
```

3.2 Compiling the msglog.o and PLI Files

The next step is to compile the msglog and the PLI files. You can compile these files on either a 32-bit or 64-bit machine. Please note that cc defaults to 64 bit. The m32 switch forces 32-bit operation on a 64-bit machine. If the compilation is done on a 32-bit machine, the m32 flag is not needed.

1. Set ccflags for the compile depending on the machine type. These flags will match the setting done in the run script from the example testbench.

   For the Linux platform, set the ccflags to the following:

   64 bits: `set ccflags = ""`

   32 bits: `set ccflags = "-m32"`

2. Compile with the correct include files and switches to generate the msglog.o file.

   a. For VCS | VCS MX:

      32 bits:

      ```
      cc -c ${ccflags} -I. -I${VCS_HOME}/include -
      I${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/include -DVCS_VERILOG
      -DUSE_VPI=1 ${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/src/msglog.c
      -o msglog.o
      ```

      64 bits:

      ```
      cc -c ${ccflags} -I. -I${VCS_HOME}/include -
      I${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/include -DVCS_VERILOG
      -DUSE_VPI=1 -DPLI_64_BIT
      ${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/src/msglog.c -o msglog.o
      ```

      NOTE for ccflags_dyn: For the Linux platform, set the ccflags_dyn to the following:

      64 bits: `set ccflags_dyn = "-fPIC"`

      32 bits: `set ccflags_dyn = "-m32 -fPIC"`

   b. For MTI:

      ```
      cc -c ${ccflags_dyn} -I. -I${MTI_HOME}/include
      -I${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/include -DQUESTA
      ${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/src/msglog.c -o msglog.o
      ```

   c. For NC:

      ```
      cc -c ${ccflags_dyn} -I. -I${CDS_INST_DIR}/tools/inca/include
      -I${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/include -DNC_VERILOG
      ${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/src/msglog.c -o msglog.o
      ```

3. Compile the verisuer file, which registers the PLI information with the simulator.

   a. For VCS | VCS MX:

```
${VCS_HOME}/bin/veriuser_to_pli_tab -include ${VCS_HOME}/include
${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/src/veriuser.c > pli.tab || rm -f
pli.tab
```

b. For MTI:

```
cc -c ${ccflags_dyn} -I. -I${MTI_HOME}/include
-I${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/include -DQUESTA
${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/src/dyn_veriuser.c
-o dyn_veriuser.o
```

c. For NC:

```
cc -c ${ccflags_dyn} -I. -I${CDS_INST_DIR}/tools/inca/include
-I${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/include -DNC_VERILOG
${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/src/dyn_veriuser.c
-o dyn_veriuser.o
```

NOTE for ldflags_dyn: For the linux platform, set the ldflags_dyn to the following:

64 bits: `set ldflags_dyn = "-shared"`

32 bits: `set ldflags_dyn = "-m32 -shared"`

4. Compile the msglog and dyn_veriuser files to generate the PLI file.

a. For MTI:

Once the dyn_veriuser.o file is generated from step 3, run this compile:

```
cc ${ldflags_dyn} -o dyn_mtipli.so msglog.o dyn_veriuser.o
```

b. For NC:

Once the dyn_veriuser.o file is generated from step 3, run this compile:

```
cc ${ldflags_dyn} -o dyn_ncvpli.so msglog.o dyn_veriuser.o
```

Once the msglog and PLI information are generated, you can simulate with the model.

4 General Concepts

This chapter describes the usage of the PCIe VIP in a OVM environment, and its user interface. This chapter discusses the following topics:

- Introduction to OVM
- Active and Passive Mode
- PCIe OVM Interface
- PCIe Gen3 Support

4.1 Introduction to OVM

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

This chapter describes the usage of the PCIe VIP in OVM environment, and its user interface. 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 OVM. For more information:

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

4.2 Active and Passive Mode

The M-PHY Adapter Physical Layer can also be configured to contain an M-PHY Monitor (svt_mphy_monitor) component associated with each M-PHY Tx and M-PHY Rx component. These monitor components may be used for functional coverage and protocol checking of the M-Phy interfaces. There are properties (for example, pcie_enable_mphy_monitor) in the PCIe configuration (svt_pcie_mphy_adapter_configuration) object class that may be used to control the behavior of the M-PHY monitors. Please refer to the class reference for more details.

4.3 PCIe OVM Interface

The Verification Compiler OVM VIP for PCIe is a suite of advanced verification components and data objects based on SystemVerilog OVM-compliant technology. The Verification Compiler OVM PCIe VIP is based on the following OVM agent architecture and data objects.

svt_pcie_device_agent

The **svt_pcie_device_agent** object defines a OVM agent that contains the following ovm_components for PCIe applications:

- Driver
- Target
- Requester
- IO target
- Memory target
- Configuration database
- Global shadow

The **svt_pcie_device_agent** object contains a OVM agent named svt_pcie_agent. The PCIe OVM subenvironment contains active PCIe applications to send and receive PCIe packets as well as OVM sequencers and sequences. Your testbench will interact mainly with OVM sequencers, which can use either Verification Compiler-provided sequences or your own sequences.

svt_pcie_agent

The **svt_pcie_agent** object defines a OVM agent that contain a layered stack of ovm_components for the Physical, Link, and Transaction layers of the PCIe protocol. The PCIe OVM agent contains active PCIe Physical, Link, and Transaction ovm_components, as well as OVM sequencers and sequences. Your testbench will interact mainly with OVM sequencers which can use either Verification Compiler-provided sequences, or your own sequences.

4.3.1 OVM Components of the PCIe Device Subenvironment

- **svt_pcie_driver_app –** A ovm_component object that implements the PCIe Driver application, which transmits PCIe packets to PCIe transaction layer.
- **svt_pcie_requester**_app **–** A ovm_component object that implements the PCIe Requester application, which supports generating Memory reads and writes towards the programmed Memory segment.
- **svt_pcie_target**_app **–** A ovm_component object that implements the PCIe Target application, which is responsible for responding to received requests by generating completion packets.
- **svt_pcie_io_target –** A ovm_component object that supports the IO segment of the PCIe system.
- **svt_pcie_mem_target –** A ovm_component object that supports the Memory segment of the PCIe system.
- **svt_pcie_cfg_database –** A ovm_component object that supports the Configuration space of the PCIe system.
- **svt_pcie_global_shadow –** A ovm_component object that implements the PCIe Device system IO, Memory and Configuration spaces.

4.3.2 OVM Components of the PCIe MAC Agent

- **svt_pcie_tl –** A ovm_component object that implements the PCIe Transaction Layer.
- **svt_pcie_dl –** A ovm_component object that implements the PCIe Link Layer.
- **svt_pcie_pl_phy –** A ovm_component object that implements the PCIe Physical Layer.

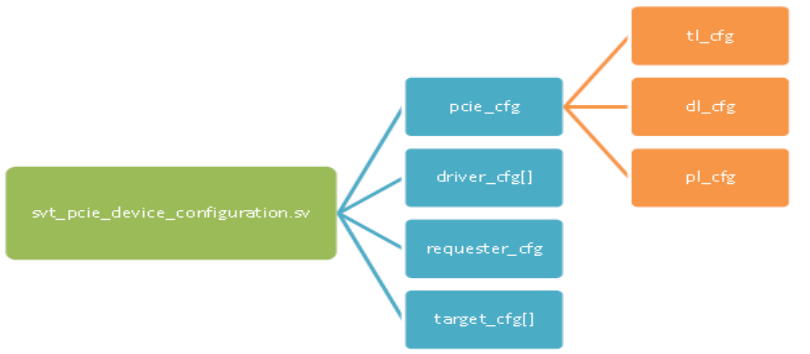
4.3.3 Configuration Data Objects

Configuration data objects are abstracted data objects that represent the content of PCIe VIP configuration data and protocol transactions. The top-level configuration data objects are:

- svt_pcie_device_configuration
  - svt_pcie_driver_app_configuration
  - svt_pcie_requester_app_status
  - svt_pcie_target_app_configuration
  - svt_pcie_configuration
    - svt_pcie_tl_configuration
    - svt_pcie_dl_configuration
    - svt_pcie_mpl_phy_configuration
    - svt_pcie_mphy_adapter_configuration

The following illustration shows the inheritance diagram for all the configuration objects.

Figure 4-1 Inheritance Diagram for All Configuration Objects



4.3.4 Status Data Objects

Status data objects are abstracted data objects that represent the content of PCIe VIP statistics. Registered status objects are updated in realtime. Separate statistics are kept per layer/application. Status objects are useful for:

- functional coverage

- reporting testcase progress
- debug

The top-level status data objects are:

- svt_pcie_device_status
  - svt_pcie_requester_app_status
  - svt_pcie_target_app_status
  - svt_pcie_io_target_status
  - svt_pcie_mem_target_status
  - svt_pcie_status
    - svt_pcie_tl_status
    - svt_pcie_dl_status
    - svt_pcie_pl_status

Following are some examples of the type of status data you can obtain.

- Target Application

  # bytes received

  # bytes sent

  # msg cpl sent

  # num tlps received
- Phy Layer

  #LTSSM state

  # hot resets initialized
- Link Layer

  # ack received

  # bad tlp sent

  # EI RX TLP withhold ack / nack
- Transaction Layer

  # bad TLPs received

  # TC5 TLPs sent

An example of waiting for link activation:

```
svt_pcie_device_status stat;
cfg = svt_pcie_device_status::type_id::create("stat");

// wait for link activation
wait(stat.port_status.pl_status.link_up == 1'b1);
```

4.3.5 Sequence Item Data Objects

The VIP supports extending OVM sequence item data classes for customizing randomization constraints. This allows you to disable some reasonable_* constraints and replace them with constraints appropriate to your

system. Individual reasonable_* constraints map to independent fields, each of which can be disabled. The following are the sequence data item classes:

- svt_pcie_driver_app_transaction
  - svt_pcie_io_target_service
  - svt_pcie_mem_target_service
  - svt_pcie_cfg_database_service
  - svt_pcie_global_shadow_service
  - svt_pcie_driver_app_service
  - svt_pcie_requester_app_service
  - svt_pcie_target_app_service
  - svt_pcie_tl_service
  - svt_pcie_dl_service
  - svt_pcie_pl_service

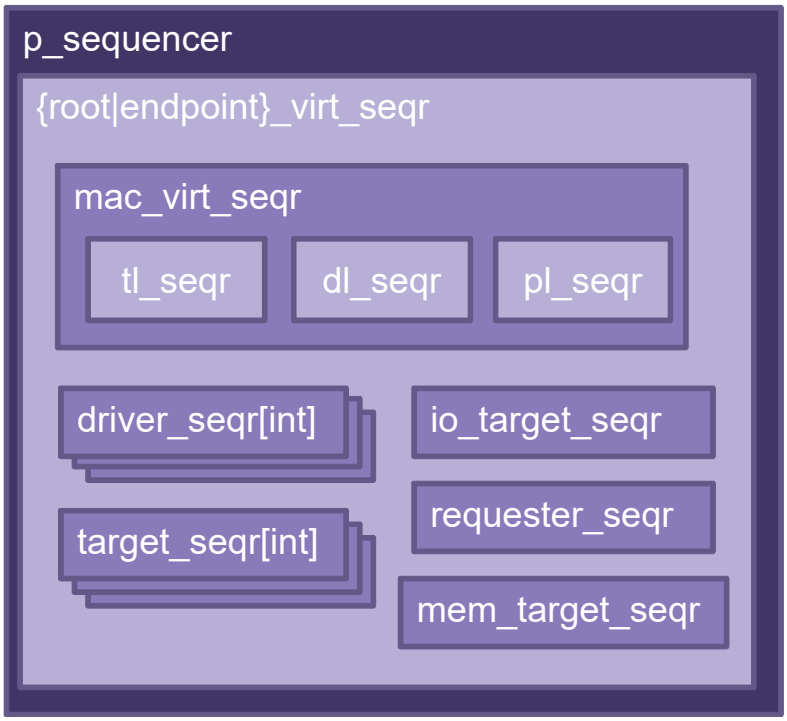
4.4 SVT Service Sequence/Sequencer

Each component in the agent has its own service sequencer. All SVT sequences are derived from ovm_sequence. Sequences or individual sequence items are executed on the appropriate sequencer.

A *p_sequencer* is instantiated with a macro in the top sequence as shown below:

```
`ovm_declare_p_sequencer
 (svt_pcie_device_system_virtual_sequencer
```

The following figure shows the elements of the `p_sequencer`:



4.5 PCIe Gen3 Support

To enable Gen3 features, the SVT_PCIE_ENABLE_GEN3 macro must be defined on the command line for VCS invocation and the svt_pcie_device_configuration::pcie_spec_ver must be set to svt_pcie_device_configuration::PCIE_SPEC_VER_3_0.

The following is an example of how to define a macro on a command line for VCS invocation:

```
vcs +define+SVT_PCIE_ENABLE_GEN3 other_switches
```

4.6 Compliance Patterns

The compliance and modified compliance patterns defined in the PCIE specification contain sequences of data that would be considered an error during normal operation. For example, at 2.5G and 5G part of the compliance pattern is to send a COM followed by data symbols that do not make a legal ordered set.

At 8G there are ordered set blocks filled with symbols that do not make up a valid ordered set. Additionally SKP ordered sets at 8G contain data associated with compliance rather than the contents of the LFSR. Because there is no way to know exactly when the DUT starts transmitting the compliance pattern vs normal link training, the VIP can and will likely flag some ordered set violations until it recognizes the compliance pattern.

This means that when the vip initially receives the compliance pattern or modified compliance pattern, the user will be required to suppress or demote some error messages until the VIP obtains lock on the pattern in order to obtain a passing test. In PIPE simulations, the VIP should recognize a compliance or modified compliance pattern by the time the pattern has completed its first cycle.

In serial simulations it will longer for the VIP to recognize compliance, because if a speed change occurs in polling.compliance, then the VIP must acquire bit lock and symbol/block alignment first. The modified compliance pattern at 8G in serial mode will take an especially long time, because the EIEOS required for block alignment occurs only once every 65792 blocks.

5 Verification Features

This chapter describes the various verification features available with the Synopsys PCIe Verification IP. This chapter discusses the following topics:

- The Transaction Logger
- The Symbol Logger
- Using Native Protocol Analyzer for Debugging
- Verification Planner
- Global Shadow Memory
- Target Memory
- Data Link Monitor

5.1 The Transaction Logger

All inbound and outbound transactions to or from the VIP (both TLPs and DLLPs) are sent to the transaction logger. These transactions are distilled and written (one transaction per line) to the transaction log file.

By default, the transaction logger is disabled. To enable it, and cause it to start writing transactions to a file, use the enable_transaction_logging member of the svt_pcie_configuration class, as shown in the following example:

```
class example extends ovm_test;
. . .
  virtual function void task build_phase(ovm_phase phase);
    super.build_phase(phase);
    . . .
    root_cfg.port_cfg.enable_symbol_logging = 1; // Enable for RC
    endpoint_cfg.prot_cfg.enable_symbol_logging = 1; // Enable for EP
    root_cfg.port_cfg.transaction_log_filename = "symbol.log"; // Recommended to only
                                                               //set the filename once
    . . .
  endfunction
endclass
```

The symbol log filename will be appended to the full hierarchical name of the port0 instance generating it.

5.1.1 Printing TLP Payload Data to a Transaction Log File

The Synopsys provided unified example shows you how to print TLP payload data to the Transaction log. You must first enable transaction logging. By default it is off. Also, set the filename of the transaction log.

```
class pcie_shared_cfg extends ovm_object;
...
  /** Setup the PCIE device system default values */
  function void setup_pcie_device_system_defaults();
    begin
...
      root_cfg.pcie_cfg.enable_transaction_logging = 1'b1;
      root_cfg.pcie_cfg.transaction_log_filename = "transaction.log";

      root_cfg.pcie_cfg.enable_symbol_logging = 1'b1;
      root_cfg.pcie_cfg.symbol_log_filename = "symbol.log";
```

Next, set how many dwords of the payload you want the model to write into the transaction log file.

```
class pcie_device_base_test extends ovm_test;
...
  virtual function void build_phase(ovm_phase phase);
    `ovm_info("build_phase", "Entered...", OVM_LOW)
    super.build_phase(phase);

...
    /** Set the payload display limit */
    svt_pcie_dl_disp_pattern::default_max_payload_print_dwords = 1024;
```

Note the default is zero. In the example, it has been set to 1024.

5.1.2 Fields of the Transaction Log Header

The fields of the transaction log header are described in this section. These fields are listed from left to right as they appear on the header.

Field: Reporter

Description:

This field represents an instance of the VIP in the test environment. The transaction log information is reported for this VIP instance.

Field: Start Time (ns)

Description:

This field represents the simulation time in ns when the transaction starts.

Field: End Time (ns)

Description:

This field represents the simulation time in ns when the transaction ends.

Field: Dir

Description:

This field represents the direction of the transaction from the VIP instance. "T" represents a transmit transaction. "R" represents a receive transaction.

Field: TLP Type

DLLP Type

Description:

This field represents the type of TLP (Transaction Layer Packet) or DLLP (Data Link Layer Packet) as defined in Table 2-3 and Table 3-1 of the PCIe Specification respectively. For TLP memory read (MRd) and memory write (MWr) packets, a "32" or "64" is appended to the type. This number represents a 32-bit or 64-bit memory addressing.

For example:

MRd32

MWr64

Field: R_ID /Tag | ST

Description:

This field has 2 different representations for a TLP transaction. The Requester ID and Tag are displayed, or the Steering Tag is displayed. This field is blank for DLLP transactions.

TLP Field	Description
R_ID/Tag	Requester ID/Tag
ST	Steering tag

For example:

TLP

Reporter	Start Time (ns)	End Time (ns)	Dir	TLP Type DLLP Type	R_ID/Tag | ST
vip0	133328.00	133340.00	T	**CfgRd0**	**0x0001/13**
vip0	159140.00	159160.00	T	**MRd32**	**0x0001/1f**

DLLP

Reporter	Start Time (ns)	End Time (ns)	Dir	TLP Type DLLP Type	R_ID/Tag | ST
vip0	133328.00	133328.00	R	**INITFC2_P_VC0**	

```
vip0        133772.00   133772.00   R       ACK
```

Field: Seq Num

Description:

This field represents the sequence number of the transaction.

Field: TC VC

Description:

This field represents the value of the Traffic Class field of the TLP.

Field: TH

Description:

This field represents the 1-bit TH field of the common TLP packet header. The TH field is an indication of the TLP Processing Hints (TPH) and the Optional TPH TLP Prefix when applicable presented in the TLP header.

Field: PH

Description:

This field represents processing hint.

Field: IDO RO

Description:

These 2 fields represent the Ordering Attribute Bits as defined in Table 2-10 of the PCIe Specification. The table is shown below.

Attribute Bit [2] (IDO)	Attribute Bit [1] (RO)	Ordering Type	Ordering Model
0	0	Default Ordering	PCI Strongly Ordered Model
0	1	Relaxed Ordering	PCI-X Relaxed Ordering Model
1	0	ID-Based Ordering	Independent ordering based on Requester/Completer ID
1	1	Relaxed Ordering plus ID-Based Ordering	Logical "OR" of Relaxed Ordering and IDO

Field: NS

Description:

This field represents the No Snoop bit value of the TLP.

Field: Address

Reg#/MsgRt/Cpl

HdrFC DataFC

Description:

This field has multiple representations. For a TLP transaction, the value(s) displayed depends on the TLP type as shown in the following table. For a DLLP transaction, the Flow Control header and data are displayed.

TLP Field	Description
< address >	**Memory request:** This field represents the memory address.
< address >	**IO request:** This field represents the IO address.
BDF: < > R: < > or BDF: < > O: < >	**Configuration request:** "BDF" represents the bus device function. "R" represents the register number. "O" represents the register byte offset. This offset is not displayed by default. To enable the display, you must set the dl_trace_options[1] attribute of the PCIe configuration class (svt_pcie_configuration). For example: <agent cfg>.pcie_cfg.dl_trace_options[1] = 1
< message >	**Message request:** This field represent the message routing as defined in Table 2-18 of the PCIe Specification.
ID: < > Stat: < >	**Completion request:** "ID" represents the Completion ID. "Stat" represents the Completion status as defined in Table 2-29 of the PCIe Specification. The status are SC, UR, CRS and CA.

DLLP Field	Description
< header > < data >	Flow Control header and data

For example:

TLP:

Reporter	Start Time (ns)	End Time (ns)	Dir	TLP Type DLLP Type	R_ID/Tag | ST	Seq Num	TC VC	T H	P H	I D O	R O	N S	Address Reg#/MsgRt/Cpl HdrFC DataFC
vip0	159140.00	159160.00	T	MRd32	0x0001/1f	139	0	0		0	0	0	**0x00245a28**
vip0	133500.00	133520.00	T	CfgWr0	0x0001/1c	2	0	0		0	0	0	**BDF:0x0000 R:0x004**

vip0	133304.00	133324.00	T	MsgD	0x0001/18	0	0	0	0	0	0	**Local Term Rcvr ID:0x0002 Stat:SC**
vip0	158868.00	158872.00	R	CplD	0x0001/1e	136	0	0	0	0	0	**BC:0004**

DLLP

Reporter	Start Time (ns)	End Time (ns)	Dir	TLP Type DLLP Type	R_ID/Tag | ST	Seq Num	TC VC	T H	P H	ID O	RO	NS	Address Reg#/MsgRt/Cpl HdrFC	DataFC
vip0	133596	133596	R	ACK		2							**116**	**230**
vip0	133124	133124	T	INITFC1_P_VC0									**102**	**1024**

Field: BE | ST

BC

MCode

Description:

This field has multiple representations of a TLP transaction.

TLP Field	Description
< byte enable >	**Memory request/IO request/Configuration request:** This field represents the byte enable.
< steering tag >	**Memory request:** When the TH field has a value of "1", this field represents the steering tag value.
BC: < >	**Completion request:** BC represents the byte count.
< message code >	**Message request:** This field represent the message code as defined in Table F-1 of the PCIe Specification.

For example:

Reporter	Start Time (ns)	End Time (ns)	Dir	TLP Type DLLP Type	R_ID/Tag | ST	Seq Num	TC VC	T H	P H	ID O	RO	NS	Address Reg#/MsgRt/Cpl HdrFC DataFC	BE | ST BC Mcode
vip0	133304.00	133324.00	T	MsgD	0x0001/18	0	0	0		0	0	0	Local Term Rcvr	**0x50**
vip0	158896.00	159124.00	T	MWr32	0x0001/09	138	0	0		0	0	0	0x00245a28	**1 c**
vip0	158868.00	158872.00	R	CplD	0x0001/1e	136	0	0		0	0	0	ID:0x0002 Stat:SC	**BC:0004**

Field: Len/Idx

DW

Description:

This field has 2 representations of a TLP transaction.

TLP Field	Description
< payload length >	For the TLP Header displayed on the first row of data, this field represents the length of the payload in double word (DW).
< data index >	For the TLP payload displayed from the second row of data and on, this field represents the accumulative count of DW data.

For example:

MWr32

Len/Idx DW	Prefix / Header / Data (All values in Hex)			
221	H400000dd	H0001091c	H00245a28	---
0	250e4795	e0b18822	9c1a2b30	a6824000
4	48105945	cafb12dd	8ed6f96b	df547dae
8	17505659	31998267	b37c242c	cc4f5f10
		< data continues >		
216	0a3336db	36bb1e9b	df73a4c9	43d8486b
220	00693366	---	---	---

In the above example, the "221" on the first row is the TLP payload length. The "0" through "220" on the subsequent rows are the data index.

CfgRd0

Len/Idx DW	Prefix / Header / Data (All values in Hex)			
1	H04000001	H00011e0f	H00020000	---

CplD

Len/Idx DW	Prefix / Header / Data (All values in Hex)			
1	H4a000001	H00010004	H00011500	---
0	06001000			

Field: Prefix / Header / Data

(All values in Hex)

Description:

This field represents the payload data values of a TLP transaction. Values with an "H" prefix represent raw header DWORDs. Values with a "P" prefix represent the PCIe Prefix data.

By default, only the TLP header data is displayed along with the header fields. You can enable the display of payload data by using one of the following methods:

a. In the build phase of the simulation, set the static attribute "default_max_payload_print_dwords" of class "svt_pcie_dl_disp_pattern" to the default maximum number of payload DWORDs to be displayed.

b. Set the "SVT_PCIE_XACT_LOG_MAX_PAYLOAD_DWORDS_DEFAULT" macro to the default maximum number of payload DWORDs to be displayed.

Payload data for CfgWr and CfgRd (corresponding CplD) are displayed with a single DWORD. By default, this DWORD value is displayed in the Big Endian format. Alternatively, you can enable the DWORD to be displayed in the Little Endian format by setting the "dl_trace_options[0]" attribute of the PCIe configuration class (svt_pcie_configuration).

For example:

Default Big Endian payload data

```
0 06001000
```

Enable Little Endian format

```
<agent cfg>.pcie_cfg.dl_trace_options[0] = 1;
```

Little Endian payload data

```
0 00100006  LittleEndian
```

Field: EP

Description:

This field represents the poison bit as defined in the PCIe Specification.

Field: ECRC

Description:

This field represents the ECRC value of a TLP as defined in the PCIe Specification.

Field: LCRC

CRC

Description:

This field has 2 representations. For a TLP transaction, the LCRC value as defined in the PCIe Specification is displayed. For a DLLP transaction, the CRC value as defined in the PCIe Specification is displayed.

Field: TX/RX

Error

Description:

This field represents the type of error injection when error injection is enabled in a transmit (tx) transaction. For a receive (rx) transaction, this field represents the detected error.

Error Injection Type	Description
BadSeq	Illegal Sequence Number
CodeViol	TX Code Violation
CrcEr	LCRC LCRC Error
Disparty	Disparity Error
DupSeq	Duplicate Sequence Number
EIErr	Scenario injects error, then vip reported this.
HdrCRC	PCIE 8G Header CRC Error
HdrPAR	PCIE 8G Header Parity Error
LCRC	LCRC Error
NAK	NAK Received for TLP
NoACK	Missing ACK for Transaction
NoEND	Missing END
NoSTART	Missing START
NoSTP	NoSTART Missing START"
NullLCRC	Nullified TLP with corrupt CRC
NullTLP	Nullified TLP
ReplCnt	Replay count of 4 exceeded

5.2 The Symbol Logger

One log file is created per simulation. All agents share the log file. Each agent must be enabled independently as shown in Example 5-1. If more than one symbol_log_filename is set, then the last one set within the simulation serves as the filename. It is recommended that you have only one agent set the filename.

Example 5-1

```
class example extends ovm_test;
   . . .
   virtual function void task build_phase(ovm_phase phase);
      super.build_phase(phase);
      . . .
      root_cfg.pcie_cfg.enable_symbol_logging = 1;                // Enable for RC
      endpoint_cfg.prot_cfg.enable_symbol_logging = 1;            // Enable for EP
     root_cfg.pcie_cfg.transaction_log_filename = "symbol.log"; // Recommended to only
set the filename once
```

```
      . . .
   endfunction
endclass
```

The transaction log filename will be appended to the full hierarchical name of the port0 instance generating it.

5.2.1 Fields of the Symbol Log Header

The fields of the symbol log header are described in this section. These fields are listed from left to right as they appear on the header.

Field: TIME

Description:

This field represents the simulation time in ns. Symbol logging is performed at the PIPE interface. If the PIPE interface is configured as multiple bytes, all bytes transferred at a time step are logged at the same time step.

Field: INSTANCE

Description:

This field represents an instance of the VIP in the test environment. The symbol log information is reported for this VIP instance.

Field: < lane symbols >

Description:

This field represents symbols on the active lane(s). The format of the field header is:

R00 [R01] [R02] … [R<n>] | T00 [T01] [T02] … [T<n>]

Where, "R" represents the receive symbol on the lane. "T" represents the transmit symbol on the lane. <n> is the number of the highest configured lane.

The encoding of the lane symbols are listed in the following tables.

Table 5-1 Special Symbol Encodings for All Link Data Rates

Symbol	Description
z	Electrical idle
?	Invalid or unknown value
.	No information available to log. This may occur at startup, at changes to link speed or link width, or if the Rx and Tx sides are operating at offset time steps at either 2.5 GT/s or 5 GT/s.

Table 5-1 Special Symbol Encodings for All Link Data Rates

Symbol	Description
q	Error injection pending: appended on each symbol that will have disparity inverted. Only applies on TX lanes.
j	Error injection pending: appended on each symbol that will have a random bit flipped. Only applies on TX lanes.
v	Error injection pending: appended on each symbol that will have an invalid encoding. Only applies on TX lanes.

For link operation at 8GT/s, symbols after the sync headers are prepended with encodings of the sync headers listed in <Xref>Table 5-2. Additional encodings for link operation at 8GT/s are listed in <Xref>Table 5-3.

Table 5-2 Sync Header Encodings for Link Operation at 8GT/s

Symbol	Description
@	2'b'00 (Reserved)
*	2'b'01 (OS block)
=	2'b'10 (Data block)
$	2'b'11 (Reserved)

Table 5-3 Special Character Encodings for Link Operation at 8GT/s

Symbol	Description
::	Data skip cycle (no valid data)
+	Start of TLP Token. Prepended on 1st symbol of token. Replaces = symbol at start of block. Only noted on TX lanes.
^	Start of DLLP Token. Prepended on 1st symbol of token. Replaces = symbol at start of block. Only noted on TX lanes.

Field: LTSSM State

Description:

This field represents the state of the LTSSM state machine as defined in section 4.2.5 of the PCIe specification. For states such as L0 and L0s where the receive (rx) and transmit (tx) LTSSM states may diverge, the rx and tx states are displayed separately. Otherwise, only a single state is displayed for both rx and tx states.

For example:

```
TIME      INSTANCE                R00    |    T00    ->    LTSSM State
-------------------------------------------------------------------------------------
```

```
   208:  root0                z    |    z    ->  initializing
   208:  endpoint0            z    |    z    ->  initializing
   212:  root0                z    |    z    ->  initializing
   212:  endpoint0            z    |    z    ->  initializing
root0                -- Detected change in link width from 1 to 4.

TIME     INSTANCE           R00  R01  R02  R03  |  T00  T01  T02  T03  ->  LTSSM State
-------------------------------------------------------------------------------------------
   216:  root0                z    z    z    z  |  z    z    z    z    ->  Det.Quiet
endpoint0            -- Detected change in link width from 1 to 4.

TIME     INSTANCE           R00  R01  R02  R03  |  T00  T01  T02  T03  ->  LTSSM State
-------------------------------------------------------------------------------------------
   216:  endpoint0            z    z    z    z  |  z    z    z    z    ->  Det.Quiet
   220:  root0                z    z    z    z  |  z    z    z    z    ->  Det.Quiet
   220:  endpoint0            z    z    z    z  |  z    z    z    z    ->  Det.Quiet
                     < symbol continues >
26840:  endpoint0            00   00   00   00 |  a1   75   47   aa   ->  tx = L0, rx = L0
26841:  root0                a1   75   47   aa |  00   00   00   00   ->  tx = L0, rx = L0
26841:  endpoint0            00   00   00   00 |  5b   2c   cd   17   ->  tx = L0, rx = L0
```

5.2.1.1 Configuration Messages in the Symbol Log

In addition to lane symbols, messages that indicate link changes are displayed in the symbol log. These messages are prepended by "--".

For example:

endpoint0 -- Detected change in link width from 1 to 4.

root0 -- Detected change in data rate to 8 Gt/s. See file header for special encodings.

5.2.2 Synchronization of Simulation Time Between Transaction Log and Symbol Log

Transaction logging times represent times at the periphery of the VIP. Symbol logging times are captured at the PIPE interface, which may be internal or external to the VIP depending on the interface type.

For Serial and PMA interface, there is no correlation between the time displayed in the transaction log and the symbol log. Due to delay through the PHY layer, symbols are logged at a different time than the transaction log for the same packet.

For PIPE interface, the simulation time displayed in the transaction log and symbol log are synchronized for the same packet. The "Start Time" of the transaction log corresponds to the time of a transaction with the "STP" or "SDP" symbol in the symbol log. The "End Time" of the transaction log corresponds to the time of a transaction with the "END" symbol in the symbol log.

Transaction Log	Symbol Log
Start Time (ns)	TIME with "STP" or "SDP" symbol

Transaction Log	Symbol Log
End Time (ns)	TIME with "END" symbol

Example 1:

Transaction Log

```
vip0    16332.00   16336.00   T   INITFC2_P_VC0                101   1025                0xb467
```

Symbol Log

```
16332:  vip0    00  00  00  00  |  SDP  c0  19  84   ->  tx = L0, rx = L0
16336:  vip0    00  00  00  00  |  00   b4  67  END  ->  tx = L0, rx = L0
```

Example 2:

Transaction Log

```
vip0    16344.00   16476.00   T   MWr32    0x0001/10    0  0  0  0  0  0  0x797ffe94    7 f
                                                29   H4000001d   H0001107f   H797ffe94      ---      0   0x88146f08
                                                 0   ee4f36e1    f7f2dc7e    2409961a    6a0e183a
                                                 4   d39af0b3    7c9a4388    6143237c    ec6e36a2
                                                 8   c56daf5b    05b6af19    49481dfc    036a8986
                                                12   70425548    b95aea17    91d4a8af    8d575fcc
                                                16   cbf0a790    ca6403af    196ebb7a    ff400faf
                                                20   67310374    6745f3f3    677e58c8    09bcfe06
                                                24   03223d90    ab52c830    df33cf23    700a0f1f
                                                28   b40b4d71      ---         ---         ---
```

Symbol Log

```
16344: vip0    00  00  00  00  |  STP  00  00  40   -> tx=L0, rx=L0
16348: vip0    00  00  00  00  |  00   00  1d  00   -> tx=L0, rx=L0
16352: vip0    00  00  00  00  |  01   10  7f  79   -> tx=L0, rx=L0
16356: vip0    00  00  00  00  |  7f   fe  94  ee   -> tx=L0, rx=L0
16360: vip0    00  00  00  00  |  4f   36  e1  f7   -> tx=L0, rx=L0
16364: vip0    00  00  00  00  |  f2   dc  7e  24   -> tx=L0, rx=L0
                < symbol continues >
16456: vip0    00  00  00  00  |  22   3d  90  ab   -> tx=L0, rx=L0
16460: vip0    00  00  00  00  |  52   c8  30  df   -> tx=L0, rx=L0
16464: vip0    00  00  00  00  |  33   cf  23  70   -> tx=L0, rx=L0
16468: vip0    00  00  00  00  |  0a   0f  1f  b4   -> tx=L0, rx=L0
16472: vip0    00  00  00  00  |  0b   4d  71  88   -> tx=L0, rx=L0
16476: vip0    00  00  00  00  |  14   6f  08  END  -> tx=L0, rx=L0
```

5.3 Using Native Protocol Analyzer for Debugging

5.3.1 Introduction

This feature enables you to invoke Protocol Analyzer from Verdi GUI. You can synchronize the Verdi wave window, smart log and the source code with the Protocol Analyzer transaction view. Protocol Analyzer can be enabled in an interactive and post-processing mode. The 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.

The VIP supports the Synopsys Protocol Analyzer (PA) tool, which is an interactive graphical application which provides protocol-oriented analysis and debugging capabilitiesallows. It allows users to view transactions, TLPs, DLLPs, ordered sets and the LTSSM state machine graphically. A transaction is made up of one request TLP and if necessary one or more completion TLPs required to complete that transaction.

The Protocol Analyzer tool supports the following PCIe features:

- TLPs
  - Request
  - One or more completions, as per protocol
- DLLPs
- Ordered Sets
- LTSSM state

5.3.2 Enabling the Protocol Analyzer

To enable protocol analyzer define the macro 'SVT_PCIE_INCLUDE_AC_PA' at compile time and set following configurations in svt_pcie_configuration class:

- enable_tl_xml_gen. Protocol file generation for the Transaction Layer
- enable_pl_xml_gen. Protocol file generation for the Physical Layer
- enable_dl_xml_gen. Protocol file generation for the Data Link Layer

The default value of each of these variable is 0, which means that protocol file generation is disabled by default.

To enable protocol file generation for a layer, set the value of the protocol generation enabling variable for that layer to 1 in the port configuration of each master or slave for which protocol file generation is desired.

The next time that the environment is simulated, protocol files will be generated according to the port configurations. The protocol files will be in XML format. Import these files into the Protocol Analyzer to view the protocol transactions.

5.3.3 Prerequisites

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

5.3.4 Compile-Time Options

The following compile-time options must be enabled:

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

  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*.

5.3.5 Run Time Options

You can set the format type for FSDB dump either through simulator command-line option or via a configuration setting.

5.3.5.1 Configuration Setting

Set the svt_pcie_configuration::pa_format_type variable to FSDB.

```
< svt_pcie_configuration>.pa_format_type=svt_xml_writer::FSDB
```

The XML type is in the process of being deprecated.

5.3.5.2 Command Line Option

The following svt_enable_pa runtime switch can be provided to your simulator:

```
+svt_enable_pa=FSDB
```

Enables FSDB output of transaction and memory information for display in Verdi.

5.3.6 Invoking Protocol Analyzer

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

5.3.6.1 Post-Processing Mode

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

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

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

5.3.6.2 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.

5.3.7 Limitations

Interactive support is available only for VCS.

5.4 Verification Planner

The PCIe VIP provides verification plans which can be used for tracking verification progress of the PCIe protocol. A set of top-level plans and sub-plans are provided. The verification plans are available at:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/VerificationPlans

For more information, refer to the README file, which is available at:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/VerificationPlans/README

5.5 Global Shadow Memory

The purpose of this global shadow memory is to provide a database of your DUT's PCI Express address space (memory and configuration space) for Write and Read transaction checking. The Global PCIe Address Space Shadow is an optional component which you can instantiate in your test environment. Returned data can be compared with the shadow copy to verify that the DUT did the original write and the read correctly.

The model captures the following data in Shadow Memory:

- Memory Writes
- Atomic Operations.
- Memory Reads The host memory must have the IN_ORDER attribute set. When the completion for that read comes back, this saved snapshot (and not the current state of the memory) will be used in the shadow comparison. Note however, the PCIe generally does not guarantee In Order write/read behavior.
- Configuration Writes. With this data you can determine various behaviors of the DUT.

The model does *not* capture the following data in Shadow Memory:

- Error Injections - we assume that an EI will cause failure of the transaction to do what it intended.
- Poisoned (EP bit) Atomic Ops - these should fail (other poisoned transactions may or may not fail - this is implementation dependent.)
- Any TLP determined to be badly formed.
- Type 1 Cfg requests (these are destined for switches, not endpoints)

The VIP captures TLPs as they go on the wire and then are passed to the shadow's transaction handler which decodes all relevant transactions and updates the expected global shadow state. For example, outbound MemWrite TLPs are inspected and (if appropriate) written to the global shadow memory. This shadow memory can be used by any checkers to allow comparisons with actual completion data (from the CplD TLP) and the expected completion data (from the global shadow memory.)

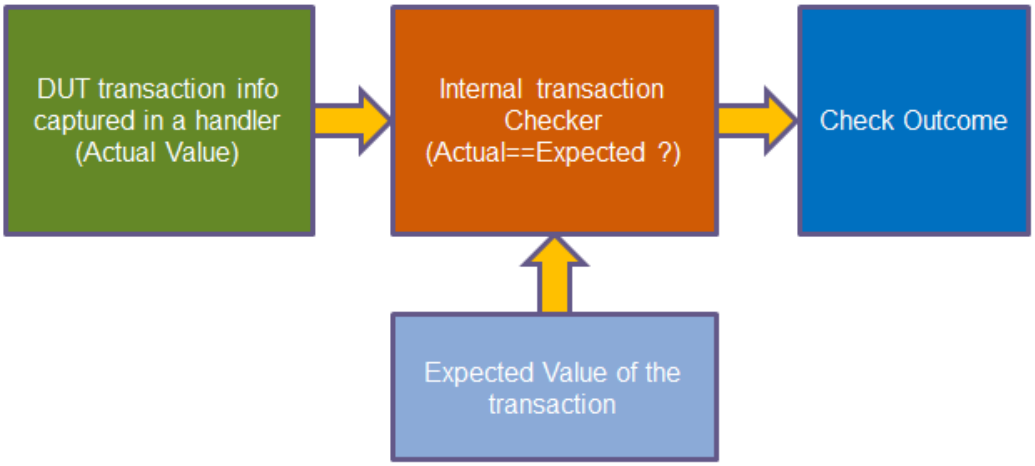
If instantiated and enabled, the Global Shadow is used by both the driver and the requester to automatically verify the results of the memory and configuration accesses made to the DUT. Memory transactions are captured and recorded in the Memory Shadow; no initial configuration of the Global System Shadow is required in most cases.

The shadow also provides for "ignored" regions of the memory. Any ignored regions will return an attribute to this effect when an application queries the shadow for expected read results. This way a checker can determine if the transaction is expected to return a predicable result or not. Occasionally there will be

memory regions (e.g. registers) that you do not want to check for correctness – write-only registers, status or statistics registers that may change sporadically, etc. T

The memory shadow has two ordering modes, the normal *non-ordered* mode, as well as a mode for strict transaction ordering. Strict ordering is only for use with DUTs that will **not** reorder any inbound transactions. The default mode is to not assume any ordering (which is normal per the PCI Express rules).

The following illustration shows an usage example:



Global shadow needs to be explicitly declared and instantiated at the top when used:

```
`define EXPERTIO_PCIESVC_GLOBAL_SHADOW_PATH  test_top.global_shadow0
pciesvc_global_shadow #( .DISPLAY_NAME( "global_shadow0." ) ) global_shadow0();
```

5.5.1 Global Shadow Memory Classes

There are two classes for using Global Shadow Memory:

1. **svt_pcie_global_shadow** . This class is OVM Driver that implements a PCIE application namely Global Shadow. It provides a SIPP [Sequence Item Pull Port] to cater to services of type svt_pcie_global_shadow_service.
2. **svt_pcie_global_shadow_service.** This class represents service transactions for a PCIE Global Shadow Memory Application. The service_type attribute is the entry point to this object.

5.5.1.1 Class svt_pcie_global_shadow Members

The following table lists important members of the svt_pcie_global_shadow. It is derived as a ovm_component.

Table 5-4 Service Class Features for Global Shadow Memory

Member	Feature/Usage
ADD_MEM_RANGE(`SVT_PCIE_GLOBAL_SHADOW_SERVICE_ADD_MEM_ RANGE)	Add supported memory range.

Table 5-4 Service Class Features for Global Shadow Memory (Continued)

Member	Feature/Usage
REMOVE_MEM_RANGE(`SVT_PCIE_GLOBAL_SHADOW_SERVICE_REMOVE_ MEM_RANGE)	Remove supported memory range.
WRITE_MEM(`SVT_PCIE_GLOBAL_SHADOW_SERVICE_WRITE_MEM)	Memory write.
READ_MEM(`SVT_PCIE_GLOBAL_SHADOW_SERVICE_READ_MEM)	Memory read.
WRITE_CFG(`SVT_PCIE_GLOBAL_SHADOW_SERVICE_WRITE_CFG)	Configuration write.
READ_CFG(`SVT_PCIE_GLOBAL_SHADOW_SERVICE_READ_CFG)	Configuration read.
WRITE_CFG_CAP(`SVT_PCIE_GLOBAL_SHADOW_SERVICE_WRITE_CFG_CA P)	Cfg Cap write.
READ_CFG_CAP(`SVT_PCIE_GLOBAL_SHADOW_SERVICE_READ_CFG_CA P)	Cfg Cap read.
TRANSACTION_COMPLETE(`SVT_PCIE_GLOBAL_SHADOW_SERVICE_TRANSACTION_ COMPLETE)	Transaction completed.
rand bit [63:0] address	Address to be read from or to be written to Global Shadow Memory.
rand bit [31:0] attributes	Attributes for ADD/REMOVE_MEM_RANGE service types.
rand bit [15:0] bdf	Bus-Device-Function for the device cfg reg in Global Shadow cfg.
rand bit [3:0] byte_enables	Byte Enables indicating enabled bytes during read/write of Global shadow memory.
svt_pcie_device_configuration cfg	Configuration pointer used to improve methods and constraints
rand bit [7:0] cfg_cap	The specific configuration-capability being read.
rand bit [31:0] cfg_reg	The specific configuration register being written in Global shadow cfg.
rand bit [31:0] data	Data to be written to Global Shadow Memory during WRITE service. When service_type is READ, it represents the data read.

Table 5-4 Service Class Features for Global Shadow Memory (Continued)

Member	Feature/Usage
rand bit [31:0] data_mask	Asserted bits indicate which bits of above data argument will be modified in the configuration-register (remaining bits will be unchanged.)
rand bit [31:0] dword_offset	Dword Offset indicating offset to the pcie_start_address. Valid offsets are 0 - data_length_in_dwords - 1.
bit error	Error indication if ADD/REMOVE_MEM_RANGE service types fail.
max_range	Upper address of the address range for ADD_MEM_RANGE and REMOVE_MEM_RANGE service types.
min_range	Lower address of the address range for ADD_MEM_RANGE and REMOVE_MEM_RANGE service types.
service_type	Memory target services
svt_sequence_item :: status_enum status	Status information about the current processing state
bit [31:0] task_status	Returns the status of READ/WRITE services.
rand bit [31:0] transaction_id	The combination of RequesterID and Tag for the transaction to cleanup.

5.5.2 Global Memory Examples

5.5.2.1 Random Memory Read using Global Shadow

User instantiates the requisite global shadow sequence in his parent sequence and reads data.

```
task user_parent_seq :: body()
…..
svt_pcie_global_shadow_service_random_rd_wr_sequence rand_rd_wr_seq;
…..

`ovm_config_db(mem_range_seq,p_sequencer.endpoint_virt_seqr.global_shadow,
{rand_rd_wr_seq == svt_pcie_global_shadow_service::    ;
 rand_rd_wr_seq. == 5; } );
```

5.5.2.2 Display Statistics of Global Shadow

Considering VIP as Endpoint. User instantiates the requisite global shadow sequence in his parent sequence and gets statistics.

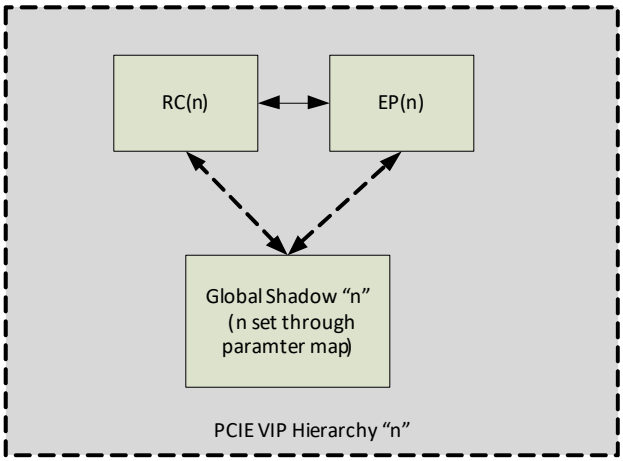
```
task user_parent_seq :: body()
          …..
svt_pcie_global_shadow_service_disp_stats_sequence     disp_stat_seq;
```

```
`ovm_config_db(mem_range_seq,p_sequencer.endpoint_virt_seqr.global_shadow,
{disp_stat_seq.service_type == svt_pcie_global_shadow_service::DISPLAY_STATS;} );
```

5.5.3 Multiple Global Shadows

Figure 5-1 shows a virtual hierarchy created by the physical connection of RC and EP and the virtual connection to the associated shadow. This feature is implemented in the model via the parameter HIERARCHY_NUMBER in the shadow and all of the instantiation models.

Figure 5-1 PCIe VIP Hierarchy Definition



5.5.3.1 Setting up Multiple Shadows

5.5.3.1.1 Module Level Construction

Instantiate a desired number of shadows (one for each PCIe VIP hierarchy described earlier). Relative location of the pciesvc_global_shadow instances to the device VIP instances in the module hierarchy is not important. The value of the associated HIERARCHY_NUMBER setting between the root, global shadow, and downstream endpoints is crucial.

```
// Global Shadow Instances ( optional )
pciesvc_global_shadow#(.DISPLAY_NAME({DISPLAY_NAME, "global_shadow0." }),
.HIERARCHY_NUMBER(0)) global_shadow0();
pciesvc_global_shadow#(.DISPLAY_NAME({DISPLAY_NAME, "global_shadow1." }),
.HIERARCHY_NUMBER(1)) global_shadow1();
…
// PCIE VIP Hierarchy 0
<pcie RC module>#(.DISPLAY_NAME(…), .HIERARCHY_NUMBER(0),…) <inst name> …
<pcie EP module>#(.DISPLAY_NAME(…), .HIERARCHY_NUMBER(0),…) <inst name> …
// PCIE VIP Hierarchy 1
<pcie RC module>#(.DISPLAY_NAME(…), .HIERARCHY_NUMBER(1),…) <inst name> …
<pcie EP module>#(.DISPLAY_NAME(…), .HIERARCHY_NUMBER(1),…) <inst name> …
```

5.5.3.1.2 SVT Global Shadow Component Instantiation

You must modify construction of the shadow agent within environments by replacing svt_pcie_global_shadow::type_id::create() with the specialized method svt_pcie_global_shadow::create_shadow() (see Table 5-5). This modification is required when you want to use multiple shadows. The hierarchy_number argument of create_shadow() must match the HIERARCHY_NUMBER parameter value of the corresponding pciesvc_global_shadow module instance.

Example instantiation of two svt_pcie_global_shadows linked to the corresponding global shadow modules 0 an 1 from the previous module instantiation:

```
global_shadow[0] = svt_pcie_global_shadow::create_shadow(0, "global_shadow[0]", this);
global_shadow[1] = svt_pcie_global_shadow::create_shadow(1, "global_shadow[1]", this);
```

Table 5-5 Multiple Shadow Support Functions

New Method	Arguments	Description
create_shadow	int hierarchy_number, // Global Shadow Hierarchy Number string name="svt_pcie_global_shadow", // Name of returned component ovm_component parent = null // Parent component	This is a function that will look up the specific shadow with the predetermined string format, then create and return the component. Similar to ::create() but will not concatenate parent and name as the lookup string. Accepts same arguments as create for "name" and "parent".
get_hierarchy_number		Returns the hierarchy number of the global shadow agent

5.5.4 Disabling Global Shadows

If no shadows are instantiated, a proper disable settings in the device configuration must be employed. If you have disabled a global shadow by not defining `EXPERTIO_PCIESVC_GLOBAL_SHADOW_PATH, then you might receive error messages (`OVM_WARNING/ERROR/FATAL`) because shadow is not available. The proper way to disable use of the global shadow is to set the proper svt_pcie_device_configuration options:

```
<device_cfg>.driver_cfg[0].enable_tx_tlp_reporting = 0;
<device_cfg>.driver_cfg[0].enable_shadow_memory_checking = 0;
<device_cfg>.requester_cfg.enable_tx_tlp_reporting = 0;
<device_cfg>.requester_cfg.enable_shadow_memory_checking = 0;
<device_cfg>.pcie_cfg.dl_cfg.enable_tx_tlp_reporting = 0;
<device_cfg>.pcie_cfg.tl_cfg.enable_shadow_cfg_lookup = 0;
```

At this point, whether the shadow is removed physically or by not defining `EXPERTIO_PCIESVC_GLOBAL_SHADOW_PATH, no message is reported.

5.6 Target Memory

All PCIe devices in the system may have memory that is accessible by writes and/or reads to/from particular memory addresses. The VIP memory is a *sparse* model, allowing a wide variety of addresses (32 and 64-bit) to be accessed by a requester. The memory is divided into pages to increase performance, match up to PCIE packets and take advantage of locality-of-reference.

There are three page-sizes available for allocating pages such that the memory buffers are neither so large that memory is wasted nor are they inefficiently small.

The basic tasks are simply Write and Read, each providing dword-sized accesses via a supplied 32 or 64-bit PCIe address.

The three page sizes are simply: Large, Small and Dword. Since all accesses to the memory target are Dword at a time, it's difficult to predict the transaction's total data length; therefore a "hint" style reservation mechanism is used to allocate pages. When a PCIE memory write is intended, the client can call the task

PREWRITEHINT() to request reservation of adequately sized pages. When the following Write() is called, the reserved pages are then used to efficiently store the data.

Note that it is up to the particular application using the target memory to choose optimal small and large page sizes based on their use-model. These applications are generally tuned to cooperate optimally with the DUT; similarly the memory target should be tuned to work with these applications. The task *DISPLAY_STATS()* can be useful in showing the allocated memory sizes during a simulation to help out in choosing optimal page sizes.

This memory is also used as the storage mechanism for the Global Shadow Memory (see Global Shadow Memory for details).

The service class *svt_pcie_mem_target_service* class represents service transactions for a PCIe Memory Target Application. The service_type attribute is the entry point to this object. The following table shows the various memory target service types.

Table 5-6 Memory Target Service Types

Service	Description
WRITE(`SVT_PCIE_MEM_TARGET_SERVICE_WRITE)	Memory write of single DWORD to Memory space of Target application. NOTE: This service call will be obsoleted in a future release.
READ(`SVT_PCIE_MEM_TARGET_SERVICE_READ)	Memory read of single DWORD to Memory space of Target application. NOTE: This service call will be obsoleted in a future release.
PRE_WRITE_HINT(`SVT_PCIE_MEM_TARGET_SERVICE_PRE_WRITE_HINT)	Add Pre-write hint to the memory. NOTE: This service call will be obsoleted in a future release.
ADD_MEM_RANGE(`SVT_PCIE_MEM_TARGET_SERVICE_ADD_MEM_RANGE)	Add supported memory range.
REMOVE_MEM_RANGE(`SVT_PCIE_MEM_TARGET_SERVICE_REMOVE_MEM_RANGE)	Remove supported memory range.
DISPLAY_STATS(`SVT_PCIE_MEM_TARGET_SERVICE_DISPLAY_STATS)	Display statistics variables.
CLEAR_STATS(`SVT_PCIE_MEM_TARGET_SERVICE_CLEAR_STATS)	Clear statistics variables.
RESET_APP(`SVT_PCIE_MEM_TARGET_SERVICE_RESET_APP)	Resets app back to its inital state. All will be lost.
WRITE_BUFFER(`SVT_PCIE_MEM_TARGET_SERVICE_WRITE_BUFFER)	Memory write of multiple DWORDs to Memory space of Target application.
READ_BUFFER(`SVT_PCIE_MEM_TARGET_SERVICE_READ_BUFFER)	Memory read of multiple DWORDs to Memory space of Target application.

A service call is used to mark a memory region as ignored. As such, the memory region will not have memory allocated for it. Read requests will be returned with random data. Write requests will have no affect.

```
/**< Add supported memory range.*/
ADD_MEM_RANGE = `SVT_PCIE_MEM_TARGET_SERVICE_ADD_MEM_RANGE,

/**< Remove supported memory range.*/
REMOVE_MEM_RANGE = `SVT_PCIE_MEM_TARGET_SERVICE_REMOVE_MEM_RANGE,

..........
//---------- Variables for ADD_MEM_RANGE and REMOVE_MEM_RANGE services ----------

  /**
   *  Lower address of the address range for ADD_MEM_RANGE and
   *  REMOVE_MEM_RANGE service types.
   */
  rand bit [63:0]      min_range = 'h0;

  /**
   *  Upper address of the address range for ADD_MEM_RANGE and
   *  REMOVE_MEM_RANGE service types.
   */
  rand bit [63:0]      max_range = 64'hFFFF_FFFF_FFFF_FFFC;

  /**
    * Attributes for ADD/REMOVE_MEM_RANGE service types.
    * - attributes[0]: Ignore. Assert this bit in the attributes to
    * disallow checking against this address range.
    * - attributes[1]: In Order. Assumes transaction will be handled in the DUT the
    * order that they were received, this provides the potential for checking some
    * alternating read/write transactions.
    */

  rand bit [31:0]      attributes = 32'h0;
```

5.6.1 Ignoring Memory Ranges

The service call `svt_pcie_mem_target_service::ADD_MEM_RANGE` (along with `REMOVE_MEM_RANGE`) are used to configure the memory range and ignored attribute. The sequence class `svt_pcie_mem_target_service_mem_range_sequence` provides access to this mechanism.

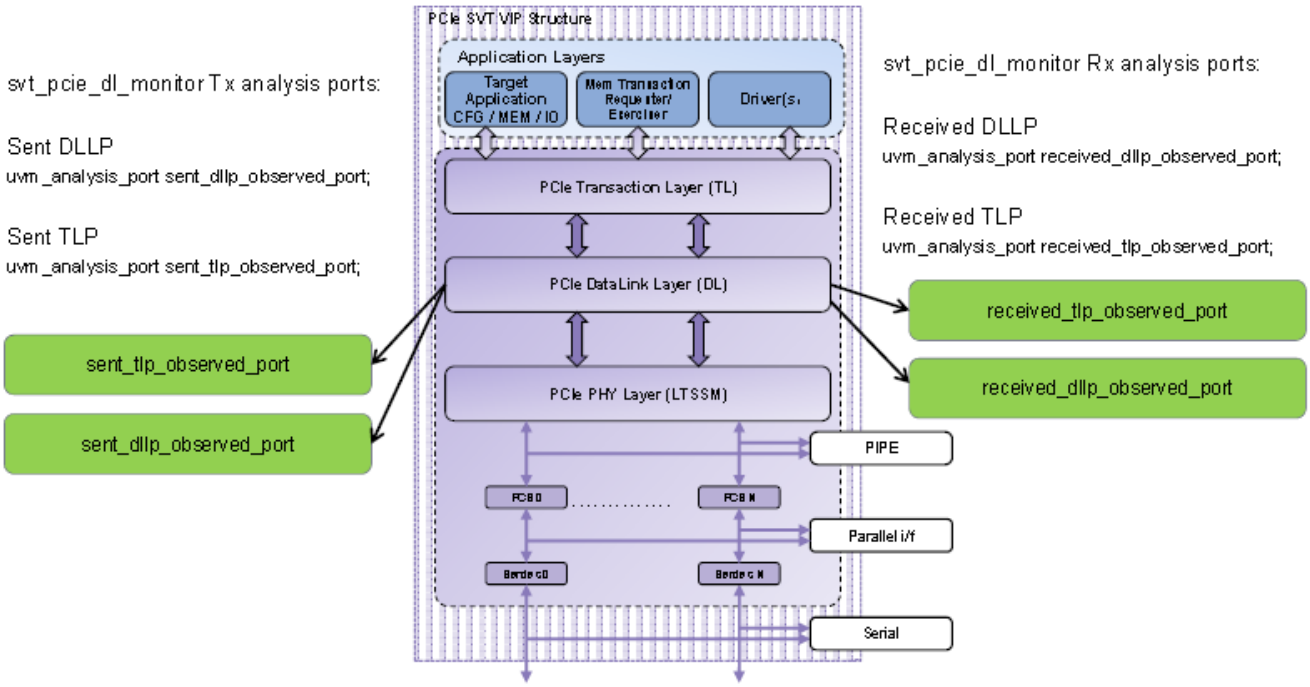
The configuration in the class `svt_pcie_target_app_configuration` for this is `uninit_mem_read_resp`, which can be set with various `UNINIT_MEM_READ_RESP_*` values.

The shadow also provides for ignored regions of the memory. Any ignored regions will return an attribute to this effect when an application queries the shadow for expected read results. In this way, a checker can determine if the transaction is expected to return a predicable result or not. Occasionally, there will be memory regions (for example, registers) that you do not want to check for correctness—write-only registers, status or statistics registers that may change sporadically, and so on. These regions in the shadow memory can be marked as `IGNORED` via the `AddMemRange()` task, which is identical as above, but is accessed via the `` `EXPERTIO_PCIESVC_GLOBAL_SHADOW_PATH `` define.

5.7 Data Link Monitor

The VIP has a Data Link Monitor which is used to indicate when TLPs and DLLP are sent and received. Figure 5-2 shows the various analysis ports for monitoring TLPs an DLLPs.

Figure 5-2 Data Link Monitor and Monitor Ports and Classes



The PCIe OVM VIP has the following TLM analysis ports in the DL to access sent/received TL packets.

- svt_pcie_dl::received_tlp_observed_port: Analysis port for to sample TLPs being received by the VIP. This port is generally used for scoreboarding.
- svt_pcie_dl::sent_tlp_observed_port: Analysis port for to sample TLPs being sent by the VIP. This port is generally used for scoreboarding.

The TLPs observed via these ports are controlled by a configuration variables in the DL namely:

- svt_pcie_dl_configuration::received_tlp_interface_mode
- svt_pcie_dl_configuration::sent_tlp_interface_mode.

For example:

```
endpoint_cfg.pcie_cfg.dl_cfg.received_tlp_interface_mode = 1;
endpoint_cfg.pcie_cfg.dl_cfg.sent_tlp_interface_mode = 1;
```

These configuration parameters are 2-bit variables. Bit 0 corresponds to the enabling of “good” packets and bit 1 corresponds to the enabling of “error” packets. Check the HTML class description for more details:

- *$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/configuration/class_svt_pcie_dl_configuration.html#item_received_tlp_interface_mode*
- *$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/configuration/class_svt_pcie_dl_configuration.html#item_sent_tlp_interface_mode*

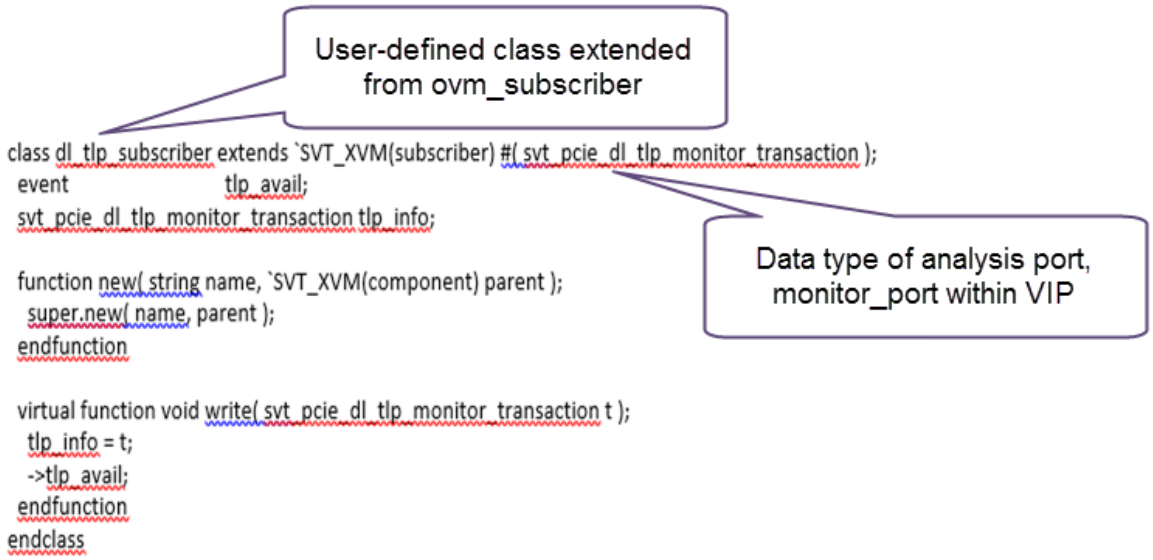
Once enabled these ports can be used to subscribe to transaction being sent/received by the VIP model with the use of OVM subscribers. The code example below illustrates the same.

Use the following flow to setup the DL Monitor.

1. Identify the analysis port on the DL monitor
2. The sent_tlp_observed_port will tell you when the TLP was sent, along with providing the TLP transaction class
3. Note the class name: svt_pcie_dl_monitor
4. Create a ovm_subscriber extension

   Goal: ovm_subscriber::write() will be called when the TLP is sent
5. Add a subscriber to your ovm_test
   a. Build phase : new your ovm_subscriber class
   b. Connect phase : connect to the ovm_analysis, monitor port in the class identified above
6. Done, when the TLP is sent, the write() method will be called.

The example which follows is annotated to explain the use of the DL Monitor following the previous steps.




```
class dl_tlp_subscriber extends `SVT_XVM(subscriber) #( svt_pcie_dl_tlp_monitor_transaction );
 event                    tlp_avail;
 svt_pcie_dl_tlp_monitor_transaction tlp_info;

 function new( string name, `SVT_XVM(component) parent );
  super.new( name, parent );
 endfunction

 virtual function void write( svt_pcie_dl_tlp_monitor_transaction t );
  tlp_info = t;
  ->tlp_avail;
 endfunction
endclass
```

```
class dl_monitors extends pcie_device_system_test_base;

 tlp_subscriber sent_tlp_subscriber;
.....
.....

 /**
  * build_phase: To build various component of class compile_test
  */

 virtual function void build();
  super.build();

  ..... |

  sent_tlp_subscriber = new( "sent_tlp_subscriber", this );

 .....

 endfunction
```

Add subscriber to your ovm_test implementation

Within the build, create an instance of your subcriber

```
virtual function void connect();
 super.connect();
 env.root.port.dl.sent_tlp_observed_port.connect( sent_tlp_subscriber.analysis_export );
endfunction
```

"Connect" to the monitor within the VIP

Use analysis_export to make the connection

```
OVM_INFO ./ts.pseudo_random_test.sv(76) @ 81793300.10 ps:
ovm_test_top.tlp_subscriber dl_tlp_subscriber: A new sent TLP available
```

from log

6 PCIe Verification Topologies

Attention Support for legacy instantiation models has been deprecated from this release. It is recommended to migrate to Unified VIP single instance model. For more details, see Chapter 10 **PCIe Verification Topologies** in *PCIe SVT UVM User Guide*.

7 Using the PCIe Verification IP

This chapter discusses the following topics:

- Up/Down Configure
- Lane Reversal
- Lane Reversal with Different Link Width Configurations
- User-Supplied Memory Model Interface
- External Clocking and Per Lane Clocking for Serial Interface

7.1 SystemVerilog OVM Example Testbenches

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

Table 7-1 SystemVerilog Example Summary

Example Name	Level	Description
`tb_pcie_svt_ovm_unified_vip_sys`	Advanced	The example consists of the following: • A top-level module, which includes tests, instantiates interfaces, HDL interconnect wrapper, generates system clock, and runs the tests • A base test, which is extended to create a directed and a random test • The tests create a testbench environment, which in turn creates PCIe System Env • PCIe System Env is configured with Root Complex and one Endpoint

The examples are located at:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/examples/sverilog/

Examples may be installed in your local design directory following the instructions in the installation chapter.

The tests in the unified example (`tb_pcie_svt_ovm_unified_vip_sys`) are:

- `ts.base_multi_link_test.sv`
- `ts.base_pipe5_gen6_full_eq_in_serdes_arch_mode_test.sv`
- `ts.base_pipe5_gen6_in_serdes_arch_mode_test.sv`
- `ts.base_pipe5_in_serdes_arch_mode_test.sv`
- `ts.base_pipe5_test.sv`
- `ts.base_pipe_test.sv`
- `ts.base_pma_test.sv`
- `ts.base_serdes5_test.sv`
- `ts.base_serdes_test.sv`

7.2 Installing and Running the Examples

Below are the steps for installing and running example, `tb_pcie_svt_ovm_unified_vip_sys`. Similar steps are applicable for other examples:

1. Install the example using the following command line:

```
% cd <location where example is to be installed>
% mkdir design_dir <provide any name of your choice>
% $DESIGNWARE_HOME/bin/dw_vip_setup -path ./design_dir  -e
    pcie_svt/tb_pcie_svt_ovm_unified_vip_sys -svtb
```

   This installs the example under:

```
<design_dir>/examples/sverilog/pcie_svt/tb_pcie_svt_ovm_unified_vip_sys
```

2. Use either one of the following to run the testbench:

   a. Use the Makefile:

      The following tests are provided in the "tests" directory:

      - `ts.base_multi_link_test.sv`
      - `ts.base_pipe5_gen6_full_eq_in_serdes_arch_mode_test.sv`
      - `ts.base_pipe5_gen6_in_serdes_arch_mode_test.sv`
      - `ts.base_pipe5_in_serdes_arch_mode_test.sv`
      - `ts.base_pipe5_test.sv`
      - `ts.base_pipe_test.sv`
      - `ts.base_pma_test.sv`
      - `ts.base_serdes5_test.sv`
      - `ts.base_serdes_test.sv`

      To run the `ts.base_serdes_test.sv` test, for example, do following:

```
gmake USE_SIMULATOR=vcsvlog base_serdes_test WAVES=1
```

      To see more options, invoke "gmake help".

   b. Use the sim script:

      To run the `ts.base_pipe_test.sv` test, for example, do following:

```
./run_pcie_svt_ovm_unified_vip_sys -w base_pipe_test vcsvlog
```

      To see more options, invoke "`./run_pcie_svt_ovm_unified_vip_sys -help`".

For more details about installing and running the example, refer to the README file in the example, located at:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/examples/sverilog/tb_pcie_svt_ovm_unified_vip_sys/README

or

<design_dir>/examples/sverilog/pcie_svt/tb_pcie_svt_ovm_unified_vip_sys/README

7.3 Error Message Usage

The control of check and error messages from the model is handled through the svt_err_check class instance "err_check" within the agent. class. Following is a message you might see from the model.

```
OVM_ERROR svt_err_check_stats.sv(817) @ 226895734.80 ps:
ovm_test_top.env.root.port0.dl0
[register_fail:AC_DL:PROTOCOL:dl_receive_nullified_tlp_lcrc]  - Received TLP with EDB
delimiter but bad LCRC = 0x8fb52aea, expected LCRC = 0x8fb52ae9
```

The main components of the message are:

- **Reporter**: "ovm_test_top.env.root.port0.dl0"
- **ID**: "register_fail:AC_DL:PROTOCOL:dl_receive_nullified_tlp_lcrc"
- **Message**: "Received TLP with EDB delimiter but bad …"

The check interface gives you the ability to change how the message is issued based on the unique message ID. For example, you can write the following code to demote the ERROR message to a NOTE/OVM_INFO message:

```
env.root.err_check.set_default_fail_effects("^AC_DL", "", svt_err_check_stats::NOTE,
"dl_receive_nullified_tlp_lcrc$");
```

The resulting output is now:

```
OVM_INFO svt_err_check_stats.sv(817) @ 226895734.80 ps: ovm_test_top.env.root.port0.dl0
[dl_receive_nullified_tlp_lcrc]  - Received TLP with EDB delimiter but bad LCRC =
0x8fb52aea, expected LCRC = 0x8fb52ae9
```

Users can change the behavior of the specified message by modifying the "effect" argument (argument three in the set_default_fail_effects() method), which is of type svt_err_check_stats::fail_effect_enum. The values available with the fail_effect_enum are:

- **IGNORE**. Ignore the check result.
- **VERBOSE**. Generate verbose message for the check results.
- **DEBUG**. Generate debug message for the check results.
- **NOTE**. Generate note message for the check results.
- **WARNING**. Generate warning message for the check results.
- **ERROR**. Generate error message for the check results.
- **EXPECTED**. Failure is expected.


Hint The NOTE, DEBUG, and VERBOSE settings equate to OVM_INFO verbosity settings of OVM_LOW, OVM_HIGH, and OVM_FULL respectively.

In addition to changing how messages are reported to you, the svt_err_check_stats class has additional features for you to track messages:

- **exec_count**. Tracks the number of times that a given check has been executed.
- **pass_count**. Tracks the number of times that a given check has PASSED.
- **fail_ignore_count** . Tracks the number of times the check has failed, with IGNORED effect.
- **fail_verbose_count**. Tracks the number of times the check has failed, with VERBOSE effect.

- **fail_debug_count**. Tracks the number of times the check has failed, with DEBUG effect.
- **fail_note_count**. Tracks the number of times the check has failed, with NOTE effect.
- **fail_warn_count**. Tracks the number of times the check has failed, with WARNING effect.
- **fail_warn_count**. Tracks the number of times the check has failed, with ERROR or FATAL effect.
- **fail_err_count**. Tracks the number of times the check has failed, with ERROR or FATAL effect.
- **fail_expected_count**. Tracks the number of times the check has failed, with EXPECTED effect.

To check the statistics count, get a handle to the stats container, using the same lookup strings as with

```
svt_err_check_stats check_stats = env.root.err_check.find("^DL$", "",
"^register_fail:DL:dl_receive_nullified_tlp_lcrc$");
```

The "check_stats.fail_note_count" would now be incremented by 1.

To disable all tracking of an ID (no statistics or coverage collection), which would supersede the above call, do the following:

```
env.root.err_check. disable_checks("^DL$", "",
"^register_fail:DL:dl_receive_nullified_tlp_lcrc$");
```

The "^" and "$" in the SVT call are the regex meta-character start/end string terminators respectively. The string arguments to the SVT err_check interface are regex expressions. Blanks are considered wildcards.

The "" argument in the previous example is the sub_group. The complete ID is register_fail:[<group>:][<sub_group>:]<unique_id>. The sub_group may or may not appear in all cases. You can use the meta-characters when trying to isolate specific groups, sub_groups, and IDs.

7.4 Some Configuration Values to Set

The following steps can help you get started in configuring the model.

1. SetAllocated Credits
   - TL: Set initial credits for a VC
   - Use: svt_pcie_tl_configuration
2. SetVCEnable
   - TL: Enable / Disable VCs; disabled by default
   - Use: svt_pcie_tl_service_set_vc_en_sequence
3. SetTrafficClassMap
   - TL: Setup TC map
   - Use: svt_pcie_tl_configuration
4. AddMemAddrAppIdMapEntry
   - TL: Sets AP ID for I/O target address range
   - Use: svt_pcie_mem_target_service_mem_range_sequence
5. SetLinkEnable
   - DL: Enable / Disable the DL
   - Use: svt_pcie_dl_service_link_en_sequence
6. SetSupportedSpeeds

- PL: Set supported speed for link training / default speed if link training is disabled
- Use: svt_pcie_pl_phy_configuration

7. SetLinkWidth
   - PL: maximum link width, also initiates LTSSM negotiation
   - Use: svt_pcie_pl_phy_configuration
8. SetLinkEnable
   - PL: Enable / Disable the PL link
   - Use: svt_pcie_pl_service_link_en_sequence

Example configuration code:

```
class pcie_shared_cfg extends ovm_object;
   rand svt_pcie_device_configuration root_cfg;
   …
   function new(string name = "pcie_shared_cfg");
      super.new(name);
      this.root_cfg = new("root_cfg");
       root_cfg.device_is_root = 1;
       root_cfg.pcie_spec_ver   = svt_pcie_device_configuration:PCIE_SPEC_VER_2_1;
       root_cfg.pcie_cfg.pl_cfg.link_width = 1;
       root_cfg.pcie_cfg.pl_cfg.target_speed = `SVT_PCIE_SPEED_2_5_G;
       root_cfg.pcie_cfg.pl_cfg.skip_polling_active = 1;
   endfunction : new
endclass : pcie_shared_cfg
```

Example to set credits.

```
class myTest extends ovm_test;
   …
    virtual function void build_phase(ovm_phase phase);
       super.build_phase(phase);
       // Setup Initial TX Credits
       cust_cfg.root_cfg.pcie_cfg.tl_cfg.init_p_hdr_tx_credits[0] = 104;
       cust_cfg.root_cfg.pcie_cfg.tl_cfg.init_p_data_tx_credits[0] = 1020;
       cust_cfg.root_cfg.pcie_cfg.tl_cfg.init_np_hdr_tx_credits[0] = 105;
       cust_cfg.root_cfg.pcie_cfg.tl_cfg.init_np_data_tx_credits[0] = 1021;
       cust_cfg.root_cfg.pcie_cfg.tl_cfg.init_cpl_hdr_tx_credits[0] = 106;
       cust_cfg.root_cfg.pcie_cfg.tl_cfg.init_cpl_data_tx_credits[0] = 1022;
      …
endclass
```

Example to enable the link:

```
class pcie_traffic_sequence extends svt_pcie_device_system_virtual_base_sequence;
  …
  task body();
      svt_pcie_dl_service_set_link_en_sequence link_en_seq;
      svt_pcie_pl_phy_service_set_phy_en_sequence phy_en_seq;
      …

        `ovm_do_on_with(link_en_seq, p_sequencer.root_virt_seqr.mac_virt_seqr.dl_seqr,
             {link_en_seq.enable == 1'b1;})
…
```

```
  endtask : body
endclass : pcie_traffic_sequence
```

7.5 OVM Reporting Levels

The OVM verbosity level for message logging is set using the +OVM_VERBOSITY runtime option. For each verbosity level, the Items that the OVM API prints to the log file are shown below.

- OVM_NONE – Only print error messages
- OVM_LOW – Print important messages that are not error messages
- OVM_MEDIUM – Drivers and monitors print transactions they transmit and receive
- OVM_HIGH – Print more detailed messages about component operation
- OVM_FULL – Print internal debug messages

7.6 Controlling Verbosity From the Command Line

The VIP can use all the OVM command line options for control of verbosity. The following table summarizes the options available. Please refer to the OVM Reference Guide for more details.

OVM Options are shown below.

Table 7-2 OVM Verbosity Options

OVM Option	Description
+OVM_VERBOSITY	setting for all components
+ovm_set_verbosity	Granular control by component and phase or time
+ovm_set_action	Action to take upon message (None, display, log, count, stop, exit, hook)
+ovm_set_severity	Severity override (upgrade or downgrade)

The remainder of this section will focus on verbosity control from the command line.

Globally

The PCIe VIP is compliant with the verbosity options within OVM. The ovm_cmdline_processor looks for the +OVM_VERBOSITY option on the simulator command line, and will set the initial verbosity for all OVM components to the supplied level.

Examples:

```
// Display only OVM_FATAL, OVM_ERROR, OVM_WARNING
simv +OVM_TESTNAME=base_pipe_test +OVM_VERBOSITY=OVM_NONE
// Display all messages
simv +OVM_TESTNAME=base_pipe_test +OVM_VERBOSITY=OVM_FULL
```

Per Component:

Also supported is the +ovm_set_verbosity which allows for more granular control. The command breakdown is as follows:

+ovm_set_verbosity=<component_name>, <id>, <verbosity>, <phase_name>

or

+ovm_set_verbosity=<component_name>, <id>, <verbosity>, time, <time>

Note, <id> can either be all, _ALL_ or a specific message id.

Example:

```
// Set all components to OVM_NONE except for the tl which is at OVM_LOW
simv +OVM_TESTNAME=base_pipe_test +OVM_VERBOSITY=OVM_NONE \
+ovm_set_verbosity=ovm_test_top.env.root.port0.tl0,_ALL_,OVM_LOW,time,0
// Set all components to OVM_NONE except for root
simv +OVM_TESTNAME=base_pipe_test +OVM_VERBOSITY=OVM_NONE \
+ovm_set_verbosity=ovm_test_top.env.root.*",_ALL_,OVM_LOW,time,0
```

Another option for component control which applies only to SVT based VIPs is the +vip_verbosity option:

Example:

```
// OVM_NONE for all with the exception that all instances of the dl are at OVM_LOW and
// all instances of the tl are at OVM_FULL
simv +vip_verbosity=svt_pcie_dl:OVM_LOW,svt_pcie_tl:OVM_FULL
```

SVT Verbosity relationship to OVM Severity Levels

`define SVT_FATAL_VERBOSITY OVM_NONE

`define SVT_ERROR_VERBOSITY OVM_NONE

`define SVT_WARNING_VERBOSITY OVM_NONE

`define SVT_NORMAL_VERBOSITY OVM_LOW

`define SVT_TRACE_VERBOSITY OVM_MEDIUM

`define SVT_DEBUG_VERBOSITY OVM_HIGH

`define SVT_VERBOSE_VERBOSITY OVM_FULL

7.7 Resetting the PCIe VIP

Reset to the VIP model is active high. Reset must be deasserted at time 0 such that a posedge is seen by the VIP. Reset should be asserted for at least 100 ns. Once deasserted, it should remain deasserted for the remainder of the simulation. DO NOT attempt to assert VIP reset to re-initialize the VIP. Do not attempt to configure the VIP until the reset has been deasserted.

To perform a reset of the DUT in mid-simulation, only the DUT should be reset. The VIP will detect the change on the bus and move to Detect. Training will resume and the bus will recover. Thus, for proper DUT verification, it is advisable to separate the SVC model reset from the DUT reset.

In a typical scenario, the following will happen:

1. Initialize and bring up link to stable after reset (normal config sequence)
2. Run traffic
3. Take down the link and check both IP/VIP are in link-down state
4. Bring link-up and check both IP/VIP are in a link-up state
5. Run traffic and verify all is well.

The svt_pcie_device_virtual_reset_sequence class will perform a mid simulation reset as described in the previous steps. This sequence implements Reset. This class resets the VIP by doing the following:

- Sets the hotplug mode to unplugged in the PL
- Calls RESET_APP on all of the applications.

Note that in order to start up the LTSSM again the user will have to use a PL service call to set the hotplug mode to HOTPLUG_DETECT

The model reset supports the following actions:

- Clears all of the Tx and Rx packet queues in the Transaction Layer. This deletes any completions in progress.
- Clears all packets queued in the driver_app, requester_app, target_app, and svt_pcie_tlp.
- All storage elements are cleared or garbage collected as appropriate.
- Deletes all packets that are in transit (for example, in the link layer or in the phy layer).
- Kills all completion timeouts in transaction layer
- Terminates compliance checks
- Resets the LTSSM back to its default initial state (Detect).

NOTE: No reconfiguration is required as the model will maintain its configuration information through the reset.

7.8 Creating and Using Custom Applications

Custom applications and test sequences can be developed for the PCIe VIP analogous to that of the Driver, Requester, and Target. You can create applications that enable specific functionality not available through the built-in applications. Custom applications allow the user to interface to the TL with TLPs enabling end-user specific functionality.

Applications examples include SRIOV and address translation. User applications are instantiated as classes in the SVT testbench. User applications that send TLPs should communicate with the VIP's tlp_seqr TLP sequencer using a TLP sequence that generates TLPs from svt_pcie_agent::tlp_seqr.

User applications co-exist and run in parallel to the Synopsys-supplied applications. Alternatively, testbenches can emulate user applications using sequences that generate TLPs with unique application IDs. The sequences that generate these TLPs run on the svt_pcie_agent::tlp_seqr sequencer.

The application_id attribute of the TLP objects generated by that application is identified by this value. The application id is available in svt_pcie_tlp::application_id. Application IDs in the range 0 – 19h are reserved for VIP internal use. The testbench should ensure that the application_id used by the applications are unique.

7.8.1 Setting Up Application ID Maps

For traffic to be directed between the MAC and the user application, a routing map must be set up. In particular, for the TLP routing to work, application IDs must be mapped to specific memory/IO address ranges, message codes and so on. This can be accomplished using the svt_pcie_tl_service transactions. A sequence of this type will run on the tl_seqr of the device agent, as shown in Example 7-1. Alternatively, the application_id to requester ID map is set up automatically by the VIP whenever a TLP with a specific RID and application ID is sent for the first time.

The following service transaction types can be used to set up application id maps:

ADD_MEM_ADDR_APPL_ID_MAP_ENTRY, ADD_IO_ADDR_APPL_ID_MAP_ENTRY
ADD_AT_ADDR_APPL_ID_MAP_ENTRY, ADD_RID_MSG_CODE_APPL_ID_MAP_ENTRY
ADD_RID_APPL_ID_MAP_ENTRY, ADD_CFG_BDF_APPL_ID_MAP_ENTRY

SEE the HTML reference documentation for more information on these service types.

Example 7-1

```
svt_pcie_tl_service add_mem_add_req;
`ovm_do_with(add_mem_add_req,{service_type ==
    svt_pcie_tl_service::ADD_MEM_ADDR_APPL_ID_MAP_ENTRY;
    appl_id == 32'h21;
    memory_addr == 0;
    memory_window == 32'h1000;})
```

7.8.2 Using Testbench Sequences to Emulate User Applications

Testbenches can add sequences that generate TLPs with unique application IDs to emulate user applications. A sequence is created that generates TLPs with unique application IDs and is run on the tlp_seqr of the PCIe agent contained in the PCI device agent. This sequencer can be accessed via the top-level virtual sequencer of type svt_pcie_device_system_virtual_sequencer:

```
`ovm_do_on(tlp_directed_seq, p_sequencer.root_virt_seqr.pcie_virt_seqr.tlp_seqr);
```

The output TLM port of this sequencer is internally connected to the sequence_item_port of the VIP's transaction layer.

The TLPs are set up with the appropriate application ID and requester ID and pushed into the seq_item_port, as indicated in Example 7-2.

Example 7-2

```
svt_pcie_tlp mem_rd_request
`ovm_create(mem_rd_request);
mem_rd_request.cfg = cfg;
mem_rd_request.tlp_type = svt_pcie_tlp::MEM_REQ;
mem_rd_request.fmt = svt_pcie_tlp::NO_DATA_3_DWORD;
mem_rd_request.length = 2 + i;
mem_rd_request.ep = 0;
mem_rd_request.at = svt_pcie_tlp::UNTRANSLATED;
mem_rd_request.first_dw_be = 4'b1111;
mem_rd_request.last_dw_be = 4'b1111;
mem_rd_request.application_id = 32'h21;
mem_rd_request.address = 32'h0000_4000 | ('h100 << i);
mem_rd_request.requester_id = 4;
`ovm_send(mem_rd_request)
```

7.8.3 Waiting for Completions

Completions are routed to the appropriate application_id using the application ID-to-requester ID map. Using code like the following the completions can be accessed from the rx_tlp_peek port whenever they are available:

```
`ovm_send(mem_rd_request)
root.tl.EVENT_RECEIVED_TLP.wait_trigger();
root.tl.rx_tlp_peek_port.peek(resp);
```

7.9 Backdoor Access to Completion Target Configuration Space

The Completion Target has access to its own Configuration space allowing reads and writes to Configuration registers (including Capabilities). In contrast with the VIP Memory and I/O targets, the Configuration space is located in a fixed 4K sized configuration block (as defined in the PCIE spec.) This block includes not only the standard PCI configuration registers, but the extended space (including extended capabilities) defined by PCIE.

Each device allocates a Configuration Pointer Table which contains pointers to all of the Configuration Blocks allocated (one for each function in the device).

7.9.1 Setting up the Configuration Space for Backdoor Access

The VIP model behavior is not defined by the configuration space as in a real device. The model behavior is defined by the attributes in the configuration and service classes. Though setting up the configuration space does not define the VIP behavior, the model can be set up to respond to any incoming configuration TLP. A user can program the configuration space of the model though the backdoor using the APIs on the cfg_database of the VIP model.

The VIP does not have a real configuration space like a RTL module. However, it has an internal memory that it uses for CfgWr/Rd transactions. The VIP stores the write value for the incoming CfgWr transactions, and use the return data from this memory while completing CfgRd TLPs.

There is a backdoor way to write/read this internal memory used by the VIP for configuration TLPs. Please note that you do not have to pre-load the configuration database to be able to use it. The VIP will respond to any configuration request, but it will have a default value of 0 in all of the registers.

To set up the configuration space in the pcie vip model without having to perform configuration write/read cycles, use the svt_pcie_cfg_database_service class. This class contains these 3 service types

To set up the configuration space in the pcie vip model without having to perform configuration write/read cycles, use the svt_pcie_cfg_database_service class. This class contains these 3 service types:

- GET_NUM_FUNCTIONS(`SVT_PCIE_CFG_DB_SERVICE_GET_NUM_FUNCTIONS) Get maximum number of functions.
- READ_CFG_DWORD(`SVT_PCIE_CFG_DB_SERVICE_READ_CFG_DWORD) Read configuration DWORD.
- WRITE_CFG_DWORD(`SVT_PCIE_CFG_DB_SERVICE_WRITE_CFG_DWORD) Write configuration DWORD.

These services also use these attributes to set up the configuration space.

- dword_addr
- dword_data
- function_num

Following shows example code on backdoor access.

```
//  This sequence creates backdoor then frontdoor (TLP) traffic sequences
class pcie_cfg_seq extends pcie_device_system_test_base_sequence;

  // Factory Registration.
  `svt_ovm_object_utils(pcie_cfg_seq)

  // Constructs the pcie_cfg_seq sequence
  //  @param name string to name the instance.
```

```
  function new(string name = "pcie_cfg_seq");
    int err_status;
    super.new(name);

  endfunction

  // Executes PCIE configuration sequences to demonstrate backdoor/frontdoor accesses

  task body();
    begin
      pcie_device_system_link_up_sequence link_up_seq;
      svt_pcie_driver_app_service_wait_until_idle_sequence wait_until_driver_idle_seq;

      cfg_read_sequence read_cfg_seq;
      //cfg_write_sequence write_cfg_seq;

       svt_pcie_cfg_database_service cfg_database_seq;
      svt_pcie_device_agent endpoint_device;
      bit [7:0]  bus, func;
      bit [15:0] remote_bdf;
      bit [31:0] cfg_rdata, cfg_wdata;

      bit [7:0]  function_num;             // Function Numberb
      bit [31:0]  cpt_ptr;                 // Configuration Pointer Table, CPT, which
                                           // contains pointers to all the
                                           //  Configuration Blocks allocated (one per
                                           // device)
      bit        cfg_space_type = 0;       // Type 0 or Type 1
      bit [3:0]  function_type = 0 ;       // PF, BF, VF, etc.
      bit [7:0]  sriov_physical_function = 0;  // The PF that is the parent Physical
                                              // function of the VF
      bit [7:0]  mriov_base_function = 0;   // The BF that is the parent Base Function
                                            // of this function
      bit [31:0] command_status;            // Returned status for allocate
      bit [15:0] req_cap_id;
      int         err_status;
      int         remote_register_num;

      int         test_data, test_addr;

      super.body();

      test_data = 32'habcd_1234;
      test_addr = $urandom_range(0, 1023);    // Scribble randomly in the cfg database

      // Housekeeping:
      // bring up link
      `svt_ovm_do(link_up_seq);

      // Can we get the BDF via these xx_device agents?  Maybe in an
      // enumerator that's considered 'cheating'?
      if(!$cast(root_device, p_sequencer.find_root_agent(this))) begin
        `ovm_fatal(get_full_name(), "Failed attempting to obtain handle to Root Device
agent.");
```

```
      end

      if(!$cast(endpoint_device, p_sequencer.find_endpoint_agent(this))) begin
        `ovm_fatal(get_full_name(), "Failed attempting to obtain handle to Endpoint
Device agent.");
      end

      // Backdoor fill in the cfg database
`define BACKDOOR_CFG_ACCESS_WORKS    1
`ifdef BACKDOOR_CFG_ACCESS_WORKS    // currently it doesn't...

      `svt_note("body", "Backdoor write started");

      function_num = 0;
      `ovm_create_on(cfg_database_seq,
                  p_sequencer.endpoint_virt_seqr.cfg_database_seqr);
       cfg_database_seq.service_type = svt_pcie_cfg_database_service::WRITE_CFG_DWORD;
       cfg_database_seq.function_num = function_num;
       cfg_database_seq.dword_addr = test_addr;
       cfg_database_seq.byte_enables = 4'b1111;
       cfg_database_seq.dword_data = test_data;   // 32'habcd_1234
       `ovm_send(cfg_database_seq);

       if(cfg_database_seq.command_status != `SVT_PCIE_CFG_DATABASE_STATUS_SUCCESSFUL)
         `ovm_error("body", $sformatf("Command status not SUCCESSFUL(0x%h) received
          0x%h", `SVT_PCIE_CFG_DATABASE_STATUS_SUCCESSFUL,
           cfg_database_seq.command_status))
       `ovm_info("body",$sformatf("Config_reg 0 is backdoor written with 0x%x",
            cfg_database_seq.dword_data), OVM_LOW);

      `ovm_create_on(cfg_database_seq,
          p_sequencer.endpoint_virt_seqr.cfg_database_seqr);
       cfg_database_seq.service_type = svt_pcie_cfg_database_service::READ_CFG_DWORD;
       cfg_database_seq.function_num = function_num;
       cfg_database_seq.dword_addr = test_addr;
       cfg_database_seq.byte_enables = 4'b1111;
       cfg_database_seq.dword_data = 32'hffff_ffff;
       `ovm_send(cfg_database_seq);
       if(cfg_database_seq.command_status != `SVT_PCIE_CFG_DATABASE_STATUS_SUCCESSFUL)
         `ovm_error("body", $sformatf("Command status not SUCCESSFUL(0x%h) received
          0x%h", `SVT_PCIE_CFG_DATABASE_STATUS_SUCCESSFUL,
          cfg_database_seq.command_status))
       `ovm_info("body",$sformatf("Config_reg 0 is backdoor read data=0x%x",
       cfg_database_seq.dword_data), OVM_LOW);

       if (cfg_database_seq.dword_data != test_data)
         begin
           `svt_error("body", $sformatf("Backdoor read of cfg addr %0d returned 0x%x,
              expected 0x%x", test_addr, cfg_database_seq.dword_data, test_data));
         end
 `endif // BACKDOOR_CFG_ACCESS_WORKS
```

The svt_pcie_* sequences that refers to the configuration space register number. The numbering is with regard to a dword address:

- SVT register number = 0 => [Device ID | Vendor ID] => PCIE registers [[3,2],[1,0]]
- SVT register number = 1 => [Status | Command] => PCIE registers [[7,6],[5,4]]

Note, most Firmware uses as a convention register numbers 0, 1, 2, 3, 4, 5, 6, and 7 as byte offset numbers as defined in the specification.

7.10 Setting VIP Lanes for Receiver Detect

To set which lanes the VIP will see as present from the DUT when the VIP performs a receiver detect in the detect.active state, use the following configuration member:

```
svt_pcie_pl_configuration::dut_receiver_present = 32'hffff_ffff
```

Each bit corresponds to a lane, with bit 0 corresponding to lane 0, and with bit 1 corresponding to lane 1, and so on.

For example, take the case of the VIP as an SPIPE configured to a width of x4. If you set dut_receiver_present to 32'h000_0007, then the VIP will behave as if it only detected a receiver on lanes 0-2. As a result of this, the VIP will try to negotiate to a width of x2. Note that this controls what lanes the VIP sees as present, not which lanes the DUT sees as present. Use dut_receiver_present for serial and SPIPE models only. MPIPE models will use the mechanism defined in the PIPE interface to determine which receivers are present.

7.11 Using ASCII Signals

The following sections document the ASCII signals you can use for viewing within a waveform viewer. Note, you may see ASCII signals prefaced with “debug”. These are only for internal Synopsys use.

7.11.1 Transaction Layer ASCII Signals

The ASCII signals listed in Table 7-3 are available for viewing within a waveform viewer. Access to the signals is through the XMR path to the TL. For example, in the examples shipped with the model, they are found at: “test_top.root0.port0.tl0.*ascii*”

Table 7-3 ASCII signals available for waveform viewers

Event Name	Description
ascii_rx_tlp_fc_type	Flow control credits associated with this TLP. Values are P, NP, CPL.
ascii_rx_tlp_type	Type of received TLP as defined by (fmt, type) fields.
ascii_rx_tlp_vc	VC on which TLP is received.
ascii_rx_tlp_xld	Received TLP transaction ID.
ascii_tx_tlp_fc_type	Flow control credits associated with this TLP. Values are P, NP, CPL.
ascii_tx_tlp_type	Type of TLP to be sent as defined by (fmt, type) fields.
ascii_tx_tlp_vc	VC on which TLP is sent.
ascii_tx_tlp_xld	Sent TLP transaction ID.

7.11.2 Data Link Layer ASCII Signals

ASCII signals on the Data Link Layer are listed in Table 7-4.

Table 7-4 Data Link Layer ASCII signals

Signal Name	Description
ascii_tx_tlp_type	Sent TLP type, defined by {fmt, type} fields.
ascii_tx_tlp_seq_num	Sent TLP sequence number.
ascii_tx_tlp_ei_code	TLP sent with this EI.
ascii_tx_dllp_type	Sent DLLP type.
ascii_tx_dllp_seq_num	Sent DLLP sequence number for ACK/NAK.
ascii_tx_dllp_credit_vc	Sent DLLP VC.
ascii_tx_dllp_credit_data_value	Sent DLLP data credit value.
ascii_tx_dllp_credit_hdr_value	Sent DLLP header credit value.
ascii_rx_tlp_type	Received TLP type, defined by {fmt, type} fields.
ascii_rx_tlp_seq_num	Received TLP sequence number.
ascii_rx_dllp_type	Received DLLP type.
ascii_rx_dllp_seq_num	Received DLLP sequence number for ACK/NAK.
ascii_rx_dllp_credit_vc	Received DLLP VC.
ascii_rx_dllp_credit_data_value	Received DLLP data credit value.
ascii_rx_dllp_credit_hdr_value	Received DLLP header credit value.
ascii_dlcmsm_state	Data Link Control Management State Machine.
ascii_vc[0-7]_fcsm_state	Flow Control State Machine for VC[0-7]
ascii_tx_dllp_ei_code	DLLP error codes.
ascii_aspm_state;	ASPM state
ascii_pm_state;	PM state
ascii_tx_callback;	Callback being executed on TX side.
ascii_rx_callback;	Callback being executed on the RX side.
ascii_fc_init_state;	Flow control init state

7.11.3 Physical Layer ASCII Signals

Physical Layer ASCII signals are listed in <Xref>Table 7-5.

Table 7-5 Physical Layer ASCII signals

Signal Name	Description
ascii_ltssm_tx_state	LTSSM state of the transmitter
ascii_ltssm_rx_state	LTSSM state of the receiver
ascii_lane*n*_rx_data	Data received on lane n, where n is a number between 0 and 31.

Table 7-5 Physical Layer ASCII signals (Continued)

Signal Name	Description
ascii_lane*n*_tx_data	Data transmitted on lane n, where n is a number between 0 and 31.
ascii_pipe_lane*n*_rx_data;	Symbol data on received on PIPE lane *n*, where n is a number between 0 and 31.
ascii_pipe_lane*n*_tx_data	Symbol data on transmitted on PIPE lane *n*, where n is a number between 0 and 31.
ascii_hotplug_mode	Current state of hotplug mode
ascii_prev_hotplug_mode	Previous state of the hotplug.
ascii_lane_reversal_mode;	Lane reversal indicates if lane reversal is enabled.

7.12 Using the Ordering Application

This component is provided as an optional feature which may be utilized by testbenches explicitly. The Ordering Application is implemented as a ovm_component. The agent components that ship with the VIP do not use this component by default. It has following functions:

- Validates ordering rules implementation of a DUT.
- It can optionally also be used to re-order outbound TLPs to act as application layer logic for a DUT that does not implement the rules in RTL.

It has following parts:

1. svt_pcie_ordering_app_configuration: This configuration class is used to configure the application component.
2. tx_tlp_in_port: This is a sequence item pull port (SIPP) that processes all transactions that are scheduled for transmission by DUT. The application expects transactions of type svt_pcie_tlp and hence it may be connected to a sequencer of type svt_pcie_tlp_sequencer.
3. rx_tlp_in_export: This is an analysis implementation port. The testbench must connect this to an analysis port that broadcasts transactions received by the VIP.
4. tx_tlp_out_port: This is a TLM put port onto which the application pushes all re-ordered TLPs (if enabled).
5. tl_status: This is a reference to the TL status the VIP maintains.

7.12.1 Steps to Use the Ordering Application:

1. Create a new configuration object of type svt_pcie_ordering_app_configuration and set the desired values to the properties.

   ```
   svt_pcie_ordering_app_configuration  ordering_app_cfg = new();
   ```

2. Create the Ordering application in build_phase() of a containing component (typically an environment or test component).

   ```
   ordering_app = svt_pcie_ordering_app::type_id::create("ordering_app", this);
   ```

3. In the build phase, set the reference of the configuration object in the ovm_config_db so it can be obtained by the application.

```
ovm_config_db#( svt_pcie_ordering_app_configuration )::set(this, "ordering_app", "cfg",
ordering_app_cfg);
```

4. Create a new sequencer instance of type svt_pcie_tlp_sequencer.

```
ovm_config_db #( svt_pcie_tl_configuration )::set( this, "ordering_app_seqr", "cfg",
dut_cfg.pcie_cfg.tl_cfg );

ordering_app_seqr = svt_pcie_tlp_sequencer::type_id::create("ordering_app_seqr", this);
```

5. In the connect_phase(), connect the ordering app with the sequencer created in step 4.

```
ordering_app.tx_tlp_in_port.connect(ordering_app_seqr.seq_item_export);
```

6. Set the reference of the svt_pcie_tl_status object (that the VIP maintains) in the ovm_config_db so it can be obtained by the application in end_of_elaboration_phase(). The application has need of this to obtain TCVC mapping and credit information.

```
ovm_config_db#( svt_pcie_tl_status)::set(this, "ordering_app", "tl_status",
<vip_agent>.pcie_agent.pcie_status.tl_status);
```

7. Connect rx_tlp_in_export port with an analysis port that broadcasts TLPs received by the VIP.

```
<analysis_port>.connect(ordering_app.rx_tlp_in_export);
```

8. Optionally connect tx_tlp_out_port with TLM blocking_put_imp of a downstream component. This will typically be responsible to send outbound transactions through DUT's application interface. This step is required if re-ordering of TLPs is enabled in the application (in case DUT does not support the Ordering rules in RTL).

```
ordering_app.tx_tlp_out_port.connect(<dut_driver_component>.<name_of_put_imp_port>);
```

9. Now, use the ordering_app_seqr created in step 4 to schedule all transactions to be transmitted by DUT.

7.13 Using the reconfigure_via_task Call

Please note, the reconfigure_via_task and the get_cfg_via_task calls have been depreciated. Use get_cfg and reconfigure functions in their place.

7.14 Configuring Trace File Output

You can configure how the trace file displays information using the svt_pcie_configuration::dl_trace_options member. These options apply to the default DL tracing format options.

- dl_trace_options[1] bit when set to 1'b1 enables optional printing of Cfg TLP Register Offset address as "O:0x???"
- dl_trace_options[1] bit when set to 1'b0 enables default printing of Cfg TLP Register Number as "R:0x???"
- dl_trace_options[0] bit when set to 1'b1 enables default printing of Cfg Access TLP Payload in Little Endian format
- dl_trace_options[0] bit when set to 1'b0 enables default printing of Cfg Access TLP Payload in Big Endian format

This first instance has the Cfg TLP Register Offset address option enabled (svt_pcie_configuration::dl_trace_options[1]=1) and printing of the CFG access payload as bigendian(svt_pcie_configuration::dl_trace_options[0]=0). See the following:

```
endpoint0  18128.000 18236.000 R CfgWr0  0x0000/06 ... BDF:0x0107 O:0x010   0 c      1 H44008001 ...
```

```
                                                                                0 fecacefa
endpoint0  18448.000 18540.000 R CfgRd0  0x0000/07 ... BDF:0x0107 O:0x010   0 f      1 H04008001 ...
endpoint0  18588.000 18696.000 T CplD    0x0000/07 ... ID:0x0107 Stat:SC  BC:0004    1 H4a008001 ...
                                                                                0 efbecefa
```

This second instance has the Cfg TLP Register Offset address option disabled (svt_pcie_configuration::dl_trace_options[1]=0) and printing of the CFG access payload as LittleEndian(svt_pcie_configuration::dl_trace_options[0]=1). See the following:

```
root0 18124.000 18232.000 T CfgWr0    0x0000/06 ... BDF:0x0107 R:0x004   0 c     1 H44008001 H0000060c ...
                                                                                 0  facecafe LittleEndian
root0 18444.000 18536.000 T CfgRd0     0x0000/07 ... BDF:0x0107 R:0x004   0 f    1 H04008001 H0000070f ...
root0 18592.000 18700.000 R CplD      0x0000/07 ... ID:0x0107 Stat:SC  BC:0004   1 H4a008001 H01070004 ...
                                                                                 0  facebeef LittleEndian
```

7.15 Setting Coefficient and Preset for Gen3 Equalization

Transmitter equalization is adopted in Gen 3 to compensate for an increased signal distortion from operating at a higher data rate. On entry to the 8.0 GT/s data rate, the link partners exchange equalization presets and coefficients to determine the transmitter and receiver settings that yield an optimal signal-to-noise ratio on each lane. This trainable equalization process consists of 4 phases. The phase information is specified in the Equalization Control (EC) field in the TS1 Ordered Sets.

In phase 0, in Recovery.RcvrCfg before transitioning to 8.0 GT/s, the Downstream port transmits the recommended preset and coefficient values to the Upstream Port. The Upstream Port uses these recommended values in phase 0 and phase1.

In phase 1, the Downstream and Upstream Ports transmit with their respective coefficients. Both ports advertise the preset and post cursor values of their transmitters, the LF and FS values. Equalization is complete in phase 1 if a finer adjustment to the preset and coefficient values is not required. If the Downstream Port requests a finer adjustment to the presets and coefficients, then the ports proceed to phase 2.

In phase 2, the transmitter coefficients of the Downstream Port are optimized. The Upstream Port can request the Downstream Port to adjust its transmitter by setting the preset and coefficient fields in the transmitted TS OS. The Downstream Port either accepts or rejects these new values. Once an optimal preset and coefficient values are determined by the Upstream Port, the Upstream Port transitions to phase 3.

In phase 3, the transmitter coefficients of the Upstream Port are optimized. The Downstream Port can request the Upstream Port to adjust its transmitter by setting preset and coefficient fields in the transmitted TS OS. The Upstream Port either accepts or rejects these new values. Once an optimal preset and coefficient values are determined by the Downstream Port, the Downstream Port transitions to Recovery.RcvrLock.

7.15.1 Enabling Equalization

Equalization checking is disabled by default In the PCIe VIP. You can use the following attributes in the PHY layer configuration class (svt_pcie_pl_configuration) to enable equalization checking.

Table 7-6 Equalization Modes

Attribute	Description
enable_equalization_verification_mode	Enables equalization verification mode
enable_equalization_coefficients_checks	Enables equalization coefficient check

Table 7-6 Equalization Modes

Attribute	Description
highest_enabled_equalization_phase	Specifies the highest equalization phase to be enabled. A value of 1 enables equalization phase 0 and phase 1. A value of 3 enables equalization phases 0, 1, 2, and 3.

For example:

```
root_cfg.pcie_cfg.pl_cfg.enable_equalization_verification_mode = 1;
root_cfg.pcie_cfg.pl_cfg.enable_equalization_coefficients_checks = 1;
root_cfg.pcie_cfg.pl_cfg.highest_enabled_equalization_phase = 3;

endpoint_cfg.pcie_cfg.pl_cfg.enable_equalization_verification_mode = 1;
endpoint_cfg.pcie_cfg.pl_cfg.enable_equalization_coefficients_checks = 1;
endpoint_cfg.pcie_cfg.pl_cfg.highest_enabled_equalization_phase = 3;
```

7.15.2 Specifying Coefficients, Presets, LF and FS Values

7.15.2.1 Initializing Presets with EQ TS OS

As a Downstream Port, the VIP transmits EQ TS OS with recommended preset values. The preset values are specified by the "upstream_preset_value" attribute in the PHY layer configuration class (svt_pcie_pl_configuration). If the recommended preset values are mapped to the valid coefficients in the DUT (Upstream Port), then the DUT transmits TS OS with the recommended preset values in equalization phases 0, 1, and 3. The TS OS coefficient fields of the DUT are specified with the corresponding coefficients from the preset mapping table of the DUT. The VIP compares the preset field of the received TS OS with the "upstream_preset_value" attribute. For more information on the mapping table, refer to the "Preset to Coefficient Mapping" section.

As an Upstream port, the VIP receives EQ TS OS with preset values recommended by the DUT. The VIP transmits TS1 OS with the recommended preset values in equalization phases 0, 1, and 3. The coefficient fields of the VIP are specified with the corresponding coefficients from the preset mapping table of the VIP in phases 0 and 3. The post cursor value is specified in phase 1.

You can use the following attributes in the PHY layer configuration class (svt_pcie_pl_configuration) to specify the coefficients, LF and FS values.

A

Table 7-7 Attributes of PHY Layer Configuration Class

ttribute	Description
upstream_preset_value	Specifies the Upstream Port preset value.
preset_to_coefficients_mapping_table	Maps received preset requests to coefficients for use in local transmitter settings. This table is indexed by a preset value. The coefficients are packed in the following format: { postcursor_coeff, cursor_coeff, precursor_coeff } The default value is { 6'h01, 6'h56, 6'h01} or 18'h01b81.
lf_value	Specifies the LF value advertised by the VIP in TS1s during equalization phase 1.

Table 7-7 Attributes of PHY Layer Configuration Class

ttribute	Description
fs_value	Specifies the FS value advertised by the VIP in TS1s during equalization phase 1.

For example:

```
 //Specify mapping table values to change default coefficients
root_cfg.pcie_cfg.pl_cfg.preset_to_coefficients_mapping_table[0]='{16{18'h00543}};

root_cfg.pcie_cfg.pl_cfg.lf_value = '{32{'d9}};
root_cfg.pcie_cfg.pl_cfg.fs_value = '{32{'d24}};
```

7.15.2.2 Preset to Coefficient Mapping

The VIP includes two preset mapping tables in the PHY layer configuration class (svt_pcie_pl_configuration).

Table 7-8 Preset PHY Mapping

Attribute	Description
preset_to_coefficients_mapping_table	Maps received preset requests to coefficients for use in local transmitter settings. This table is indexed by a preset value. The coefficients are packed in the following format: { postcursor_coeff, cursor_coeff, precursor_coeff } The default value is { 6'h01, 6'h56, 6'h01} or 18'h01b81.
expected_preset_to_coefficients_mapping_table	Verifies the preset to coefficient mappings in the DUT. This table should be programmed with the same value as the preset table in the DUT. This table is indexed by a preset value. The coefficients are packed in the following format: { postcursor_coeff, cursor_coeff, precursor_coeff } The default value is { 6'h0c, 6'h24, 6'h00} or 18'h0c900.

As a Downstream Port, the VIP verifies the DUT preset mapping in Phase 0 and again in Phase 3. As an Upstream Port, the VIP verifies the DUT preset mapping in Phase 2. If the preset maps to an invalid entry, the VIP disables mapping check, transmits with the recommended coefficients specified by the DUT, and set the reject preset bit.

7.15.2.3 LF and FS Values

The LF and FS values are advertised in phase 1. The VIP uses the values advertised by the DUT to determine the DUT's acceptance or rejection of its recommended presets and coefficients. If the DUT does not accept or reject the presets and coefficients, the VIP issues a warning message.

For the LF and FS values advertised by the VIP, the VIP verifies that the presets and coefficients recommended by the DUT does not violate its LF and FS values.

7.15.2.4 Rejecting Presets or Coefficients

The VIP has built-in checks for preset or coefficient requests from the DUT as defined in the PCIe specification. In addition, the VIP includes a mode to manually force a rejection on a per lane basis regardless of the results of the built-in check.

The VIP issues a notice message when the DUT rejects a coefficient.

7.15.2.5 Automatic Rejection

For each preset or coefficient request, the VIP verifies that the mapped coefficients in the preset case or received coefficients does not violate the LF and FS rules as defined in section 4.2.3.1 of the PCIE Specification. If a violation occurs, the VIP asserts the "reject coefficient values" bit in the transmitted TS OS on that particular lane. This bit is also asserted for preset requests that do not map to valid coefficients. When a new set of presets and coefficients are received, the VIP performs a new check.

7.15.2.6 Manual Rejection

Manual rejection can be specified for any preset or coefficient request from the DUT on any lane. The VIP asserts the reject bit in the TS OS in the appropriate phase.

You can use the following attribute in the PHY layer configuration class (svt_pcie_pl_configuration) to reject preset or coefficient values requested by the DUT.

Attribute	Description
reject_preset_coefficient_request	Each bit maps to a corresponding lane. When a bit is set to 1'b1, the corresponding lane rejects the new preset and coefficient values. This bit is applicable only in equalization phase 2 for Downstream Ports and equalization phase 3 for Upstream Ports.

For example:

```
root_cfg.pcie_cfg.pl_cfg.reject_preset_coefficient_request = 32'h0
```

7.16 Target Application

The Target Completer Application provides the following features:

- Provides completer services by responding to various inbound requests: CFG, Memory, IO and Interrupts
- Will break up reads into multiple completions
- Interleaved with other read completions
- Highly configurable
  - min:max read data size
  - Completion boundary (align)
  - Max payload size
  - min:max latency (mem, io, cfg ; all independent)
  - Un-Initialized mem mode: 0, completer abort

To configure the Target Completer Application you use the svt_pcie_target_app_configuration class. Consult the HTML class reference for additional information. Follow are some useful settings:

- completer_id. Default Completer ID used by the Target application in the generated completions until Configuration Write reuqests are received on the link to program the completer ID. This ID is concatenation of Bus number, Device number and a Function number.

- max_io_cpl_latency. The variable represents maximum latency in ns for each completion packet generated by the application in response to inbound IO requests.
- min_cfg_cpl_latency. The variable represents minimum latency in ns for the completion packet generated by the application in response to inbound Configuration requests.
- read_completion_boundary_in_bytes. The variable read_completion_boundary_in_bytes specifies the RCB value. The Target application uses this while creating completions.
- max_payload_size_in_bytes. The variable specifies maximum payload size in bytes. Any TLP payload cannot exceed this value in size.

7.16.0.1 Target Application Callbacks

The target application is the component responsible for handling the auto-generated completions in the VIP model. The model has has two callbacks defined at this application layer namely:

1. post_rx_tlp_get(): Called by the component after recognizing a TLP transaction received immediately from the link.
2. pre_tx_tlp_put(): Called by the component after scheduling a TLP transaction for transmission on the link, just prior to framing.

These callback can be used to inject errors into transactions using exception objects. The following example illustrates how to set the error poison (EP) bit in a completion TLP generated by the target application.

```
// Callback Class
class set_ep_target_app_callback extends svt_pcie_target_app_callback;

  `svt_ovm_object_utils(set_ep_target_app_callback)

  // Exception List and Exception class objects
  svt_pcie_tlp_exception_list my_tlp_exc_list = new("my_tlp_exc_list");
  svt_pcie_tlp_transaction_exception my_tlp_exc = new("my_tlp_exc");

  function new(string name = "set_ep_target_app_callback");
    super.new();
  endfunction

// Callback Function Implementation
virtual function void pre_tx_tlp_put(svt_pcie_target_app target_app, svt_pcie_tlp
transaction, ref bit drop);
 // Add any conditional statement here to look for a specific TL packet (if necessary).
 // The illustration below is unconditional so it would end up setting the EP bit on all
 // completion packets being transmitted by the target application.
    my_tlp_exc.error_kind = svt_pcie_tlp_transaction_exception::CORRUPT_EP;
    my_tlp_exc.corrupted_data = 0;
    my_tlp_exc.corrupted_data[0] = 1;
    my_tlp_exc_list.add_exception(my_tlp_exc);
    `svt_note("pre_tx_tlp_get",$sformatf("ERP - pre_tx_tlp_put: Attaching exception
      list - corrupting TLP EP field (was=1'b%b now=1'b%b).\n", transaction.ep,
    my_tlp_exc.corrupted_data[0]));
    $cast(transaction.exception_list, my_tlp_exc_list.`SVT_DATA_COPY());
  endfunction

endclass
```

```
// OVM Test Class
class base_pipe_test extends pcie_device_base_test ;

  set_ep_target_app_callback set_ep_target_cb;
…
  virtual function void end_of_elaboration_phase(ovm_phase phase);
    super.end_of_elaboration_phase(phase);

    set_ep_target_cb = new("set_ep_target_cb");

ovm_callbacks#(svt_pcie_target_app,svt_pcie_target_app_callback)::add(env.endpoint.targ
et[0], target_cb);
  endfunction

endclass
```

The same approach can be used to attach other errors on completions generated by the target application. To check the list of errors supported by the model refer to <point to a table or list (the list can be found inside the enumerated type svt_pcie_tlp_exception::error_kind_enum) in the doc which lists all the error types that can be attached to a TLP.

7.17 Requester Application

Requester Application is used for generating background traffic. The Application generates PCIe Read/Write transactions to a given target. You can randomize the following:

- [minimum:maximum] address range(s)
- Number or writes
- Number of reads
- [minimum:maximum] data length
- Configure bandwidth
- Time between packets
- # requests per second

It performs a synchronize when it is finished.

You use the class svt_pcie_requester_app_configuration for the Target Application. For additional information on configuration members, consult the HTML Class Reference. Following are some configuration members.

- bandwidth_mode. Specifies the mode which controls the read/write request generation mechanism. BANDWIDTH_MODE_REQUESTS_PER_SEC mode specifies the read/write request generation rate as number of requests to be generated per second. BANDWIDTH_MODE_NS_DELAY_IN_REQUESTS mode specifies the read/write request generation rate in terms of delay between successive requests generated.
- completion_timeout_ns. Completion timeout in nanoseconds.
- num_mem_read. Number of memory read transactions to be transmitted.
- max_time_between_packets. The variable is applicable when bandwidth_mode is set to BANDWIDTH_MODE_NS_DELAY_IN_REQUESTS. The value of this variable specifies maximum delay in NS in the successive packets to be generated.

7.18 What Are Blocking and Non-blocking Reads in PCIe SVT?

'Blocking' read prevents other transactions from being queued. Whereas in case of non-blocking read, a process does not wait for the completion. As a result the transaction is queued.

The driver application layer uses the block attribute to control when the driver queues the next transaction. When it is set to 1, the driver will wait until the transaction is completed before the next transaction is queued.

When set to 0, the driver does not wait for transaction to complete and drives the sub-sequent transaction. The function of the block bit is to not return control to the user until the completion is received in the case of a read.

```
` * block = 1, block until completion is received.
* block = 0, non-block.
For cfg_rd request:
            `ovm_do_on_with(cfg_rd_rq,p_sequencer.root_virt_seqr.driver_transaction_seq
r[0],{
            bdf == 16'h0100;
            block == 1'b1; // block until completion is received.
            first_dw_be == 4'hf;
            register_number inside {[0:50]};
            });
```

For Mem_rd request:

```
`ovm_do_on_with(mem_rd_request_seq,vip_seqr.driver_transaction_seqr[0],{
address == mem_wr_request_seq.address;
traffic_class == mem_wr_request_seq.traffic_class;
length == mem_wr_request_seq.length;
block == 1'b0; // it will not wait for completion
first_dw_be == mem_wr_request_seq.first_dw_be;
last_dw_be == mem_wr_request_seq.last_dw_be;
});
```

7.19 Using SKP Ordered Sets

The SKP interval transmission and reception can be controlled in the PCIe VIP through the following attributes in the svt_pcie_pl_configuration class.

The VIP will transmit a SKP OS based on these settings:

- Gen1 and Gen2
  - min_tx_skp_interval_in_symbol_times
  - max_tx_skp_interval_in_symbol_times
- Gen3
  - min_tx_skp_interval_in_blocks
  - max_tx_skp_interval_in_blocks

The VIP will check the reception of the SKP OS from the DUT based on these settings:

- Gen1 and Gen2
  - min_rx_skp_interval_in_symbol_times
  - max_rx_skp_interval_in_symbol_times
- Gen3
  - min_rx_skp_interval_in_blocks
  - max_rx_skp_interval_in_blocks

This table shows the allowed ranges and the default setting for the SKP interval attributes.

Table 7-9 SKP Ordered Set Configuration Members

Type	Range	Default	Description
max_tx_skp_interval_in_symbol_times			
Integer	32 - large value	1538	Maximum number of symbol times before the upper phy will schedule the insertion of a SKP ordered set (2.5GT/s and 5 GT/s)
min_tx_skp_interval_in_blocks			
Integer	2 - large value	375	Minimum number of blocks before the upper phy must schedule the insertion of a SKP ordered set (8GT/s)
max_tx_skp_interval_in_blocks			
Integer	2 - large value	375	Maximum number of blocks before the upper phy must schedule the insertion of a SKP ordered set (8GT/s)
min_rx_skp_interval_in_symbol_times			
Integer	32 - large value	1180	Minimum number of symbol times before the upper phy flag an error due to the lack of a SKP ordered set (2.5GT/s and 5 GT/s)
max_rx_skp_interval_in_symbol_times			
Integer	32 - large value	1538	Maximum number of symbol times before the upper phy will flag an error due to the lack of a SKP ordered set (2.5GT/s and 5 GT/s)
min_rx_skp_interval_in_blocks			
Integer	2 - large value	375	Minimum number of blocks before the upper phy flags an error due to the lack of a SKP ordered set (8GT/s)
max_rx_skp_interval_in_blocks			
Integer	2 - large value	375	Maximum number of blocks before the upper phy flags an error due to lack of a SKP ordered set (8GT/s)
min_tx_skp_symbols_in_ordered_set			
Integer	1-5	3	Minimum number of SKP symbols in an ordered set at 2.5GT/s and 5GT/s.

Table 7-9 SKP Ordered Set Configuration Members (Continued)

Type	Range	Default	Description
max_tx_skp_symbols_in_ordered_set			
Integer	1-5	3	Maximum number of SKP symbols in an ordered set at 2.5GT/s and 5GT/s.
min_tx_skp_symbols_in_ordered_set_8g			
Integer	1-5	3	Minimum number of SKP symbols in an ordered set at 8GT/s. Acceptable values: 1 - 5.
max_tx_skp_symbols_in_ordered_set_8g			
Integer	1-5	3	Maximum number of SKP symbols in an ordered set at 8GT/s. Acceptable values: 1 - 5.

The min/max_tx_skp_interval_in_<xxx >settings are randomized in the VIP to the min and max settings of the attributes. For example, the default setting for min_tx_skp_interval_in_symbol_times is 1180 symbol times. The default setting for max_tx_skp_interval_in_symbol_times is 1538 symbol times. The PCIe VIP will transmit the SKP OS based on these 2 settings. The SKP interval transmission will be randomized between the min and max values.

If the SKP OS interval needs to be set to a specific value, then set the min and max values to the same number. For example, to have the PCIe VIP transmit the SKP interval at 1275, the following setting would be done.

```
 //Configure min/max skip interval to a set value.
cfg.rc_cfg.pcie_cfg.pl_cfg.min_tx_skp_interval_in_symbol_times = 1275;
cfg.rc_cfg.pcie_cfg.pl_cfg.max_tx_skp_interval_in_symbol_times = 1275;
```

Similarly for the SKP interval in blocks for gen3, you would do the same thing.

```
//Configure min/max skip interval to a set block value.
cfg.rc_cfg.pcie_cfg.pl_cfg.min_tx_skp_interval_in_blocks = 372;
cfg.rc_cfg.pcie_cfg.pl_cfg.max_tx_skp_interval_in_blocks = 372;
```

For the min/max_rx_skp_interval_in_symbol_times and min/max_rx_skp_interval_in_blocks, the PCIe VIP will check for the reception of the SKP OS from the DUT. Again, the SKP interval will be checked based on the randomized min and max values. If the VIP is required to check the SKP interval reception for a particular value, then set the min and max values to be the value that is needed.

The PCIe VIP also allows for the number of SKP symbols to be included in the SKP OS. The default setting for both min and max is 3, so 3 SKP symbols will be sent in the SKP OS. The SKP OS can be adjusted to send a random number of SKP symbols and set to another value such as 5 by setting the min and max numbers to be different or the same number.

7.20 Using Service Class Reset App

The VC VIP provides functionality to support mid-simulation reset. The scenario is that the VIP is connected to the DUT, and that the DUT is reset sometime into the test after the initial reset of the VIP and DUT. The purpose of the test is to ensure the DUT recovers and that the link retrains. During this time you want to mimic that a reset also happening on the VIP. This means all activity should cease, all buffers should be cleared, and you should go back into our initial state waiting for training sets.

In terms of implementation note that the VIP has its own reset, and that it is only allowed to toggle once, typically at the beginning of a simulation. Further, the VIP reset should never be connected to the reset of the DUT.

Since the VIP reset doesn't toggle, a mid-sim reset is performed with a combination of hot plug control and application resets. The PL provides a mechanism to 'unplug' and then 'plug' the svt_pcie_pl_service_request_hot_plug_mode_sequence. All applications provide a mechanism which resets the apps meaning they are re-initialized, svt_pcie_*_reset_app_sequence. Table 7-10 lists each one for each service class.

Synopsys also provides a sequence which wraps the hot plug and reset calls: svt_pcie_device_virtual_reset_sequence.

The following outlines the steps in a mid-sim reset.

1. Unplug VIP from bus (HOTPLUG_UNPLUG)
2. Assert Reset on the DUT
3. Reset apps (all, some, or none – user choice)
4. Re-enable the VIP on the bus (HOTPLUG_DETECT)
5. De-assert Reset on the DUT
6. Continue with the test
   - Suggest monitoring for a change in LTSSM state; PL : WaitForLtssmStateChange()
7. Initialize DUT and VIP, run to a point in the test where a mid-sim reset is to be performed

   Option 1:
   - Unplug VIP from bus, svt_pcie_pl_service_request_hot_plug_mode_sequence with HOTPLUG_UNPLUG
   - Assert reset on DUT
   - foreach (app) ResetApp (call on apps to reset; not necessary to reset all apps)
   - execute svt_pcie_pl_service_request_hot_plug_mode_sequence with HOTPLUG_DETECT
   - De-assert reset on DUT

   Option 2:
   - In parallel, assert DUT's reset line and execute: svt_pcie_device_virtual_reset_sequence
   - After completion of sequence, reassert DUT's reset line
8. Continue with the test
   - Suggest monitoring for a change in LTSSM state, or L0

     ```
     wait (agent.status.pcie_status.pl_status.ltssm_state == svt_pcie_types::L0)
     ```

Table 7-10 Service Class App Sequence Resets

Reset Sequence for Application	Description
svt_pcie_requester_app_service_reset_app_sequence	• This sequence implements ResetApp • RESET_APP is a reset for the Driver • The effect of resetting this application is to drop all queued and partially completed requests. After a reset the driver will check for completions on sent requests or check for timeouts. • It is not necessary to call Reset App at the beginning of simulation
svt_pcie_io_target_service_reset_app_sequence	• This sequence implements Reset App • RESET_APP resets the application back to its initial state. All data will be lost. • It is not necessary to call this at start of sim.
svt_pcie_mem_target_service_reset_app_sequence	• This sequence implements Reset App • RESET_APP resets the memory target back to its initial state. All data will be lost. • It is not necessary to call this at start of sim
svt_pcie_requester_app_service_reset_app_sequence	• This sequence implements Reset App • RESET_APP is a reset for the requester. Any outstanding requests will be dropped, and there will be no timeouts for these dropped requests. If completions come in for the dropped requests they will be treated as unexpected completions. • It is not necessary to call RESET_APP at the start of simulation.
svt_pcie_target_app_service_reset_app_sequence	• This sequence implements ResetApp • RESET_APP resets the target application. All outstanding requests are dropped and will not be completed. • It is not necessary to call RESET_APP at the start of simulation.

Following is a code fragment showing midsim reset.

```
task midsim_reset_sequence:: body();

    pcie_device_system_link_up_sequence link_up_seq;

    svt_pcie_requester_app_service_mem_range_sequence req_mem_range_seq;
    svt_pcie_requester_app_service_app_sequence req_app_seq;
    svt_pcie_requester_app_service_clr_stats_sequence req_clr_stats_seq;
    svt_pcie_requester_app_service_disp_stats_sequence req_disp_stats_seq;
    svt_pcie_requester_app_service_reset_app_sequence reset_requester_app_seq;
    svt_pcie_target_app_service_reset_app_sequence reset_target_app_seq;
    svt_pcie_pl_service_request_hot_plug_mode_sequence request_hot_plug_mode_seq;
    svt_pcie_device_virtual_reset_sequence  device_reset_vseq;

...
```

```
    // bring up link
    `svt_ovm_do(link_up_seq);

// Add memory ranges to Requester application

`svt_ovm_do_on_with(req_mem_range_seq,p_sequencer.endpoint_virt_seqr.requester_seqr, {
req_mem_range_seq.service_type == svt_pcie_requester_app_service::ADD_MEM_RANGE;
                                                                                ...//
Reset after a few TLPs have been sent
     wait(ep_agent.status.pcie_status.tl_status.num_tlps_sent == 3);


     // Now reset both sides of the link
     /*
     // Note that this code has been left in intentionally to serve as an example on how
to reset only certain parts of the VIP
     // The reset virtual sequence called below will reset all app layers.

`svt_ovm_do_on_with(request_hot_plug_mode_seq,p_sequencer.endpoint_virt_seqr.pcie_virt_
seqr.pl_seqr,{mode == svt_pcie_pl_service::HOT_PLUG_UNPLUG;});


`svt_ovm_do_on_with(request_hot_plug_mode_seq,p_sequencer.root_virt_seqr.pcie_virt_seqr
.pl_seqr,{mode == svt_pcie_pl_service::HOT_PLUG_UNPLUG;});

     // App resets are optional, and are reset individually with a service call
     `svt_ovm_do_on_with(req_app_seq,p_sequencer.endpoint_virt_seqr.requester_seqr, {
req_app_seq.service_type == svt_pcie_requester_app_service::RESET_APP; });

     `svt_ovm_do_on(reset_target_app_seq,p_sequencer.root_virt_seqr.target_seqr[0]);
     */

     ->seq_to_test1;
     ovm_report_info( "TEST", "Triggering seq_to_test1 event.", OVM_NONE );

     // Just use the reset virtual sequence
     `svt_ovm_do_on(device_reset_vseq, p_sequencer.root_virt_seqr);

     // EP should dected the EIOS and automatically enter detect.
     wait(ep_agent.status.pcie_status.pl_status.ltssm_state ==
svt_pcie_types::RECOVERY_RCVRLOCK);

     ->seq_to_test2;
     ovm_report_info( "TEST", "Triggering seq_to_test2 event.", OVM_NONE );

     `svt_ovm_do_on(device_reset_vseq, p_sequencer.endpoint_virt_seqr);

     // Wait some time and start everything back up
     #1000;

`svt_ovm_do_on_with(request_hot_plug_mode_seq,p_sequencer.endpoint_virt_seqr.pcie_virt_
seqr.pl_seqr,{mode == svt_pcie_pl_service::HOT_PLUG_DETECT;});
```

```
`svt_ovm_do_on_with(request_hot_plug_mode_seq,p_sequencer.root_virt_seqr.pcie_virt_seqr
.pl_seqr,{mode == svt_pcie_pl_service::HOT_PLUG_DETECT;});

     wait(ep_agent.status.pcie_status.pl_status.ltssm_state == svt_pcie_types::L0);

     `svt_ovm_do_on_with(req_app_seq,p_sequencer.endpoint_virt_seqr.requester_seqr, {
req_app_seq.service_type == svt_pcie_requester_app_service::START_APP; });


     `svt_ovm_do_on_with(req_app_seq,p_sequencer.endpoint_virt_seqr.requester_seqr, {
req_app_seq.service_type == svt_pcie_requester_app_service::IS_APP_FINISHED; });

     // Wait until Requester application has completed pumping in specified memory
requests
     if(req_app_seq.is_finished == 'b0) begin
       // wait until requester application is finished.
      `svt_ovm_do_on_with(req_app_seq,p_sequencer.endpoint_virt_seqr.requester_seqr, {
req_app_seq.service_type == svt_pcie_requester_app_service::WAIT_UNTIL_FINISHED; });
     end
...
    // Now reset both sides of the link
    /*
    // Note that this code has been left in intentionally to serve as an example on how
to reset only certain parts of the VIP
    // The reset virtual sequence called below will reset all app layers.

`svt_ovm_do_on_with(request_hot_plug_mode_seq,p_sequencer.endpoint_virt_seqr.pcie_virt_
seqr.pl_seqr,{mode == svt_pcie_pl_service::HOT_PLUG_UNPLUG;});


`svt_ovm_do_on_with(request_hot_plug_mode_seq,p_sequencer.root_virt_seqr.pcie_virt_seqr
.pl_seqr,{mode == svt_pcie_pl_service::HOT_PLUG_UNPLUG;});

    // App resets are optional, and are reset individually with a service call
    `svt_ovm_do_on_with(req_app_seq,p_sequencer.endpoint_virt_seqr.requester_seqr, {
req_app_seq.service_type == svt_pcie_requester_app_service::RESET_APP; });

    `svt_ovm_do_on(reset_target_app_seq,p_sequencer.root_virt_seqr.target_seqr[0]);
    */

     ->seq_to_test1;
     ovm_report_info( "TEST", "Triggering seq_to_test1 event.", OVM_NONE );

     // Just use the reset virtual sequence
     `svt_ovm_do_on(device_reset_vseq, p_sequencer.root_virt_seqr);
...

`svt_ovm_do_on_with(request_hot_plug_mode_seq,p_sequencer.endpoint_virt_seqr.pcie_virt_
seqr.pl_seqr,{mode == svt_pcie_pl_service::HOT_PLUG_DETECT;});


`svt_ovm_do_on_with(request_hot_plug_mode_seq,p_sequencer.root_virt_seqr.pcie_virt_seqr
.pl_seqr,{mode == svt_pcie_pl_service::HOT_PLUG_DETECT;});
```

```
      wait(ep_agent.status.pcie_status.pl_status.ltssm_state == svt_pcie_types::L0);

      `svt_ovm_do_on_with(req_app_seq,p_sequencer.endpoint_virt_seqr.requester_seqr, {
req_app_seq.service_type == svt_pcie_requester_app_service::START_APP; });

      `svt_ovm_do_on_with(req_app_seq,p_sequencer.endpoint_virt_seqr.requester_seqr, {
req_app_seq.service_type == svt_pcie_requester_app_service::IS_APP_FINISHED; });

      // Wait until Requester application has completed pumping in specified memory
requests
      if(req_app_seq.is_finished == 'b0) begin
        // wait until requester application is finished.
        `svt_ovm_do_on_with(req_app_seq,p_sequencer.endpoint_virt_seqr.requester_seqr, {
req_app_seq.service_type == svt_pcie_requester_app_service::WAIT_UNTIL_FINISHED; });
      end

      `svt_ovm_do_on(req_disp_stats_seq,p_sequencer.endpoint_virt_seqr.requester_seqr);
      `svt_ovm_do_on(req_clr_stats_seq,p_sequencer.endpoint_virt_seqr.requester_seqr);
      `svt_ovm_do_on(req_disp_stats_seq,p_sequencer.endpoint_virt_seqr.requester_seqr);
endtask
```

7.21 Using FLR

Perform the following steps to enable FLR support:

1. Initiate `cfg_wr` in the Device Control Register (Offset 08h) to initiate FLR
2. Wait until the device enters `flr_active`—that is, state `max_expected_time_to_enter_flr_active_in_ns`
3. Initiate the traffic from VIP
4. The function must complete the FLR within 100 ms. After 100 ms, there will be no traffic pending for that function.

FLR support can be enabled using RC instance of the VIP—that is, by Config Read/Write sequences on the Device Control Register.

Example 7-3 Sample code

Driver App Transaction's Sequences:

```
svt_pcie_driver_app_transaction_cfg_wr_sequence       cfg_wr_sequence;
svt_pcie_driver_app_transaction_cfg_rd_sequence         cfg_rd_sequence;
```

Read the control register's content:

```
  `ovm_do_on_with(cfg_rd_sequence,
p_sequencer.root_virt_seqr.driver_transaction_seqr[0], {cfg_rd_sequence.address ==
16'h0100;
          cfg_rd_sequence.block == 1'b1;
          cfg_rd_sequence.register_number == 'h08;})

// Set FLR bit to 1
tmp_data = cfg_rd_sequence.req.payload[0] | 32'h0000_8000;
//
```

Now update the Register:

```
        `svt_xvm_do_on_with(cfg_wr_sequence,
p_sequencer.root_virt_seqr.driver_transaction_seqr[0], {cfg_wr_sequence.address ==
16'h0100;
          cfg_wr_sequence.block == 1'b1;
          cfg_wr_sequence.payload == tmp_data;
          cfg_wr_sequence.register_number == 'h08;})
```

For Config commands:

- *Address* field must be the Base address of the DUT EP
- *Register number* field must be the offset

7.22 Programming Hints and Tips

7.22.1 PIPE Polarity

You can use the svt_pcie_pl_configuration::invert_tx_polarity configuration member to programs polarity inversion on all lanes. It is a 32-bit vector where bit 0 control polarity inversion on lane 0. When a bit is set, the corresponding lane will invert polarity on all outgoing data. Note that it only works in serial mode.

7.22.2 Address Translation Services

The PCIe VIP supports Address Translation Services (ATS). In particular, the model supports access to the AT field within the Memory Read and Memory Write TLPs. If the bit is set, then the provided request gets translated through the ATS protocol.

7.22.2.1 VIP Services to Map Translation inside RC VIP for ATS

The following services are present for ATS at `svt_pcie_mem_target` level:

- **WRITE_ATPT:** Populate ATPT table within RC VIP instance to support enhanced ATS implementation.
- **READ_ATPT:** Read ATPT table within RC VIP instance to support enhanced ATS implementation.
- **INVALIDATE_ATPT:** Backdoor deletion of entry in ATPT table within RC VIP instance to support enhanced ATS implementation.

7.22.2.2 VIP Service Sequences with ATS Services

The following VIP service sequences are available with the ATS services:

- `svt_pcie_mem_target_service_write_atpt_sequence`: This sequence implements WRITE_ATPT service.
- `svt_pcie_mem_target_service_read_atpt_sequence`: This sequence implements READ_ATPT service.

7.22.2.3 Knobs to Control RC VIP ATS Operation

The RC VIP ATS operation is controlled by the following VIP configurations:

- `svt_pcie_driver_app_configuration`
  - `enable_enhanced_address_translation`
  - `stu`: Smallest translation unit size that is used for address translation requests
  - `completion_timeout_nsvariable` used for Non-Posted requests initiated by RC
  - `read_completion_boundary_in_bytes`
- `svt_pcie_target_app_configuration`
  - `stu`: smallest translation unit size that is used for address translation requests
  - `enable_enhanced_address_translation`
  - `one_translation_completion_response_wt`
  - `two_translation_completion_response_wt`
  - `enable_truncated_translation_completion`
  - `page_response_code_invalid_wt`

- page_response_code_unused_wt
- page_response_code_failure_wt
- read_completion_boundary_in_bytes

7.22.2.4 TL Status Class Events to Track Translation

See Table 7-11 for TL status class events that are used by VIP to track translation.

Table 7-11 TL Status Class Events

TL Status Class Events	Description
num_tlp_msg_ats_invalidate_req_received	Indicates the number of ATS Invalidate Req message TLPs received with data.
num_tlp_msg_ats_invalidate_req_sent	Indicates the number of ATS Invalidate Req message TLPs sent with data.
num_tlp_msg_ats_invalidate_cpl_received	Indicates the number of ATS Invalidate Cpl message TLPs received without data.
num_tlp_msg_ats_invalidate_cpl_sent	Indicates the number of ATS Invalidate Cpl message TLPs sent without data.
num_tlp_msg_ats_page_req_received	Indicates the number of ATS Page Req message TLPs received without data.
num_tlp_msg_ats_page_req_sent	Indicates the number of ATS Page Req message TLPs sent without data.
num_tlp_msg_ats_prg_response_received	Indicates the number of ATS PRG Response message TLPs received without data.
num_tlp_msg_ats_prg_response_sent	Indicates the number of ATS PRG Response message TLPs sent without data.

7.22.2.5 Automatic Translated Response from VIP:

VIP also provides capability to automatically respond to address translation requests by automatically creating an address translation entry in the ATPT with an address within the boundary specified by automatic_at_response_min_mem_range and automatic_at_response_max_mem_range.

The following configurations are used to enable this capability:

- svt_pcie_target_app_configuration::enable_automatic_at_response
- svt_pcie_target_app_configuration::automatic_at_response_min_mem_range
- svt_pcie_target_app_configuration::automatic_at_response_max_mem_range

Note After enabling this capability, you are not required to implement **WRITE_ATPT** service from test/sequence.

7.22.3 Calls For Analysis Port Set Up and Usage

The following configuration members help you setup analysis ports on the monitor.

- rand int unsigned attribute svt_pcie_dl_configuration::received_dllp_interface_mode = 0.
  Select DLLPs available at Receive DLLP analysis port. DLLPs are filtered based on th bits enabled in received_dllp_interface_mode bit vector. See svt_pcie_dl::received_dllp_observed_port for accessing received DLLPs.

  Bits are defined in include/svt_pcie_common_defines.v as SVT_PCIE_SENT_DLLP_INTERFACE_MODE_[sel]_BIT where [sel] is defined as:

  - GOOD_PACKETS_BIT

  - ERR_PACKETS_BIT

- rand int unsigned attribute svt_pcie_dl_configuration::received_tlp_interface_mode = 0
  Select TLPs available at Receive TLP analysis port. TLPs are filtered based on the bits enabled in received_tlp_interface_mode bit vector. See svt_pcie_dl::received_tlp_observed_port for accessing received TLPs.

  Bits are defined in include/svt_pcie_common_defines.v as SVT_PCIE_RECEIVED_TLP_INTERFACE_MODE_[sel]_BIT where [sel] is defined as:

  - GOOD_PACKETS_BIT

  - ERR_PACKETS_BIT

- rand int unsigned attribute svt_pcie_dl_configuration::replay_timeout
  Length of the replay timer in symbols. If called, timeout value is sticky. Setting to value = 0 will enable automatic updates.

- rand int unsigned attribute svt_pcie_dl_configuration::sent_dllp_interface_mode = 0
  Select DLLPs available at Sent DLLP analysis port. DLLPs are filtered based on the bits enabled in sent_dllp_interface_mode bit vector. See svt_pcie_dl::sent_dllp_observed_port for accessing sent DLLPs.

  Bits are defined in include/svt_pcie_common_defines.v as SVT_PCIE_SENT_DLLP_INTERFACE_MODE_[sel]_BIT where [sel] is defined as:

  - ALL_PACKETS_BIT

- rand int unsigned attribute svt_pcie_dl_configuration::sent_tlp_interface_mode = SVT_PCIE_SENT_TLP_INTERFACE_MODE_DEFAULT
  Select TLPs available at Sent TLP analysis port. TLPs are filtered based on the bits enabled in sent_tlp_interface_mode bit vector. See svt_pcie_dl::sent_tlp_observed_port for accessing sent TLPs.

  Bits are defined in include/svt_pcie_common_defines.v as SVT_PCIE_SENT_TLP_INTERFACE_MODE_[sel]_BIT where [sel] is defined as:

  - ALL_PACKETS_BIT.

You must set the sent_tlp_interface_mode and received_tlp_interface_mode members to enable the analysis ports--otherwise, no transactions appear.

The sent_tlp_interface_mode parameter is for enabling the analysis port. The following code hows the enabling of the analysis ports:

```
// Enable analysis ports.
cust_cfg.root_cfg.pcie_cfg.dl_cfg.sent_tlp_interface_mode = 1;
cust_cfg.root_cfg.pcie_cfg.dl_cfg.received_tlp_interface_mode = 1;
```

```
cust_cfg.root_cfg.pcie_cfg.dl_cfg.sent_dllp_interface_mode = 1;
cust_cfg.root_cfg.pcie_cfg.dl_cfg.received_dllp_interface_mode = 1;
```

If these configuration variables (sent_tlp_interface_mode and received_tlp_interface_mode) are set to 1, then you can use the class svt_pcie_dl_dllp_monitor_transaction to obtain sent and received TLPs using the analysis ports.

Following are the steps to set up the dl_monitors:

1. Use these connections.

   ```
   ep_agent.pcie_agent.dl.sent_tlp_observed_port.connect(sent_tlp_port);
   ep_agent.pcie_agent.dl.received_tlp_observed_port.connect(received_tlp_port);
   ```

   Refer to the following example file: ts.directed_pipe_test.sv

   ```
   env.root.pcie_agent.dl.sent_tlp_observed_port.connect(
    sent_tlp_subscriber.analysis_export );
   env.root.pcie_agent.dl.received_tlp_observed_port.connect(
    rcvd_tlp_subscriber.analysis_export );
   env.root.pcie_agent.dl.sent_dllp_observed_port.connect(
    sent_dllp_subscriber.analysis_export );
   env.root.pcie_agent.dl.received_dllp_observed_port.connect(
    rcvd_dllp_subscriber.analysis_export );
   ```

2. The following connections need to be enabled:

   svt_pcie_device_configuration -> svt_pcie_configuration -> svt_pcie_dl_configuration

   Use the following code:

   ```
   root_cfg.pcie_cfg.dl_cfg.sent_tlp_interface_mode = 1;
   root_cfg.pcie_cfg.dl_cfg.received_tlp_interface_mode = 1;
   root_cfg.pcie_cfg.dl_cfg.sent_dllp_interface_mode = 1;
   root_cfg.pcie_cfg.dl_cfg.received_dllp_interface_mode = 1;
   ```

7.22.4 Sequences and the ovm_sequence :: get_response Task

The driver does not know at what time a transaction is queued, and what its tag ID will be. But for every request queued into the driver, the driver issues a unique command number. You access the command number by referring to svt_pcie_driver_app_transaction::**command_num.** The command_num attribute is assigned when the transaction is queued**.**

Every sequence that is executed will generate a response, and thus ovm_sequence :: get_response should always be called for every request that has been queued. For posted commands, it still needs to be called. For nonposted commands, if a block is called, then when you call ovm_sequence :: get_response, you will have the completion information available.

If a block is set to '0', then you want to save the command_num. You would pick up commands later on the source_rx_transaction_out_port,. You can match the completion by checking command_num. If you want to wait for a completion or check if a request has completed, then there are service calls for that, and they all use command_num.

7.22.5 Setting the TH and PH Bits Using the Driver Application Class

You can set the TH and PH bits using the Driver App interface class and transaction class: svt_pcie_driver_app_transaction. Note the following members to implement this capability:

```
/**
   * Transaction Hint bit, indicates presence of TLP Processing Hints.
   */
```

```
 rand bit th    = 0;

/**
   *  Processing Hints field.
   */
 rand ph_enum    ph = BIDIRECTIONAL;

/**
Steering tag used when TLP processing hint is present. Bits [7:0] are part of the
request header. If the TH bit is set, then the steering tag field will always be
substituted for the tag field for memory writes. In addition, the steering tag field
will always be substituted for the byte enables for memory reads/atomic operations.
  */
 rand bit [7:0] st;
```

7.22.6 Fast Link Training

A common way speed up link training is to decrease the number of training sets transmitted in Polling.Active. The VIP supports this option, which is controlled by the Physical Layer configuration member:

rand int unsigned attribute svt_pcie_pl_configuration::*num_tx_ts1_in_polling_active* = SVT_PCIE_NUM_TX_S1_IN_POLLING_ACTIVE_DEFAULT

This member sets the number of training sets the LTSSM must transmit in Polling.Active before exiting this state. This parameter and all LTSSM timeout parameters should be set to match whatever values are used in the DUT in order to obtain valid results during abbreviated link training.

7.22.7 When to Invoke Service Calls

You should not make any service calls until after the VIP is properly configured and initialized. For example, you should not call the hot unplug service call while the LTSSM is still in it's initial state. The hot unplug call may immediately kick the LTSSM into detect before it has a chance to finish initializing.

7.22.8 Exceptions and Scrambler Control Bits

The svt_pcie_pl_proxy code has been implemented so that if there is a svt_pcie_symbol_exception, then the svt_pcie_symbol_exception->scrambler_control bits are used (except when error_kind == "NO_ERROR").

You must set the scrambler_control bits for all symbols when using svt_pcie_symbol_exception class. The following table shows the values available to you with the scrambler_control enum:

Table 7-12 Values for Setting Control Bits of Scrambler

Name	Value	Description
NONE	b'00	Disables scrambler for the specified symbol.
FORCE_SCRAMBLE	b'01	Forces the scrambler to scramble the symbol to be transmitted (even if it is not supposed to be scrambled as per PCIe rules).
INIT_SCRAMBLER	b'10	Resets and initializes scrambler. The specified symbol is not scrambled.

Table 7-12 Values for Setting Control Bits of Scrambler

Name	Value	Description
INIT_AND_FORCE_SCRAMBLER	b'11	Resets, initializes scrambler and forces the scrambler to scramble the symbol to be transmitted (even if it is not supposed to be scrambled as per PCIE rules).

Summary: when a symbol is changed then the scrambler_control needs to reflect what you wants to occur for this symbol.

7.22.9 User TS1 Ordered Set Notes

User TS1s Ordered Sets can only be used in legitimate LTSSM states. You can formulate and send user TS OS only in those LTSSM states where it is legitimate to send a TS OS. The Ordered Set creation task would simply stop those default TS Ordered Sets, and let the user TS go on the bus.

Note the following:

- User TS1 OS replace outgoing TS1's, but they do not override all outgoing data.
- Users will never see a user TS1 OS in Recovery.Idle because the LTSSM is required to send idles.
- For users requiring TS1 to be sent in Recovery.Idle, the best way to do this is to start up the user TS in the state before Recovery.Idle (recovery.rcvrcfg), and turn on user TS2. The LTSSM will not make a state transition while the user TS are turned on.

```
`ovm_info(get_type_name(),$sformatf("Begin process set_tx_ts1_pattern_seq 2nd
time\n"),OVM_NONE);
   set_tx_ts1_pattern_seq.randomize() with {
        set_tx_ts1_pattern_seq.user_tx_ts_enable         == 1'b1;
        set_tx_ts1_pattern_seq.min_user_tx_ts_burst      == 1;
        set_tx_ts1_pattern_seq.max_user_tx_ts_burst      == 1;
        set_tx_ts1_pattern_seq.min_user_tx_ts_spacing    == 2;
        set_tx_ts1_pattern_seq.max_user_tx_ts_spacing    == 2;
};
```

7.23 PCIe VIP Bare COM Support

7.23.1 Background

In 2.5 GT/s or 5.0 GT/s transmission (8b10b encoding only), a normal transmitted SKP Ordered Set consists of a *COM* Symbol (K28.5) followed by three SKP Symbols (K28.0). The term *Bare COM* is used in reference to the PIPE interface during either 2.5 GT/s or 5.0 GT/s transmission where the datum being received across that interface is a *COM* symbol (K28.5 symbol) with the `RxStatus` equal to 2 indicating "1 SKP removed".

In a real PCIe system, the *Bare COM* PIPE scenario will be the result of a number of repeaters or re-timers each removing "1 SKP" Ordered Set. The last one in the string of three will then result in this *Bare COM* with the remaining three SKP symbols having been removed by previous repeaters leaving only the *COM* and the "1 SKP removed" `RxStatus` of 2 as indicated.

7.23.2 Enabling VIP Bare COM transmission (to mimic the system scenario)

When the PCIe VIP's `MIN_TX_SKP_SYMBOLS_IN_ORDERED_SET_VAR` variable is set to 0 in a test for the SPIPE model (`IS_PIPE_MASTER` is 0 and `PHY_INTERFACE_TYPE` is 0), the possibility of transmitting a *Bare COM* as a SKP Ordered Set is enabled. If that PCIe VIP MPIPE model's `MAX_TX_SKP_SYMBOLS_IN_ORDERED_SET_VAR` variable is also set to 0 in a test, every SKP Ordered Set is assured to be a *Bare COM*. This is the method that guarantees *Bare COM* transmission.

The above scenario results in a *COM* with `RxStatus` equal to 2 being transmitted on the PIPE interface (*Bare COM*).

7.23.2.1 Misconfiguration Warning Message

If the `MIN_TX_SKP_SYMBOLS_IN_ORDERED_SET_VAR` is set to 0 and either the VIP model's `IS_PIPE_MASTER` parameter is 0 (SPIPE) or the `PHY_INTERFACE_TYPE` is not 0 (Serdes or PMA models), then the VIP will issue the following warning (and, as indicated, will set the number of SKP symbols to the normal 3):

```
WARNING: endpoint0.port0.pl0.: 'Bare COM' (num_skp_symbols_to_send == 0) is illegal for
'IS_PIPE_MASTER' == 1 (s/b 0) OR 'PHY_INTERFACE_TYPE = 0'(s/b 0 - PIPE) to support 'Bare
COM' SKP OS) - set 'num_skp_symbols_to_send' to 3.
```

7.23.3 Enabling VIP Bare COM Reception

The reception of a *Bare COM* in any PCIe VIP PIPE model release that includes the *Bare COM* support in "Enabling VIP Bare COM transmission (to mimic the system scenario)" needs no special setup and is available by default for 2.5 GT/s and 5.0 GT/s data rates.

7.23.3.1 Ordered Set Checker Warnings

Bare COM reception in a PCIe MPIPE model will result in the following warning:

```
WARNING: endpoint0.port0.pl0.ordered_set_checker0.: Detected undersize SKP ordered set
with 1 COM and 1 SKP symbols. Expecting 3 SKP symbols.
```

The warning indicates "1 COM and 1 SKP" (rather than the expected "0 SKP") due to the `ordered_set_checker` interpreting the `RxStatus` equal to 2 being received with the *COM* as having removed a received SKP ("1 SKP removed" status) and therefore including that SKP Symbol as having been received (as would have been the case if a real PHY was attached to the PIPE).

To suppress the above warnings in any test that intentionally transmits *Bare COM*s, all the receiving model's `ordered_set_checker` modules requires message suppression using the `MSGCODE_PCIESVC_ORDERED_SET_CHECKER_UNDERSIZE_SKIP` message suppression code.

7.24 Up/Down Configure

As part of bandwidth management and means to optimize power consumption, PCIe protocol supports changing the link width even after the TLP traffic has started (that is, Up/Down configure implies changing link width when link_up is 1). RC and EP VIP instances support this protocol feature and can act an initiator or target of the link width change request.

The following methods, service requests, and class variables are required to achieve the desired functionality. For more details, see HTML class reference documentation.

Service Requests

```
svt_pcie_pl_service
```

Control Methods

```
svt_pcie_pl_configuration::set_link_width_values( , ,)
svt_pcie_device_agent::reconfigure_via_task(). Alternatively one can use
svt_pcie_device_agent_service  requests with svt_pcie_device_agent_service::service_type
= svt_pcie_device_agent_service::REFRESH_CFG
```

Control Properties

```
svt_pcie_pl_configuration::upconfigure_capable
svt_pcie_pl_configuration::link_width
svt_pcie_pl_configuration::mpipe_turn_off_unused_lanes_after_initial_link_training
svt_pcie_pl_service::service_type = svt_pcie_pl_service::INITIATE_LINK_WIDTH_CHANGE
```

Status Properties

```
svt_pcie_pl_status::initial_negotiated_link_width
svt_pcie_pl_status::negotiated_link_width
```

Use model for SVT PHY layer service request Initiate_Link_Width_Change

The service request direct the LTSSM to change the link width of a link that is already in link up state. The VIP LTSSM shall service this request from following states L0/TX_L0_Rx_L0s/L1. Issuing the service request when VIP LTSSM is not in one of the above listed states will result in the service request being ignored. This service request must be used in conjunction with the properties in svt_pcie_pl_configuration class.

1. Set the desired link width and/or supported link width and/or expected link width properties using the method set_link_width_values provided in class svt_pcie_pl_configuration.
2. Invoke reconfigure_via_task or issue REFRESH_CFG service request so that the values communicated via above step percolate to VIP local copy of the configuration object.
3. Issue svt_pcie_pl_service::INITIATE_LINK_WIDTH change service request and wait for the LTSSM to transition from L0/L1/RX.L0s -> Recovery ->Config -> L0.
4. In case the VIP instance is the target of link width change request initiated by its partner (that is, the DUT), the testbench can set the appropriate value of expected_link_width by using set_link_width_values.

7.25 Lane Reversal

The order of lanes in a multi-lane PCIe link may require a change as part of the physical link training. This feature is known as lane reversal and is traditionally verified by redoing the testbench connections between VIP instance and DUT instance thus adding a compile dependency. PCIe SVT VIP offers run time control to select the order of lanes.

Control Properties

svt_pcie_pl_configuration::lane_reversal_mode can take any of the four enumerated values UNSUPPORTED,FORCED,SUPPORTED, FORCED_AND_SUPPORTED. For more details, see HTML class reference documentation.

Use Model for Lane Reversal Feature

- svt_pcie_pl_configuration::lane_reversal_mode must be modified while the VIP LTSSM is in DETECT state (before the training is initiated).

- svt_pcie_pl_configuration::lane_reversal_mode = FORCED_AND_SUPPORTED is not applicable to EP VIP instance.
- Note that the lane number value passed to other VIP API (equalization coefficients control, receiver present control, lane polarity control, and so on) use logical lane number, therefore enabling lane reversal will not have any impact on the existing tests.

7.26 Lane Reversal with Different Link Width Configurations

Usage notes for supported link width with lane reversal enabled.

- Set the MAX_SUPPORTED_DUT_LINK_WIDTH = <user_link_width>
- Set svt_pcie_pl_configuartion attribute lane_reversal_mode to FORCED.
- Call the set_link_width_values function for changing the link width

  The set_link_width_values function accepts the following three inputs:

  - link_width_value (svt_pcie_pl_configuartion: link_width_value)
  - supported_link_width_vector_value (svt_pcie_pl_configuartion: supported_link_width_vector_value)
  - expected_link_width_value (svt_pcie_pl_configuartion: expected_link_width_value)
    - The link_width_value (svt_pcie_pl_configuartion: link_width_value) must be same as DUT link width value.
    - The supported_link_width_vector_value (svt_pcie_pl_configuartion: supported_link_width_vector_value) must have all the possible link widths a VIP can support from 1 up to link_width_value value.

    i. When unset (second argument) in the function call, it prompts VIP to set supported_link_width_vector_value to support all the possible link widths from 1 up to link_width_value value.

    The controls for supported link width vector are as follows:

Table 7-13 Controls for Supported Link Width Vector

Link Width	supported_link_width_vector
1	32'h01
2	32'h03
4	32'h07
8	32'h0F
12	32'h1F
16	32'h3F;
32	32'h7F;

    - expected_link_width_value (svt_pcie_pl_configuartion: expected_link_width_value): The expected negotiated link width value. This value must be same as supported link width vector's link width value.

i. When unset in the function call, it prompts VIP to set expected_link_width value same as the value of link_width_value argument.

Example 7-4 VIP-DUT Setup

- MAX_SUPPORTED_DUT_LINK_WIDTH = 16
- Set the set_link_width_values (16, 32'h03, 2) for VIP // Here Supported link width vector 32'h03 and expected link width is 2
- Set the lane_reversal_mode to FORCED for VIP
- The link up happens at X2 on [Lane 3, Lane 2]

Example 7-5 VIP-DUT Setup

- MAX_SUPPORTED_DUT_LINK_WIDTH = 32
- Set the set_link_width_values (32, 32'h01, 1) for VIP // Here Supported link width vector 32'h01 and expected link width is 1
- Set the lane_reversal_mode to FORCED for VIP
- The link up happens at X1 on [Lane 31].

Example 7-6 VIP-DUT Setup

- MAX_SUPPORTED_DUT_LINK_WIDTH = 8
- Set the set_link_width_values (8, 32'h07, 4) for VIP // Here Supported link width vector 32'h04 and expected link width is 4
- Set the lane_reversal_mode to FORCED for VIP
- The link up happens at X4 [Lane7, Lane6, Lane5, Lane4]

Example 7-7 VIP-VIP Setup or PHY-DUT Setup

- MAX_SUPPORTED_DUT_LINK_WIDTH = 8
- Set the set_link_width_values (8, 32'h01, 1) for VIP(RC) // Here Supported link width vector 32'h04 and expected link width is 1
- Set the lane_reversal_mode for VIP(RC) to FORCED
- Set set_link_width_values (8, 32'h07, 4) for VIP(EP) // Here Supported link width vector 32'h04 and expected link width is 4
- Set the lane_reversal_mode for VIP(EP) to SUPPORTED
- The link up happens at X1 [Lane7]

Example 7-8 VIP-VIP Setup or PHY-DUT Setup

- MAX_SUPPORTED_DUT_LINK_WIDTH = 8, Lane Reversal ENABLED
- Set the set_link_width_values (8) for VIP(RC) // Here Supported link width vector 32'h0F (depends upon the first Argument) and expected link width is 8(depends upon the first argument)
- Set the lane_reversal_mode for VIP(RC) to FORCED
- Set the set_link_width_values (8) for VIP(EP) // Here Supported link width vector 32'h0F (depends upon the first Argument) and expected link width is 8 (depends upon the first argument)

- Set the lane_reversal_mode for VIP(EP) to SUPPORTED
- The link up happens at X8 [Lane7, Lane6, Lane5, Lane4, Lane3, Lane2, Lane1, Lane0]

Note

- For detailed description of lane_reversal_mode and set_link_width_values method, see svt_pcie_pl_configuartion class in the HTML class reference documentation.
- The description in the previous section shows the use of set_link_width_values to control the initial link width (before physical link up transitions from 0 to 1). In subsequent course of simulation (that is, post link up is set to 1) if test intends to change the link width, then link_width_value (first argument of method set_link_width_values) value must be modified and it is recommended to retain supported_link_width_vector_value as is by resupplying its current value as input to the method.

7.27 User-Supplied Memory Model Interface

The user-supplied memory interface allows you to direct the SVT PCIe VIP application layer to utilize an external memory model to store TLP payloads. This can be useful in systems where memory is allocated from a central resource.

The package class `svt_mem_backdoor_base` provides the base API between the `svt_pcie_mem_target` and a desired memory model. `svt_pcie_mem_target_gmem_model` (generic memory model) is the default implementation of this interface and provides the base memory model of the application layer `mem_target` component. The model provides an example of the currently utilized and required minimum features of the `svt_mem_backdoor_base` API by the mem_target.

You can extend this class to implement linkage to your memory model and map the interface through the config_db for mem_target use. Required functions to overload in the extended user memory interface class are `peek_base`, `poke_base` and `free_base`.

The config_db must be set during initial configuration in the build phase. Overrides of the user memory model will be ignored during subsequent phases and reconfiguration operations.

Perform the following steps to connect a user memory interface object:

1. Extend the model `svt_pcie_mem_target_gmem_model` and overload the functions `poke_base`, `peek_base` and `free_base` to communicate with your memory model.
2. In test `build_phase` of the test or environment class, construct an extended `mem` model object.
3. Pass the memory interface object handle to `mem_target` instance through `config_db` as `user_gmem_model`.

Example override code:

```
class user_gmem_model extends svt_pcie_mem_target_gmem_model;
 …
endclass

…
virtual function void build_phase(ovm_phase phase);
  // handle to the user's gmem implementation
  user_gmem_model user_model;

  // Build up default test and environment
  super.build_phase(phase);

  /*
   * Create and assign a user override model IN BUILD PHASE
```

```
        */
       user_model = new("user_model", this);
       svt_config_object_db#(svt_mem_backdoor_base)::set(this, "<relative path to pcie
device>.mem_target", "user_gmem_model", user_model);

  endfunction
```

`mem_target` can return the memory interface object handle at any time with `svt_pcie_mem_target::get_backdoor()`:

```
memory interface handle = <svt_pcie_vip_instance>.mem_target.get_backdoor();
```

For more information about the API features of `svt_pcie_mem_target_gmem_model` and `svt_mem_backdoor_base`, see HTML class reference documentation.

7.28 External Clocking and Per Lane Clocking for Serial Interface

The PCIe VIP supports the following two clocking modes:

- Internal transmit bit clock mode
  - VIP serial transmission depends on internally generated clock.
- External transmit bit clock mode
  - VIP expects external bit clock at physical transmission rate fed to the VIP as input.
  - In this mode, VIP assumes that the jitter if any present is applied to the externally supplied clock.

7.28.1 Enabling External Clocking and Per Lane Clocking Modes

- External Clocking mode: External clocking is disabled by default in PCIe VIP. You can use the following Verilog parameter to enable external transmit bit clock mode.

Attribute	Type	Description	Comments
TRANSMIT_BIT_CLOCK_MODE	Verilog Parameter	Verilog parameter to specify clocking mode. 0 => internal bit clocks are used to transmit serial data. 1 => external bit clocks are used to transmit serial data.	Attribute is applicable for all lanes.

For example,

Set up for configuring external clocking mode from RC:

```
spd_0.SVT_PCIE_UI_TRANSMIT_BIT_CLOCK_MODE  = 1;
```

- External clocking signaling interface: The VIP model has per lane per link speed clocking speed inputs for external bit transmit clock mode. This gives you an option to connect to the VIP model in external transmit bit clock mode when you do not have link speed multiplexed clocking pin. If you

have a single output wire per lane for gen1/gen2/gen3/gen4 link speed, then you can connect the VIP model per lane per link speed pin with the link speed multiplexed on per lane basis. You can use the following signals to connect for external clocking mode.

Attribute	Type	Description	Comments
svt_pcie_ext_clk_intf::logic [31:0] tx_clk_2_5g	Input	Per lane external bit clock at Gen1(2.5GTS) rate. The interface signal is defined for all 32 lanes but you must connect the maximum number of physical lanes your design supports	Attribute is applicable for each lane.
svt_pcie_ext_clk_intf::logic [31:0] tx_clk_5g	Input	Per lane external bit clock at Gen2(5GTS) rate. The interface signal is defined for all 32 lanes but you must connect the maximum number of physical lanes your design supports.	Attribute is applicable for each lane.
svt_pcie_ext_clk_intf::logic [31:0] tx_clk_8g	Input	Per lane external bit clock at Gen3(8GTS) rate. The interface signal is defined for all 32 lanes but you must connect the maximum number of physical lanes your design supports.	Attribute is applicable for each lane.
svt_pcie_ext_clk_intf::logic [31:0] tx_clk_16g	Input	Per lane external bit clock at Gen4(16GTS) rate. The interface signal is defined for all 32 lanes but you must connect the maximum number of physical lanes your design supports.	Attribute is applicable for each lane.

For example,

Set up for configuring external clocking mode from RC:

```
spd_0.SVT_PCIE_UI_TRANSMIT_BIT_CLOCK_MODE  = 1;
assign port_if_0.ext_clk_if.tx_clk_2_5g = spd_1.m_ser.port0.tx_bit_clk;
assign port_if_0.ext_clk_if.tx_clk_5g = spd_1.m_ser.port0.tx_bit_clk;
assign port_if_0.ext_clk_if.tx_clk_8g = spd_1.m_ser.port0.tx_bit_clk;
assign port_if_0.ext_clk_if.tx_clk_16g = spd_1.m_ser.port0.tx_bit_clk;
```

- Per lane clocking mode: Per lane clocking is disabled by default in PCIe VIP. You can use following Verilog parameter to enable per lane clocking in conjunction with enabling external clocking mode.

Attribute	Type	Description	Comments
ENABLE_PER_LANE_CLOCKING	Verilog Parameter	Verilog parameter to enable per lane clocking in vip model.	Attribute is applicable for all lanes.

For example,

```
spd_0.SVT_PCIE_UI_ENABLE_PER_LANE_CLOCKING  = 1;
spd_1.SVT_PCIE_UI_ENABLE_PER_LANE_CLOCKING  = 1;
```

Note

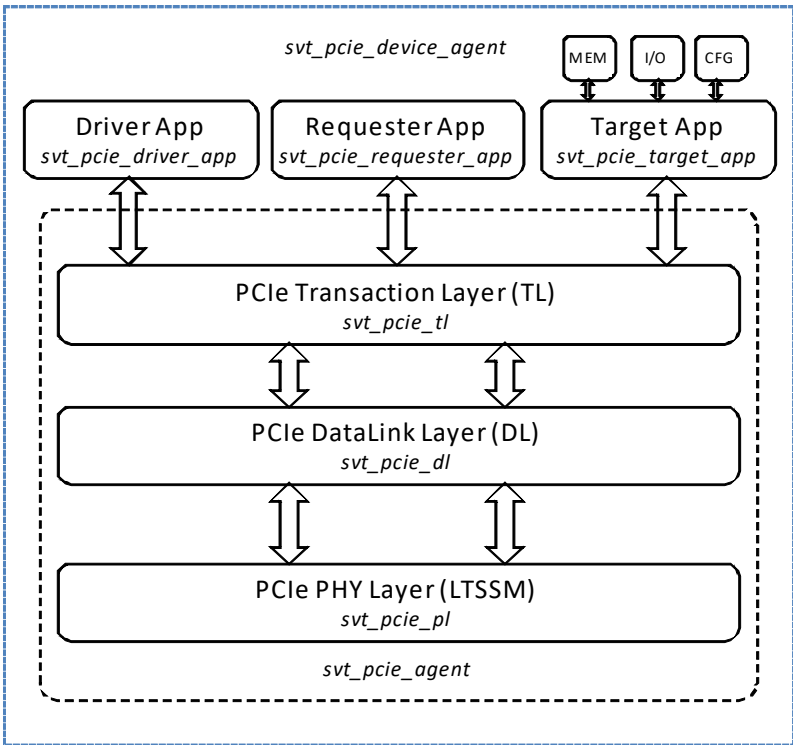
- If you are not running with external transmit bit clock mode, then it is not required to connect to the VIP model ext_*_tx_bit_clk clocking signals.
- If per lane clocking is disabled and VIP is running in external clocking mode then you only need to connect to VIP lane 0 clock (tx_clk_2_5g[0], tx_clk_8g[0], tx_clk_8g[0], tx_clk_16g[0]).
- VIP is required to have correct external clocks even in low power states where reference clock might be switched off, so the onus on providing the correct clock even in case of low power scenario is on the testbench.

8 PCIe Device Agent

8.1 Overview

The PCIe OVM VIP, at its highest level, is composed of the `svt_pcie_device_agent` class, which encapsulates the PCIe Agent (Class type=`svt_pcie_agent`)—an application layer that comprises of Driver Application, Requester Application, and Target Application (Memory, I/O, Configuration database).

Figure 8-1 Block Diagram – PCIe OVM VIP



- The Application Layer: The PCIe VIP has a layer on top of the PCIe stack representing the software layer in a real application of the PCIe bus. The application layer is responsible for generating and handling transactions. The application layer of the PCIe VIP is the layer that is typically programmed by the test to generate stimulus or respond to the incoming requests. The application layer of the PCIe VIP has the following blocks that perform specific functions:
  - Driver Application (Type=`svt_pcie_driver_app`, Instance=`driver[0]`): The Driver application provides a simple interface that can be used to quickly create PCIe transaction requests (memory read/write request, I/O read/write requests etc.,). The application deals with driver application transaction objects (`svt_pcie_driver_app_transaction`) which is an abstract description of the transaction layer packet.
  - Requester Application (Type=`svt_pcie_requester_app`, Instance=`requester`): The requester application can be used to generate PCIe memory/read transaction to a remote target. The application can be configured to choose addresses at random (constrained by minimum/maximum configuration parameters) and have varying lengths (again, constrained by minimum/maximum configuration parameters) and generate traffic at a requested bandwidth (again, constrained by minimum/maximum configuration parameters). This application will be useful where a background exerciser of write/read request is required.
  - Target Application (Type=`svt_pcie_target_app`, Instance=`target[0]`): The target application is the block that automatically responds to various inbound requests. Read transaction requests can be optionally broken up into multiple completions, potentially interleaved with other read completions by configuration. The target application has auxiliary blocks that represent the completion memory used while generating completions. These blocks include Memory Target, IO Target and Configuration Database. The blocks comprise of a sparse memory allowing a wide variety memory/IO addresses/registers to be accessed by a DUT.
  - Memory Target (Type=`svt_pcie_mem_target`, Instance=`mem_target`): The memory target is the PCIe VIP's sparse memory model used to store write data and return read data to the incoming memory requests. This sparse memory model responds to a wide variety of addresses (32-bit and 64-bit) when accessed by a requester. The memory target has APIs to write into its sparse memory via backdoor or read from its sparse memory via backdoor to meet different kinds of testing requirements.
  - I/O Target (Type=`svt_pcie_io_target`, Instance=`io_target`): The I/O target is the PCIe VIP's sparse memory model used to store write data and return read data to the incoming I/O requests. This sparse memory model responds to a wide variety of addresses (32-bit and 64-bit) when accessed by a requester. The I/O target has APIs to write into its sparse memory via backdoor or read from its sparse memory via backdoor to meet different kinds of testing requirements.
  - Configuration Database (Type=`svt_pcie_cfg_database`, Instance=`svt_pcie_mem_target::cfg_database`): The configuration database is the PCIe VIP's sparse memory model used to store write data and return read data to the incoming configuration requests. This sparse memory model responds to a wide variety of addresses (type 0, type 1, extended capability registers, and so on) when accessed by a requester. The configuration database has APIs to write into its sparse memory via backdoor or read from its sparse memory via backdoor to meet different kinds of testing requirements.
- PCIe Agent: The PCIe Agent encapsulates the OVM drivers that represents the Transaction Layer, the Data-link Layer and the Physical Layer of the PCIe protocol stack. For more details, see PCIe Agent.

8.2 Configuration

The PCIe Device Agent is configured using an object of class `svt_pcie_device_configuration`. This class has other class objects defined within it to form a hierarchy that corresponds to the hierarchy inside the PCIe Device Agent.

```
svt_pcie_device_configuration
            |
            |--------> driver_cfg[]  (type=svt_pcie_driver_app_configuration)
            |
            |--------> requester_app (type=svt_pcie_requester_app_configuration)
            |
            |--------> target_cfg[]  (type=svt_pcie_target_app_configuration)
            |
            |--------> pcie_cfg      (type=svt_pcie_configuration)
                           |
                           |--------> tl_cfg (type=svt_pcie_tl_configuration)
                           |
                           |--------> dl_cfg (type=svt_pcie_dl_configuration)
                           |
                           |--------> pl_cfg (type=svt_pcie_pl_configuration)
```

The PCIe Device Agent is configured using an object of `svt_pcie_device_configuration` class. This class is comprised of direct variables and class objects that are used to configure other agents/drivers that are part of the device agent. For details about the attributes of this class, see HTML class description of the `svt_pcie_device_configuration` class available at the following location:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/configuration/class_svt_pcie_device_configuration.html

Note The variable `model_instance_scope` is a string variable that must be set to a value that is the Verilog Cross-module reference to the Verilog instantiation model that is instanced and connected to the DUT. This configuration parameter binds the OVM agent to its Verilog counterpart. If this variable is not set correctly, then it results in an error at runtime.

8.2.1 Initial Configuration

The initial configuration of the PCIe Device Agent (and all of its sub-components) is established using the configuration database (`ovm_config_db`) class defined in OVM. The PCIe Device Agent—the recipient of the configuration object has a `ovm_config_db::get()` defined within the `build_phase()` and so the parent test/environment has to specify a `ovm_config_db::set()` in its `build_phase()`. Before calling the `ovm_config_db:set()`, the desired configuration attributes must be set to user-defined values. The example below illustrates the initial configuration step.

Example 8-1

```
class pcie_device_unified_vip_env extends ovm_env;
   …
   svt_pcie_device_agent root;
   svt_pcie_device_configuration root_cfg;
   …
   function void build_phase( ovm_phase phase );
      super.build_phase( phase );
```

```
            …
            root_cfg = new ("root_cfg");
            // Functions that programs values to different configuration parameter.
            setup_system_defaults(root_cfg);
            // Setting up the OVM agent with its Verilog counterpart.
            root_cfg.model_instance_scope = "test_top.root";
            // Setting the type of the device. In this example a root complex.
            root_cfg.device_is_root = 1;

            // Call the ovm_config_db::set() to set the configuration of the OVM agent.
            // Arg1 & Arg2: context & instance name. Hierarchical path to the object data
            // Arg3: Field name, "cfg" is the field name used by the agent class
            // Arg3: Object/Value being set
            ovm_config_db#(svt_pcie_device_configuration)::set(this,"root", "cfg",  root_cfg);
            root = svt_pcie_device_agent::type_id::create( "root", this );
            …
         endfunction
…
         function void setup_system_defaults (svt_pcie_device_configuration cfg);
            cfg.pcie_spec_ver = svt_pcie_device_configuration::PCIE_SPEC_VER_2_1;

            // Programming configuration attributes of the Applications
            cfg.driver_cfg[0].requester_id = 'h300;
            cfg.driver_cfg[0].percentage_use_tlp_digest = 50;

            cfg.target_cfg[0].completer_id = 'h300;
            cfg.target_cfg[0].percentage_use_tlp_digest = 50;
            cfg.target_cfg[0].max_read_cpl_data_size_in_bytes = 512;

            // Programming the configuration attributes of the PCIe protocol layers
            cfg.pcie_cfg.tl_cfg.credit_starvation_timeout = 8000;
            cfg.pcie_cfg.tl_cfg.completion_timeout = 400000;

            cfg.pcie_cfg.dl_cfg. updatefc_timeout_ns = 40000;

            cfg.pcie_cfg.pl_cfg.set_link_width_values(4);
            cfg.pcie_cfg.pl_cfg.set_link_speed_values(`SVT_PCIE_SPEED_2_5G);
            cfg.pcie_cfg.pl_cfg.skip_polling_active = 1;
         endfunction
      endclass
```

The `ovm_config_db::set()` in the illustration above will program the agent and all of its sub-components. In the Example 8-1, only a few parameters from the different layers are shown as being modified and for the rest, a default value is assumed. For the complete list of the configuration attributes and their default values check the HTML class description of `svt_pcie_device_configuration` at the following location: *$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/configuration/class_svt_pcie_device_configuration.html*

8.2.2 Dynamic Configuration

The configuration set during the build of the OVM agent can be dynamically (during the `run_phase`) modified if the test requires a change in the VIP's configuration. The agent provides the following three ways to reconfigure the agent or its sub-components:

1. Using `svt_pcie_device_agent::reconfigure_via_task()`: This task can be called and a new configuration object with the desired programming can be specified as an input to this task.

2. Using `svt_pcie_device_agent::refresh_cfg()`: This task can be called to refresh the configuration of the agent or its sub-components based on a `ovm_config_db::set()` that is issued before calling `refresh_cfg()`.
3. Using the `REFRESH_CFG` service request: The PCIe Device Agent supports a service request interface via the service sequencer which is discussed under the sequencers sub-section. Using the service sequencer in the agent class a service request can be placed to refresh the configuration of the agent or sub-components based on a `ovm_config_db::set()` that is issued before requesting a `REFRESH_CFG` service.

The targeted modification can be on a specific layer of the VIP or across multiple layers depending on what the test is trying to achieve. In the Example 8-2, a typical use of dynamic reconfiguration is shown by using `reconfigure_vias_task()` task. In this example, the intent of reconfiguration is to cause the model to operate at a new PIPE width, PIPE clock rate and link speed after exiting `HOT_RESET`.

Example 8-2

```
class pcie_pipe_speed_width extends ovm_test;
   `ovm_component_utils(pcie_pipe_speed)
   …
   task run_phase (ovm_phase phase);
      svt_configuration temp_cfg = null;
      svt_pcie_device_configuration new_cfg = null;

      super.run_phase(phase);
      …
      // Assumptions:
      // The OVM test has an instance of OVM environment named 'env'
      // The OVM environment has an instance of the PCIe device agent named 'root'
      // VIP's DL is enabled.
      // LTSSM gets operational at 2.5G and uses a PIPE model.
      // Link operates at default PCLK rate/PIPE width defined in the PL.
      // PCLK at 2.5G = 250MHz; PIPE width = 8-bits
      // PCLK at 5.0G = 500MHz; PIPE width = 8-bits
      // PCLK at 8.0G = 1000MHz; PIPE width = 8-bits
      // The test initiates a transition to take the LTSSM to HOT_RESET

      wait(env.root.status.pcie_status.pl_status.ltssm_state ==
svt_pcie_types::HOT_RESET);
      // After a timeout the LTSSM is expected to be in DETECT
      wait(env.root.status.pcie_status.pl_status.ltssm_state ==
svt_pcie_types::DETECT_QUIET);

      // Fetch the current configuration of the PCIe device agent
      env.root.get_cfg_via_task(temp_cfg);
      $cast(new_cfg, temp_cfg.clone();

      // Program the new values for PCLK rate and PIPE width.
      // Note: the configuration variables are defined in the PL
      // Indices 0, 1 and 2 PCLK rate/PIPE width at Gen1, Gen2 and Gen3
      new_cfg.pcie_cfg.pl_cfg.pclk_rate[0] = svt_pcie_pl_configuration::PCLK_1000_MHZ;
      new_cfg.pcie_cfg.pl_cfg.pclk_rate[1] = svt_pcie_pl_configuration::PCLK_1000_MHZ;
      new_cfg.pcie_cfg.pl_cfg.pclk_rate[2] = svt_pcie_pl_configuration::PCLK_1000_MHZ;
      new_cfg.pcie_cfg.pl_cfg.pipe_width[0] = svt_pcie_pl_configuration::PIPE_8_BITS;
      new_cfg.pcie_cfg.pl_cfg.pipe_width[1] = svt_pcie_pl_configuration::PIPE_8_BITS;
      new_cfg.pcie_cfg.pl_cfg.pipe_width[2] = svt_pcie_pl_configuration::PIPE_8_BITS;

      // Reconfigure the VIP with the newly defined configuration
```

```
      new_cfg.pcie_cfg.pl_cfg.set_link_speed_values(`SVT_PCIE_SPEED_8_0G |
`SVT_PCIE_SPEED_5_0G | `SVT_PCIE_SPEED_2_5G);
      env.root.reconfigure_via_task(new_cfg);
   endtask
endclass
```

Similarly, a reconfiguration of the PCIe Device Agent or its sub-components can be performed using the `refresh_cfg()` function or by requesting a `REFRESH_CFG` service on the agent as illustrated in the Example 8-3.

Example 8-3

```
class pcie_pipe_speed_width extends ovm_test;
   `ovm_component_utils(pcie_pipe_speed)
   …
   task run_phase (ovm_phase phase);
      svt_configuration temp_cfg = null;
      svt_pcie_device_configuration new_cfg = null;
      svt_pcie_device_agent_service_sequence dev_agent_serv_seq;

      super.run_phase(phase);
      …
      // Assumptions:
      // The OVM test has an instance of OVM environment named 'env'
      // The OVM environment has an instance of the PCIe device agent named 'root'

      // Did a number of things…

      // About to change the configuration of the PCIe VIP

      // Fetch the current configuration of the PCIe device agent
      env.root.get_cfg_via_task(temp_cfg);
      $cast(new_cfg, temp_cfg.clone();

      // Modify members inside new_cfg
      // Call the ovm_config_db::set() to set the new configuration.
      ovm_config_db#(svt_pcie_device_configuration)::set(this,"env.root", "cfg",
new_cfg);

      // Refresh the agent with the new configuration (option 1)
      env.root.refresh_cfg();

         … OR …

      // Refresh the agent with the new configuration (option 2)
      dev_agent_serv_seq = new();
      dev_agent_serv_seq.service_type = svt_pcie_device_agent_service::REFRESH_CFG;
      dev_agent_serv_seq.start(env.root.device_agent_service_seqr);

      // Do other things
   endtask
endclass
```

8.3 Status

The PCIe Device Agent provides a set of state values representing the status of its sub-components at anytime in the test simulation. And these state values are encapsulated within the

svt_pcie_device_status class. The device agent class has an object of type svt_pcie_device_status instanced as status. For details about the attributes of this class, see HTML class description of the svt_pcie_device_status class available at the following location:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/status/class_svt_pcie_device_status.html

A test can access the status object of the device agent in the following two ways:

1. Directly accessing object svt_pcie_device_agent::status as illustrated in Example 8-2 where the test check for the LTSSM state as a control variable.
2. The test can use the svt_pcie_device_agent::get_device_status() function to retrieve the status of the agent and use a local copy of the status object as shown in Example 8-4.

Note The get_device_status function takes an input which is passed by reference.

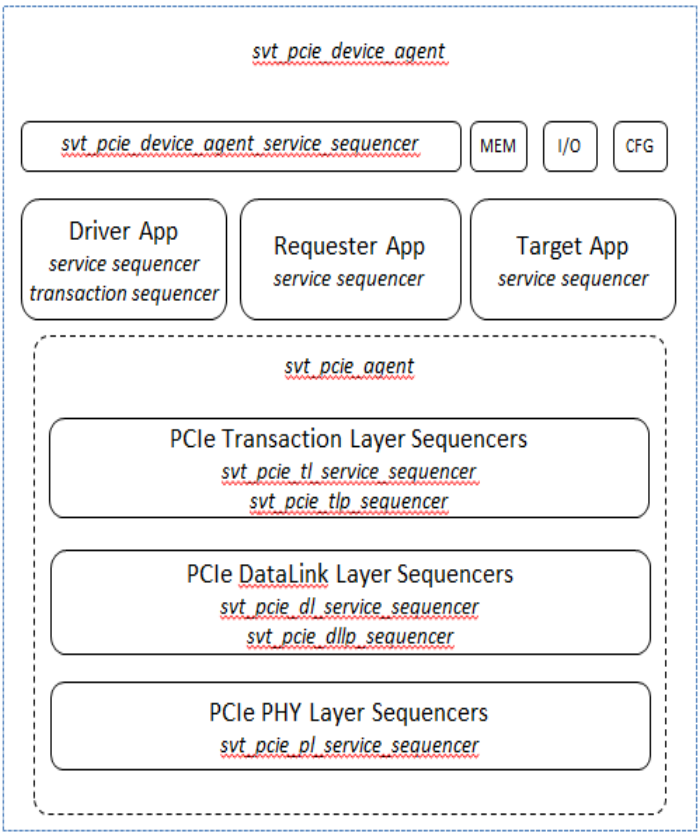
Example 8-4

```
class pcie_pipe_speed_width extends ovm_test;
   `ovm_component_utils(pcie_pipe_speed)
   …
   task run_phase (ovm_phase phase);
      svt_pcie_device_status root_dev_status;
      super.run_phase(phase);
      …
      // Assumptions:
      // The OVM test has an instance of OVM environment named 'env'
      // The OVM environment has an instance of the PCIe device agent named 'root'

      env.root.get_device_status(root_dev_status);

      // Use root_device_status to check the current status of any layer.
      // An example, LTSSM state value.

      // Do other things
   endtask
endclass
```

8.4 Sequencers

The PCIe Device Agent class (svt_pcie_device_agent) has eight OVM sequencer objects to schedule transaction requests or service requests on any of its subcomponent Drivers. A OVM sequencer is an arbiter that controls the transaction flow from multiple stimulus generators. The sequencers communicate with drivers using the TLM interfaces.

Figure 8-2 Block Diagram – Sequencers



Note Blocks named Mem, I/O and CFG are service sequencers corresponding to memory target, I/O target and configuration database OVM drivers.

Sequencers in the PCIe VIP are broadly classified as service sequencers and transaction sequencers.

8.4.1 Service Sequencers

A service sequencer is used to schedule sequences that are referred to as services on any OVM driver in the device agent class. An example of a service request can be a request on the memory target to write/read data in an attempt to load/store the completion memory of the VIP. Service sequences will not generate PCIe transactions on the PCIe bus but they are used to request a change in the behavior or sample a state value of the OVM driver.The device agent class includes the following seven service sequencer objects:

1. Configuration Database Service Sequencer (Type=`svt_pcie_cfg_database_service_sequencer`, Instance=`cfg_database_seqr`): A sequencer used to schedule services such as, backdoor write/read services to configuration database. It feeds service transactions of type `svt_pcie_cfg_database_service` to Sequencer Item Pull Port (SIPP) `ovm_seq_port` of OVM driver `svt_pcie_cfg_database` class.
2. PCIe Device Agent Service Sequencer (Type=`svt_pcie_device_agent_service_sequencer`, Instance=`device_agent_service_seqr`): A sequencer used to schedule services such as, refresh

configuration of the agent. It feeds service transactions of type `svt_pcie_device_agent_service` to SIPP `device_agent_service_seq_item_port` of OVM agent `svt_pcie_device_agent` class.

3. Driver Application Service Sequencer (Type=`svt_pcie_driver_app_service_sequencer`, Instance=`driver_seqr[0]`): A sequencer used to schedule services such as, applying reset to the driver application. It feeds service transactions of type `svt_pcie_driver_service` to the SIPP `service_seq_item_port` of OVM driver `svt_pcie_driver_app` class.
4. IO Target Service Sequencer (Type=`svt_pcie_io_target_service_sequencer`, Instance=`io_target_seqr`): A sequencer used to schedule services such as, backdoor write/read services to the I/O completion space of the VIP. It feeds service transactions of type `svt_pcie_io_target_service` to the SIPP `seq_item_port` of OVM driver `svt_pcie_io_target` class.
5. Memory Target Service Sequencer (Type=`svt_pcie_mem_target_service_sequencer`, Instance=`mem_target_seqr`): A sequencer used to schedule services such as, backdoor write/read services to the memory completion space of the VIP. It feeds service transactions of type `svt_pcie_mem_target_service` to the SIPP `seq_item_port` of OVM driver `svt_pcie_mem_target` class.
6. Requester Application Service Sequencer (Type=`svt_pcie_requester_app_service_sequencer`, Instance=`requester_seqr`): A sequencer used to schedule services such as starting the requester application. It feeds service transactions of type `svt_pcie_requester_app_service` to the SIPP `seq_item_port` of OVM driver `svt_pcie_requester_app` class.
7. Target Application Service Sequencer (Type=`svt_pcie_target_app_service_sequencer`, Instance=`target_seqr[0]`): A sequencer used to schedule services such as, starting the requester application. It feeds service transactions of type `svt_pcie_requester_app_service` to the SIPP `seq_item_port` of OVM driver `svt_pcie_requester_app` class.

Example 8-5 shows how you can request a service on the driver application using the driver application service sequencer. The requested service is to wait for the idle state of the driver application.

Example 8-5

```
class my_pcie_test extends ovm_test;
   `ovm_component_utils(my_pcie_test)
   …
   task run_phase (ovm_phase phase);
      svt_pcie_driver_app_service_wait_until_idle_sequence drv_app_serv_seq;
         …
      // Assumptions:
      // The OVM test has an instance of OVM environment named 'env'
      // The OVM environment has an instance of the PCIe device agent named 'root'
      // VIP's DL is enabled.
      // LTSSM is in L0
      // the test generated a number of transaction requests using the driver
application
      drv_app_serv_seq = new();
      drv_app_serv_seq.start(env.root.driver_seqr[0]);
      // End of test. The driver app is idle and so all transaction requests are done.
   endtask
endclass
```

8.4.2 Transaction Sequencers

A transaction sequencer is used to schedule sequences that cause PCIe transactions (TLPs and DLLPs) on the PCIe bus. There is only a single transaction sequencer object in the device agent class:

- Driver Application Transaction Sequencer (Type=`svt_pcie_driver_app_transaction_sequencer`, `Instance=driver_transaction_seqr[0]`): A sequencer used to schedule TLP transactions on the driver application of the VIP. It feeds transactions of type `svt_pcie_driver_app_transaction` to the SIPP `seq_item_port` of OVM driver class `svt_pcie_driver_app`.

Example 8-6 shows how you can send a transaction request to the driver using the driver application transaction sequencer.

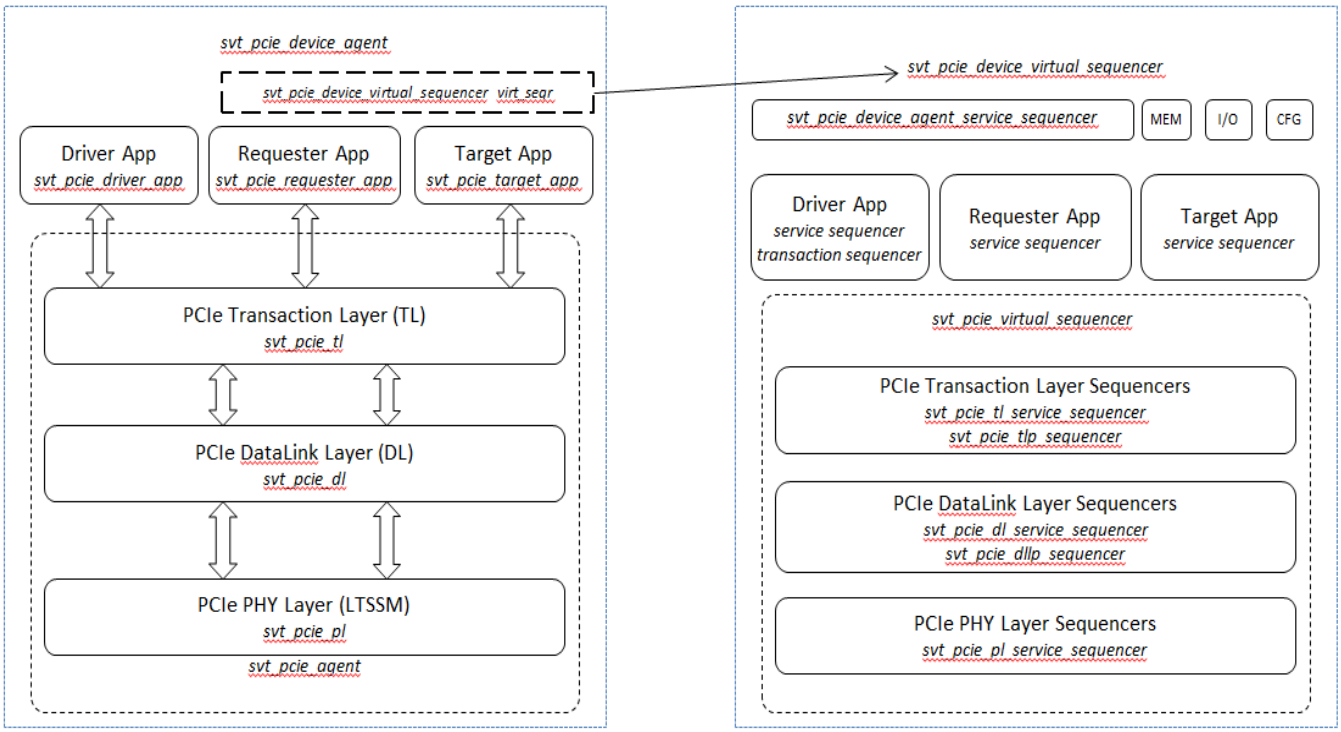
Example 8-6

```
class my_pcie_test extends ovm_test;
   `ovm_component_utils(my_pcie_test)
   …
   task run_phase (ovm_phase phase);
      svt_pcie_driver_app_transaction_mem_write_sequence mem_wr_seq;
         …
      // Assumptions:
      // The OVM test has an instance of OVM environment named 'env'
      // The OVM environment has an instance of the PCIe device agent named 'root'
      // VIP's DL is enabled.
      // LTSSM is in L0
      mem_wr_seq = new();
      mem_wr_seq.randomize with {
         address == 'h8000;
         length == 2;
         first_dw_be == 0;
         last_dw_be == 0;
         traffic_class == 0;
         address_translation == 2'b00;
         foreach(payload[i])
           payload[i] == 'hc0de_0000 + i;
      };
      mem_wr_seq.start(env.root.driver_transaction_seqr[0]);
      // Add end of test criteria.
      // End of test.
   endtask
endclass
```

8.4.3 Virtual Sequencer

The device agent class has a virtual sequencer object of type `svt_pcie_device_virtual_sequencer` instanced as `virt_seqr` that connects to all sequencers that are defined under its hierarchy. The sequencer class object has a hierarchical structure that mirrors `svt_pcie_device_agent` class. Instead of having OVM drivers/agents as sub-components, it has the OVM sequencer of the corresponding OVM driver/agent.

Figure 8-3 Block Diagram – Virtual Sequencer



Note Blocks named Mem, I/O and CFG are service sequencers corresponding to memory target, I/O target and configuration database OVM drivers.

The example that illustrates the use of the driver application service sequencer and driver application transaction sequencer can alternatively access these sequencers via the virtual sequencer instance as illustrated in Example 8-7 and Example 8-8.

Example 8-7

```
class my_pcie_test extends ovm_test;
   `ovm_component_utils(my_pcie_test)
   …
   task run_phase (ovm_phase phase);
      svt_pcie_driver_app_service_wait_until_idle_sequence drv_app_serv_seq;
         …
      // Assumptions:
      // The OVM test has an instance of OVM environment named 'env'
      // The OVM environment has an instance of the PCIe device agent named 'root'
      // VIP's DL is enabled.
      // LTSSM is in L0
      // the test generated a number of transaction requests using the driver
application
      drv_app_serv_seq = new();
      drv_app_serv_seq.start(env.root.virt_seqr.driver_seqr[0]);
      // End of test. The driver app is idle and so all transaction requests are done.
   endtask
```

```
endclass
```

Example 8-8

```
class my_pcie_test extends ovm_test;
   `ovm_component_utils(my_pcie_test)
   …
   task run_phase (ovm_phase phase);
      svt_pcie_driver_app_transaction_mem_write_sequence mem_wr_seq;
         …
      // Assumptions:
      // The OVM test has an instance of OVM environment named 'env'
      // The OVM environment has an instance of the PCIe device agent named 'root'
      // VIP's DL is enabled.
      // LTSSM is in L0
      mem_wr_seq = new();
      mem_wr_seq.randomize with {
         address == 'h8000;
         length == 2;
         first_dw_be == 0;
         last_dw_be == 0;
         traffic_class == 0;
         address_translation == 2'b00;
         foreach(payload[i])
           payload[i] == 'hc0de_0000 + i;
      };
      mem_wr_seq.start(env.root.virt_seqr.driver_transaction_seqr[0]);
      // Add end of test criteria.
      // End of test.
   endtask
endclass
```

The examples above shows the virtual sequencer as an alternative mechanism for generating services and transactions on the driver application without showcasing its real value. The real value of the virtual sequencer is seen when the PCIe device agent is part of a test environment that has other agents that drive stimulus to other possible interfaces on the DUT. As an example, consider a system-level environment that has both PCIe and USB VIP's being used to drive stimulus to the PCIe and USB interfaces of the SoC. To simplify test writing, a system wide virtual sequencer class can be defined to access both the PCIe VIP and USB VIP sequencers. And this system wide sequencer can be defined in the system environment and connected to the PCIe and USB VIP agents as shown in the Example 8-9.

Example 8-9

```
class my_system_virtual_sequencer extends ovm_sequencer;
   …
   svt_pcie_device_virtual_sequencer pcie_dev_virt_seqr;
   usb_device_virt_sequencer usb_virt_seqr;

   function new(…);
      …
   endfunction

endclass
class my_system_env extends ovm_env;
   svt_pcie_device_agent root;
   usb_device_agent usb_dev;
   my_system_virtual_sequencer sys_virt_seqr;
```

```
   …

   function void build_phase(ovm_phase phase);
      super.build_phase(phase);

      …
      this.sys_virt_seqr = my_system_virtual_sequencer::create("sys_virt_seqr", this);

   endfunction

   function void connect_phase(ovm_phase phase);
      super.connect_phase(phase);
      this.sys_virt_seqr.pcie_dev_virt_seqr = root.virt_seqr;
      this.sys_virt_seqr.usb_virt_seqr = usb_dev.virt_seqr;

      …
   endfunction
endclass
```

With this system-level sequencer defined, the OVM test will have a single reference to all sequencers that are part of the system. The test can drive stimulus to both the PCIe and USB interface by being oblivious to the agents or the hierarchies under them.

Example 8-10

```
class my_pcie_test extends ovm_test;
   `ovm_component_utils(my_pcie_test)
   …
   task run_phase (ovm_phase phase);
      svt_pcie_driver_app_transaction_mem_write_seqeunce pcie_wr_seq;
      usb_device_transaction_sequence usb_tr_seq;
         …

      pcie_wr_seq = new();
      pcie_wr_seq.randomize with {
         address == 'h8000;
         length == 2;
         first_dw_be == 4'b1111;
         last_dw_be == 4'b1111;
         traffic_class == 0;
         address_translation == 2'b00;
         foreach(payload[i])
           payload[i] == 'hc0de_0000 + i;
      };

      usb_tr_seq = new();
      usb_tr.randomize with {
         …
      };

      fork

pcie_wr_seq.start(env.sys_virt_seqr.pcie_dev_virt_seqr.driver_transaction_seqr[0]);
      usb_tr.start(env.sys_virt_seqr.usb_virt_seqr.ss_pkt_seqr);
      join
      // Add end of test criteria.
      // End of test.
   endtask
endclass
```

The USB VIP agent used in the example above is purely hypothetical. It does not represent the USB VIP product from Synopsys or any other vendor.

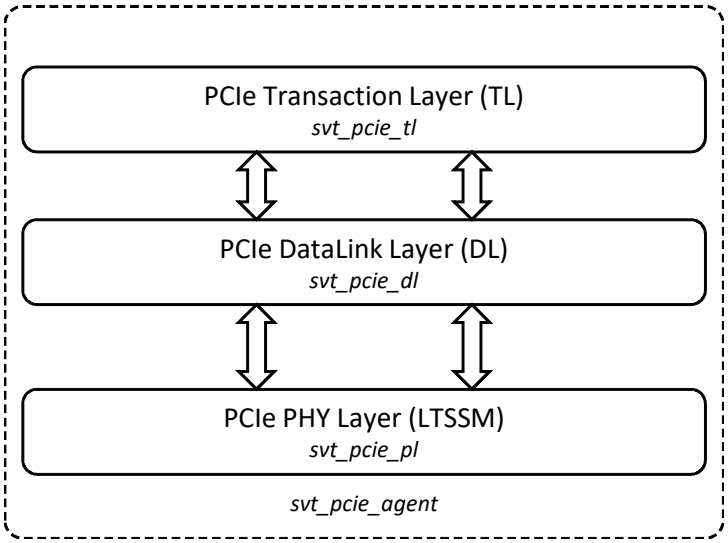
9 PCIe Agent

9.1 Overview

The PCIe Agent encapsulates the OVM drivers that represent the layered stack specified in the PCIe specification. Class `svt_pcie_agent` represents this encapsulation in the PCIe VIP. The PCIe agent class is instanced as `pcie_agent` within `svt_pcie_device_agent`. The agent class consists of the following layers:

1. The Transaction Layer (Type=`svt_pcie_tl`, Instance=`pcie_tl`): Class `svt_pcie_tl` defines the functions of the transaction layer (TL) in the PCIe VIP. The applications transfer transaction requests to the TL for transmission. The TL composes the transaction layer packet (TLP) and hands it down to the data-link layer located below it. The transaction layer also receives TLPs from the data-link layer which gets routed up to the correct application. It is also possible for the test to interface directly with the TL to generate TLPs. But it is recommended to use the application layers.
2. The Data-link Layer (Type=`svt_pcie_dl`, Instance=`pcie_dl`): Class `svt_pcie_dl` defines the functions of the data-link (DL) layer in the PCIe VIP. The TL transfers TLPs to the DL to be framed with a sequence number and a CRC and ensure the remote receiver receives the packet without any errors. The DL also performs the other standard functions of link management using data-link layer packets (DLLPs).
3. The Physical Layer (Type=`svt_pcie_pl`, Instance=`pcie_pl`): Class `svt_pcie_pl` defines the functions of the physical layer (PL) in the PCIe VIP. The PL breaks down packets it receives (from the DL) into symbols and encodes them as per the specification before transmission. It also composes packets from data received on the receive data lanes and sends them back to the DL. It also performs other functions to maintain the link such as the LTSSM and functions for low power and so on.

Figure 9-1 Block Diagram – PCIe Agent



9.2 Configuration

The PCIe Agent is configured using an object of class type `svt_pcie_configuration`. An object of this type with an instance name of `pcie_cfg` is defined in `svt_pcie_device_configuration` class. This class is comprised of data members and class object members. The object members represent the configuration of sub-components of the PCIe Agent class.

```
svt_pcie_device_configuration
            |
            |--------> driver_cfg[]   (type=svt_pcie_driver_app_configuration)
            |
            |--------> requester_app  (type=svt_pcie_requester_app_configuration)
            |
            |--------> target_cfg[]   (type=svt_pcie_target_app_configuration)
            |
            |--------> pcie_cfg       (type=svt_pcie_configuration)
                           |
                           |--------> tl_cfg (type=svt_pcie_tl_configuration)
                           |
                           |--------> dl_cfg (type=svt_pcie_dl_configuration)
                           |
                           |--------> pl_cfg (type=svt_pcie_pl_configuration)
```

The `svt_pcie_configuration` class also has data members that define the behavior of the agent. For example, `svt_pcie_configuration::enable_cov` is a configuration variable that enables functional coverage.

Note `svt_pcie_configuration::enable_cov` is a 6-bit variable and each bit enables a specific kind of coverage. For details about this variable and other variables, see HTML class description of the `svt_pcie_configuration` class available at the following location: *$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/configuration/class_svt_pcie_configuration.html*

9.2.1 Initial Configuration

The initial configuration of the `svt_pcie_agent` or its sub-components is set as illustrated in "Initial Configuration" of the PCIe Device Agent section. The configuration attributes defined within the `svt_pcie_device_configuration::pcie_agent` and its sub-configuration objects can be programmed before the `ovm_config_db::set()` call illustrated in Example 8-2.

9.2.2 Dynamic Configuration (reconfiguration)

Dynamic configuration changes to the configuration of the `svt_pcie_agent` or its sub-components can be made as defined in the "Dynamic Configuration" of the PCIe Device Agent section. The members of `svt_pcie_device_agent::pcie_cfg` can be modified before the calls to reconfigure the configuration of the device agent as illustrated in Example 8-2 and Example 8-3.

9.3 Status

The PCIe Agent has a set of state values representing the status of its constituent blocks at any given time in the test simulation. And these state values are encapsulated within the `svt_pcie_status` class. The `svt_pcie_device_status` class has an object of type `svt_pcie_status` instanced as `pcie_status`.

For details about the attributes of this class, see HTML class description of the `svt_pcie_status` class available at the following location:

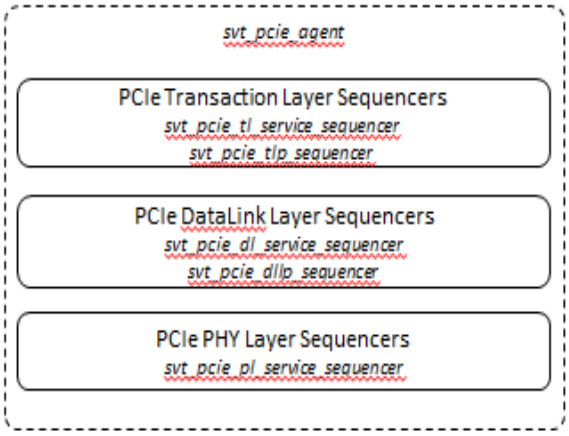
$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/status/class_svt_pcie_status.html

The test can access these state variables as described by the status section of the PCIe device agent class. Example 8-2 referenced by that section uses `svt_pcie_device_status::status.pl_status.ltssm_state` as a control variable.

9.4 Sequencers

The PCIe Agent class (`svt_pcie_agent`) has six OVM sequencer objects to schedule transaction requests or service requests on any of its subcomponent drivers. A OVM sequencer is an arbiter that controls the transaction flow from multiples stimulus generators. The sequencers communicate with drivers using TLM interfaces.

Figure 9-2 Block Diagram – Sequencers



Sequencers in the PCIe Agent are broadly classified as service sequencers and transaction sequencers.

9.4.1 Service Sequencers

A Service Sequencer is used to schedule sequences that are referred to as services on any OVM driver in the PCIe Agent class. An example of a service request can be a request on the Physical layer to initiate a link width change. Service sequences will not generate PCIe transactions on the PCIe bus, but they are used to request a change in the behavior or sample a state value of the OVM driver. The PCIe Agent class includes the following three service sequencer objects:

1. Data-link Layer Service Sequencer (Type=`svt_pcie_dl_service_sequencer`, Instance=`dl_seqr`): A sequencer used to schedule services such as enabling of the Data-link layer driver. It feeds service transactions of type `svt_pcie_dl_service` to SIPP (Sequencer Item Pull port) `seq_item_port` of OVM driver `svt_pcie_dl` class.
2. Physical Layer Service Sequencer (Type=`svt_pcie_pl_service_sequencer`, Instance=`pl_seqr`): A sequencer used to schedule services such as speed change on the Physical layer driver. It feeds service transactions of type `svt_pcie_pl_service` to SIPP (Sequence Item Pull Port) `seq_item_port` of OVM driver `svt_pcie_pl` class.
3. Transaction Layer Service Sequencer (Type=`svt_pcie_tl_service_sequencer`, instance=`tl_seqr`): A sequencer used to schedule services such as setting up a traffic class map. It feeds service transactions of type `svt_pcie_tl_service` to the SIPP (Sequence Item Pull Port) `service_seq_item_port` of OVM driver `svt_pcie_tl` class.

Example 9-1 shows how you can request a service on the TL via the TL service sequencer. The requested service is to map a given traffic class to a VC.

Example 9-1

```
class my_pcie_test extends ovm_test;
   `ovm_component_utils(my_pcie_test)
   …
   task run_phase (ovm_phase phase);
      svt_pcie_tl_service_set_tc_map_sequence tl_serv_seq;
         …
      // Assumptions:
      // The OVM test has an instance of OVM environment named 'env'
```

```
      // The OVM environment has an instance of the PCIe device agent named 'root'
      tl_serv_seq = new();

      // Enable TC value 1. By default only TC=0 is enabled and mapped to VC0
      // Map TC=1 to VC1
      tl_serv_seq.tc_enable = 1;
      tl_serv_seq.tc_num    = 1;
      tl_serv_seq.vc_num    = 1;
      tl_serv_seq.start(env.root.pcie_agent.tl_seqr);

      // Map TC=2 to VC2
      tl_serv_seq.tc_enable = 1;
      tl_serv_seq.tc_num    = 2;
      tl_serv_seq.vc_num    = 2;
      tl_serv_seq.start(env.root.pcie_agent.tl_seqr);

      // Map TC=2 to VC2
      tl_serv_seq.tc_enable = 1;
      tl_serv_seq.tc_num    = 3;
      tl_serv_seq.vc_num    = 3;
      tl_serv_seq.start(env.root.pcie_agent.tl_seqr);

      Etc.,

      // Generate transaction requests.

      // End of test. Check for an idle state of testbench blocks before ending test.
   endtask
endclass
```

9.4.2 Transaction Sequencers

A transaction sequencer is used to schedule sequences that cause PCIe transactions (TLPs, DLLPs) on the PCIe bus. The two transaction sequencers in the PCIe Agent class are as follows:

1. TLP Transaction Sequencer (Type= `svt_pcie_tlp_sequencer`, Instance=`tlp_seqr`): A sequencer used to schedule TLP transactions on the transaction layer of the VIP. It feeds transactions of type `svt_pcie_tlp` to the SIPP (Sequence Item Pull Port) `seq_item_port` of OVM driver class `svt_pcie_tl`.
2. DLLP Transaction Sequencer (Type=`svt_pcie_dllp_sequencer`, Instance=`dllp_seqr`): A sequencer used to schedule DLLP transactions on the data-link layer of the VIP. It feeds transactions of type `svt_pcie_dllp` to the SIPP (Sequence Item Pull Port) `seq_item_port` of OVM driver class `svt_pcie_dl`.

Example 9-2 shows how you can schedule a TLP transaction request to the TL using the TLP sequencer.

Example 9-2

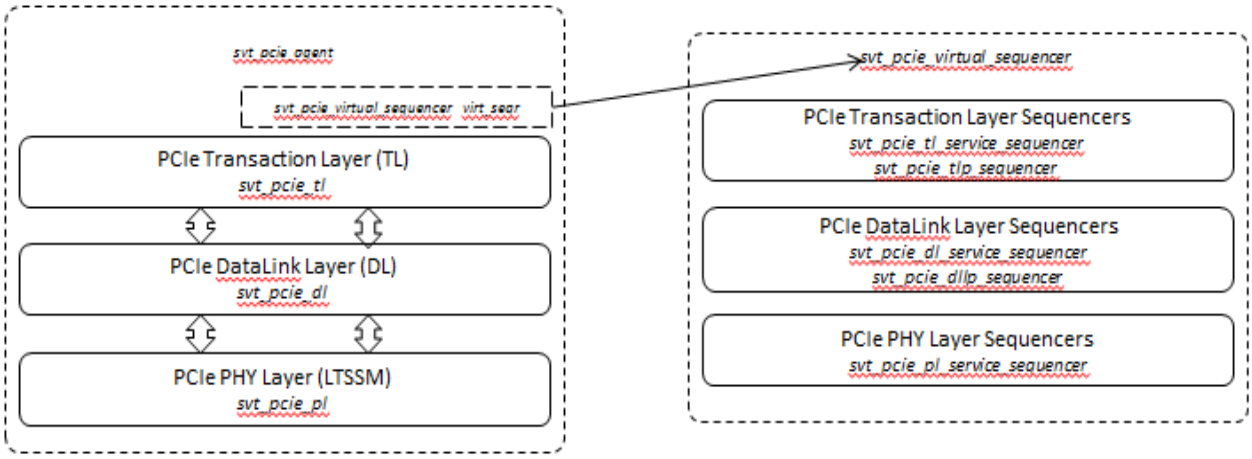
```
class my_pcie_test extends ovm_test;
   `ovm_component_utils(my_pcie_test)
   …
   task run_phase (ovm_phase phase);
      svt_pcie_tlp_mem_request_sequence mem_tlp_seq;
         …
      // Assumptions:
      // The OVM test has an instance of OVM environment named 'env'
      // The OVM environment has an instance of the PCIe device agent named 'root'
```

```
        // VIP's DL is enabled.
        // LTSSM is in L0
        mem_tlp_seq = new();
        mem_tlp_seq.randomize with {
           address == 'h8000;
           length == 2;
           requester_id == 'h100;
           tlp_type == svt_pcie_tlp::MEM_REQ;
           fmt == svt_pcie_tlp::WITH_DATA_3_DWORD;
           // Program other header fields.
            …
           foreach(payload[i])
             payload[i] == 'hc0de_0000 + i;
        };
        mem_tlp_seq.start(env.root.pcie_agent.tlp_seqr);
        // Add end of test criteria.
        // End of test.
    endtask
endclass
```

9.4.3 Virtual Sequencer

The PCIe Agent class has a virtual sequencer object of type `svt_pcie_virtual_sequencer` instanced as `virt_seqr` that connects to all sequencers that are defined under its hierarchy. The sequencer class object has a hierarchical structure similar to the `svt_pcie_agent` class. Instead of having OVM drivers/agents as subcomponents, it has the OVM sequencer of the corresponding OVM drivers/agents.

Figure 9-3 Block Diagram – Virtual Sequencer



The example that illustrates the use of the TL service sequence and TLP transaction sequencer can alternatively access these sequencers via the virtual sequencer instance as illustrated in Example 9-3 and Example 9-4.

Example 9-3

```
class my_pcie_test extends ovm_test;
   `ovm_component_utils(my_pcie_test)
   …
   task run_phase (ovm_phase phase);
```

```
      svt_pcie_tl_service_set_tc_map_sequence tl_serv_seq;
         …
      // Assumptions:
      // The OVM test has an instance of OVM environment named 'env'
      // The OVM environment has an instance of the PCIe device agent named 'root'
      tl_serv_seq = new();

      // Enable TC value 1. By default only TC=0 is enabled and mapped to VC0
      // Map TC=1 to VC1
      tl_serv_seq.tc_enable = 1;
      tl_serv_seq.tc_num    = 1;
      tl_serv_seq.vc_num    = 1;
      tl_serv_seq.start(env.root.pcie_agent.virt_seqr.tl_seqr);

      // Map TC=2 to VC2
      tl_serv_seq.tc_enable = 1;
      tl_serv_seq.tc_num    = 1;
      tl_serv_seq.vc_num    = 1;
      tl_serv_seq.start(env.root.pcie_agent.virt_seqr.tl_seqr);

      // Map TC=2 to VC2
      tl_serv_seq.tc_enable = 1;
      tl_serv_seq.tc_num    = 1;
      tl_serv_seq.vc_num    = 1;
      tl_serv_seq.start(env.root.pcie_agent.virt_seqr.tl_seqr);

      Etc.,

      // Generate transaction requests.

      // End of test. Check for an idle state of testbench blocks before ending test.
   endtask
endclass
```

Example 9-4

```
class my_pcie_test extends ovm_test;
   `ovm_component_utils(my_pcie_test)
   …
   task run_phase (ovm_phase phase);
      svt_pcie_tlp_mem_request_sequence mem_tlp_seq;
         …
      // Assumptions:
      // The OVM test has an instance of OVM environment named 'env'
      // The OVM environment has an instance of the PCIe device agent named 'root'
      // VIP's DL is enabled.
      // LTSSM is in L0
      mem_tlp_seq = new();
      mem_tlp_seq.randomize with {
         address == 'h8000;
         length == 2;
         requester_id == 'h100;
         tlp_type == svt_pcie_tlp::MEM_REQ;
         fmt == svt_pcie_tlp::WITH_DATA_3_DWORD;
         // Program other header fields.
          …
         foreach(payload[i])
           payload[i] == 'hc0de_0000 + i;
      };
```

```
      mem_tlp_seq.start(env.root.pcie_agent.virt_seqr.tlp_seqr);
      // Add end of test criteria.
      // End of test.
   endtask
endclass
```

Example 9-3 and Example 9-4 illustrate the use of the `svt_pcie_agent::virt_seqr` to access sequencers that are part of `svt_pcie_agent` class. But it finds practical usage in cases where the PCIe VIP agent is part of a test environment that has other agents that drive stimulus to other possible interfaces on the DUT. With such a system-level testbench, a system wide virtual sequencer class can be defined to access both PCIe VIP and sequencers that are part of other VIP agents. And this system wide sequencer can be defined in the system environment and connected to the sequencers of the PCIe VIP and sequencers of other VIP agents. Example 9-5 shows the typical usage.

Example 9-5

```
class my_system_virtual_sequencer extends ovm_sequencer;
   …
   svt_pcie_device_virtual_sequencer pcie_dev_virt_seqr;
   usb_device_virt_sequencer usb_virt_seqr;

   function new(…);
      …
   endfunction
endclass
class my_system_env extends ovm_env;
   svt_pcie_device_agent root;
   usb_device_agent usb_dev;
   my_system_virtual_sequencer sys_virt_seqr;

   …

   function void build_phase(ovm_phase phase);
      super.build_phase(phase);

      …
      this.sys_virt_seqr = my_system_virtual_sequencer::create("sys_virt_seqr", this);

   endfunction

   function void connect_phase(ovm_phase phase);
      super.connect_phase(phase);
      this.sys_virt_seqr.pcie_dev_virt_seqr = root.virt_seqr;
      this.sys_virt_seqr.usb_virt_seqr = usb_dev.virt_seqr;

      …
   endfunction
endclass
class my_pcie_test extends ovm_test;
   `ovm_component_utils(my_pcie_test)
   …
   task run_phase (ovm_phase phase);
      svt_pcie_tlp_mem_request_sequence mem_tlp_seq;
      usb_device_transaction_sequence usb_tr_seq;
         …

      mem_tlp_seq = new();
      mem_tlp_seq.randomize with {
```

```
            address == 'h8000;
            length == 2;
            requester_id == 'h100;
            tlp_type == svt_pcie_tlp::MEM_REQ;
            fmt == svt_pcie_tlp::WITH_DATA_3_DWORD;
            // Program other header fields.
             …
            foreach(payload[i])
              payload[i] == 'hc0de_0000 + i;
          };

          usb_tr_seq = new();
          usb_tr.randomize with {
            …
          };

          fork
          mem_tlp_seq.start(env.sys_virt_seqr.pcie_dev_virt_seqr.pcie_virt_seqr.tlp_seqr);
          usb_tr.start(env.sys_virt_seqr.usb_virt_seqr.ss_pkt_seqr);
          join
          // Add end of test criteria.
          // End of test.
        endtask
      endclass
```

Note The USB VIP agent used in the example above is purely hypothetical. It does not represent the USB VIP product from Synopsys or any other vendor.

Note Example 9-5 accesses the TLP sequencer using the device agent virtual sequencer PCIe Device agent (`svt_pcie_device_agent`) class `svt_pcie_device_agent::virt_seqr.pcie_virt_seqr.tlp_seqr`. And this is same as accessing the TLP sequencer via the virtual sequencer defined inside `svt_pcie_agent` class `svt_pcie_agent::virt_seqr.tlp_seqr`. The latter is illustrated in Example 9-4.

10 Using the Transaction Layer

10.1 Transaction Layer

The Transaction Layer, TL, is implemented as a ovm_driver, svt_pcie_tl. The TL contains a configuration object, svt_pcie_tl_configuration (see "Transaction Layer Configuration"), a service sequencer (see "Transaction Layer Sequencer and Sequences"), which work together to setup and control the behavior of the TL. Additionally, the TL offers callbacks with exception capability (see "Transaction Layer Callbacks and Exceptions"), as well as a status object "Transaction Layer Status".

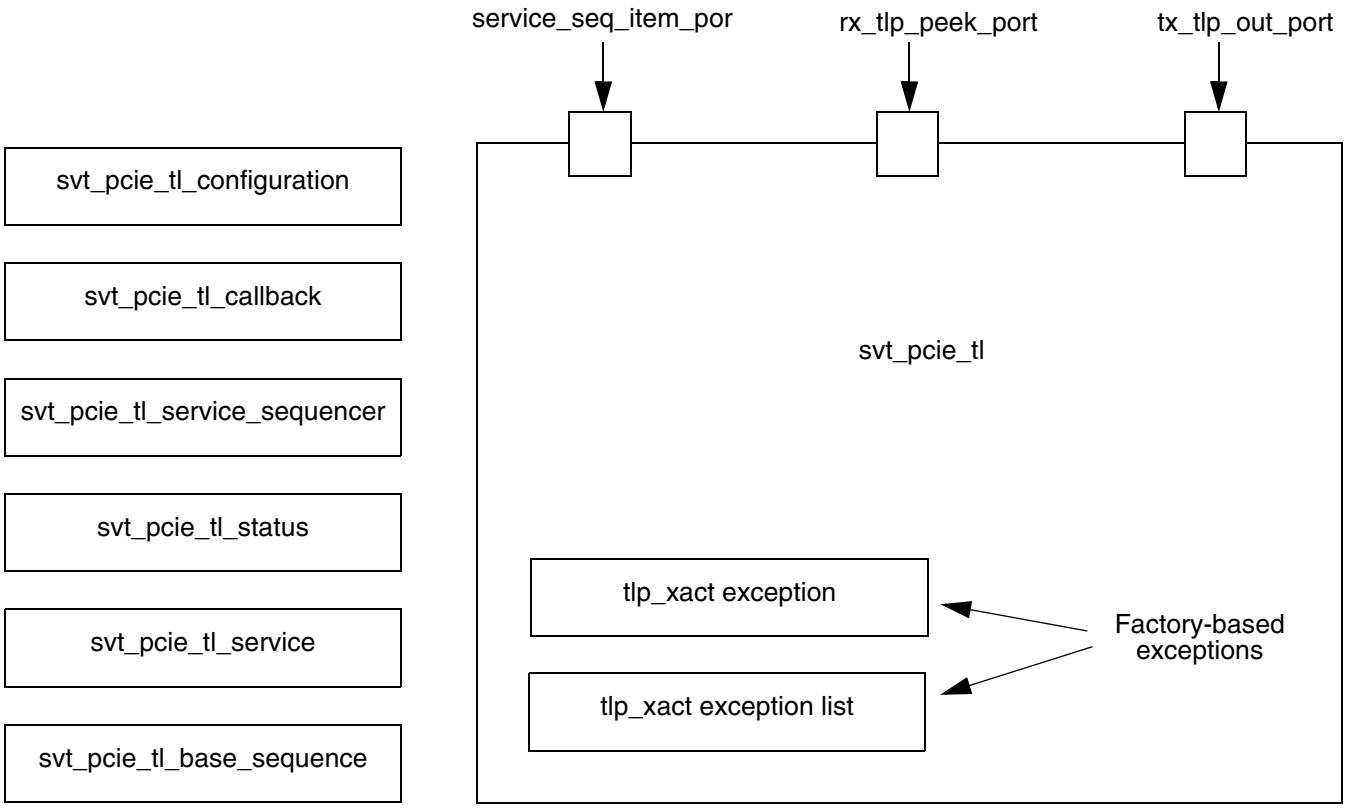
Attention Note, the descriptions for the classes and applications show the most important and often used members/features. Consult the PCIe OVM HTML Class Reference for a complete listing of the members and their data types.

Consult the following to get information on TLP related programming tasks.

- SolvNetPlus PCIe VIP Articles.
- PCIe SVT FAQ

A block diagram of the Transaction Layer elements is shown in Figure 10-1.

Figure 10-1 Transaction Layer block diagram



The VIP TL layer is analogous to that of the transaction Layer of the PCIe specification. The Transaction Layer encapsulates transactions generated by an application into TLPs. It also performs traffic class (TC) to virtual channel (VC) mapping, utilizes a credit-based flow control with the remote link, and checks and enforces TLP ordering rules. VC0 is automatically initialized with default credits.

Note If you use Virtual Channels other than VC0, those Virtual Channels must be initialized. To initialize VC1-VC7, credits must be initialized. Use svt_pcie_tl_configuration::init*_tx_credits followed by a call to svt_pcie_tl_service_tl_set_vc_en_sequence to set up the VCs.

10.2 Transaction Layer Configuration

The transaction layer configuration class is `svt_pcie_tl_configuration`. The members within the class control the settings of the Transaction Layer, TL.

The class is accessed via the following instance hierarchy:

instance name of svt_pcie_device_configuration.pcie_cfg.tl_cfg

For details about the attributes of this class, see HTML class description of the `svt_pcie_tl_configuration` class available at the following location:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/configuration/class_svt_pcie_tl_configuration.html

10.2.1 Verilog Configuration Parameters

Not all configuration items are currently controlled from within the OVM interface. At this time the items in "Compile-time Verilog Parameters" and "Runtime-changeable Verilog Parameters" are controlled only via Verilog.

10.2.1.1 Compile-time Verilog Parameters

Parameters that are only changeable when instantiating the TL as part of the instantiation model are listed in Table 10-1.

Table 10-1 Transaction Layer runtime Verilog parameters

Parameter Name	Type	Range	Default Value	Description
DEFAULT_ROUTE_AT_APPL_ID				
	Integer	0 - large value	0	Default Application ID to route Address Translation requests to.
NUM_APPL_ID				
	Integer	8-128	8	Max number of unique Application IDs. IDs assigned to applications must be less than this value.
RID_APPLID_TABLE_SIZE				
	Integer	4-4096	64	Number of unique RID to Appl_id map entries.
RID_MSGCODE_APPLID_TABLE_SIZE				
	Integer	4-4096	64	Number of unique {RID,msgcode} to Appl_id map entries.
MEM_ADDR_ADDPLID_TABLE_SIZE				
	Integer	4-4096	64	Number of unique Mem Address to Appl_id map entries.
IO_ADDR_ADDPLID_TABLE_SIZE				
	Integer	4-4096	64	Number of unique I/O Address to Appl_id map entries.
AT_ADDR_ADDPLID_TABLE_SIZE				
	Integer	4-4096	64	Number of unique AT Address to Appl_id map entries.

10.2.1.2 Runtime-changeable Verilog Parameters

Transaction Layer parameters that are changeable at runtime are listed in Table 10-2.

Table 10-2 Transaction Layer runtime Verilog parameters

Parameter Name	Type	Range	Default Value	Description
MAX_NUM_END_TO_END_PREFIXES				
	Integer		4	Max number of prefixes allowed. Version 3 only.

10.3 Transaction Layer Sequencer and Sequences

The transaction layer supports both service and transaction sequences. All TL sequences run on the svt_pcie_tl_service_sequencer. The TL sequencer is accessed through one of the following paths:

Virtual Sequencer:

This is the path through the virtual sequencer, virt_seqr. The virtual sequencer contains references to all of the nonvirtual sequencers. This is made available to enable the development of a high level virtual sequence that coordinates between all the nonvirtual sequencers. The path is

instance name of svt_pcie_agent.virt_seqr.tl_seqr

Sequencer:

This is the instantiated path to the nonvirtual sequencer:

instance name of svt_pcie_agent.tl_seqr

The non-virtual and virtual sequencers both are valid access points to the TL sequencer. Either may be used, though the virtual version is recommended when coordinating between multiple non-virtual sequencers.

Services are commands to give the model that are related to behavior or configuration. They are not transaction items that are to be sent across the bus. For the TL there is a base service, svt_pcie_tl_service, and there is a base service sequence, svt_pcie_tl_service_base_sequence.

The service, svt_pcie_tl_service, is a sequence_item that supports all the transaction types supported by the TL. A service is selected by constraining the service_type_enum of the class to one of the enumerated service types.

Alternatively, the model provides several sequences that contain the base svt_pcie_tl_service that can be used for test development.

Example 10-1

```
svt_pcie_tl_check_final_credits_sequence fc_seq;
…
`ovm_do_on (fc_seq, p_sequencer.root_virt_seqr.pcie_virt_seqr.tl_seqr);
…
```

For details about the service sequences of this class, see the `'TL_SERVICE_SEQUENCES'` tab under HTML class description available at the following location:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/sequencepages.html

In addition to the sequence library, the Transaction Layer provides the `svt_pcie_tl_service`, which is a sequence item that can be used to build custom sequences. Enumerated values of the service type and their associated attributes are listed in Table 10-3.

Table 10-3 Service type enumerated parameters

Parameter	Attributes	I/O	Attribute Description
ADD_MEM_ADDR_APPL_ID_MAP_ENTRY Used to map memory addresses to an application.			
	memory_addr [63:0]	I	Base address of the memory range.
	memory_window [63:0]	I	Window of addresses that will cause a match of entry.
	appl_id [31:0]	I	Application ID to map TLP to.
	error [1]	O	Indication that addition of new entry failed.
ADD_IO_ADDR_APPL_ID_MAP_ENTRY Used to map I/O addresses to an application.			
	memory_addr [63:0]	I	Base address of the memory range .
	memory_window [63:0]	I	Window of addresses that will cause a match of entry.
	appl_id [31:0]	I	Application ID to map TLP to.
	error [1]	O	Indication that addition of new entry failed.
ADD_IO_ADDR_APPL_ID_MAP_ENTRY Used to map memory addresses that need address translation to an application.			
	memory_addr [63:0]	I	Base address of the memory range.
	memory_window [63:0]	I	Window of addresses that will cause a match of entry.
	appl_id [31:0]	I	Application ID to map TLP to.
	error [1]	O	Indication that addition of new entry failed.
ADD_AT_ADDR_APPL_ID_MAP_ENTRY Used to map memory addresses that need address translation to an application.			
	memory_addr [63:0]	I	Base address of the memory range.
	memory_window [63:0]	I	Window of addresses that will cause a match of entry.
	appl_id [31:0]	I	Application ID to map TLP to.
	error [1]	O	Indication that addition of new entry failed.
ADD_CFG_BDF_APPL_ID_MAP_ENTRY Used to map Config Request {Bus, Device, Function} to applications. Used when the VIP is the upstream port of a link. This mapping is enabled via the ENABLE_ROUTE_CFG_ TYPE[0|1]_TO_ FUNCTION parameter.			
	config_type [1]	I	If true, the onfiguration is a Type 1 request. If false, the configuration is a Type 0 request..
	bdf [15:0]	I	{bus, device, function} of the request.
	appl_id [31:0]	I	Application ID to map TLP to.
	error [1]	O	Indication that addition of new entry failed.

Table 10-3 Service type enumerated parameters (Continued)

ADD_RID_APPL_ID_MAP_ENTRY Used to map Requester IDs to applications. This table is automatically populated when TLPs are sent by the Transaction Layer. This mapping is always enabled.			
	requester_id [15:0]	I	Requester ID to map.
	appl_id [31:0]	I	Application ID to map TLP to.
	error [1]	O	Indication that addition of new entry failed.
ADD_RID_MSG_CODE_APPL_ID_MAP_ENTRY Used to map {Requester Ids, msgcode} of MSG TLPs to applications.			
	requester_id [15:0]	I	Requester ID to map.
	msgcode [7:0]	I	Msgcode of TLP. Same value as msgcode field in TLP header. See Include/pciesvc_parms.v file for defines.
	appl_id [31:0]	I	Application ID to map TLP to.
	error [1]	O	Indication that addition of new entry failed.
DISPLAY_MEM_ADDR_APPL_ID_MAP Displays all memory addressses.			
DISPLAY_IO_ADDR_APPL_ID_MAP Displays all I/O address to application ID map entries.			
DISPLAY_AT_ADDR_APPL_ID_MAP Displays all memory address to application ID map entries that require address translation.			
DISPLAY_CFG_BDF_APPL_ID_MAP Displays all configuration Type 0 and Type 1 {Bus, Device, Function} to application ID map entries. Use when the VIP is the upstream port of a link.			
DISPLAY_RID_APPL_ID_MAP Displays all Requester ID to application ID map entries. Use when the VIP is the upstream port of a link.			
DISPLAY_RID_MSG_CODE_APPL_ID_MAP Displays all {Requester ID, MsgCode} to application ID map entries.			
DISPLAY_STATS Displays all stats in the Transaction Layer.			
CLEAR_STATS Clears all stats in the Transaction Layer.			
CHECK_FINAL_CREDITS Compares initial allocated credits to final allocated – received credit values. Compares initial Limit credits to final limit – consumed credit values. Warnings are issued if any credits are lost.			
SET_VC_ENABLE Enable or disable a virtual channel. Called for each VC to enable. VC0 is enabled by default.			

Table 10-3 Service type enumerated parameters (Continued)

	rand bit vc_enable	I	Enable or disable the VC.
	rand bit[31:0] vc_num	I	Virtual channel number.
IS_TL_IDLE Indicates whether the Transaction Layer is currently idle. **Note**: IS_TL_IDLE will be deprecated in a future release. The preferred way to check whether the TL is idle is to use the status object at shown in "Determining if the Transaction Layer is Idle".			
	rand bit tl_idle	O	Returns the TL status: tl_idle == 0 means not idle, tl_idle == 1 means idle.

10.4 Transaction Layer Callbacks and Exceptions

The Transaction Layer provides a callback class, svt_pcie_tl_callback, for applying exceptions to TLP transactions. For more information on the TL callbacks and exceptions, refer to Chapter 16.6.

10.5 Transaction Layer Status

The transaction layer provides a status class that provides statistics regarding the TL layer. The class is accessed using the following instance hierarchy:

instance name of svt_pcie_agent.status.tl_status

Refer to the HTML Reference for a full listing of status members.

10.5.1 Determining if the Transaction Layer is Idle

The TL provides the svt_pcie_tl_status::is_idle member for determining if the Transaction Layer is idle (that is, all queues are empty). This is useful for determining end-of-test.

Example 10-2

```
svt_pcie_device_status root_status = p_sequencer.get_root_shared_status(this);
wait(root_status.pcie_status.tl_status.is_idle);
```

Additionally, you can use the svt_pcie_tl_service to run a service to check on the status. See IS_TL_IDLE in Table 10-3.

Note In a future release the IS_TL_IDLE parameter will be deprecated. The preferred way to check whether the Transaction Layer is idle is to use the status object as shown in Example 10-2.

10.6 Transaction Layer TLMs

The svt_pcie_tl provides TLMs for access to the received TLPs. There is a ovm_blocking_put_port and a ovm_blocking_peek_imp, which are rx_tlp_out_put and rx_ltp_peek_port, respectively. You can connect to those ports for access to all received TLPs.

Additonally, there is a ovm_seq_item_pull_port, service_seq_item_port that is used for service requests and user applications.

10.7 Transaction Layer Verilog Interface

The Verilog component of the TL is instantiated within the MAC. It can be found at:

path to instantiation model.port0.tl0

The signals at this level are useful for debugging. A few of the signals are highlighted in the following sections.

10.7.1 Transaction Layer Module IOs

The Transaction Layer module I/O signals listed in Table 10-4 are the Verilog module port connections to the TL layer. These are useful for browsing the VIP reset and checking if a particular VC is initialized.

Table 10-4 Transaction Layer Module IOs

Name	I/O	Description
reset	I [1]	Active high reset. Must only be asserted for 100ns ONCE per simulation at the beginning.
dl_status	I [31:0]	Bits (use predefined parameters for access): [0] = link_up. [8] = VC0 initialized [9] = VC1 initialized [10] = VC2 initialized [11] = VC3 initialized [12] = VC4 initialized [13] = VC5 initialized [14] = VC6 initialized [15] = VC7 initialized

11 Data Link Layer Features and Classes

11.1 Classes and Applications for Using the VIP's Data Link Layer

The following classes have members and tasks to implement Data Link Layer features and operation.

- Power Management: Describes power management at DL Layer.
- Component Class svt_pcie_dl: A ovm_component which implements the PCIe Data Link Layer. This class is included in the svt_pcie_agent. When you instantiate the PCIe OVM agent, you will also instantiate this component.
- Configuration class svt_pcie_dl_configuration: This class contains class members to configure the behavior of the Data Link Layer. For example, the class has the member svt_pcie_dl_configuration::replay_timeout which you would use to configure the length of the replay timer in symbols.
- Status class svt_pcie_dl_status: Used for returning status and statistics back to your testbench.
- Service Class svt_pcie_dl_service: Service transactions for Link layer module. For example, if you want to initiate a transition to the L0 state from the PM low power state, then you would use the member INITIATE_PM_EXIT(10).

The following section lists and describes the members and tasks of the major classes to implement various features for your testbench.

Attention Note, the descriptions for the classes and applications show the most important and often used members/features. Consult the PCIe OVM HTML Class Reference for a complete listing of the members and their data types.

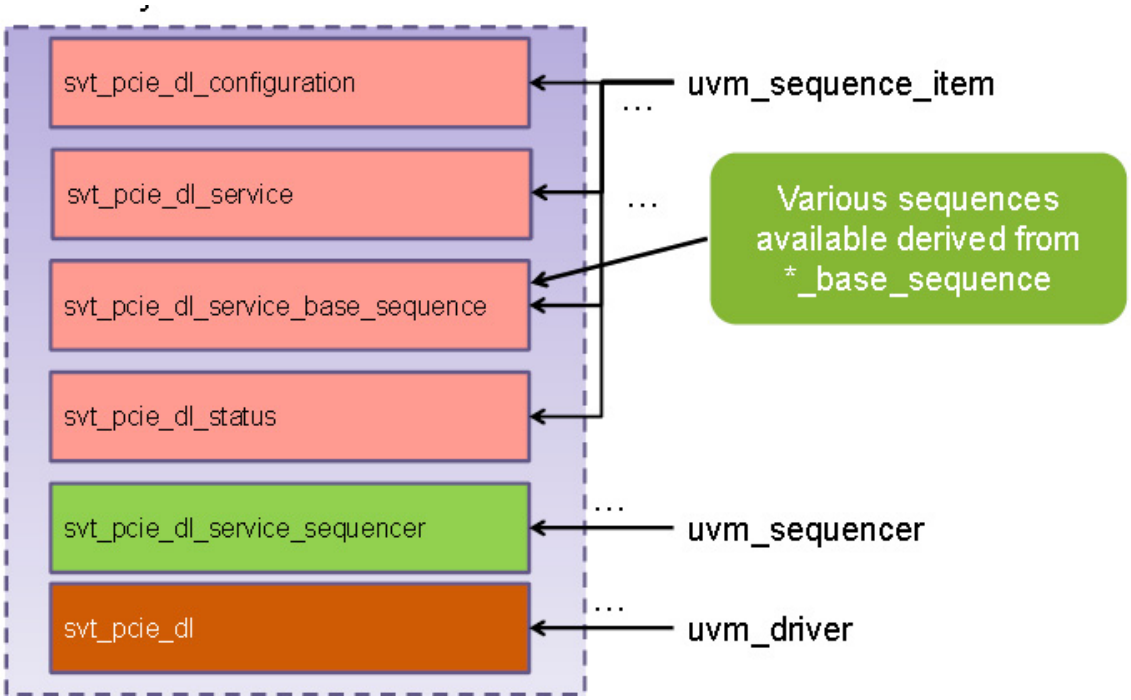
11.2 Additional Documentation on DL Programming Tasks

Consult the following to get information on DL related programming tasks.

- SolvNetPlus PCIe VIP Articles
- PCIe SVT FAQ

11.3 Class Elements of the Link Layer

The following illustration shows the classes making up the Link Layer. They will be discussed in various sections of the chapter.



11.4 Power Management

The Data Link Layer in the SVT PCIE VIP provides support for PM/ASPM functionality similar to the specification. As defined by the specification, the VIP can be directed into and out of particular power states. The VIP can also be configured to automatically enter states as specified by the protocol. Exit from low power states can be initiated by the VIP as well if it needs to transmit a TLP or DLLP.

NOTE: ASMP L1 entry must be enabled by the Data Link configuration, but PM L1 does not (due to specification controls).

The VIP has built in checkers to make sure the handshake process occurs per specification. Timeouts are used to make sure handshakes occur within a reasonable time frame.

11.4.1 ASPM

11.4.2 L0s Entry

For L0s entry, the VIP can be configured to automatically transition to Tx L0s when an idle period is reached. Or it can be directed to Tx L0s. The idle timer, when set to a non-zero value, enables the VIP's automatic entry to L0s. The DL can also be sent immediately to Tx L0s via a service request.

For exit from L0s to L0, the VIP can be directed or autonomously transition. User directed exit from Tx L0s is initiated via DL INITIATE_ASPM_EXIT service request. Autonomous exit occurs when any DLLP or TLP needs to be transmitted.

:

Table 11-1 DL L0s Configuration

Member	Description
l0s_idle_timer_limit_ns	When set to non-zero value, VIP will automatically enter ASPM L0s when transmitter is idle for * this time. If set to 0, automatic ASPM L0s entry is disabled. For directed entry into L0s, use * InitiateASPML0sEntry task.

Table 11-2 DL Service Requests

Member	Description
INITIATE_ASPM_L0S_ENTRY	Initiates VIP to enter ASPM Tx L0s low power state.
NITIATE_ASPM_EXIT	Initiates VIP to transition back to L0 from ASPM low power state.

11.4.2.1 L1 Entry

ASPM L1 entry must be enabled by DL configuration variables listed below. The entry/exit to/from ASPM L1 is initiated by DL service requests INITIATE_ASPM_L1_ENTRY/ INITIATE_ASPM_EXIT.

For exit from L1 to L0, the VIP can be directed or autonomously transition. User directed exit from L1 is initiated via a DL INITIATE_ASPM_EXIT service request. Autonomous exit occurs when any DLLP or TLP needs to be transmitted.

:

Table 11-3 DL Service Requests

enable_aspm_l1_entry	Enable ASPM L1 entry
INITIATE_ASPM_L1_ENTRY	Initiates VIP to enter PM L1 low power state
INITIATE_ASPM_EXIT	Initiates VIP to transition back to L0 from ASPM low power state.

11.4.2.2 L1 Substate Entry

The VIP supports entry into L1 substates. Entry must be enabled via DL configuration variables listed below. These vars must be set in addition to the vars listed in the L1 section. If L1_1 and L1_2 are both enabled, the VIP will transition to the highest power savings state, which is L1_2. Entry/exit to/from L1 substates is initiated just like L1 entry/exit.

Table 11-4 DL Configuration Members for L1 Substrate Entry

Member	Description
enable_aspm_l1_2_entry	The variable enables ASPM L1.2 entry. Must be used in conjunction with enable_aspm_l1_entry and INITIATE_ASPM_L1_ENTRY service request
enable_aspm_l1_1_entry	The variable enables ASPM L1.1 entry. Must be used in conjunction with enable_aspm_l1_entry and INITIATE_ASPM_L1_ENTRY service request.

11.4.2.3 Active State NAK

Active state NAK TLP msgs must be initiated by the test case. Active State NAK TLPs received from the DUT can be forwarded to the test case via

11.4.3 PM

The VIP assumes the test case has performed the proper PM handshake in order for the VIP to transition to low power states. The VIP does not respond to PM TLP messages nor will it initiate any PM TLP messages. In order for the test case to complete the handshake with the VIP, the received PM TLP messages must be forwarded to the test case by the VIP. This is accomplished by routing PM TLPs to the testcase via the TLs mapping tables using ADD_RID_MSG_CODE_APPL_ID_MAP_ENTRY service request.

Once the test case has determined the functions in the DUT are ready for transition to low power states, the test case can initiate VIP transition per sections below. If the DUT is initiating the transition, the VIP will respond as if its functions are ready for PM low power transition.

11.4.3.1 L1

PM L1 does NOT have to be enabled by DL configuration. This behavior is enabled by default, similar to the specification. The entry/exit to/from PM L1 is initiated by DL service requests INITIATE_PM_L1_ENTRY/ INITIATE_PM_EXIT.

For exit from L1 to L0, the VIP can be directed or autonomously transition. User directed exit from L1 is initiated via the DL INITIATE_PM_EXIT service request. Autonomous exit occurs when any DLLP or TLP needs to be transmitted.

No configuration members exist for DL D1 configuration.

Table 11-5 L1 DL Service Requests

Member	Description
INITIATE_PM_L1_ENTRY	Initiates VIP to enter PM L1 low power state
INITIATE_PM_EXIT	Initiates VIP to transition back to L0 from PM low power state.

11.4.3.2 L1 Substate Entry

The VIP supports entry into L1 substates. Entry to L1 substates must be enabled via DL configuration variables listed below. If L1_1 and L1_2 are both enabled, then the VIP will transition to the highest power savings state, which is L1_2. Entry/exit to/from L1 substates is initiated just like L1 entry/exit.

Table 11-6 L1 Substrate Entry Members

Member	Description
enable_pm_l1_2_entry	The variable enables PM L1.2 entry. Must be used in conjunction with enable_pm_l1_entry and INITIATE_PM_L1_ENTRY service request
enable_pm_l1_1_entry	The variable enables PM L1.1 entry. Must be used in conjunction with enable_pm_l1_entry and INITIATE_PM_L1_ENTRY service request.

11.4.3.3 L2/3 Entry

PM L2/3 entry does NOT need to be enabled by DL configuration. This behavior is enabled by default, similar to the specification The entry/exit to/from PM L2/3 is initiated by DL service requests INITIATE_PM_L23_ENTRY/INITIATE_PM_EXIT.

For exit from L2/3 to L0, the VIP can be directed or autonomously transition. User directed exit from L2/3 is initiated via DL INITIATE_PM_EXIT service request. Autonomous exit occurs when any DLLP or TLP needs to be transmitted

There is no configuration member for D1 L1.

Table 11-7 DL Service Requests

Member	Description
INITIATE_PM_L23_ENTRY	Initiates VIP to enter PM L1 low power state
INITIATE_PM_EXIT	Initiates VIP to transition back to L0 from PM low power state.

11.4.4 VIP PM/ASPM Checks

The VIP has automatic checking for PM/ASPM functionality. The checks can be demoted if the behavior is expected. The timeouts are configurable via DL configuration. By default, any PM request is expected to

complete successfully. In the event that an ASPM active state NAK TLP is received, the VIP will flag this as a OVM_WARNING.

Table 11-8 Pm/ASPM Checks

Member	Description
MSGCODE_PCIESVC_DL_ASPM_L1_HANDSHAKE_TIMEOUT	• f DUT fails to respond to ASPM request handshake for this: • # symbols, NOTICE will be issued • and ASPM entry will be aborted. aspm_timeout_cnt_limit = `SVT_PCIE_ASPM_TIMEOUT_CNT_LIMIT_DEFAULT;
MSGCODE_PCIESVC_DL_ASPM_L1_1_HANDSHAKE_TIMEOUT	
MSGCODE_PCIESVC_DL_ASPM_L1_2_HANDSHAKE_TIMEOUT	
MSGCODE_PCIESVC_DL_PM_L1_HANDSHAKE_TIMEOUT	• If DUT fails to respond to PM request handshake for this: • # symbols, NOTICE will be issued and • * PM entry will be aborted. */ rand int unsigned pm_timeout_cnt_limit = `SVT_PCIE_PM_TIMEOUT_CNT_LIMIT_DEFAULT;
MSGCODE_PCIESVC_DL_PM_L1_1_HANDSHAKE_TIMEOUT	
MSGCODE_PCIESVC_DL_PM_L1_2_HANDSHAKE_TIMEOUT	
MSGCODE_PCIESVC_DL_PM_L23_HANDSHAKE_TIMEOUT	
MSGCODE_PCIESVC_DL_ASPM_L1_RX_ACTIVE_STATE_NAK	ASPM L1 Handshake terminated by receiving ACTIVE_STATE_NAK
MSGCODE_PCIESVC_DL_RECEIVED_UNEXPECTED_PM_ACK	Received PM_REQUEST_ACK DLLP when PM was not requested
MSGCODE_PCIESVC_DL_RECEIVED_TLP_ASPM_L1_STARTED	Received TLP when ASPM L1 entry is in progress. Dut should hold TLP until L1 hanshake is complete. Spec 5.4.1.2.1
MSGCODE_PCIESVC_DL_RECEIVED_TLP_PM_L1_STARTED	Received TLP when PM L1 entry is in progress. Dut should hold TLP until L1 hanshake is complete. Spec 5.3.2.1
MSGCODE_PCIESVC_DL_RECEIVED_TLP_PM_L23_STARTED	Received TLP when PM L2/L3 entry is in progress. Dut should hold TLP until L2/L3 hanshake is complete. Spec 5.3.2.3

11.5 Component Class svt_pcie_dl

The OVM component class svt_pcie_dl is responsible for the following features:

- Implements Link layer module
- Implements static and dynamic configuration. Dynamic configuration is the ability of the model to configure and re-configure at run time. Static configuration is done before at time zero (simulation time). The following table shows the members supporting dynamic and static configuration and general status monitoring.
- Responsible for reconfigure PCIE
- Provides status of the application.
- Provides a SIPP [Sequence Item Pull Port] to cater to services of type svt_pcie_dl_service.
- Provides classes and members for error injection

Following table lists significant functions and data members.

Table 11-9 Members and Features for svt_pcie_dl

Member	Feature
Functions	
post_tlp_framed_in_get(...)	Called by the component after recognizing a TLP Transaction received on the link.
pre_dllp_out_putt(...)	Callback issued by the component just prior to putting a Vendor Specific DLLP transaction received on the link on the put port.
pre_dllp_transmission_svc_callback t(...)	Method used to apply TX DLLP exceptions.
pre_phy_pkt_w_framing_transmission_svc_callback t(...)	Method used to apply PHY packet framing exceptions.
pre_tlp_framed_out_put t(...)	Callback issued by the component after scheduling a TLP transaction for transmission on the link, just prior to framing.
pre_tlp_transmission_svc_callback t(...)	Method used to apply TX TLP exceptions.
rx_dllp_post_deframed_callback t(...)	Called from the SVC Link Layer to report an inbound DLLP for callback.
rx_dllp_startedt(...)	Callback issued by the component immediately after receiving a User DLLP transaction on the set_item_port prior to its further processing.
rx_tlp_post_deframed_callback t(...)	Called from the SVC Link Layer to report an inbound TLP for callback.
tx_dllp_pre_framed_callback t(...)	Called from the SVC Link Layer to report an outbound DLLP for callback.
tx_dllp_startedt(...)	Called by the component after building a DLLP Transaction just prior to its further processing.

Table 11-9 Members and Features for svt_pcie_dl (Continued)

Member	Feature
tx_tlp_pre_framed_callbackt(...)	Called from the SVC Link Layer to report an outbound TLP for callback.
get_cfg t(...)	Overrides the base method to generate an error.
reconfiguret(...)	Overrides the base method to generate an error.
set_err_checkt(...)	Used to set the err_check object and to fill in all of the local checks.
Following are derived from various base classes for port tracking and error injection	
svt_pcie_dl_tlp_exception dl_tlp_xact_rx_exception	Randomization factory to create RX TLP exception (error etc.) to be inserted in transaction
svt_pcie_dl_tlp_exception_list dl_tlp_xact_rx_exception_list	Randomization factory to create RX TLP exception list for a TLP transaction
svt_pcie_dl_tlp_exception dl_tlp_xact_tx_exception = null;	Randomization factory to create TX TLP exception list for a TLP transaction
svt_pcie_dl_tlp_exception_list dl_tlp_xact_tx_exception_list	Randomization factory to create TX TLP exception list for a TLP transaction
svt_pcie_dllp_exception dllp_xact_exception	Randomization factory to create TX DLLP exception list for a TLP transaction
svt_pcie_dllp_exception_list dllp_xact_exception_list	Randomization factory to create TX DLLP exception list for a TLP transaction
svt_pcie_phy_transaction_exception phy_xact_exception	Randomization factory to create TX PHY transaction framing exception (error etc.) to be inserted in packet
phy_xact_exception_list	Randomization factory to create TX PHY transaction framing exception list for a packet
svt_debug_opts_analysis_port received_dllp_observed_port	Analysis port for received DLLPs. This port is generally used for scoreboarding. The DLLPs observed via this analysis port are controlled by svt_pcie_dl_configuration :: received_dllp_interface_mode
svt_debug_opts_analysis_port received_tlp_observed_port	Analysis port for received TLPs. This port is generally used for scoreboarding. The TLPs observed via this analysis port are controlled by svt_pcie_dl_configuration :: received_tlp_interface_mode.
svt_debug_opts_blocking_put_port rx_dllp_out_port	RX DLLP Put Port Provides a mechanism for external components to receive vendor specific DLLPs from the DAta Link Layer. The handle to this DLLP put port can be set or obtained through the driver's public member rx_dllp_out_port

Table 11-9 Members and Features for svt_pcie_dl (Continued)

Member	Feature
svt_debug_opts_blocking_peek_imp_port rx_dllp_peek_port RX DLLP Peek port.	Provides a mechanism for external components to retrieve Vendor Specific DLLPs from the Data Link Layer. The handle to this DLLP peek port can be set or obtained through the driver's public member rx_dllp_peek_port .
svt_debug_opts_analysis_port sent_dllp_observed_port	Analysis port for sent DLLPs. This port is generally used for scoreboarding. The DLLPs observed via this analysis port are controlled by svt_pcie_dl_configuration :: sent_dllp_interface_mode.
svt_debug_opts_analysis_port sent_tlp_observed_port	Analysis port for sent TLPs. This port is generally used for scoreboarding. The TLPs observed via this analysis port are controlled by svt_pcie_dl_configuration :: sent_tlp_interface_mode.

11.6 Configuration class svt_pcie_dl_configuration

Use the svt_pcie_dl_configuration to define the overall behavior of the Data Link Layer. You can define the following behaviors for over 88 different features. In addition, note that most configurable attributes are defined as SystemVerilog RAND types. This allows your testbench to randomize them following predefined constraints provided by Synopsys. Consult the PCIe HTML Class Reference on declared data types.

11.6.1 Members and Features

For details about the attributes of this class, see HTML class description of the `svt_pcie_dl_configuration` class available at the following location:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/configuration/class_svt_pcie_dl_configuration.html

11.6.2 Calculating Ack/Nak Latency Values

Ack/Nak latency values are calculated as shown in the following example:

- attached_acknak_latency_timer_limit

```
attached_acknak_latency_timer_limit
                                    = attached_max_nak_latency +
                                      attached_internal_delay +
                                      internal_delay +
                                      internal_phy_delay +
                                      attached_internal_phy_delay
```

- acknak_latency_timer_limit

  `acknak_latency_timer_limit` is a random value between `min_acknak_latency` and `max_acknak_latency`.

- acknak_latency

```
acknak_latency
            = (((MAX_PAYLOAD_SIZE_VAR + TLP_OVERHEAD) * ack_factor) / smallest_width) +
              internal_delay
```

- attached_acknak_latency

```
attached_acknak_latency
        = (((MAX_PAYLOAD_SIZE_VAR + TLP_OVERHEAD) * ack_factor) / smallest_width) +
          attached_internal_delay
```

11.7 Status class svt_pcie_dl_status

This class makes available the specific status information as it relates to the Data Link Layer. For example you can get the number of:

- NACK DLLPs received and sent
- TLPs and DDLPs received and sent with errors
- Tx alignment errors injected with two STPs per symbol.
- Tx DLLP code violation errors injected.
- Tx DLLPs with non-zero reserved bits injected.
- Number of packets that had to be retransmitted.
- Number of credits on every virtual channel.

For details about the attributes of this class, see HTML class description of the `svt_pcie_dl_status` class available at the following location:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/status/class_svt_pcie_dl_status.html

11.8 Service Class svt_pcie_dl_service

The Data Link Layer service class is responsible for the implementing the following major tasks and features.

- Initiating transitions for the VIP to enter low power states:
  - ASPM Tx L0s
  - ASPM L1 low power state.
  - PM L1
  - PM L2/L3
  - Back to L0 from ASPM
  - Back to L0 from PM
- Setting ACKFactor value
- Displaying DLL statistics
- Allowing DLCMSM to transition out of Disabled state.
- Enabling and disabling gating
- Gating FC type of INITFC frames
- Setting the Virtual channel of the INITFC frames to be gated
- Getting Status information about the current processing state

For details about the service sequences of this class, see tab `'DL_SERVICE_SEQUENCES'` under HTML class description available at the following location:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/sequencepages.html

12 PHY Layer Features and Classes

12.1 Classes and Applications for Using the VIP's PHY Layer

The following classes have members and tasks to implement Data Link Layer features and operation.

- Service Class svt_pcie_pl_service
- OVM Component Class svt_pcie_pl
- PHY Layer Configuration Class

The following section lists and describes the members and tasks of the major classes to implement various features for your testbench.

Attention Note, the descriptions for the classes and applications show the most important and often used members/features. Consult the PCIe OVM HTML Class Reference for a complete listing of the members and their data types.

12.2 Additional Documentation on PHY Programming Tasks

Consult the following to get information on PHY related programming tasks.

- SolvNetPlus PCIe VIP Articles.
- PCIe SVT FAQ

Add a new section 10.3 in chapter 10 "Phy layer feature and classes"

12.3 External Tx Bit Clk Use Model

This is a special use model which can be used in common ref clk mode. In this mode, the VIP is capable of transmitting serial data with respect.to serial bit clocks provided from the Test Bench. To enable this model VIP has to be configured in following manner from the Test Bench.

```
defparam <vip_top_level_inst_path>.port0.USE_EXTERNAL_BIT_CLK = 1;
assign <vip_top_level_inst_path>.ext_bit_clk_gen1 =
endpoint0.port0.SVC_CLKGEN4_INST.clkgen_txclk0.bit_clk;clk;
assign <vip_top_level_inst_path>.ext_bit_clk_gen2 =
endpoint0.port0.SVC_CLKGEN4_INST.clkgen_txclk0.bit_clk;clk;
assign <vip_top_level_inst_path>.ext_bit_clk_gen3 =
endpoint0.port0.SVC_CLKGEN4_INST.clkgen_txclk0.bit_clk;clk;
assign <vip_top_level_inst_path>.ext_bit_clk_gen4 =
endpoint0.port0.SVC_CLKGEN4_INST.clkgen_txclk0.bit_clk;clk;
```

Where root0 is the top level instantiation absolute path in the Test Bench.

If a device does not support Gen2/Gen3/Gen4, then leave following wires open:

- ext_bit_clk_gen2
- ext_bit_clk_gen3
- ext_bit_clk_gen4

12.4 Service Class svt_pcie_pl_service

This transaction class supports all of the Service requests which can be processed by the PHY Layer.

Table 12-1 Class svt_pcie_pl_service

Member	Description
assert_clkreq_n	Controls whether the VIP will assert bidirectional signal clkreq_n or not. Note there is a soft pullup if clkreq is not asserted When '1' clkreq_n will be asserted (ie driven to 'b0). When '0' clkreq_n will not be asserted by the VIP.
corrupt_disparity_byte_wt[16]	Automatic OS corruption weight, Tx OS Bytes 0-15.
corrupt_lane_mask	Automatic OS corruption and FORCE_LANE_TX_ELEC_IDLE lane mask.
corrupt_tx_idle_data_enable	Automatic EIOS corruption percentage.
corrupt_tx_os_percentage	Automatic OS corruption percentage.
cursor_coeff	Cursor value for an equalization request.
direction_change_response	Direction change response.
ei_bytenum	User Task Tx TS EI Byte Number.
ei_code	User Task Tx TS EI Code.
ei_tx_phy_data_bit_flip_weight	Tx Data Pattern Bit Flip weighting - weighting to replace outgoing data with an invalid codeword when an error is injected. 2.5G/5G only.
ei_tx_phy_data_corrupt_data_weight	Tx Data Pattern Corrupt Data weighting - weighting to replace outgoing data with random data when an error is injected.
ei_tx_phy_data_corrupt_disparity_weight	Tx Data Pattern Corrupt Disparity weighting - weighting to replace outgoing data with an invalid codeword when an error is injected. 2.5G/5G only.
ei_tx_phy_data_invalid_codeword_weight	Tx Data Pattern Invalid Codeword weighting - weighting to replace outgoing data with an invalid codeword when an error is injected. 2.5G/5G only.
ei_tx_phy_data_lane_mask	Tx Data Pattern Lane mask- Each bit corresponds to a lane. Lanes with a 'b0 will not have errors injected.
ei_tx_phy_data_pattern_enable	Tx Data Pattern Enable - enable the phy data error pattern.
eq_lane_num	Programs the lane number for equalization information.
expect_reject	Reject response from link partner.
figure_of_merit	Figure Of Merit response.
hot_plug_mode	This attribute controls Hot plug mode. Table 12-2 shows the various hot plug modes.

Table 12-1 Class svt_pcie_pl_service (Continued)

Member	Description
hot_reset_mode	This attribute controls Hot reset mode. Refer to Table 12-3 for a listing of hot reset modes.
internal_condition	Type of internal condition. Refer to Table 12-4 for a listing of internal conditions.
invalid_codeword_byte_wt[16]	Automatic OS corruption weight, Tx OS Bytes 0-15.
invalid_data_byte_wt[16]	Automatic OS corruption weight, Tx OS Bytes = 0-15.
lane_enabled	Set to enable the association.
lane_num	Lane Number.
loopback_enable	Initiate Loopback Enable.
max_ei_tx_phy_data_pattern_burst	Max Tx Data Pattern Burst - maximum number of symbols in an error burst.
max_ei_tx_phy_data_pattern_spacing	Max Tx Data Pattern Spacing - max number of symbols between error bursts.
max_user_tx_ts_burst	User Task Max Tx TS Burst.
max_user_tx_ts_spacing	User Task Max Tx TS Spacing.
min_ei_tx_phy_data_pattern_burst	Min Tx Data Pattern Burst - minimum number of symbols in an error burst.
min_ei_tx_phy_data_pattern_spacing	Min Tx Data Pattern Spacing - min number of symbols between error bursts.
min_user_tx_ts_burst	User Task Min Tx TS Burst.
min_user_tx_ts_spacing	User Task Min Tx TS Spacing.
phy_enable	When clear the LTSSM will attempted to enter the DISABLED state. This is not the same thing as turning off the LTSSM!!! To completely disable the LTSSM the hot_plug_mode should be set to HOT_PLUG_UNPLUG.
phy_id	Phy number to configure.
phy_response_code	Reject response from link partner.
postcursor_coeff	Post Cursor value for an equalization request.
precursor_coeff	Pre Cursor value for an equalization request.
preset_valid	Preset valid for an equalization request.
preset_value	Preset value for an equalization request.

Table 12-1 Class svt_pcie_pl_service (Continued)

Member	Description
reject_coefficient_preset_requests	For the side that is receiving requests during equalization, setting this will force rejection of incoming requests. clearing this bit will allow requests to once again be accepted.
service_type	Transaction command. For a complete listing of all the service requests supported by the VIP, see tab `'PL_SERVICE_SEQUENCES'` under HTML class description available at the following location: *$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_uvm_class_reference/html/sequencepages.html*
status	Status information about the current processing state.
symbol<0-15>	User Task Tx TS Symbol <0-15>
tx_deemph	Tx Deemphasis Value
user_tx_ts_enable	User Task Tx TS Enable.

The following table shows the various types of hot plug modes. The member "hot_plug_mode" sets the mode type.

Table 12-2 Hot Plug Modes

Mode	Description
HOT_PLUG_UNPLUG	Hot plug mode is unplug. Applicable to VIP in active and passive modes.
HOT_PLUG_MONITOR	Hot plug mode is monitor. Only applicable to VIP in active mode.
HOT_PLUG_WAIT	Hot plug mode is wait. Only applicable to VIP in active mode.
HOT_PLUG_DETECT	Hot plug mode is detect. Only applicable to VIP in active mode.

The following table list the various hot reset modes. The member "hot_reset_mode" sets the reset mode.

Table 12-3 Hot Reset Modes

Mode	Description
HOT_RESET_INACTIVE	LTSSM is not instructed to enter reset. If LTSSM initiated the hot reset and is in the hot reset state, setting to incative will direct the LTSSM out of hot reset.
HOT_RESET_FORCE	Instruct the LTSSM to enter hot reset and remain in this state.
HOT_RESET_WAIT	Instruct the LTSSM to enter hot reset, then wait for the attached link to enter hot reset. Once the attached link has shut down its transmitters LTSSM will automatically exit to detect and reset the hot reset mode to INACTIVE.

The following table lists various internal conditions. The member "internal_condition" sets these conditions.

Table 12-4 Internal Condition States

Internal Condition	Description
INT_COND_RX_PATH_BLOCK_ALIGNMENT_ACHIEVED	Indicates Block alignment status on RX path.
INT_COND_TX_PATH_BLOCK_ALIGNMENT_ACHIEVED	Indicates Block alignment status on TX path.
INT_COND_RX_PATH_8G_SPEED_FIRST_PKT_RECEIVED	Received first packet on RX path at 8G speed.
INT_COND_TX_PATH_8G_SPEED_FIRST_PKT_RECEIVED	Received first packet on TX path at 8G speed.

For details about the service sequences of this class, see tab `'PL_SERVICE_SEQUENCES'` under HTML class description available at the following location:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/sequencepages.html

12.5 OVM Component Class svt_pcie_pl

This class is OVM Driver that implements Physical layer module. The class is responsible to reconfigure PCIE SVC Physical layer module. It is also responsible to provide status of the application. It provides a SIPP [Sequence Item Pull Port] to cater to services of type svt_pcie_pl_service. Note, the class supports all OVM phases.

Table 12-5 Class svt_pcie_pl

Member	Description
pre_symbol_out_put	Called by the component after gathering all the symbols to be transmitted on the PCIe link. This is the last chance to the user to corrupt any symbol before it goes on the link.
reconfigure_via_task	Depreciated.
tx_os_started	Called by the component after building an OS Transaction. The callback gives user a chance to attach exception list to the OS transaction prior to it transmission on the link.
tx_ts_os_started	Called by the component after building a TS OS Transaction. The callback gives user a chance to attach exception list to the TS OS transaction prior to it transmission on the link.
os_xact_exception	Randomization factory to create TX OS exception (error etc.) to be inserted in transaction.
os_xact_exception_list	Randomization factory to create TX OS exception list for an OS transaction.
svc_in_port	PL Service TLM Sequence Item Pull Port. Provides a mechanism for submitting PL Service transactions recognized by the PL Layer. The handle to this TLM sequence item pull port can be set or obtained through the driver's public member svc_in_port.

Table 12-5 Class svt_pcie_pl (Continued)

Member	Description
symbol_exception	Randomization factory to create TX symbols exception (error etc.) to be inserted in symbols.
symbol_exception_list	Randomization factory to create TX symbols exception list for symbols to be transmitted.
ts_os_xact_exception	Randomization factory to create TX TS OS exception (error etc.) to be inserted in transaction.
ts_os_xact_exception_list	Randomization factory to create TX TS OS exception list for a TS OS transaction.

12.6 PHY Layer Configuration Class

For details about the attributes of this class, see HTML class description of the `svt_pcie_pl_configuration` class available at the following location:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/configuration/class_svt_pcie_pl_configuration.html

13 Using the Driver Application

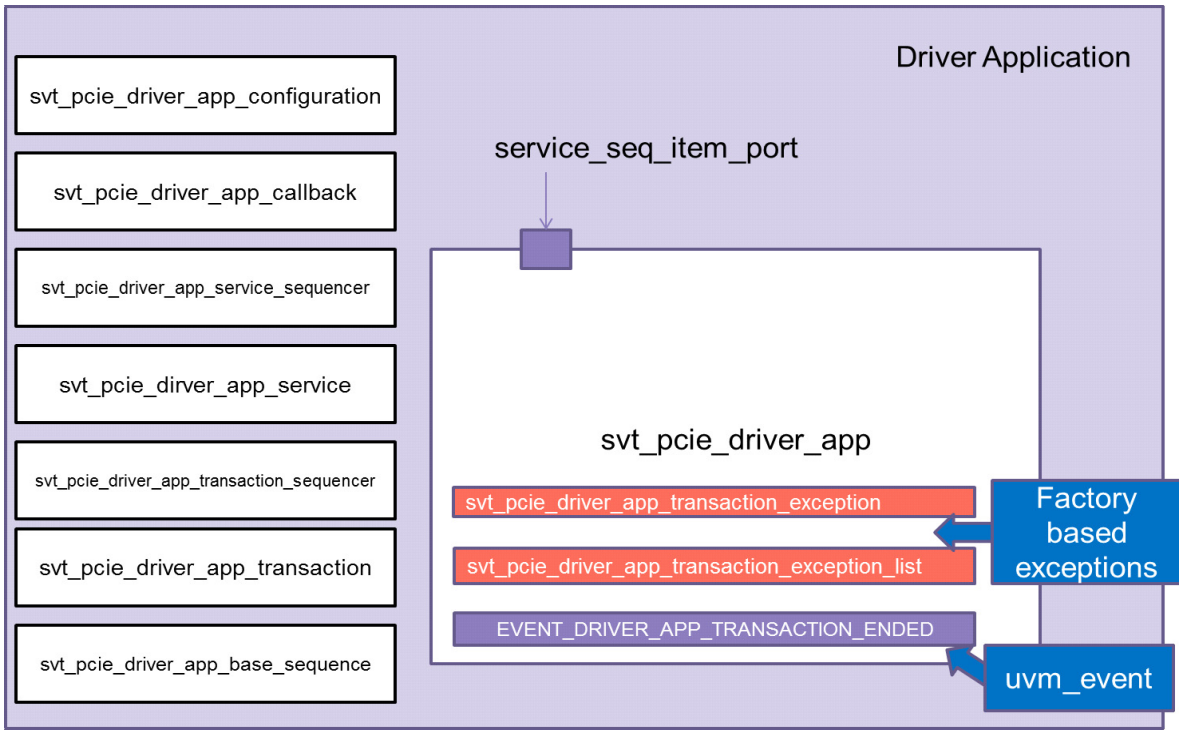
13.1 Introduction

The Driver application is implemented as a ovm_driver component. The Driver has the following functions:

- Provides configuration and service sequences, and transaction sequences for the creation of PCIe transactions
- Tracks completions for a given transaction
- Interfaces with the Global Shadow and built-in scoreboard to validate data in the PCIe bus

The driver consists of several components, as shown in Figure 13-1.

Figure 13-1 PCIe Driver application components



13.2 Driver Application Configuration

The Driver application configuration class is svt_pcie_driver_app_configuration. The members within the class control the settings of the Driver.

The Driver has a few parameters that are only configurable in Verilog instantiation models. See "Verilog Configuration Parameters and Tasks" for information about those parameters.

The svt_pcie_driver_app_configuration class is accessed via the following instance hierarchy:

instance-name-of-svt_pcie_device_configuration.pcie_cfg.driver_cfg[*int*]

For more details, such as members, class inheritance UML, list of tasks, list of constraints and so on, see the HTML class description of `svt_pcie_driver_app_configuration` at the following location:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/configuration/class_svt_pcie_driver_app_configuration.html

13.3 Verilog Configuration Parameters and Tasks

The items in "Compile-time Verilog Configuration Parameters" and "Driver Application Sequencer and Sequences" are controlled only via Verilog.

13.3.1 Compile-time Verilog Configuration Parameters

Parameters that are only changeable when instantiating the Driver application as part of the instantiation model are listed in Table 13-1.

Table 13-1 Driver application runtime Verilog parameters

Parameter Name	Type	Range	Default Value	Description
CMB_TABLE_SIZE				
	integer	16-256	256	Size of the command management block, which is used to track pending and outstanding transactions.
DISPLAY_NAME				
	string		"pciesvc_driver"	Default display name for the driver. This is not typically changed by the user.
MAX_NUM_TAGS				
	integer	1-256	32	Maximum number of tags that can be used. If greater than 32 it is assumed that the extended tag bits are legal to use.

13.3.2 Runtime Configuration Parameters

Driver application parameters that are changeable at runtime are listed in Table 13-2.

Table 13-2 Driver application runtime Verilog parameters

Parameter Name	Type	Range	Default Value	Description
PCIE_SPEC_VER				
	real	1.1, 2.0, 2.1, 3.0	PCIE_SPEC_VER_2_1	See Include/pciesvc_parms.v: PCIE_SPEC_VER_* Note: Set this parameter in the model. It is not changed at this level but rather at the agent level

13.3.3 Runtime Verilog Tasks

Verilog tasks that are changeable at runtime are listed in Table 13-3.

Table 13-3 Driver application runtime Verilog tasks

Parameter Name	Arguments	I/O	Description
AddTLPPrefix Adds the prefix or prefixes contained in prefix_array to the next queued command.	logic [31:0] prefix_array[]	I	A dynamic array containing the prefixes to be added. It is up to the user to build and set the contents of the array.

13.4 Driver Application Sequencer and Sequences

The Driver layer supports both service and transaction sequences. Service sequences run on the svt_pcie_driver_app_service_sequencer while transaction sequences run on the svt_pcie_driver_app_transaction_sequencer.

Test writers can access the service sequencer via the following path:

instance-name-of-svt_pcie_device_agent.driver_seqr[*int*]

Test writers can access the transaction sequencer via the following path:

instance-name-of-svt_pcie_device_agent.driver_transaction_seqr[*int]*

By default, there is one instance of the transaction sequencer driver_seqr[0] and one instance of the driver_transaction_seqr[0]. Implementing them as arrays allows for future expansion.

13.4.1 Service Sequences

Services are commands to give the model that are related to behavior or configuration; they are not a transaction item which is to be sent across the bus. For the Driver there is a base service, svt_pcie_driver_app_service, and there is a base service sequence, svt_pcie_driver_app_service_base_sequence.

The service, svt_pcie_driver_app_service, is a sequence_item which supports all the service transaction types supported by the Driver. A service is selected by constraining the service_type_enum of the class to one of the enumerated service types.

Alternatively, the model provides several sequences that contain the base svt_pcie_driver_app_service that can be used for test development.

For example:

```
svt_pcie_driver_app_service_wait_until_idle_sequence  wait_seq;
…
  `ovm_do_on (wait_seq, p_sequencer.root_virt_seqr. driver_ seqr[0]);
…
```

The service sequencer is described in Table 13-4.

The service sequences are listed in Table 13-5.

Table 13-4 Driver application service sequencer

svt_pcie_driver_app_service_sequencer This class is sequencer that provides stimulus for the svt_pcie_driver_app_service_driver class. The svt_pcie_driver_app_service_agent class is responsible for connecting this ovm_sequencer to the driver if the agent is configured as OVM_ACTIVE.

Table 13-5 Driver application service sequences

svt_pcie_driver_app_service_base_sequence This sequence is the base class for the svt_pcie_driver_app_service sequence. All the other sequences are extended from this sequence.
svt_pcie_driver_app_service_is_trans_complete_sequence This sequence implements Is Transaction Complete. IS_TRANSACTION_COMPLETE creates a request to check if a transaction is complete. The sequence is useful to query the VIP about whether the Driver transaction that is queued to the Driver application component is complete. If the transaction with the specified command_num is complete, returned status is 1'b1, otherwise the returned status is 1'b0.
svt_pcie_driver_app_service_null_sequence This sequence implements Null Traffic. This class creates a null sequence that can be associated with a sequencer but generates no traffic.
svt_pcie_driver_app_service_wait_for_compl_sequence This sequence implements Wait For Completion. WAIT_FOR_COMPLETION creates a request to wait for completion for the specified transaction. The command_num variable is the ID for the transaction which is available once the transaction is queued to the Driver. The sequence blocks until all the completion data for the specified request is returned.
svt_pcie_driver_app_service_wait_until_idle_sequence This sequence implements Wait Until Driver Idle. WAIT_UNTIL_DRIVER_IDLE creates a request to wait until the Driver application is idle. When the Driver has some queued transactions to be transferred over the link, or if the Driver has not yet received any completion of transferred transactions (that is, if transaction are outstanding), then the Driver is said to be in a non-idle condition. When all of the outstanding transactions are complete and there are no transactions queued to the Driver, the Driver is said to be in an idle condition. This sequence blocks until the Driver is idle.

13.4.2 Transaction Sequences

Transactions correspond to a transaction item that is to be sent across the bus. For the Driver there is a base transaction, svt_pcie_driver_app_transaction, and there is a base transaction sequence, svt_pcie_driver_app_transaction_base_sequence.

The transaction, svt_pcie_driver_app_transaction, is a sequence_item that supports all the transaction types supported by the Driver. Transactions are defined by constraining the fields listed in Table 13-6.

Table 13-6 Transaction fields

											Config only		Message only			Responses only		
Transaction Type	**address**	**length**	**first_dw_be**	**last_dw_be**	**traffic_class**	**address_translation**	**ep**	**payload**	**exception_list**	**block**	**cfg_type**	**rgister_ number**	**routing_type**	**message_code**	**vendor_fields**	**payload**	**command_num**	**completion _status**
MEM_RD	**x**	**x**	**x**	**x**	**x**	**x**			**x**	**x**						**x**	**x**	**x**
MEM_RD_LK	x	x	x	x	x	x			x	x						x	x	x
MEM_WR	x	x	x	x	x	x	x	x	x	x							x	x
IO_RD	x		x						x	x						x	x	x
IO_WR	x							x	x									
CFG_RD	x*		x						x	x	x	x				x*	x	x
CFG_WR	x*						x	x*	x	x	x	x					x	x
MSG					x		x		x	x				x	x		x	x
ATOMIC_OP_FETCH_ADD	x	x	x	x	x	x	x	x	x	x							x	x
ATOMIC_OP_SWAP	x	x	x	x	x	x	x	x	x	x							x	x
ATOMIC_OP_CAS	x	x	x	x	x	x	x	x	x	x							x	x

*CFG types: address is {B,D,F}, payload is payload[0] only

Alternatively, the model provides several sequences that contain the base svt_pcie_driver_app_transaction that can be used for test development.

The transaction sequencer is described in Table 13-7.

The transaction sequences are listed in Table 13-8.

Note Transaction sequences for a given type of transaction use the fields shown in Table 13-6.

Table 13-7 Driver application transaction sequencer

svt_pcie_driver_app_transaction_sequencer This class is sequencer that provides stimulus for the svt_pcie_driver_app_transaction_driver class. The svt_pcie_driver_app_transaction_agent class is responsible for connecting this ovm_sequencer to the Driver if the agent is configured as OVM_ACTIVE.

Table 13-8 Driver application transaction sequences

svt_pcie_driver_app_transaction_atomicop_fetchadd_sequence This sequence generates an atomic operation fetchadd sequence.
svt_pcie_driver_app_transaction_atomicop_cas_sequence This sequence generates an atomic operation CAS (compare and swap) request.
svt_pcie_driver_app_transaction_atomicop_swap_sequence This sequence generates an atomic operation swap request.
svt_pcie_driver_app_transaction_base_sequence This sequence is the base class for svt_pcie_driver_app_transaction sequences.All the other sequences are extended from this sequence. This sequence takes care of managing the objections by making the manage_objection bit 1
svt_pcie_driver_app_transaction_cfg_rd_sequence This sequence generates a configuration read request.
svt_pcie_driver_app_transaction_cfg_wr_sequence This sequence generates a configuration write request.
svt_pcie_driver_app_transaction_io_rd_sequence This sequence generates an I/O read request.
svt_pcie_driver_app_transaction_io_wr_sequence This sequence generates an I/O write request.
svt_pcie_driver_app_transaction_mem_rd_sequence This sequence generates a memory read request.
svt_pcie_driver_app_transaction_mem_wr_sequence This sequence generates a memory write request.
svt_pcie_driver_app_transaction_null_sequence This sequence implements Null Traffic. This class creates a null sequence which can be associated with a sequencer but generates no traffic.
svt_pcie_driver_app_transaction_vendor_msg_0_sequence This sequence generates a vendor-defined message type 0 request.

Table 13-8 Driver application transaction sequences (Continued)

svt_pcie_driver_app_transaction_vendor_msg_1_sequence This sequence generates a vendor-defined message type 1 request.
svt_pcie_driver_app_transaction_vendor_msgd_0_sequence This sequence generates a vendor-defined message with data type 0 request.
svt_pcie_driver_app_transaction_vendor_msgd_1_sequence This sequence generates a vendor-defined message with data type 1 request.

13.5 Driver Application Callbacks and Exceptions

The Driver provides a callback class, svt_pcie_driver_app_callback. It is available for observation of transactions only. Modification of the packet at this level would have no meaning.

```
class svt_pcie_driver_app_callback extends svt_callback;
   function void new ( string name = "svt_pcie_driver_app_callback" )
   virtual function void transaction_ended ( svt_pcie_driver_app driver,
      svt_pcie_driver_app_transaction transaction )
endclass
```

The transaction_ended method is called at the conclusion of a transaction. It is called by the Driver whenever the transaction is completed by the link partner. Completion data returned by the link partner is available in the payload. The transaction_ended method is not called for partial completions.

13.5.1 Transaction Layer Exceptions

Driver level exceptions are applied via the transaction item, itself or via the factory rather than via callbacks. The svt_pcie_driver_app_transaction class contains an exception list, svt_pcie_driver_app_exception_list. By default the list is null, thus indicating no exception is to be applied. Exceptions are added by adding a handle to an exception list that is not null.

Reviewing the svt_pcie_driver_app_exception_list, once can setup a particular exception via the svt_pcie_driver_app_exception class.

13.6 Driver Status

The driver application provides a set of state values representing its status at any given time in the test simulation. These state values are encapsulated within `svt_pcie_driver_app_status` class. The driver application class has an object of type `svt_pcie_driver_application_status` instanced as status.

For more details, such as members, class inheritance UML, list of tasks, list of constraints, and so on, see the HTML class description of `svt_pcie_driver_app_status` at the following location:

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/status/class_svt_pcie_driver_app_status.html

13.6.1 Determining if the Driver Application is Idle

The Driver provides the svt_pcie_driver_app_service_wait_until_idle sequence for determining if the Driver is idle, i.e. all queues are empty and all transactions have been completed. This is useful for determining end-of-test.

Example:

```
svt_pcie_driver_app_service_wait_until_idle_sequence  wait_seq;
```

```
…
`ovm_do_on (wait_seq, p_sequencer.root_virt_seqr. driver_ seqr[0]);
…
```

13.7 Driver Application Events

The driver provides an ovm_event, EVENT_DRIVER_APP_TRANSACTION_ENDED which fires at the completion of each transaction. It can be used for synchronization of activities with the driver.

13.8 Driver Application TLMs

A port named ovm_seq_item_pull_port, service_seq_item_port is used for service requests.

13.9 Driver Application Transactions

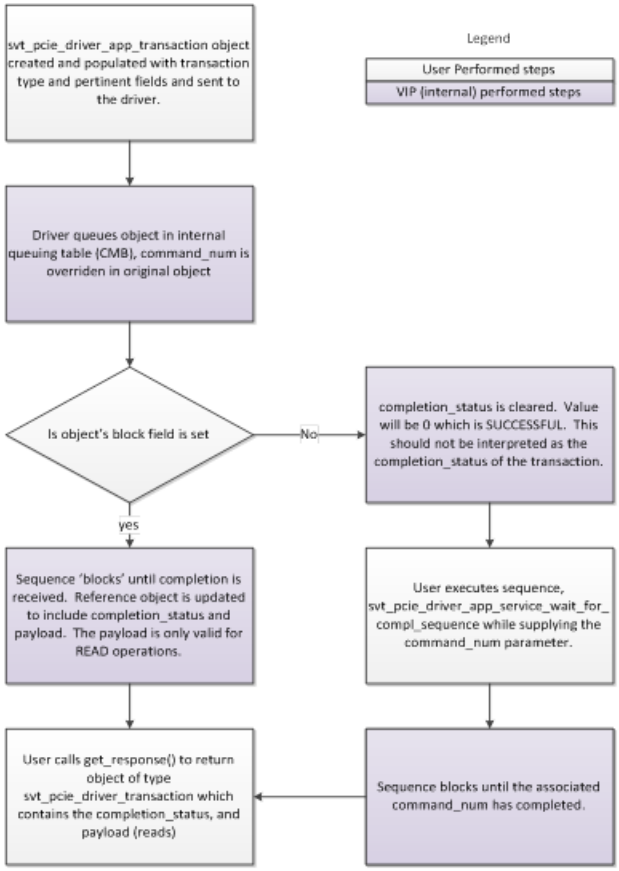
When utilizing the driver and its corresponding sequences for bus transactions, you may want to retrieve the completion status and optional data from reads. This section explains how this can be done in both blocking and non-blocking order.

The svt_pcie_driver_app_transaction driver application provides a single transaction_item representing all transaction types.

You can use the svt_pcie_driver_app_transaction in a custom sequence or in one of the pre-supplied sequences. All the pre-built driver application transaction sequences populate an object of type svt_pcie_driver_app_transaction and consequently will return a transaction item of type svt_pcie_driver_app_transaction when using the get_response() method.

Data Flow

The following diagram highlights the data flow for submitting and retrieving responses to the driver:



13.9.1 Blocking and Non-Blocking Transactions

13.9.1.1 Posted

For a posted transaction, a non-blocking transaction will queue the posted transaction and return. You can rely on the command_num being updated in the return to get_response(). For a blocking transaction, the behavior is similar, in that the transaction is queued and then executed but there is no completion, therefore only command_num is updated in the return to get_response(). For a blocking transaction the command will be considered completed once the driver application sends to the TL layer that is, it does not wait for the transaction to go out on the wire or an ACK to come back.

13.9.1.2 Non-Posted

For non-posted transactions, completions are expected so the behavior of the blocking and non-blocking is more particular. For a non-blocking transaction, the transaction will be queued and you can get the command_num from the return of get_response(). For a blocking transaction, the transaction is queued, executed, and all completions are returned. When calling get_response(), the return data will continue the command_num as well as payload and completion_status.

Example 13-1

```
svt_pcie_driver_app_transaction read_tran;
svt_pcie_driver_app_service_wait_for_compl_sequence wait_for_compl_seq;
```

```
`ovm_create(write_tran);
// populate read_tran
read_tran.block = 0;
`ovm_send(read_tran)
get_response(read_tran);
`ovm_do_on_with(wait_for_compl_seq,
                p_sequencer.root_virt_seq.driver_seqr[0],
                {wait_for_compl_seq.command_num=read_tran.command_num;
} )
get_response(wait_for_comple_seq);
// read_tran.payload now contains the completion data
```

14 Functional Coverage

The PCIe VIP provides notification routines which users can utilize for functional coverage. The notifications are called inside of a class which can easily be extended by users to meet their specific needs. A set of default covergroups is provided, which you can use some or all of.

14.1 Enabling Functional Coverage

To enable functional coverage, perform the following steps:

1. Set the following variable in the `svt_pcie_configuration` class:

   ```
   enable_cov = 6'b111111; // Bitwise enable
   ```

2. Set the `OVM_HOME` environment variable to point to the simulator's OVM package files.

   ```
   setenv OVM_HOME  ../<path_to_vcs_or_vcsmx_installation>/vcs_version/etc/ovm-2.1.2
   ```

3. Configure the following defines:

   ```
   +define+SVT_OVM_TECHNOLOGY
   +define+SVT_PCIE_ENABLE_COMMON_COV
   ```

The configuration variable `enable_cov` is a 6-bit vector, where each bit corresponds to enabling coverage for subsections listed in the table.

Table 14-1 Coverage Controls

Bit Position	Coverage Control
`enable_cov[0]`	When set to 1'b1, enables PIPE functional coverage.

Table 14-1 Coverage Controls

Bit Position	Coverage Control
`enable_cov[1]`	When set to 1'b1, enables functional coverage in PL Layer
`enable_cov[2]`	When set to 1'b1, enables functional coverage in DL Layer.
`enable_cov[3]`	When set to 1'b1, enables functional coverage in TL Layer.
`enable_cov[4]`	When set to 1'b1, enables LTSSM functional coverage.
`enable_cov[5]`	When set to 1'b1, enables 8b/10b symbol functional coverage (only valid for SERDES interface).

14.2 Class Structure and Callbacks

The classes described in this section are unencrypted and can be viewed in Include/pciesvc_coverage_pkg.sv.

All of the functional coverage classes have an abstract base class(ending is _base) which contains the variables which are to be used by coverage groups as well as pure virtual declarations for Update() and Sample() routines. The Update() callback routines are used to unpack data passed in the callback function into class variables. The Sample() callbacks are used to trigger the coverage groups.

Derived from each base class is a data class where the implementation for all of the Update() tasks is defined. Users that do not wish to use any of the provided functional coverage can extend their own coverage class from the data class.

A functional coverage class is extended from the data class, and in the functional coverage class there is an implementation of the Sample() callbacks along with a number of different functional coverage groups. Users that wish to utilize some or all of the provided functional coverage but modify or add to the existing coverage should extend from the functional coverage classes. Individual covergroups and/or coverpoints can be turned off/adjusted by using standard SystemVerilog syntax such as option.weight. Please refer to the SystemVerilog LRM for more details.

For users who wish to modify the Update() implementation in the _data class, it is recommended to call super.Update() in the child implementation to ensure that the data is unpacked correctly.

14.3 Overriding the Default Coverage Class

Each layer in the VIP protocol stack has a pointer to the corresponding coverage class. The Physical Layer has a pointer to both phy coverage as well as pipe coverage. Any class which replaces the default coverage class must have the _data class for the appropriate layer as its parent.

14.3.1 Overriding With OVM

OVM users should override the default coverage class by using the factory to replace the _data class with the desired class, as shown in the following example:

```
factory.set_type_override_by_type(
   pciesvc_coverage_pkg::link_fc_data::get_type(),
   a_different_coverage_class::get_type(),
   1
);
```

14.3.2 Overriding for SystemVerilog Users

There is an override function for each of the four types of coverage classes: tl, link, phy and pipe. The transaction override function is in the Transaction Layer, the link override function is in the Data Link Layer, and the phy and pipe functions are in the Physical Layer. You must first instantiate the class that you want to use for coverage and then call `new()` on it. Once the class has been constructed the object handle is passed through the override function call. All override function calls are described in Table 14-2. These tasks may also be called through the SystemVerilog API.

Table 14-2 Transaction override functions

Function Name	Arguments	Layer
SetTransactionCoverageClass	new_class – handle to the new coverage class. The new class must be derived from pciesvc_tl_fc_data or pciesvc_tl_fc_coverage.	*SVC_PATH*.port0.tl0
SetLinkCoverageClass	new_class – handle to the new coverage class. The new class must be derived from pciesvc_link_fc_data or pciesvc_link_fc_coverage.	*SVC_PATH*.port0.dl0
SetPhyCoverageClass	new_class – handle to the new coverage class. The new class must be derived from pciesvc_phy_fc_data or pciesvc_phy_fc_coverage.	*SVC_PATH*.port0.phy0
SetPipeCoverageClass	new_class – handle to the new coverage class. The new class must be derived from pciesvc_pipe_fc_data or pciesvc_pipe_fc_coverage.	*SVC_PATH*.port0.phy0

14.4 Transaction Layer

All methods and variables are declared in pciesvc_tl_fc_base, which is located in Include/pciesvc_coverage_pkg.sv. Implementation of the Update() functions is in the pciesvc_tl_fc_data class. The covergroups and implementation of the sample() functions are provided in the class pciesvc_tl_fc_coverage.

14.4.1 Transaction Layer Functional Coverage

Table 14-3 lists the covergroups, coverpoints and bins present in the Transaction Layer coverage class.

Table 14-3 Covergroups, coverpoints and bins in the Transaction Layer coverage class

Covergroup	Coverpoints	Bins

Table 14-3 Covergroups, coverpoints and bins in the Transaction Layer coverage class (Continued)

cg_tx_tc_vc_mapping	cp_tc	tc_0
		tc_1
		tc_2
		tc_3
		tc_4
		tc_5
		tc_6
		tc_7
	cp_vc	vc_0
		vc_1
		vc_2
		vc_3
		vc_4
		vc_5
		vc_6
		vc_7
	cp_tc_cross_vc	All TC and VC combinations
cg_rx_tc_vc_mapping	cp_tc	tc_0
		tc_1
		tc_2
		tc_3
		tc_4
		tc_5
		tc_6
		tc_7
	cp_vc	vc_0
		vc_1
		vc_2
		vc_3
		vc_4
		vc_5
		vc_6
		vc_7
	cp_tc_cross_vc	All TC and VC combinations

14.4.2 Transaction Layer Callbacks

Transaction Layer functional coverage class callbacks and arguments are listed in Table 14-4.

Table 14-4 Transaction Layer functional coverage class callbacks and arguments

Task Name	Arguments	I/O	Values
UpdateTxTcVcMapping Called every time the Transaction Layer passes a packet to the link.	tx_tc	I	Traffic class of the transmitted TLP
	tx_vc	I	VC which maps to the traffic class of the transmitted TLP
UpdateRxTcVcMapping Called every time the Transaction Layer receives a packet from the DL.	rx_tc	I	Traffic class of the received TLP
	rx_vc	I	VC which maps to the traffic class of the received TLP
SampleTxTcVcMapping Called immediately following UpdateTxTcVcMapping()	N/A	N/A	N/a
SampleRxTcVcMapping Called immediately following UpdateRxTcVcMapping()	N/A	N/A	N/a

14.5 Data Link Layer

All methods and class variables are declared in pciesvc_link_fc_base, which is located in Include/pciesvc_coverage_pkg.sv. Implementation of the Update() functions is in the pciesvc_link_fc_data class. The covergroups and implementation of the sample() functions are provided in the pciesvc_link_fc_coverage class.

Please note for the TLP and DLLP Update tasks that all fields in the argument list may not be valid depending on the type of packet. For example, the message_code field is valid only for message TLPs, and should be disregarded on other TLPs. The provided covergroups cover most aspects of TLP/DLLP transmission, but not all TLP fields have a coverpoint. However, all TLP fields are updated during the UpdateTxTLP/UpdateRxTLP callbacks so that users can create their own coverage if necessary.

14.5.1 Data Link Layer Functional Coverage

Table 14-5 lists the covergroups, coverpoints and bins present in the Data Link Layer layer coverage class. Note that the type field for TLPs and DLLPs is sampled in several coverpoints. Having different types of TLPs in different coverpoints allows users to easily change weights/goals within the coverpoints to cover only the types of packets they are interested in.

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class

Covergroup	Coverpoint	Bins	Comment

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

cg_tx_dllp	cp_dllp_type_acknak	ACK	ACK/NAK DLLPs
		NAK	
	cp_dllp_type_pm	PM Enter L1	Power Management DLLPS
		PM Enter L23	
		PM Active State Request	
		PM Request Ack	
	cp_dllp_type_vendor_specific	Vendor Specific	Vendor Specific
	cp_dllp_type_fc_vc0	initfc1_p_vc0	VC0 Flow Control DLLPs
		initfc1_np_vc0	
		initfc1_cpl_vc0	
		initfc2_p_vc0	
		initfc2_np_vc0	
		initfc2_cpl_vc0	
		updatefc_p_vc0	
		updatefc_np_vc0	
		updatefc_cpl_vc0	
	cp_dllp_type_fc_vc1	initfc1_p_vc1	VC1 Flow Control DLLPs
		initfc1_np_vc1	
		initfc1_cpl_vc1	
		initfc2_p_vc1	
		initfc2_np_vc1	
		initfc2_cpl_vc1	
		updatefc_p_vc1	
		updatefc_np_vc1	
		updatefc_cpl_vc1	
	cp_dllp_type_fc_vc2	initfc1_p_vc2	VC2 Flow Control DLLPs
		initfc1_np_vc2	
		initfc1_cpl_vc2	
		initfc2_p_vc2	
		initfc2_np_vc2	
		initfc2_cpl_vc2	
		updatefc_p_vc2	
		updatefc_np_vc2	
		updatefc_cpl_vc2	

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_dllp_type_fc_vc3	initfc1_p_vc3	VC3 Flow Control DLLPs
		initfc1_np_vc3	
		initfc1_cpl_vc3	
		initfc2_p_vc3	
		initfc2_np_vc3	
		initfc2_cpl_vc3	
		updatefc_p_vc3	
		updatefc_np_vc3	
		updatefc_cpl_vc3	
	cp_dllp_type_fc_vc4	initfc1_p_vc4	VC4 Flow Control DLLPs
		initfc1_np_vc4	
		initfc1_cpl_vc4	
		initfc2_p_vc4	
		initfc2_np_vc4	
		initfc2_cpl_vc4	
		updatefc_p_vc4	
		updatefc_np_vc4	
		updatefc_cpl_vc4	
	cp_dllp_type_fc_vc5	initfc1_p_vc5	VC5 Flow Control DLLPs
		initfc1_np_vc5	
		initfc1_cpl_vc5	
		initfc2_p_vc5	
		initfc2_np_vc5	
		initfc2_cpl_vc5	
		updatefc_p_vc5	
		updatefc_np_vc5	
		updatefc_cpl_vc5	

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_dllp_type_fc_vc6	initfc1_p_vc6	VC6 Flow Control DLLPs
		initfc1_np_vc6	
		initfc1_cpl_vc6	
		initfc2_p_vc6	
		initfc2_np_vc6	
		initfc2_cpl_vc6	
		updatefc_p_vc6	
		updatefc_np_vc6	
		updatefc_cpl_vc6	
	cp_dllp_type_fc_vc7	initfc1_p_vc7	VC7 Flow Control DLLPs
		initfc1_np_vc7	
		initfc1_cpl_vc7	
		initfc2_p_vc7	
		initfc2_np_vc7	
		initfc2_cpl_vc7	
		updatefc_p_vc7	
		updatefc_np_vc7	
		updatefc_cpl_vc7	
	cp_hdr_fc	less_8	HDR FC Value (sampled only on flow control type DLLPs)
		less_32	
		less_128	
		less_255	
	cp_data_fc	less_128	DATA FC Value (sampled only on flow control type DLLPs)
		less_512	
		less_1024	
		less_4096	
	cp_hdr_cross_fc_vc0	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc0	hdr cross flow control type for VC0
	cp_hdr_cross_fc_vc1	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc1	hdr cross flow control type for VC1
	cp_hdr_cross_fc_vc2	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc2	hdr cross flow control type for VC2

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_hdr_cross_fc_vc3	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc3	hdr cross flow control type for VC3
	cp_hdr_cross_fc_vc4	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc4	hdr cross flow control type for VC4
	cp_hdr_cross_fc_vc5	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc5	hdr cross flow control type for VC5
	cp_hdr_cross_fc_vc6	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc6	hdr cross flow control type for VC6
	cp_hdr_cross_fc_vc7	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc7	hdr cross flow control type for VC7
	cp_data_cross_fc_vc0	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc0	data cross flow control type for VC0
	cp_data_cross_fc_vc1	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc1	data cross flow control type for VC1
	cp_data_cross_fc_vc2	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc2	data cross flow control type for VC2
	cp_data_cross_fc_vc3	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc3	data cross flow control type for VC3
	cp_data_cross_fc_vc4	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc4	data cross flow control type for VC4

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_data_cross_fc_vc5	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc5	data cross flow control type for VC5
	cp_data_cross_fc_vc6	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc6	data cross flow control type for VC6
	cp_data_cross_fc_vc7	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc7	data cross flow control type for VC7
	cp_dllp_error_injections	corrupt_crc	Error injections for transmitted DLLPs
		unknown_type	
		rsvd_non_zero	
		duplicate_ack	
		missing_start	
		missing_end	
		corrupt_disparity	
		code_violation	
cg_rx_dllp	cp_dllp_type_acknak	ACK	ACK/NAK DLLPs
		NAK	
	cp_dllp_type_pm	PM Enter L1	Power Management DLLPS
		PM Enter L23	
		PM Active State Request	
		PM Request Ack	
	cp_dllp_type_vendor_specific	Vendor Specific	Vendor Specific

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_dllp_type_fc_vc0	initfc1_p_vc0	VC0 Flow Control DLLPs
		initfc1_np_vc0	
		initfc1_cpl_vc0	
		initfc2_p_vc0	
		initfc2_np_vc0	
		initfc2_cpl_vc0	
		updatefc_p_vc0	
		updatefc_np_vc0	
		updatefc_cpl_vc0	
	cp_dllp_type_fc_vc1	initfc1_p_vc1	VC1 Flow Control DLLPs
		initfc1_np_vc1	
		initfc1_cpl_vc1	
		initfc2_p_vc1	
		initfc2_np_vc1	
		initfc2_cpl_vc1	
		updatefc_p_vc1	
		updatefc_np_vc1	
		updatefc_cpl_vc1	
	cp_dllp_type_fc_vc2	initfc1_p_vc2	VC2 Flow Control DLLPs
		initfc1_np_vc2	
		initfc1_cpl_vc2	
		initfc2_p_vc2	
		initfc2_np_vc2	
		initfc2_cpl_vc2	
		updatefc_p_vc2	
		updatefc_np_vc2	
		updatefc_cpl_vc2	

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_dllp_type_fc_vc3	initfc1_p_vc3	VC3 Flow Control DLLPs
		initfc1_np_vc3	
		initfc1_cpl_vc3	
		initfc2_p_vc3	
		initfc2_np_vc3	
		initfc2_cpl_vc3	
		updatefc_p_vc3	
		updatefc_np_vc3	
		updatefc_cpl_vc3	
	cp_dllp_type_fc_vc4	initfc1_p_vc4	VC4 Flow Control DLLPs
		initfc1_np_vc4	
		initfc1_cpl_vc4	
		initfc2_p_vc4	
		initfc2_np_vc4	
		initfc2_cpl_vc4	
		updatefc_p_vc4	
		updatefc_np_vc4	
		updatefc_cpl_vc4	
	cp_dllp_type_fc_vc5	initfc1_p_vc5	VC5 Flow Control DLLPs
		initfc1_np_vc5	
		initfc1_cpl_vc5	
		initfc2_p_vc5	
		initfc2_np_vc5	
		initfc2_cpl_vc5	
		updatefc_p_vc5	
		updatefc_np_vc5	
		updatefc_cpl_vc5	

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_dllp_type_fc_vc6	initfc1_p_vc6	VC6 Flow Control DLLPs
		initfc1_np_vc6	
		initfc1_cpl_vc6	
		initfc2_p_vc6	
		initfc2_np_vc6	
		initfc2_cpl_vc6	
		updatefc_p_vc6	
		updatefc_np_vc6	
		updatefc_cpl_vc6	
	cp_dllp_type_fc_vc7	initfc1_p_vc7	VC7 Flow Control DLLPs
		initfc1_np_vc7	
		initfc1_cpl_vc7	
		initfc2_p_vc7	
		initfc2_np_vc7	
		initfc2_cpl_vc7	
		updatefc_p_vc7	
		updatefc_np_vc7	
		updatefc_cpl_vc7	
	cp_hdr_fc	less_8	HDR FC Value (sampled only on flow control type DLLPs)
		less_32	
		less_128	
		less_255	
	cp_data_fc	less_128	DATA FC Value (sampled only on flow control type DLLPs)
		less_512	
		less_1024	
		less_4096	
	cp_hdr_cross_fc_vc0	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc0	hdr cross flow control type for VC0

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_hdr_cross_fc_vc1	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc1	hdr cross flow control type for VC1
	cp_hdr_cross_fc_vc2	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc2	hdr cross flow control type for VC2
	cp_hdr_cross_fc_vc3	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc3	hdr cross flow control type for VC3
	cp_hdr_cross_fc_vc4	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc4	hdr cross flow control type for VC4
	cp_hdr_cross_fc_vc5	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc5	hdr cross flow control type for VC5
	cp_hdr_cross_fc_vc6	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc6	hdr cross flow control type for VC6
	cp_hdr_cross_fc_vc7	all combinations of cp_hdr_fc bins and cp_dllp_type_fc_vc7	hdr cross flow control type for VC7
	cp_data_cross_fc_vc0	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc0	data cross flow control type for VC0
	cp_data_cross_fc_vc1	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc1	data cross flow control type for VC1
	cp_data_cross_fc_vc2	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc2	data cross flow control type for VC2
	cp_data_cross_fc_vc3	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc3	data cross flow control type for VC3
	cp_data_cross_fc_vc4	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc4	data cross flow control type for VC4
	cp_data_cross_fc_vc5	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc5	data cross flow control type for VC5
	cp_data_cross_fc_vc6	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc6	data cross flow control type for VC6
	cp_data_cross_fc_vc7	all combinations of cp_data_fc bins and cp_dllp_type_fc_vc7	data cross flow control type for VC7

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

cg_tx_tlp	cp_mem_requests	mem_rd_req_32	fmt/type for Memory Requests
		mem_rd_req_64	
		mem_wr_req_32	
		mem_wr_req_64	
	cp_mem_rd_lk_requests	mem_rd_req_lk_32	fmt/type for Mem Read Locked Requests
		mem_rd_req_lk_64	
	cp_io_requests	io_rd_req	fmt/type for I/O Requests
		io_wr_req	
	cp_cfg_type0_requests	cfg_rd_req0	fmt/type for Type 0 Config Requests
		cfg_wr_req0	
	cp_cfg_type1_requests	cfg_rd_req1	fmt/type for Type 1 Config Requests
		cfg_wr_req01	
	cp_msg	Msg	fmt/type for Message TLPs
		MsgD	
	cp_cpl	Cpl	fmt/type for Completion TLPs
		CplD	
	cp_lk_cpl	CplLk	fmt/type for Completion Locked TLPs
		CplDLk	

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_atomic_ops	FetchAdd32	fmt/type for Atomic Ops
		FetchAdd64	
		Swap	
		Swap64	
		CAS32	
		CAS64	
	cp_traffic class	tc0	Traffic Class
		tc1	
		tc2	
		tc3	
		tc4	
		tc5	
		tc6	
		tc7	
	cp_transaction_hint	0/1	Transaction hint
	cp_tlp_digest	0/1	TLP digest bit
	cp_error_poison	0/1	Error Poison bit
	cp_address_transalation	default_untranslated	Address transaction bit
		translation_request	
		translated	
	cp_length	length_1	length field
		length_2_thru_1023	
		length_1024	
	cp_attr_id_order	0/1	ID ordering attribute bit
	cp_attr_relax_order	0/1	relaxed ordering attribute bit
	cp_attr_no_snoop	0/1	nosnoop attribute bit

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_first_dw_be	autobins for 4'b000-4'b1111	First DW byte enable. Fires only on mem, I/O and CFG type TLPs
	cp_last_dw_be	autobins for 4'b0-4'b1111	Last DW byte enable. Fires only on mom, I/O and CFG type TLPs
	cp_ph	0/1	processing hint
	cp_msg_code	cp_assert_inta	Message Code Type
		cp_assert_intb	
		cp_assert_intc	
		cp_assert_intd	
		cp_deassert_inta	
		cp_deassert_intb	
		cp_deassert_intc	
		cp_deassert_intd	
		pm_active_state_nak	
		pm_pme	
		pm_pme_turn_off	
		pm_pme_to_ack	
		err_cor	
		err_non_fatal	
		err_fatal	
		unlock	
		set_slot_power_limit	
		OBFF	
	cp_completion_status	successful completion	Completion Status
		unsupported request	
		completer abort	
		CRS	

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_sequence_num	seq_num_0	TLP Sequence Number
		seq_num_1_thru_4095	
		seq_num_4095	
	cp_ei_code	ei_none	Error Injection Codes
		ei_corupt_crc	
		ei_illegal_seq_num	
		ei_duplicate_seq_num	
		ei_nullified	
		ei_nullified_good_lcrc	
		ei_nullified_corrupt_lcrc	
		ei_corrupt_disparity	
		ei_code_violation	
		ei_missing_start	
		ei_missing_end	
		ei_8g_corrupt_header_crc	
		ei_8g_corrupt_header_parity	
		ei_corrupt_ecrc	
		ei_ignore_credit	
		ei_expect_ur	
		ei_expect_crs	
		ei_expect_ca	
		ei_expect_timeout	
cg_rx_tlp	cp_mem_requests	mem_rd_req_32	fmt/type for Memory Requests
		mem_rd_req_64	
		mem_wr_req_32	
		mem_wr_req_64	

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_mem_rd_lk_requests	mem_rd_req_lk_32	fmt/type for Mem Read Locked Requests
		mem_rd_req_lk_64	
	cp_io_requests	io_rd_req	fmt/type for I/O Requests
		io_wr_req	
	cp_cfg_type0_requests	cfg_rd_req0	fmt/type for Type 0 Config Requests
		cfg_wr_req0	
	cp_cfg_type1_requests	cfg_rd_req1	fmt/type for Type 1 Config Requests
		cfg_wr_req01	
	cp_msg	Msg	fmt/type for Message TLPs
		MsgD	
	cp_cpl	Cpl	fmt/type for Completion TLPs
		CplD	
	cp_lk_cpl	CplLk	fmt/type for Completion Locked TLPs
		CplDLk	
	cp_atomic_ops	FetchAdd32	fmt/type for Atomic Ops
		FetchAdd64	
		Swap	
		Swap64	
		CAS32	
		CAS64	
	cp_traffic class	tc0	Traffic Class
		tc1	
		tc2	
		tc3	
		tc4	
		tc5	
		tc6	
		tc7	

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_th	0/1	Transaction hint
	cp_td	0/1	TLP digest bit
	cp_ep	0/1	Error Poison bit
	cp_at	0/1	Address transaction bit
	cp_length	length_1 length_2_thru_1023	length field
		length_1024	
	cp_attr_id_order	0/1	ID ordering attribute bit
	cp_attr_relax_order	0/1	relaxed ordering attribute bit
	cp_attr_no_snoop	0/1	nosnoop attribute bit
	cp_first_dw_be	autobins for 4’b000-4’b1111	First DW byte enable. Fires only on mem, I/O and CFG type TLPs
	cp_last_dw_be	autobins for 4'b0-4'b1111	Last DW byte enable. Fires only on mom, I/O and CFG type TLPs
	cp_ph	0/1	processing hint

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

	cp_msg_code	cp_assert_inta	Message Code Type
		cp_assert_intb	
		cp_assert_intc	
		cp_assert_intd	
		cp_deassert_inta	
		cp_deassert_intb	
		cp_deassert_intc	
		cp_deassert_intd	
		pm_active_state_nak	
		pm_pme	
		pm_pme_turn_off	
		pm_pme_to_ack	
		err_cor	
		err_non_fatal	
		err_fatal	
		unlock	
		set_slot_power_limit	
		OBFF	
	cp_completion_status	successful completion	Completion Status
		unsupported request	
		completer abort	
		CRS	
	cp_sequence_num	seq_num_0	Sequence number assigned to TLP
		seq_num_1_thru_4094	
		seq_num_4095	
cg_tx_ipg	cp_tx_ipg	ipg_0	Inter packet gap of transmitted packets
		ipg_1	
		ipg_2	
		ipg_3_to_4	
		ipg_5_to_8	
		ipg_9_to_16	
		ipg_16_to_32	
		ipg_greater_than_32	

Table 14-5 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)

cg_rx_ipg	cp_rx_ipg	ipg_0	Inter packet gap of received packets
		ipg_1	
		ipg_2	
		ipg_3_to_4	
		ipg_5_to_8	
		ipg_9_to_16	
		ipg_16_to_32	
		ipg_greater_than_32	

14.5.2 Link Layer Callbacks

Data Link Layer callbacks are listed in Table 14-6.

Table 14-6 Data Link Layer callbacks

Function Name	Arguments	I/O	Values
UpdateTxDLLP This function is called every time the link finishes sending a DLLP.	dllp[63:0]	I	64 bit array containing the DLLP data
	ei_code[31:0]	I	Error injection code associated with the DLLP.
UpdateRxDLLP Called every time the link received a DLLP.	dllp[63:0]	I	64 bit array containing the DLLP data
	rx_status[31:0]	I	Status of the received DLLP. Status bits are defined in Include/pciesvc_parms.v under RECEIVED_TLP_STATUS*

Table 14-6 Data Link Layer callbacks (Continued)

UpdateTxTLP Called after the link sends the last byte of a TLP.	tlp_fmt[2:0]	I	Format field
	tlp_type[4:0]	I	Type field
	tc[2:0]	I	Traffic class
	th	I	Transaction hint
	td	I	TLP Digest bit
	ep	I	Error/Poison bit
	attr_id_order	I	ID Order attribute bit
	attr_relax_order	I	Relaxed order attribute bit
	attr_no_snoop	I	No snoop attribute attribute
	at[1:0]	I	address translation
	length[9:0]	I	length field
	ecrc[31:0]	I	ECRC(digest) value
	lcrc[31:0]	I	Link CRC value
	sequence_num (int)	I	Sequence number of the TLP
	requester_id[15:0]	I	Requester ID field
	tag[7:0]	I	Tag value
	first_dw_be[3:0]	I	First DW byte enable field.
	last_dw_be[3:0]	I	Last DW byte enable field.
	address[63:0]	I	Address field.

Table 14-6 Data Link Layer callbacks (Continued)

	ph[1:0]	I	Processing hint
	bus_num[7:0]	I	bus number
	device_num[2:0]	I	device number
	function_num[2:0]	I	function number
	register_num[9:0]	I	Combines reg and ext_reg field for config TLPs
	message_code[7:0]	I	Message code field
	message_dword2[31:0]	I	2nd dword of message TLP. This would be used for vendor specific messages.
	message_dword3[31:0]	I	3rd dword of message TLP.
	completer_id[15:0]	I	Completer ID field
	completion_status[2:0]	I	Completion status
	bcm	I	Byte count modified field
	byte_count[11:0]	I	Byte count field
	lower_address[6:0]	I	Lower address field.
	steering_tag[7:0]	I	Steering tag.
	payload_data (dynamic array)	I	Payload data, if any present.
	ei_code [31:0]	I	Error injection code associated with the TLP.
UpdateRxTLP Called after the link receives the last byte of a TLP.	tlp_fmt[2:0]	I	Format field
	tlp_type[4:0]	I	Type field
	tc[2:0]	I	Traffic class
	th	I	Transaction hint
	td	I	TLP Digest bit
	ep	I	Error/Poison bit
	attr_id_order	I	ID Order attribute bit
	attr_relax_order	I	Rleaxed order attribute bit
	attr_no_snoop	I	No snoop attribute attribute

Table 14-6 Data Link Layer callbacks (Continued)

	at[1:0]	I	address translation
	length[9:0]	I	length field
	ecrc[31:0]	I	ECRC(digest) value
	lcrc[31:0]	I	Link CRC value
	sequence_num (int)	I	Sequence number of the TLP
	requester_id[15:0]	I	Requester ID field
	tag[7:0]	I	Tag value
	first_dw_be[3:0]	I	First DW byte enable field.
	last_dw_be[3:0]	I	Last DW byte enable field.
	address[63:0]	I	Address field.
	ph[1:0]	I	Processing hint
	bus_num[7:0]	I	bus number
	device_num[2:0]	I	device number
	function_num[2:0]	I	function number
	register_num[9:0]	I	Combines reg and ext_reg field for config TLPs
	message_code[7:0]	I	Message code field
	message_dword2[31:0]	I	2nd dword of message TLP. This would be used for vendor specific messages.
	message_dword3[31:0]	I	3rd dword of message TLP.
	completer_id[15:0]	I	Completer ID field
	completion_status[2:0]	I	Completion status
	bcm	I	Byte count modified field
	byte_count[11:0]	I	Byte count field
	lower_address[6:0]	I	Lower address field.I
	steering_tag[7:0]	I	Steering tag.
	payload_data (dynamic array)	I	Payload data, if any present.
UpdateTxIpg Called every time a new packet starts transmission.	ipg(int)	I	Number of bytes between current packet and previously transmitted packet.
UpdateRxIpg Called on the start of a new received packet.	ipg(int)	I	Number of bytes between current packet and previously received packet

Table 14-6 Data Link Layer callbacks (Continued)

UpdateDLCMSMState This function is called every time the DLCMSM changes state.	state[31:0]	I	Current state of the DLCMSM (states are defined in Verilog/Link_Layer/pciesvc_ll_parms.v)
UpdateFCState Called every time the FC state machine changes state.	state[31:0]	I	Current state of the flow control state machine. States are defined in Verilog/Link_Layer/pciesvc_ll_parms.v
SampleTxDLLP Called immediately after UpdateTxDLLP()	n/a	n/a	n/a
SampleRxDLLP Called immediately after UpdateRxDLLP()	n/a	n/a	n/a
SampleTxTLP Called immediately after UpdateTxTLP()	n/a	n/a	n/a
SampleRxTLP Called immediately after SampleRxTLP	n/a	n/a	n/a
SampleTxIPG Called immediately after UpdateTxIPG()	n/a	n/a	n/a
SampleRxIPG Called immediately following SampleRxIPG	n/a	n/a	n/a
SampleDLCMSMState Called immediately following UpdateDLCMSMState()	n/a	n/a	n/a
SampleFCState Called immediately following UpdateFcState()	n/a	n/a	n/a

14.6 Physical Layer

All methods and class variables are declared in pciesvc_phy_fc_base, which is located in Include/pciesvc_coverage_pkg.sv. Implementation of the Update() functions is in the pciesvc_phy_fc_data class. The covergroups and implementation of the sample() functions are provided in the class pciesvc_phy_fc_coverage.

A covergroup with all of the legal transitions in the LTSSM has been provided, though users should note that the PCIESVC LTSSM hitting a certain state doesn't necessarily imply that the DUT has successfully entered that state. Depending on whether or not the VIP is upstream or downstream not all state transitions may apply. Finally, in many cases there are multiple conditions which may trigger a transition from one state to the next (example: transitioning from L0 to recover due to receiving a training set, or going from L0 to recover for a speed change). Additional coverage will be required to capture these conditions.

14.6.1 Physical Layer Functional Coverage

Covergroups, coverpoints and bins in the Physical Layer coverage class are described in Table 14-7.

Table 14-7 Covergoups, coverpoints and bins in the Physical Layer coverage class

Covergroup	Coverpoint	Bins	Comment
cg_negotiated_data_rate	cp_negotiated_data_rate	speed_2_5G	Data rate upon entry into L0
		speed_5_0G	
		speed_8_0G*	
cg_negotiated_link_width	cp_negotiated_link_width	link_width_1	Link width upon entry into L0
		link_width_2	
		link_width_4	
		link_width_8	
		link_width_12	
		link_width_16	
		link_width_32	

Table 14-7 Covergoups, coverpoints and bins in the Physical Layer coverage class (Continued)

cg_ltssm_state_transitions cp_ltss_state_transitions	detect_quiet_to_detect_active	State transitions of all LTSSM states except for the L0s substates, which have their own separate coverage.
	detect_active_to_polling_active	
	polling_active_to_polling_compliance	
	polling_active_to_polling_configuration	
	polling_active_to_detect_quiet	
	polling_compliance_to_detect_quiet	
	polling_compliance_to_polling_active	
	polling_configuration_to_configuration_linkwidth_start	
	polling_configuration_to_detect_quiet	
	configuration_linkwidth_start_to_disabled	
	configuration_linkwidth_start_to_loopback_entry	
	configuration_linkwidth_start_to_configuration_linkwidth_accept	
	configuration_linkwidth_start_to_detect_quiet	
	configuration_linkwidth_accept_to_configuation_lanenum_wait	
	configuration_linkwidth_accept_to_detect_quiet	
	configuration_lanenum_accept_to_configuration_complete	

Table 14-7 Covergoups, coverpoints and bins in the Physical Layer coverage class (Continued)

	configuration_lanenum_accept_to_configuration_lanenum_wait	
	configuration_lanenum_accept_to_detect_quiet	
	configuration_lanenum_wait_to_configuration_lanenum_accept	
	configuration_lanenum_wait_to_detect_quiet	
	configuration_complete_to_configuration_idle	
	configuration_complete_to_detect_quiet	
	configuration_idle_to_l0	
	configuration_idle_to_detect_quiet	
	configuration_idle_to_recovery_rcvrlock	
	recovery_rcvrlock_to_recovery_equalization_phase0*	
	recovery_rcvrlock_to_recover_equalization_phase1*	
	recovery_rcvrlock_to_recovery_rcvrcfg	
	recovery_rcvrlock_to_recovery_speed	
	recovery_rcvrlock_to_configuration_linkwidth_start	
	recovery_rcvrlock_to_detect_quiet	
	recovery_equalization_phase0_to_recovery_speed*	
	recovery_equalization_phase0_to_recovery_equalization_phase1	
	recovery_equalization_phase1_to_recovery_rcvrlock*	
	recovery_equalization_phase1_to_recovery_speed*	

Table 14-7 Covergoups, coverpoints and bins in the Physical Layer coverage class (Continued)

		recovery_equalization_phase2_to_recovery_speed*	
		recovery_equalization_phase2_to_recovery_equaliztion_phase3*	
		recovery_equalization_phase3_to_recovery_speed*	
		recovery_equalization_phase3_to_recovery_rcvrlock*	
		recovery_speed_to_recovery_rcvrlock	
		recovery_rcvrcfg_to_recovery_idle	
		recovery_rcvrcfg_to_configuration_linkwidth_start	
		recovery_rcvrcfg_to_recovery_idle	
		recovery_rcvrcfg_to_configuration_linkwidth_start	
		recovery_rcvrcfg_to_detect_quiet	
		recovery_idle_to_disabled	
		recovery_idle_to_hot_reset	
		recovery_idle_to_configuration_linkwidth_start	
		recovery_idle_to_loopback_entry	
		recovery_idle_to_l0	
		recovery_idle_to_detect_quiet	
		recovery_idle_to_recovery_rcvrlock	
		l0_to_recovery_rcvrlock	
		l0_to_l1_entry	
		l1_entry_to_l1_idle	
		l1_entry_to_recovery_rcvrlock	
		l1_idle_to_l1_1_idle*	

Table 14-7 Covergoups, coverpoints and bins in the Physical Layer coverage class (Continued)

		l1_idle_to_l1_2_entry*	
		l1_2_entry_to_l1_2_idle*	
		l1_2_idle_to_l1_2_exit*	
		l1_2_exit_to_l1_idle*	
		l1_1_idle_to_l1_idle*	
		l1_1_idle_to_recovery_rcvrlock*	
		l0_to_l2_idle	
		l2_idle_to_detect_quiet	
		disabled_to_detect_quiet	
		loopback_entry_to_loopback_activ e	
		loopback_entry_to_loopback_exit	
		loopback_active_to_loopback_exit	
		loopback_exit_to_detect_quiet	
		hot_reset_to_detect_quiet	
cg_tx_l0s_substate_ transitions	cp_tx_l0s_substate	l0_to_l0s_entry	L0s substate transitions for the transmit side
		l0s_entry_to_l0s_idle	
		l0s_idle_to_l0s_fts	
		l0s_fts_to_l0	
cg_rx_l0s_substate_ transitions	cp_rx_l0s_substate_ transitions	l0_to_l0s_entry	L0s substate transitions for the receive side
		l0s_entry_to_l0s_idle	
		l0s_idle_to_l0s_fts	
		l0s_fts_to_l0	
		los_fts_to_recovery_rcvrlock	
*Exist for 8G models only			

14.6.2 Physical Layer Callbacks

Callbacks in the Physical Layer are defined in Table 14-8.

Table 14-8 Callbacks in the Physical Layer

Function Name	Arguments	I/O	Values
UpdateNegotiatedDataRate Called every time the LTSSM enters the L0 state.	rate [2:0]	I	Pipe rate value upon entering L0
UpdateNegotiatedLinkWidth Called every time the LTSSM enters the L0 state.	width (int)	I	Link width upon entering L0
UpdateLtssmState Called every time the LTSSM enters a new state.	state [31:0]	I	Current LTSSM state
UpdateTxL0sSubstate Called every time the LTSSM transmit side enters a new L0s substate.	tx_l0s_substate[31:0]	I	Current LTSSM tx substate
UpdateRxL0sSubstate Called every time the LTSSM receive side enters a new L0s substate.	rx_l0s_substate[31:0]	I	Current LTSSM rx substate
SampleNegotiatedDataRate Called immediately following UpdateNegotiatedDataRate()	n/a	n/a	n/a
SampleNegotiatedLinkWidth Called immediately following UpdateNegotiatedLinkWidth	n/a	n/a	n/a
SampleLtssmState Called immediately following UpdateLtssmState	n/a	n/a	n/a
SampleTxL0sSubstate Called immediately following UpdateTxL0sSubstate	n/a	n/a	n/a
SampleRxL0sSubstate Called immediately following UpdateRxL0sSubstate	n/a	n/a	n/a

14.7 PIPE Interface

All methods and class variables are declared in pciesvc_pipe_fc_base, which is located in Include/pciesvc_coverage_pkg.sv. Implementation of the Update() functions is in the pciesvc_pipe_fc_data class. The covergroups and implementation of the Sample() functions are provided in the class pciesvc_pipe_fc_coverage.

14.7.1 PIPE Functional Coverage

PIPE functional covergroups coverpoints, and bins are listed in Table 14-9.

Table 14-9 PIPE covergroups, coverpoints and bins

Covergroup	Coverpoint	Bins	Comment
cg_rate	cp_rate	PIPE_RATE_2_5G	Valid values for the pipe rate signal
		PIPE_RATE_5G	
		PIPE_RATE_8G*	
cg_powerdown	cp_powerdown	P0	Valid values for the pipe powerdown signal
		P0s	
		P1	
		P2	
data_bus_width	cp_data_bus_width	bus_width_8_bits	Valid values for the data_bus_width signal.
		bus_width_16_bits	
		bus_width_32_bits	

* Exists for 8G models only

14.7.2 PIPE Interface Callbacks

Callbacks in the PIPE interface are listed in Table 14-10.

Table 14-10 PIPE callbacks

Function Name	Arguments	I/O	Values
UpdateRate Called every time the pipe signal rate changes.	rate [1:0]	I	Pipe rate value
UpdataPowerDown Called every time the pipe signal powerdown changes.	powerdown[2:0]	I	Pipe powerdown value
UpdateDataBusWidth This function is called every time the pipe signal data_bus_width changes value.	data_bus_width[1:0]	I	Pipe data bus width value

Table 14-10 PIPE callbacks (Continued)

UpdateTxPipeLane This function is called once per lane per pipe clock cycle and copies over all of the tx signals for 1 lane into class variables.	lane_number (int)	I	lane number to be updated
	tx_data[31:0]	I	transmit data
	tx_data_k[3:0]	I	transmit data control bits
	tx_compliance	I	transmit compliance
	tx_data_valid*	I	data_valid
	tx_start_block*	I	start block
	tx_sync_header*	I	transmit sync header
	tx_elec_idle	I	transmit electrical idle
UpdateRxPipeLane This function is called once per lane per pipe clock cycle and copies over all of the rx signals for 1 lane into class variables.	rx_data[31:0]	I	receive data
	rx_data_k[3:0]	I	receive data control bits
	rx_status[1:0]	I	receive status
	rx_valid	I	receive data valid
	rx_elec_idle	I	receive electricle idle
	rx_data_valid*	I	receive data valid
	rx_start_block*	I	received start of a new block
	rx_sync_header*	I	value of sync header
	invert_rx_polarity	I	invert received polarity
SampleRate Called immediately after UpdateRate()	n/a	n/a	n/a
SamplePowerDown Called immediately after SamplePowerDown()	n/a	n/a	n/a
SampleDataBusWidth Called immediately after UpdateDataBusWidth().	n/a	n/a	n/a
SampleTxPipeLane Called once per pipe clock after all tx lanes are updated	n/a	n/a	n/a
SampleRxPipeLane Called once per pipe clock after all rx lanes are updated.	n/a	n/a	n/a
*These signals present in 8G models only			

15 M-PHY Adapter Layer

15.1 Overview

This chapter contains the following topics:

The PCIe VIP implements the M-PHY interface through a ovm_component named svt_pcie_mphy_adapter. The PCIe VIP M-PHY Adapter component object extends from the ovm_component class.

When an M-PCIe interface is being simulated, the M-PHY Adapter layer rather than the Physical layer models the interface to a DUT. The PCIe VIP M-PHY Adapter component provides the features defined in the M-PCIE ECN specification related to the PHY Adapter (PA) block and instances the necessary M-PHY VIP components.

Note The VIP-MPCIE-OPT-SVT license key is required to run the M-PCIe VIP.

Figure 15-1 shows the relationship of the M-PHY Adapter Layer to the other stacked layers of the VIP.

Figure 15-1 VIP Layered Architecture with M-PHY Serial Interface

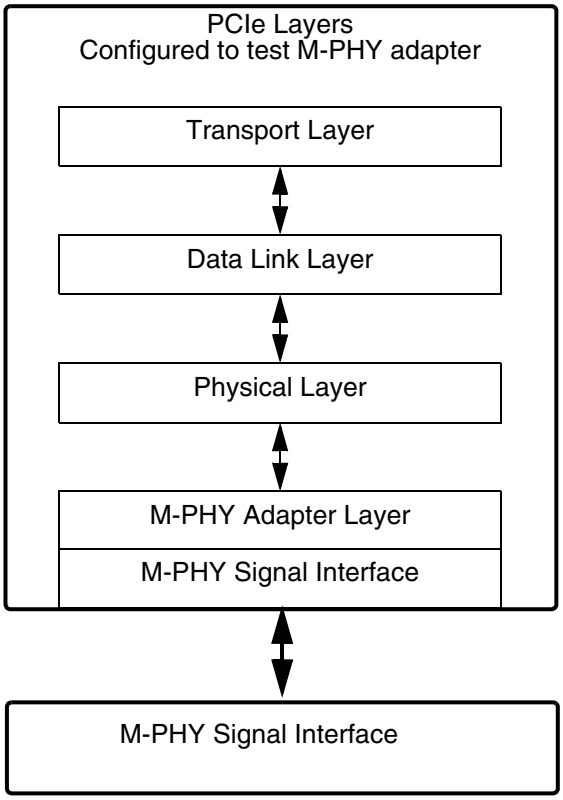
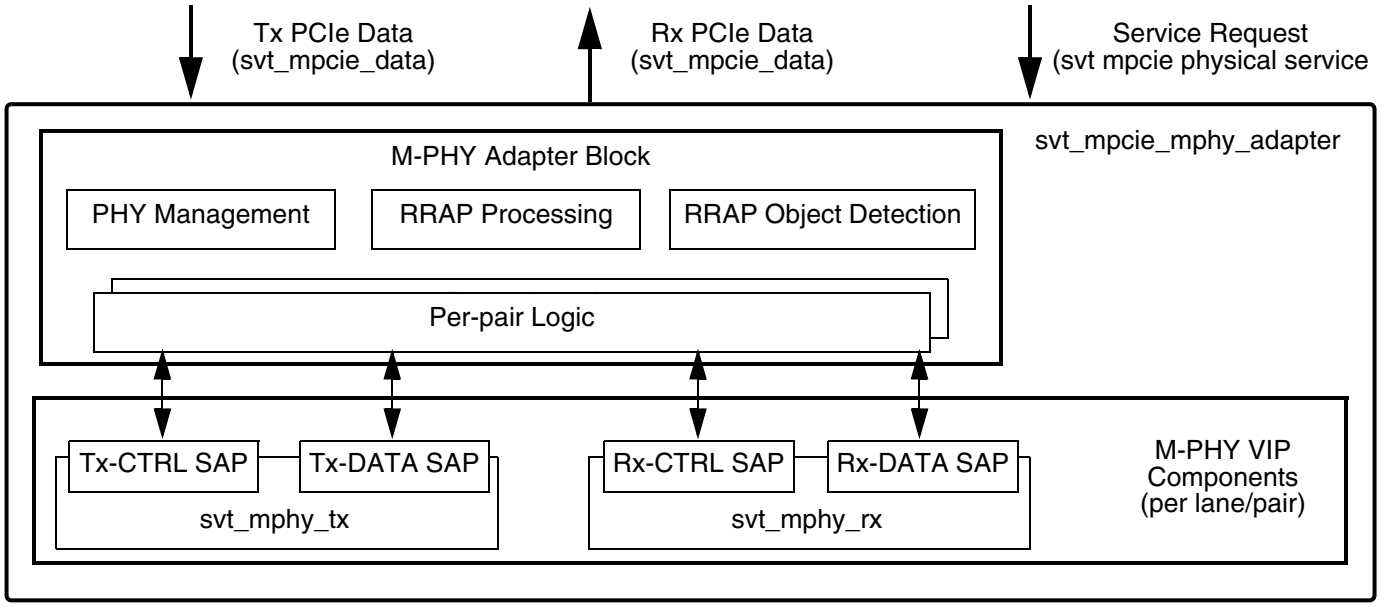


Figure 15-2 shows the make up of the M-PHY Adapter Layer.

Figure 15-2 Functionality of the M-PHY Adapter Layer



The VIP may be configured with 1, 2, or 4 Rx M-PHY lanes and 1, 2 or 4 Tx M-PHY lanes.

- The number of Rx and Tx lanes may be different.
- M-PCIe also supports the notion that a logical lane may map to a different physical lane. For example logical lane 0 may have its data sent/received on physical lane 2. This is supported by the VIP's svt_pcie_mphy_adapter_configuration:mpcie_[rx|tx]_pair_to_lane[$} array, which holds the physical lane number for each logical lane (pair) number.
- The M-PHY lanes are independent of each other, coordinated only by the PCIe M-PHY Adapter layer.

Each lane also supports simultaneous CRTL and DATA accesses:

- Connection to/from the M-PHY Adapter layer, and between Phy Adapter and M-PHY is channel/port based.
  - For Tx, PCIe Data (svt_pcie_symbol) objects produced by the PCIe Agent's Physical Layer are passed to the Agent's M-PHY Adapter Layer. There they are transformed into CTRL SAP/ DATA SAP objects and passed to the M-PHY components.
  - For Rx, activity on the signal interface is detected and interpreted by the M-PHY components to create CTRL SAP/DATA SAP objects, which are passed to the PCIe Agent's M-PHY Adapter Layer where they are interpreted and transformed into PCIe Data (svt_pcie_symbol objects. The PCIe Data objects are then passed to the Agent's Physical Layer.
  - CTRL SAP and DATA SAP are both svt_mphy_transaction objects.
  - Callbacks allow testbench access to data objects passing through the various channels/ports.
- Service Requests allow the Link Layer and the testbench to trigger the M-PHY Adapter layer to take specific actions (for example, to direct M-PHYs to go to the HIBERN8 state).

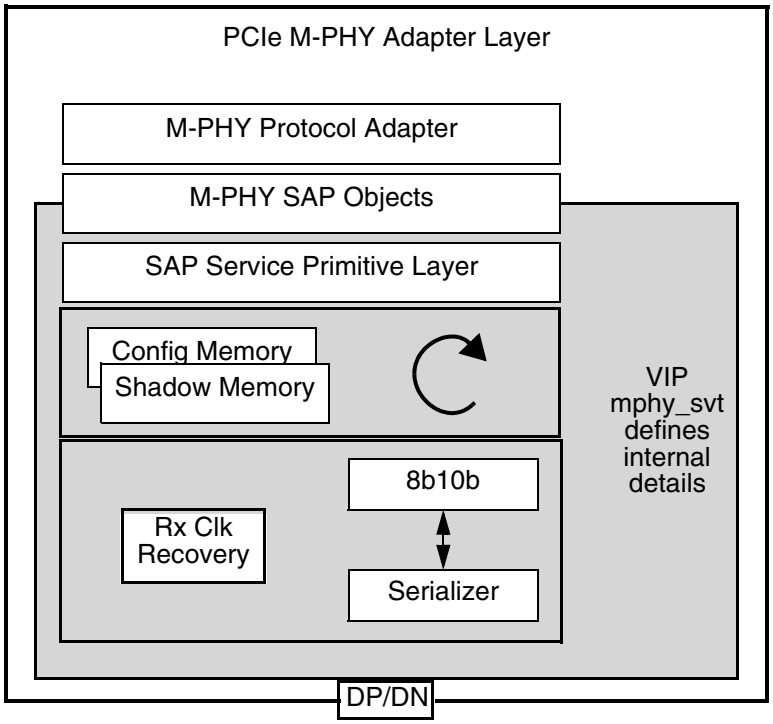
Refer to Service Transactions for additional information on Service Transactions.

Attention

While not shown in any of the figures, the M-PHY Adapter Layer can also be configured to contain an M-PHY Monitor (svt_mphy_monitor) component associated with each M-PHY Tx and M-PHY Rx component. These monitor components may be used for functional coverage and protocol checking of the M-PHY interfaces. There are properties (for example, pcie_enable_mphy_monitor) in the svt_mphy_adapter_configuration PCIe configuration object class that may be used to control the behavior of the M-PHY monitors.Refer to the class reference for more details.

Figure 15-3 is a high level block diagram of the M-PHY Adapter component.

Figure 15-3 Overview of M-PHY Adapter Layer



15.2 Configuration Classes

To configure the M-PHY Adapter, you use two sets of configuration classes: One for the M-PHY Adapter Layer and another for the M-PHY layer.

- svt_mphy_adapter_configuration

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

  In addition to the sub-configurations for the M-PHY component there are a number of configuration properties specific to M-PCIe. Please refer to the class reference for more details. This is the top level configuration class for M-PCIe as it includes the following class within the following arrays:

- pcie_mphy_rx_cfg[$]: an array of type svt_pcie_mphy_configuration
- pcie_mphy_tx_cfg[$]: an array of type svt_pcie_mphy_configuration

- svt_pcie_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 9 and 10 of the ECN_M-PCIE specification.

  Note: This class has M-Tx and/or M-Rx capability attributes from the M-PHY specification Tables 48 and 52.

15.3 Signal Interfaces

Unlike other PCIe components, the PCIe M-PHY Adapter components communicate through signal interfaces as well as through channels. However the PCIe M-PHY Adapter component does not use a port instance like the PCIe Physical component does. Instead the PCIe M-PHY Adapter instances the number of svt_mphy_tx and svt_mphy_rx components required by the total number of lanes being simulated. The svt_mphy_tx and svt_mphy_rx components communicate either through channel connects or directly with the M-PHY interface specified in the svt_mphy_configuration class supplied to the M-PHY components.

15.4 Data Factory Objects

The following are M-PHY Adapter layer data factory objects.

The following factory allocates new data descriptor instances to represent data placed in data out channel. This factory is always present. To cause the M-PHY Adapter component to use an extended data descriptor, replace this factory with the extended version:

- Attribute Name: mpcie_data_factory. M-PHY Adapter out factories

The following factories allocate new data descriptor instances to represent data placed in SAP DATA and SAP CTRL channels. These factories are always present. Replace these factories with the extended versions to cause the M-PHY Adapter component to use an extended data descriptor.

- Attribute Names: sap_data_factory and sap_ctrl_factory. M-PHY Adapter RRAP factories

The following factories allocate new data descriptor instances to represent RRAP activity occurring in the model or across the bus.

- Attribute Names: rrap_packet_factory and rrap_transaction_factory

15.5 Exception List Factories

The M-PHY Adapter component supplies the following exception list factories. They are used for creating exceptions for the processing of RRAP packets or RRAP transactions:

- randomized_mpcie_rrap_packet_tx_exception_list
- randomized_mpcie_rrap_transaction_exception_list

15.6 Error injection

The M-PHY Adapter layer provides the following error injection methods:

- RRAP transactions

  For RRAP target processing can inject invalid response, no response, and incorrect processing. See the HTML documentation for svt_mpcie_rrap_transaction::rrap_response_type_enum

- RRAP Packets
  - Parity error
  - Reserved field error
  - Invalid packet type

15.7 Transactions

The next sections describe data transaction classes that support M-PCIe functionality in the VIP.

15.7.1 Data Transactions

The following data transaction classes support M-PCIe functionality in the VIP.

- svt_pcie_symbol

  Transfers data symbols between the Link, Physical, and M-PHY Adapter layers.

- svt_mphy_transaction

  Produced or consumed by the M-PHY Adapter Layer to communicate CTRL and DATA SAPs to or from the M-PHY interface component.

- svt_mpcie_rrap_transaction

  Represents an M-PCIe Remote Register Access Protocol command/response pair. Its implementation array consists of two svt_mpcie_rrap_packet data objects.

- svt_mpcie_rrap_packet

  Represents a single M-PCIe Remote Register Access Protocol command or response.

15.7.2 Service Transactions

The svt_pcie_mphy_adapter_service data class is used to request specific non-dataflow actions from the M-PHY Adapter Layer:

- MPCIE_RRAP
  - Requests execution of the transaction defined by rrap_transaction. Although RRAP is the protocol for accessing the registers of the remote device, this service is used for both accessing remote registers and accessing the registers of the local device. The execute_locally member in the rrap_transaction object designates whether the transaction is targeting the local or remote configuration attributes.
- MPCIE_MTX_EXIT_HIBERN8
  - Requests Tx PHYs to exit theHibern8 state.
- MPCIE_MTXRX_CFG_FOR_HIBERN8
  - Requests Tx and Rx PHYs to be configured to allow entry to the Hibern8 state. If issued while in HS_BURST, upon exiting HS_BURST CFGRDY CTRL SAP objects are issued to all local M-PHY instances.
- MPCIE_MTX_EXIT_BURST
  - Requests M-PCIe Tx PHYs to exit BURST state.
  - If an MPCIE_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_pcie_mphy_adapter_status indicate STALL has been reached. The command is not ENDED until both PHY states indicate HIBERN8 has been reached.

- MPCIE_MTX_ENTER_BURST
  - Requests Tx PHYs to enter BURST state.
- MPCIE_MTX_LINE_RESET
  - Requests Tx PHYs to perform LINE Reset.
- PCIE_MPHY_RESET
  - Requests Tx PHYs to perform the reset operation selected by mphy_reset_kind.

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

- mphy_reset_kind
- rrap_packet
- rrap_transaction

15.8 Using the M-PCIe Interface

When an M-PCIe interface is being simulated, each connection to the other components in the design is made through one of six SystemVerilog interfaces representing the possible interfaces of the M-PHY.

The following two interfaces are used by the VIP if the DUT subsystem uses an RMMI interface and includes the link. For these interfaces the VIP models the link partner functionality above the RMMI interface that is on the far side of the link from the DUT:

- svt_mphy_rmmi_mtx_dut_phy_if
- svt_mphy_rmmi_mrx_dut_phy_if

The following two interfaces are used if the DUT subsystem uses an RMMI interface and does not include the link. For these interfaces the VIP models the link partner, the link, and the M-PHY functionality up to the RMMI interface of the DUT:

- svt_mphy_rmmi_mtx_dut_controller_if
- svt_mphy_rmmi_mrx_dut_controller_if

The following two interfaces are used if both the VIP and the DUT contain or model PHYs, and both connect to the serial signal interfaces:

- svt_mphy_serial_tx_if
- svt_mphy_serial_rx_if

The testbench must create and provide one of these interfaces for each M-PHY TX and M-PHY RX in the M-PHY Adapter.

With OVM, the interfaces are assigned through the config_db.

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

15.9 Shared Status

The following properties in the svt_pcie_mphy_adapter class are accessible as shared status information related to M-PCIe and M-PHY state:

- mphy_tx_status[$]

  An 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_pcie_mphy_adapter layer component.

- mphy_rx_status[$]

  An 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_pcie_mphy_adapter 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.

- pcie_mphy_tx_aggregate_state_stable

  A bit that is high when all the M-PHY Tx components have the same FSM state, and low otherwise.

- pcie_mphy_rx_aggregate_state_stable

  A bit that is high when all the M-PHY Rx components have the same FSM state, and low otherwise.

In addition to the above (included here due to their connection to the M-PHY components) there are a number of other notifications and status variables related to M-PCIe, with names like mpcie_* and NOTIFY_MPCIE_*. 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 PCIe VIP agent.

15.10 Configuring the PCIe M-PHY Adapter for an RMMI Interface

The PCIe Agent's configuration class, svt_pcie_configuration, has two svt_pcie_mphy_adapter_configuration objects: mphy_adapter_cfg and remote_mphy_adapter_cfg. For DUTs that use the serial interface the remote_mphy_adapter_cfg remains null. If the DUT has an RMMI interface then the configuration process for the VIP depends on whether the DUT subsystem includes the link.

15.10.1 Configuring the VIP for a DUT Subsystem That Does Not Include the Link

If the PCIe VIP is being used to test a DUT subsystem that has an RMMI interface and that does not include the link (that is, if the DUT's connection is from the MAC perspective of an M-PCIe MAC/PHY RMMI interface), then the VIP requires both the mphy_adapter_cfg and remote_mphy_adapter_cfg. In this case, the VIP effectively contains two virtual physical layers: mphy_adapter_cfg represents the VIP's local side of the link and remote_mphy_adapter_cfg represents the remote side of the link.

The process for creating the necessary configurations is as follows:

1. Create an instance of the top level configuration, svt_pcie_configuration.

2. Customize the top level variables such as mpcie_signal_interface in the mphy_adapter_cfg instance.
3. Invoke the svt_pcie_configuration function create_remote_mphy_adapter_cfg() to create the remote_mphy_adapter_cfg. It is cloned from the mphy_adapter_cfg, preserving the customized settings.
4. Invoke the svt_pcie_mphy_adapter_configuration function create_mphy_cfgs() for the mphy_adapter_cfg configuration instances to create the M-PHY configuration objects.
5. Invoke the svt_pcie_mphy_adapter_configuration function create_mphy_cfgs() for the remote_mphy_adapter_cfg configuration instances to create the M-PHY configuration objects.
6. Customize the M-PHY configurations as necessary.

Steps 4 and 5 may be done in either order, but must follow steps 1-3, which must be in the order shown.

15.10.2 Configuring the VIP for a DUT Subsystem That Includes the Link

If the PCIe VIP is being used to test a DUT subsystem that has an RMMI interface and that includes the link (that is, if the DUT's connection is from the PHY perspective of an M-PCIe MAC/PHY RMMI interface), then the remote_mphy_adapter_cfg is not used and should remain null. In this case, the VIP effectively contains one virtual physical layer (its local PHY) with the configuration controlled by the mphy_adapter_cfg object.

The process for creating the necessary configurations is as follows:

1. Create an instance of the top level configuration, svt_pcie_configuration.
2. Customize the top level variables such as mpcie_signal_interface in the mphy_adapter_cfg instance.
3. Invoke the svt_pcie_mphy_adapter_configuration function create_mphy_cfgs() for the mphy_adapter_cfg configuration instances to create the M-PHY configuration objects.
4. Customize the M-PHY configurations as necessary.

15.10.3 cfg Attribute Settings

A testbench would most likely modify the following members of the mphy_adapter_cfg object:

1. mpcie_signal_interface: This should be set to svt_pcie_mphy_adapter_configuration::RMMI_IF for an RMMI interface. If the testbench invokes the create_remote_mphy_adapter_cfg(), the local mphy_adapter value will be changed to TLM and the remote_mphy_adapter value will remain set to RMMI_IF. This value must not be changed by reconfigure().
2. max_tx_lane_width: Sets the maximum number of TX lanes to be supported. It determines the number of TX M-PHY instances and must not be changed when recon figuring.
3. max_rx_lane_width: Sets the maximum number of RX lanes to be supported. It determines the number of RX M-PHY instances and must not be changed when reconfiguring.
4. default_mpcie_link_tx_min_activatetime: Sets the default value of the mpcie_link_tx_min_activatetime attribute. The mpcie_link_tx_min_activatetime attribute is normally set via the MPCIE_RRAP service transaction (with execute_locally set to 1), but since it controls the time spent in Detect.Active which occurs before entry to LS_BURST and consequently before the VIP accepts MPCIE_RRAP service requests, a testbench would set this default to control that timeout. The timeout must be at least one cycle of the symbol clock and should be set to at least 6us for 20 bit RMMI and at least 12us for 40 bit RMMI interfaces.

In addition the configuration object includes variables for all M-PCIe capability attributes, any of which may be changed when reconfiguring.

15.10.4 Quick Discovery Configuration Mode

The MPCIe discovery sequence svt_pcie_mphy_adapter_discovery_sequence includes a "discovery_quick_configuration" mode that restricts the number of RRAP transactions over the wire to one, and utilizes passed-in values to assign TX configuration parameters. This done rather than reading the RX parameters of the other side of the link via RRAP transfers. This shortens the discovery sequence to tens of microseconds rather than hundreds. In the quick configuration mode, the discovery sequence only operates on the local agent. As a result, in a multiple VIP situation the discovery sequence would need to be run on each instance.

The svt_pcie_mphy_adapter_discovery_sequence has three flow control bits and several randomization values (which are fully detailed in the HTML) as follows.

Flow control:

bit	discovery_quick_configuration
bit	host_execution
bit	disable_correct_spec_rules

Random VIP configuration values:

rand bit [7:0]	c_refclk_tx_hs_g1_sync_length_control;
rand bit [7:0]	c_refclk_tx_hs_g2_sync_length_control;
rand bit [7:0]	c_refclk_tx_hs_g3_sync_length_control;
rand bit [7:0]	nc_refclk_tx_hs_g1_sync_length_control;
rand bit [7:0]	nc_refclk_tx_hs_g2_sync_length_control;
rand bit [7:0]	nc_refclk_tx_hs_g3_sync_length_control;
rand bit [3:0]	link_tx_hs_g1_prepare_length_control;
rand bit [3:0]	link_tx_hs_g2_prepare_length_control;
rand bit [3:0]	link_tx_hs_g3_prepare_length_control;
rand bit [7:0]	link_tx_hibern8_time;
rand bit [3:0]	link_tx_min_activatetime;

The mpcie_device_system_test_sequence_collection.sv::mpcie_device_system_rmmi_startup_sequence adds control to enable the execution of the svt_pcie_mphy_adapter_discovery_sequence on both the root and endpoint agents.

bit	discovery_quick_configuration

Full information on each property is in the HTML documentation.

16 Using Callbacks

16.1 Introduction

Callbacks provide a means to examine or modify transactions at various points in the protocol stack. This chapter describes their basic usage, provides some examples, and gives tips for debugging them.

Within a trans ac at ion, you can use the transaction handle to:

- Understand where in the protocol layers a particular transaction is being processed.
- Examine a particular field that was added to a transaction (for example, the Link Layer Sequence Number).
- Collect statistics and functional coverage or provide transactions to a scoreboard.
- Modify a transmitted TLP to cause a particular error to occur in the DUT and then examine the received TLP to verify that the error actually occurred as planned.
- Modify a received transaction to cause the VIP to respond abnormally to that transaction (for example, by injecting an illegal CRC value.)

16.2 How Callbacks Are Used

Callbacks occur at specific places within the VIP, where callback “hooks” have been provided by the VIP. Follow these steps to implement and use a callback:

1. Identify which callback you need to use.
2. Sub-class the appropriate data type and implement the appropriate callback method with a user-defined action each time the callback is made.
3. Create an object of this type and add it to the queue of callback objects on the appropriate VIP instance.

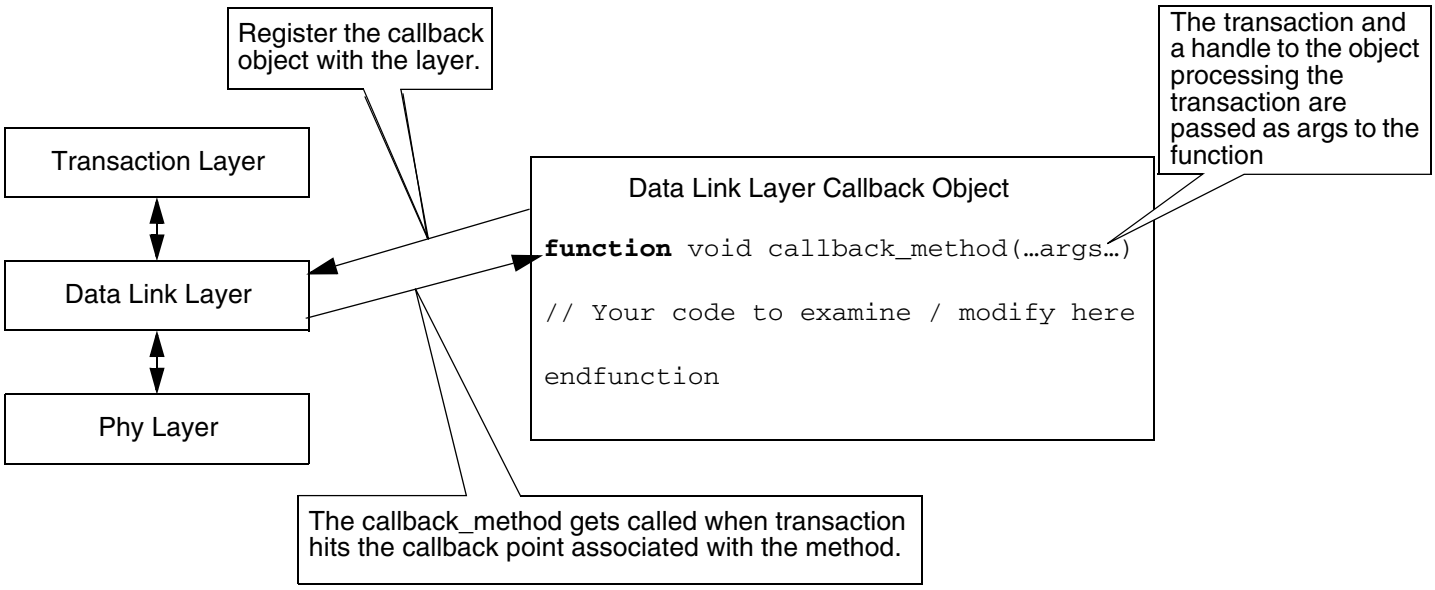
You can request a callback by specifying:

1. What VIP layer you are interested in (for example, the Data Link Layer)
2. Which direction (transmit or receive) and location within that layer you want to get the callback from. (Typically, there is more than one callback point you can specify).
3. The particular object you want registered to receive callbacks.

4. Code within a method of the above object to examine and modify the transaction.

While a test is running, whenever the callback you have implemented occurs (that is, when the transaction reaches the point in the protocol flow that causes the callback to occur), your registered callback method will be called. Once your method has finished its processing, it returns from the callback and the protocol processing continues. Figure 16-1 is a diagram of the callback process for a callback that you request in the Data Link Layer.

Figure 16-1 Callback operation for a callback in the Data Link Layer



16.2.1 Other Uses for Callbacks

Although callbacks are typically used to examine and modify the transactions as they move through the protocol stack, there are additional uses for callbacks:

- Examining if a previous callback or error injection has occurred on the transaction
- Causing an exception to occur on the associated transaction

16.2.2 Callback Hints

Callbacks are a tool that should be used conservatively:

- Limit callback modifications to one per callback.
- If multiple modifications are truly needed, use multiple callbacks.
- If multiple callbacks are used, it is important to understand the order in which the callbacks will be called.

16.2.3 When *Not* to Use a Callback

Although callbacks are a flexible mechanism, OVM analysis ports are preferred for statistics, coverage, scoreboarding, and so on. However OVM ports do not allow modification of the transaction within the caller, so use callbacks if you want to modify the transaction within the caller.

The VIP has many mechanisms for examining transactions, altering frame data, timing, and so on that might be easier to use. Here are some examples:

- When an exception already exists for the modification you have in mind, use the exception. It will not only inject the error, it will also verify that the response is correct for that error, and automatically recover. If a control already exists in the configuration or via a service request, use that control instead of a callback.
- Statistics – There already are statistics available that count many protocol items. There is no need to rewrite these.
- Delays of packets – Callbacks must be called in "zero time". They are coded as functions to enforce this.
- Scoreboards – Do not use a callback to send data to your scoreboard if there is an available analysis port at the same location.

16.3 Detailed Usage

This section describes how callbacks are used in several example testcases. The first example is a working testcase, although it is missing many features that you would normally use. The second example expands on those additional useful mechanisms.

16.3.1 A Basic Testcase

These are the main mechanisms, classes and methods used in the basic testcase example:

- Test Case Class – Each method is a different OVM phase:
  - new() constructor
  - build_phase() – Set up a test-specific configuration.
  - end_of_elaboration_phase() – Create and register the callback object
  - run_phase() – A placeholder. Nothing to do here.
- Callback Class – Each method is a specific callback from the given layer:
  - new() constructor
  - pre_tlp_framed_out_put() – Callback called just prior to framing the TLP.

Example 16-1 Code for a basic testcase

```
typedef class tx_dl_tlp_callback; // Forward declaration

// The testcase class:
class dl_tlp_basic_callback_test extends basic ;
   // Factory registration
   `ovm_component_utils( dl_tlp_basic_callback_test )

   // Callback instance:
   tx_dl_tlp_callback dl_tlp_cb;

   // Constructor:
   function new(string name="dl_tlp_basic_callback_test",ovm_component parent=null);
      super.new(name,parent);
   endfunction : new
```

```
   // Configure/build various components:
   virtual function void build_phase(ovm_phase phase);
      super.build_phase(phase);

      // Set test-specific cfg values:
      cust_cfg.root_cfg.pcie_cfg.pl_cfg.set_link_width_values(4);
      cust_cfg.endpoint_cfg.pcie_cfg.pl_cfg.set_link_width_values(4);

      //Register the cfg with the registry so that it will be picked up by the env.
      svt_config_object_db#(pcie_device_system_configuration)::set(this,  , {"env",
         ".", "cfg"}, cust_cfg);
   endfunction : build_phase

   function void end_of_elaboration_phase(ovm_phase phase);
      super.end_of_elaboration_phase(phase);

      // Create a callback object instance:
      dl_tlp_cb = tx_dl_tlp_callback::type_id::create("dl_tlp_cb");

      // Register the callback object with the DL component:
      ovm_callbacks#(svt_pcie_dl,svt_pcie_dl_callback)::add(env.endpoint.port.dl,
         dl_tlp_cb) ;
   endfunction : end_of_elaboration_phase

   task run_phase( ovm_phase phase );
      // In this case, there is not much to do in this phase
      `ovm_info(get_full_name(), "Running test dl_tlp_basic_callback_test .\n",
         OVM_LOW);
   endtask : run_phase

endclass : dl_tlp_basic_callback_test

// This is the callback class that is instantiated in the test class above:
class tx_dl_tlp_callback extends svt_pcie_dl_callback;

   // Factory registration:
   `ovm_object_utils(tx_dl_tlp_callback)

   // Error counter - static to allow it to act globally across all callback instances:
   static int error_count = 0;

   function new(string name = "tx_dl_tlp_callback");
      super.new(name);
   endfunction : new

   // pre_tlp_framed_out_put: this is the actual callback function. It will be
   // called every time a DL/TLP is ready to be framed for transmission:
   virtual function void pre_tlp_framed_out_put( svt_pcie_dl dl,
            svt_pcie_dl_tlp transaction, ref bit drop);

      `ovm_info(get_full_name(),$sformatf("\nA callback prior to transmitting TLP.
         Current TC=%0d and ECRC=0x%x, error count=%0d.\n",
         transaction.tlp.traffic_class, transaction.tlp.ecrc, error_count), OVM_LOW);
```

```
      if(error_count < 1 ) begin    // Just corrupt one TLP
         // Force a new traffic class field. Note that the traffic class (since it's in
         // the TLP header) is not directly a member of the transaction, but of the
         // tlp object encapsulated by the DL/TLP object.  The only directly
         // modifiable member of the svt_pcie_dl_tlp_transaction class is the sequence
         // number.
         transaction.tlp.traffic_class = 3;
         error_count++;
      end
   endfunction : pre_tlp_framed_out_put
endclass : tx_dl_tlp_callback
```

16.4 Advanced Topics in Callbacks

As mentioned above, there are other features you can incorporate (or, depending on what you need to do – must incorporate) in the testcase. This section discusses the concept of *exceptions*, a mechanism to request changes to a transaction that works a bit differently than directly modifying the fields of the transaction. (For example that is how the traffic class is modified in Example 16-1).

16.4.1 Exceptions – a "Delayed" Transaction Modification Request

When you make direct changes to fields in the transaction, they take effect immediately. In many cases, this works fine, but there are situations where you want to hold off on updating the transaction until all the changes are in place. A typical example of this is CRC fields. It makes little sense to calculate a CRC field if another transaction field on which the CRC depends will be changing.

There is a mechanism that allows you to schedule that CRC change (for example) when all of the transactions changes have been incorporated. This delayed transaction update is handled with an object called an *exception*.

An exception is created to request a particular change to the transaction. Once an exception is created, it is then associated with the transaction by placing it on that transaction's *exception list*. Just prior to the callback transaction being placed back into the data flow, the exception is examined and the transaction is updated based on the contents of that exception.

16.4.2 Creating an Exception

As with any object, a handle first must be declared for the exception. Its type will be based on the type of transaction with which it is associated. In the HTML class reference, follow the svt_pcie_dl class to the class attributes and find the exception class svt_pcie_dl_tlp_transaction_exception.

To create an exception, first create a handle of this type and an object associated with it:

Perhaps show the complete callback class?

```
// Exception handle and object
svt_pcie_dl_tlp_exception dl_tlp_exc = new("my_dl_tlp_exc");
```

Next, create the handle and object for the exception list:

```
// Exception list handle and object
dl_tlp_exception_list_via_callback my_dl_tlp_exc_list = new("my_dl_tlp_exc_list");
```

Now set the appropriate fields in the exception to make the request (in this case, a DL/TLP LCRC error):

```
my_dl_tlp_exc.error_kind = svt_pcie_dl_tlp_exception::CORRUPT_LCRC;

// The CORRUPT_LCRC causes the 'corrupted_data' value to be XOR'd with the
// (correctly) calculated LCRC field.
my_dl_tlp_exc.corrupted_data = 32'h0000_0001; // This will invert bit 0 of LCRC.
```

Put the exception into the exception list:

```
my_dl_tlp_exc_list.add_exception(my_dl_tlp_exc);
```

Finally, cast the exception list into the transaction:

```
$cast(transaction.tx_exception_list, my_dl_tlp_exc_list.`SVT_DATA_COPY());
```

Now every time the transaction is handled, if there is an exception in the transaction's exception list, it will be evaluated just prior to the transaction being put back into the data flow.

16.4.3 Creating a Factory Exception

In addition to the above callback exceptions, there is another type of exception: Instead of it being associated with user-specified callback code, it occurs automatically each time the callback function is called (even if there is no user-specified callback code). This allows you to generate randomized error injections.

Note that factory exceptions and callbacks are mutually exclusive: If a user modifies the transaction in the callback, the factory exception will not occur.

The code to set up a factory exception is similar to the code in "Creating an Exception" above, with a few minor differences.

First, define a derived exception list class:

```
class dl_tlp_exception_list_via_factory extends
   svt_pcie_dl_tlp_transaction_exception_list;
   svt_pcie_dl_tlp_exception xact_exc = new("xact_exc");
   function new(string name = "dl_tlp_exception_list_via_factory",
         svt_pcie_dl_tlp_exception xact_exc = null);
      super.new(name,xact_exc);
      xact_exc = this.xact_exc;
      xact_exc.NO_ERROR_wt = 0;
      xact_exc.AUTO_CORRUPT_LCRC_wt = 0;
      xact_exc.ILLEGAL_SEQ_NUM_wt = 0;
      xact_exc.DUPLICATE_SEQ_NUM_wt = 0;
      xact_exc.NULLIFIED_TLP_wt = 0;
      xact_exc.NULLIFIED_TLP_GOOD_LCRC_wt = 0;
      xact_exc.NULLIFIED_TLP_AUTO_CORRUPT_LCRC_wt = 0;
      xact_exc.MISSING_START_wt = 0;
      xact_exc.MISSING_END_wt = 0;
      xact_exc.CORRUPT_DISPARITY_wt = 0;
      xact_exc.CODE_VIOLATION_wt = 0;
      xact_exc.CORRUPT_8G_HEADER_CRC_wt = 0;
      xact_exc.CORRUPT_8G_HEADER_PARITY_wt = 0;
      xact_exc.RETAIN_LCRC_wt = 0;
      xact_exc.RECALC_LCRC_wt = 0;
```

```
      xact_exc.FORCE_LCRC_wt = 0;
      xact_exc.CORRUPT_LCRC_wt = 1; // This will cause a corrupt LCRC
      xact_exc.corrupted_data=32'hffff_ffff;  // This value will be XOR'd with the LCRC
      this.randomized_exception = xact_exc;
   endfunction : new
endclass
```

Now, in the testcase, create an exception in the list handle:

```
rand dl_tlp_exception_list_via_factory dl_tlp_exc_list;
```

In the build_phase of the testcase, create and randomize the exception list object, then set it to the per-layer component configuration database:

```
dl_tlp_exc_list = new("dl_tlp_exc_list");
dl_tlp_exc_list.randomize();
ovm_config_db#(svt_pcie_dl_tlp_exception_list)::set(this,
   "env.endpoint.port0.dl0",
   "tlp_xact_exception_list",
   dl_tlp_exc_list);
```

Whenever the transaction callback is called, (assuming there is no user callback), the transaction will be modified based on the exception above. If there is a user callback, it will be called *after* the exception code has modified the transaction and you then have the option to further modify the transaction.

16.4.4 Error Injection Using Application Layer TX Callbacks

This section shows you how to use Application Layer TX callbacks for error injection.

16.4.4.1 Basic Target Application Completion Callbacks

Whenever a target has built a completion TLP for transmission (to the TL layer, and ultimately to the remote device) the model calls a completion callback (class: svt_pcie_target_app_callback , method: pre_tx_tlp_put()). The testbench can use this callback by creating an object of a derived class and using ovm_callbacks() and add() to include that callback object in the list of callback objects. This, of course, is standard OVM. Some minor additions are available.

16.4.4.2 Methods to Modify the Transaction

There are three ways to modifity the transaction:

- Simple Modifications.
- Exceptions
- Error Injection Exceptions

16.4.4.2.1 Simple Modifications

When a callback is called, it is passed a TLP object (of type *svt_pcie_tlp*) transaction that contains various fields that can be modified in several ways; the transaction can be modified directly, for example:

```
virtual function void pre_tx_tlp_put(svt_pcie_target_app target_app,
                                     svt_pcie_tlp transaction,
                                     ref bit drop);
    transaction.ep = 1;// Set the Poison Bit in the TLP
endfunction  // pre_tx_tlp_put()
```

In the example, you simply set the value of the TLP poison bit (ep) to 1. The callback will hand the TLP back to the protocol stack , and that TLP's Poison bit will be set.

Note that there is not an explicit Error Injection code being added. However , an implicit virtual EI code *is* added: *USER_MODIFIED_TLP*. This code does two main things:

1. Marks the TLP as having been modified, so later callbacks can know.
2. Keeps any later "automatic" error injections from occurring.

Note that although you should not need to add USER_MODIFIED_TLP manually, it will be placed on the TLP automatically when the TLP has been modified in a callback. (As you will see below, there is an analogous EI code *MALFORMED_TLP* that will be added to a TLP that is explicitly Malformed.)

16.4.4.2.2 Exceptions

Another way the transaction can be modified is via an *exception* – essentially a delayed update to that transaction, for example:

```
virtual function void pre_tx_tlp_put(svt_pcie_target_app target_app,
                                     svt_pcie_tlp transaction,
                                     ref bit drop);
  svt_pcie_tlp_exception_list my_tlp_exc_list;    // Exception list
  svt_pcie_tlp_transaction_exception my_tlp_exc;  // Exception obj
  my_tlp_exc_list = new("my_tlp_exc_list");
  my_tlp_exc = new("my_tlp_exc");
  my_tlp_exc.error_kind =    // Set the exc type
      svt_pcie_tlp_transaction_exception::CORRUPT_ECRC;
  my_tlp_exc.corrupted_data = 32'hffff_ffff; // XOR with ECRC
  my_tlp_exc_list.add_exception(my_tlp_exc);  // Add exc to list
  $cast(transaction.exception_list,
        my_tlp_exc_list.`SVT_DATA_COPY() );   // Add exc list to TLP
endfunction  // pre_tx_tlp_put()
```

In the previous example, the ECRC will be corrupted (see the documentation for details, but essentially the ECRC will be calculated, and the XOR'd with the provided *corrupted_data* (i.e. 32'hffff_ffff). However, this does not occur immediately. Since this is a calculation that is done on the entire TLP, you need to ensure that all the fields that you may intend to change have been changed prior to the ECRC calculation. Using an exception allows you to do this – an exception is applied to the TLP until that TLP is actually getting *pack*'ed, just prior to its being handed back to the protocol stack.

As previous stated, once the exception is applied and the TLP modified, it will be tagged with the USER_MODIFIED_TLP EI code.

Error Injection Exceptions

In addition to modifying the TLP contents directly, an exception can be used to propagate an *Error Injection* (EI) to the layers below it. For example, although the Application layer isn't able to modify the LCRC of a TLP to be transmitted, you can request via an EI that the LCRC be corrupted:

```
virtual function void pre_tx_tlp_put(svt_pcie_target_app target_app,
                                     svt_pcie_tlp transaction,
                                     ref bit drop);
  svt_pcie_tlp_exception_list my_tlp_exc_list;    // Exception list
  svt_pcie_tlp_transaction_exception my_tlp_exc;  // Exception obj
  my_tlp_exc_list = new("my_tlp_exc_list");
  my_tlp_exc = new("my_tlp_exc");
  my_tlp_exc.error_kind =    // Set the exc type
      svt_pcie_tlp_transaction_exception::AUTO_TX_CORRUPT_LCRC;
  my_tlp_exc_list.add_exception(my_tlp_exc);  // Add exc to list
  $cast(transaction.exception_list,
        my_tlp_exc_list.`SVT_DATA_COPY() );   // Add exc list to TLP
endfunction  // pre_tx_tlp_put()
```

The previous example provides the exception, which will be 'translated' to an Error Injection to be handed down the protocol stack. Once it goes into the DataLink Layer (where the LCRC is calculated), then the LCRC is corrupted (as instructed above).

In addition, since the prefix to the exception is ***AUTO_***, this implies that not only will the error injection occur, but that the VIP will automatically do the following:

- Determine if the LCRC actually was recognized correctly by the remote device.
- Recover from the error, and retransmit any required TLPs.
- Suppress any error messages associated with the above.

Note: once a TLP has an EI Exception attached to it, you should not attempt to modify in any other way. Error injections work due to a controlled injection of the error – if multiple errors are attempted simultaneously, the EI will generally will fail in a typically difficult-to-debug way!

Unlike the previous examples note that since the user has *not* changed the TLP (adding the EI request doesn't count as a change to the actual TLP header/data), only the actual EI is attached; the USER_MODIFIED_TLP is **not.**

16.4.4.3 Malformed and Nullified TLPs

If you intend to create a TLP that is *Malformed* (see PCIe spec for details), it is up to the testbench to inform the VIP of that fact. This is done simply via the transactions *set_malformed(value)* method. For example:

```
virtual function void pre_tx_tlp_put(svt_pcie_target_app target_app,
                                     svt_pcie_tlp transaction,
                                     ref bit drop);
 … various manipulations on TLP …// Corrupt the TLP
   // Set TLP to be treated as Malformed
   transaction.set_malformed(1);
endfunction  // pre_tx_tlp_put()
```

Note that if an Error Injection is set on a TLP marked as Malformed, that Error Injection will be canceled (for reasons given above – the requirement of a controlled Error Injection has been broken.) Note also that in the TL layer, credits are not counted for a Malformed transaction – as it is assumed that (being Malformed) the receiver will neither recognize nor count credits for the associated transaction.

Recall previously that we added USER_MODIFIED_TLP as a virtual EI code to TLPs that a user has modified. In this (Malformed) case, the virtual EI code MALFORMED_TLP will be added instead. It has the same basic effect to remind lower layers that this TLP has been modified; in addition it also tells those same lower layers (including callbacks in those layers) that this has been modified so as to be Malformed.

If the transaction did not already have an error injected, then the TL would have consumed credits as appropriate. If the TLP is malformed or nullified via callback (using exceptions), then credit consumption by the VIP cannot be changed. Similarly, a TL layer exception cannot be canceled or changed via this callback.

16.4.5 A More Comprehensive Example

In addition to the Test Case and Callback classes used in Example 16-1, three other classes are added in Example 16-2:

- Exception Classes – Each transaction type has its own exception class, which contains objects of its exception class:
  - new() constructor
  - Various per-transaction fields to set up the exception
- Exception List Classes – Each transaction has a class for its exception list
- Error Handler – Extended from ovm_report_catcher. It has several important methods:
  - new() constructor
  - pattern_match() – Matches the actual error string with the “expected” string
  - catch() – Called upon an error message being potentially displayed. You can filter with this.

Example 16-2 Code for a more comprehensive testcase

```
// Forward declarations
typedef class tlp_exc_list_via_callback;
typedef class dl_tlp_exception_list_via_callback;
typedef class dl_tlp_err_catcher;
typedef class tx_dl_tlp_callback;

// The testcase class:
class dl_tlp_example_callback extends basic ;
   // Factory registration:
   `ovm_component_utils( dl_tlp_example_callback )
   // Callback instance:
   tx_dl_tlp_callback dl_tlp_cb;
   // Exception list class instance:
   dl_tlp_exception_list_via_callback dl_tlp_exc_list;
   // Error catcher:
   dl_tlp_err_catcher err_catcher;
   // Constructor "new":
   function new(string name="tlp_exception_via_callback", ovm_component parent=null);
      super.new(name,parent);
   endfunction : new

   // build_phase: To build various components of class:
   virtual function void build_phase(ovm_phase phase);
      super.build_phase(phase);
      // Load test specific cfg values:
      cust_cfg.root_cfg.pcie_cfg.pl_cfg.set_link_width_values(4);
      cust_cfg.endpoint_cfg.pcie_cfg.pl_cfg.set_link_width_values(4);
```

```
      // Register config with the registry so that it will be picked up by the env:
      svt_config_object_db#(pcie_device_system_configuration)::set(this,  ,
         {"env", ".", "cfg"}, cust_cfg);

      // Create dl_tlp_exc_list:
      dl_tlp_exc_list = new("dl_tlp_exc_list");

      // Pass the constrained exception list to Data Link Layer:
      ovm_config_db#(svt_pcie_dl_tlp_exception_list)::set(this,
         "env.endpoint.port0.dl0",
         "dl_tlp_xact_exception_list",
         dl_tlp_exc_list);

      // Create error report catcher object and register it:
      err_catcher = new();
      ovm_report_cb::add(null, err_catcher);
   endfunction : build_phase

   function void end_of_elaboration_phase(ovm_phase phase);
      super.end_of_elaboration_phase(phase);
      // Create the callback object:
      dl_tlp_cb = new("dl_tlp_cb");
      dl_tlp_cb.randomize();
      // Register the callback object with the DL component:
      ovm_callbacks#(svt_pcie_dl,svt_pcie_dl_callback)::add(env.endpoint.port.dl,
         dl_tlp_cb);
      `ovm_info(get_full_name(), "Exiting...", OVM_HIGH);
   endfunction : end_of_elaboration_phase

   task run_phase( ovm_phase phase );
      `ovm_info(get_full_name(), "Running test dl_tlp_example_callback .\n",
         OVM_LOW);
   endtask : run_phase
endclass : dl_tlp_example_callback

// This is the callback class. It is instantiated in the test class above:
class tx_dl_tlp_callback extends svt_pcie_dl_callback;
   // Factory registration:
   `ovm_object_utils(tx_dl_tlp_callback)
   //`svt_ovm_object_utils(tx_dl_tlp_callback)

   // There are two basic ways to modify a value in a transaction (which is
   // really what we are aiming to do with the callback):
   //   1. Explicitly modify the value (e.g. transaction.<field> = <value> )
   //   2. Use an exception to cause a modification (typically for 'calculated' fields
   //      such as CRC).
   // To use an exception, you need two things:
   //   1. The "exception" - one per modification to the transaction (note that the
   //      type [class] of this object is specific to the transaction).
   //   2. The "exception list" - holds the exception(s) (again, the
   //      exception-list class is specific to the transaction.)

   // Exceptions - Use one each for the TLP transaction and the DL/TLP (which is
```

```
   // essentially a TLP transaction with a sequence number and LCRC added):
   // For a TLP transaction (within the DL/TLP transaction):
   svt_pcie_tlp_exception my_tlp_exc = new("my_tlp_exc");
   // For DL/TLP transaction:
   svt_pcie_dl_tlp_exception my_dl_tlp_exc = new("my_dl_tlp_exc");

   // Exception lists - Use one per transaction type that you intend to modify:
   tlp_exc_list_via_callback my_tlp_exc_list = new("my_tlp_exc_list");
   dl_tlp_exception_list_via_callback my_dl_tlp_exc_list = new("my_dl_tlp_exc_list");

   // Error counter:
   static int error_count = 0;
   rand   bit do_lcrc_err;

  // Constructor:
   function new(string name = "tx_dl_tlp_callback");
      super.new(name);
   endfunction : new

   // pre_tlp_framed_out_put: this is the actual callback function. It will be
   // called every time a DL/TLP is ready to be framed for transmission:
   virtual function void pre_tlp_framed_out_put( svt_pcie_dl dl,
         svt_pcie_dl_tlp transaction,
         ref bit drop);
      `ovm_info(get_full_name(),
         $sformatf("\nA callback prior to transmitting TLP. Current TC=%0d and
            ECRC = 0x%x, error count=%0d.\n",
            transaction.tlp.traffic_class,
            transaction.tlp.ecrc, error_count),
            OVM_LOW);
      if(error_count < 1 ) begin    // Just corrupt one TLP
         if (do_lcrc_err) begin   // inject an LCRC err
            // The AUTO_TX_FORCE_LCRC causes the 'corrupted_data' value to
            // be forced into the LCRC field:
            my_dl_tlp_exc.corrupted_data = 32'hbaad_baad; // Forced LCRC value
            my_dl_tlp_exc.error_kind = svt_pcie_dl_tlp_exception::AUTO_TX_FORCE_LCRC;
         end
         else begin
           my_dl_tlp_exc.error_kind=svt_pcie_dl_tlp_exception::AUTO_TX_ILLEGAL_SEQ_NUM;
         end
         // Put the exc into the list, then copy/cast the list into the transaction:
         my_dl_tlp_exc_list.add_exception(my_dl_tlp_exc);
         `svt_note("pre_tlp_framed_out_put",
            $sformatf(" Attaching DL/TLP exception list .\n."));
         $cast(transaction.exception_list, my_dl_tlp_exc_list.`SVT_DATA_COPY());
         error_count++;
      end
   endfunction
endclass : tx_dl_tlp_callback

///////// Various helper classes: exception lists, error catcher ///////////////
// tlp_exc_list_via_callback: see above for details of exception lists.
// This is for the TLP transaction encapsulated in the DL/TLP transaction
class tlp_exc_list_via_callback extends svt_pcie_tlp_transaction_exception_list;
```

```
      // The default exception, in case we have no others defined:
      svt_pcie_tlp_exception xact_exc = new("xact_exc");

      // Constructor:
      function new(string name = "tlp_exc_list_via_callback",
         svt_pcie_tlp_exception xact_exc = null);
         super.new(name,xact_exc);
         xact_exc = this.xact_exc;
         xact_exc.NO_ERROR_wt = 1;              // Implies no errors
         xact_exc.set_constraint_weights(0);
         this.randomized_exception = xact_exc; // insert this exception into our list
      endfunction : new
   endclass : tlp_exc_list_via_callback

   // dl_tlp_exception_list_via_callback: see above for details of exception lists.
   // This is for actual the DL/TLP transaction.
   class dl_tlp_exception_list_via_callback extends svt_pcie_dl_tlp_exception_list;
      // The default exception - in case we have no others defined.
      svt_pcie_dl_tlp_exception xact_exc = new("xact_exc");
      // Constructor:
      function new(string name = "dl_tlp_exception_list_via_callback",
         svt_pcie_dl_tlp_exception xact_exc = null);
         super.new(name,xact_exc);
         xact_exc = this.xact_exc;
         xact_exc.NO_ERROR_wt = 1;
         xact_exc.set_constraint_weights(0);
         this.randomized_exception = xact_exc;
      endfunction : new
   endclass : dl_tlp_exception_list_via_callback

   // Class to demote OVM_WARNING and OVM_ERROR messages:
   class dl_tlp_err_catcher extends ovm_report_catcher;
      // Constructor:
      function new(string name="error catcher");
         super.new(name);
      endfunction

      // catch(): This function is where we handle the actual OVM WARNING/ERROR
      // messages; it suppresses the errors that would occur due to the corrupted
      // CRC, etc.
      virtual function action_e catch();
         // Error catcher required if CORRUPT_TC is injected:
         if(get_severity() == OVM_ERROR) begin
            if ((ovm_is_match("*Received TLP with invalid seq num*", get_message())))
               begin
                  set_severity(OVM_INFO);
               end
         end
         else if (get_severity() == OVM_WARNING) begin
            if ((ovm_is_match("*Received TLP with bad LCRC*", get_message()))) begin
               set_severity(OVM_INFO);
            end
         end
         return THROW;
```

```
   endfunction : catch
endclass : dl_tlp_err_catcher
```

16.5 SVT VIP Callbacks Reference

The callbacks that currently are supported and their arguments are listed in Table 16-1.

Callbacks and their methods for each layer are listed in Table 16-2.

Table 16-1 Supported PCIe callbacks

Function Name	Arguments (type and name)	I/O	Values
pre_tlp_framed_out_put DL Layer Called just prior to framing a TLP for transmission to the Phy Layer.	svt_pcie_dl dl	I	The component object of the calling layer.
	svt_pcie_dl_tlp transaction	I	The DL/TLP transaction to be examined/modified. Note that it also contains a TLP object.
	drop	ref	When set, the transaction is dropped prior to transmission. NOTE: Currently not implemented
post_tlp_framed_in_get DL Layer Called just after reception from the Phy Layer.	svt_pcie_dl dl	I	The component object of the calling layer.
	svt_pcie_dl_tlp transaction	I	The DL/TLP transaction to be examined/modified. Note that it also contains a TLP object.
	bit drop	ref	When set, the transaction is dropped prior to transmission. NOTE: Currently not implemented
tx_dllp_started DL Layer Called just prior to transmitting a DLLP to the Phy Layer.	svt_pcie_dl dl	I	The component object of the calling layer.
	svt_pcie_dllp transaction	I	The DLLP transaction to be examined/modified.
	bit drop	ref	When set, the transaction is dropped prior to transmission. NOTE: Currently not implemented
rx_dllp_started DL Layer Called just after reception from the Phy Layer.	svt_pcie_dl dl	I	The component object of the calling layer.
	svt_pcie_dllp transaction	I	The dllp transaction t be examined/modified.
	bit drop	ref	When set, the transaction is dropped prior to transmission. NOTE: Currently not implemented

Table 16-1 Supported PCIe callbacks (Continued)

pre_tx_tlp_put Target Application Called by the component just after scheduling a TLP transaction for transmission on the link, just prior to framing.	svt_pcie_target_app target_app	I	The component object of the calling layer.
	svt_pcie_tlp transaction	I	The transaction to be examined/modified.
	bit drop	ref	When set, the transaction is dropped prior to framing and is not transmitted.
post_rx_tlp_get Target Application Called by the component after receiving a TLP transaction from the link.	svt_pcie_target_app target_app	I	The component object of the calling layer.
	svt_pcie_tlp transaction	I	The transaction to be examined/modified.
	bit drop	ref	When set, the transaction is dropped without any further processing.
transaction_ended Driver Application Called by the component when the transaction is complete	svt_pcie_driver_app driver	I	The component object issuing the callback
	svt_pcie_driver_app_transaction transaction	I	The transaction to be examined/modified.
tx_ts_os_started Phy Layer Called by the component after building a TS OS transaction.	svt_pcie_pl pl	I	The component object issuing this callback.
	svt_pcie_os transaction	I	The transaction to be examined/modified.
tx_os_started Phy Layer Called by the component after building a non-TS OS Transaction	svt_pcie_pl pl	I	The component object issuing this callback.
	svt_pcie_os transaction	I	The transaction to be examined/modified.
pre_symbol_out_put Phy Layer Called by the component at every symbol time after gathering all the symbols to be transmitted on the link	svt_pcie_pl pl	I	The component object issuing this callback.
	svt_pcie_symbol symbols[]	I	The array of symbols to be examined/modified.

Table 16-1 Supported PCIe callbacks (Continued)

symbol_stripe_ended Phy Layer Called by the component when the symbol stripe has been completed	svt_pcie_pl pl	I	The component object issuing this callback.
	svt_pcie_symbol_stripe symbol_stripe	I	The symbol stripe object to be examined.

Table 16-2 Callback classes and methods by layer

Layer	Callback Class	Method	Description
Driver App	svt_pcie_driver_app_callback	transaction_ended	Called by the component once the transaction is completed by the link partner. Completion data returned by the link partner is now available in the payload.
Target App	svt_pcie_target_app_callback	post_rx_tlp_get	Called by the component after recognizing a TLP transaction received immediately from the link.
		pre_tx_tlp_put	Called by the component after scheduling a TLP transaction for transmission on the link, just prior to framing.
TL Layer	svt_pcie_tl_callback	post_seq_item_get	Called by the component after pulling a TLP out of its TLP input, but before acting on the TLP.
		pre_tlp_out_put	Called by the component once the TLP is completely received and prior to putting the TLP on the rx port.
DL Layer	svt_pcie_dl_callback	post_tlp_framed_in_get	Called by the component after recognizing a TLP transaction received immediately from the link.
		pre_tllp_framed_out_put	Called by the component after scheduling a TLP transaction for transmission on the link, just prior to framing.
		rx_dllp_started	Called by the component after receiving user DLLP transaction from an input port.
		tx_dllp_started	Called by the component after constructing a DLLP transaction just prior to its further processing.

Table 16-2 Callback classes and methods by layer (Continued)

PL Layer	svt_pcie_pl_callback	pre_symbol_out_put	Called by the component at every symbol time after gathering all the symbols to be transmitted on the link. Last chance to corrupt any symbol before it goes on the link.
		tx_os_started	Called by the component after building an OS Transaction. The callback gives the user a chance to attach an exception list to the OS transaction prior to its transmission on the link.
		tx_ts_os_started	Called by the component after building a TS OS transaction. The callback gives the user a chance to attach an exception list to the TS OS transaction prior to its transmission on the link.
		symbol_stripe_ended	Called by the component when the symbol stripe has been completed. The symbol_stripe object contains the information necessary to log symbol data. Writes to the symbol_stripe object are ignore.

16.6 Transaction Layer Callbacks and Exceptions

The Transaction Layer provides a callback class, svt_pcie_tl_callback. This callback class is used to apply exceptions that can modify or observe data.

```
class svt_pcie_tl_callback extends svt_callback;
   extern function void new ( string name = "svt_pcie_tl_callback" );
   extern virtual function void pre_tlp_out_put ( svt_pcie_tl tl , svt_pcie_tlp tlp ,
      ref bit drop );
   endfunction;
   virtual function void post_seq_item_get(svt_pcie_tl tl, svt_pcie_tlp tlp,
      ref bit drop);
   endfunction
endclass
```

Table 16-3 Transaction Layer Callbacks

Callback	VIP Direction	Behavior
post_tlp_framed_in_get	TX	Callback issued by the component after pulling a TLP transaction out of its TLP input, but before acting on the TLP in any way
pre_tlp_framed_out_put	RX	Callback issued by the component after a TLP transaction is received completely and prior to transmitting the received TLP to the RX port

16.6.1 Transaction Layer Exceptions

Transaction Layer exceptions can be applied using the svt_pcie_tl_callback class. The callback methods in this class support the svt_pcie_tlp transaction class. The svt_pcie_tlp transaction class includes an exception list, svt_pcie_tlp_exception_list, which contains an exception, svt_pcie_tlp_exception. The svt_pcie_tlp_exception exception is used to be specify the type of exception to be applied.

By default the exception list is null, thus indicating no exception is to be applied. Exceptions are added by adding a handle to an exception list that is not null. For more information on the available exception types, refer to the HTML class reference of the exception class, svt_pcie_tlp_exception.

16.7 Datalink Layer Callbacks and Exceptions

The Datalink Layer provides a callback class, svt_pcie_dl_callback. This callback class is used to apply exceptions that can modify or observe data.

```
class svt_pcie_dl_callback extends svt_callback;
   extern function new(string name = "svt_pcie_dl_callback");
   extern virtual function void pre_tlp_framed_out_put(svt_pcie_dl dl,
                                       svt_pcie_dl_tlp transaction, ref bit drop);
   extern virtual function void post_tlp_framed_in_get(svt_pcie_dl dl,
                                       svt_pcie_dl_tlp transaction, ref bit drop);
   extern virtual function void tx_dllp_started(svt_pcie_dl dl,
                                       svt_pcie_dllp transaction, ref bit drop);
   extern virtual function void rx_dllp_started(svt_pcie_dl dl,
                                       svt_pcie_dllp transaction, ref bit drop);
endclass
```

Table 16-4 Datalink Layer Callbacks

Callback	VIP Direction	Behavior
post_tlp_framed_in_get	RX	Callback issued by the component after recognizing a TLP transaction received immediately from a link
pre_tlp_framed_out_put	TX	Callback issued by the component after scheduling a TLP transaction for transmission on the link, just prior to framing
rx_dllp_started	RX	Callback issued by the component after receiving User DLLP transaction from an input port
tx_dllp_started	TX	Callback issued by the component after building a DLLP transaction just prior to its further processing

16.7.1 Datalink Layer Exceptions

Datalink Layer exceptions can be applied using the svt_pcie_dl_callback class. Each callback method in this class supports a particular transaction class. Each transaction class includes an exception list which contains an exception that is used to be specify the type of exception to be applied.

By default the exception list is null, thus indicating no exception is to be applied. Exceptions are added by adding a handle to an exception list that is not null.

For example, the post_tlp_framed_in_get callback method supports the svt_pcie_dl_tlp transaction class. The svt_pcie_dl_tlp transaction class includes an exception list, svt_pcie_dl_tlp_exception_list, that contains an exception, svt_pcie_dl_tlp_exception. The svt_pcie_dl_tlp_exception exception is used to specify the exception type. Tables 16-5 highlights the relationships between the DL callback methods and their associated classes.

For more information on the available exception types, refer to the HTML class reference of the exception classes in Tables 16-5.

Table 16-5 Datalink Layer Callback Method and Associated Classes

Callback	Transaction Class	Exception List Class	Exception Class
post_tlp_framed_in_get	svt_pcie_dl_tlp	svt_pcie_dl_tlp_exception_list	svt_pcie_dl_tlp_exception
pre_tlp_framed_out_put	svt_pcie_dl_tlp	svt_pcie_dl_tlp_exception_list	svt_pcie_dl_tlp_exception
rx_dllp_started	svt_pcie_dllp	svt_pcie_dllp_exception_list	svt_pcie_dllp_exception
tx_dllp_started	svt_pcie_dllp	svt_pcie_dllp_exception_list	svt_pcie_dllp_exception

16.8 Physical Layer Callbacks and Exceptions

The Physical Layer provides a callback class, svt_pcie_pl_callback. This callback class is used to apply exceptions that can modify or observe data.

```
class svt_pcie_pl_callback extends svt_xactor_callback;
   extern function new(string name = "svt_pcie_pl_callback");
   extern virtual function void tx_ts_os_started(svt_pcie_pl pl,
                                                  svt_pcie_os transaction);
   extern virtual function void tx_os_started(svt_pcie_pl pl, svt_pcie_os transaction);
   extern virtual function void pre_symbol_out_put(svt_pcie_pl pl,
                                                  svt_pcie_symbol symbols[]);
   extern virtual function void symbol_stripe_ended(svt_pcie_pl pl,
                                        svt_pcie_symbol_stripe symbol_stripe);
endclass
```

Table 16-6 Physical Layer Callbacks

Callback	VIP Direction	Behavior
pre_symbol_out_put	TX	Callback issued by the PL at every symbol time after gathering all the symbols to be transmitted on the PCIe link. This is the last chance the user has to corrupt any symbol before it goes on the link. Note, for muli-lane configures the symbols are striped per lane into symbols[].
tx_os_started	TX	Callback issued by the component after building an OS Transaction. The callback gives user a chance to attach exception list to the OS transaction prior to its transmission on the link
tx_ts_os_started	TX	Callback issued by the component after building a TS OS Transaction. The callback gives user a chance to attach exception list to the TS OS transaction prior to its transmission on the link

Table 16-6 Physical Layer Callbacks

Callback	VIP Direction	Behavior
symbol_stripe_ended	TX	Called by the component when the symbol stripe has been completed. The symbol_stripe object contains the information necessary to log symbol data. Writes to the symbol_stripe object are ignore.

16.8.1 Physical Layer Exceptions

Physical Layer exceptions can be applied using the svt_pcie_pl_callback class. Each callback method in this class supports a particular transaction class. Each transaction class includes an exception list which contains an exception that is used to be specify the type of exception to be applied.

By default the exception list is null, thus indicating no exception is to be applied. Exceptions are added by adding a handle to an exception list that is not null.

For example, the pre_symbol_out callback method supports the svt_pcie_symbol transaction class. The svt_pcie_symbol transaction class includes an exception list, svt_pcie_symbol_exception_list, which contains an exception, svt_pcie_symbol_exception. The svt_pcie_symbol_exception exception is used to specify the exception type. Tables 16-7 highlights the relationships between the PL callback methods and their associated classes.

For more information on the available exception types, refer to the HTML class reference of the exception classes in Tables 16-7.

Table 16-7 Physical Layer Callback Methods and Associated Classes

Callback	Transaction Class	Exception List Class	Exception Class
pre_symbol_out_put	svt_pcie_symbol	svt_pcie_symbol_exception_list	svt_pcie_symbol_exception
tx_os_started	svt_pcie_os	svt_pcie_os_exception_list	svt_pcie_os_exception
tx_ts_os_started	svt_pcie_os	svt_pcie_os_exception_list	svt_pcie_os_exception

16.9 Controlling Completion Timing and Size Using Callbacks

There are two ways to control completion timing:

1. By specifying a delay for the current completion.
2. By specifying a delay for the next completion.

Control of completion size is applied only to the next completion, and not the current completion.

16.9.1 Controlling Delay for the Current Completion

To specify a delay for the current completion, use the completion_delay_in_ns attribute in the svt_pcie_tlp class.

This is used only in associated with the following callback:

- svt_pcie_target_app_callback::pre_tx_tlp_put

It is ignored in all other use.

When used at the specified callback this attribute controls the delay between the time the callback occurs (at creation of the completion) and the time the completion is sent from the target application to the transaction layer. It does not incorporate the delay through the layers or the time required to actually transmit the completion, which may be significant for large memory read completions on narrow links.

This value is ignored if:

- This TLP is not a completion.
- The value is not set during the specified callbacks.
- The value is negative.

16.9.2 Controlling Delay for the Next Completion

To specify a delay for the next completion, use the next_completion_delay_in_ns attribute in the svt_pcie_tlp class . It specifies the delay in ns to the creation of a completion that follows the current TLP.

This is used only in association with the following callbacks:

- svt_pcie_target_app_callback::post_rx_tlp_get
- svt_pcie_target_app_callback::pre_tx_tlp_put

It is ignored in all other use.

When used at the specified callbacks it controls the delay between:

- The execution of the callback, and
- The target application creating the subsequent completion

It does not incorporate the delay through the layers or the time required to actually transmit the completion, which may be significant for large memory read completions on narrow links.

The delay is used in two cases:

1. A non-posted request received by the vip: the value specifies the delay to the first completion transmitted by the vip in response.
2. A memory read completion transmitted by the vip, if there will be at least one subsequent completion for the same request: the value specifies the delay to the next completion transmitted by the VIP.

To control completion timing using this value, set it in either the pre_tx_tlp_put callback or the post_rx_tlp_get callback in the svt_pcie_target_app_callback class.

This value is ignored if:

- This TLP is neither a non-posted request received by the vip nor a memory read completion transmitted by the VIP.
- This TLP is the last completion of a memory read request.
- The value is not set during the specified callbacks.
- The value is negative.

16.9.3 Controlling Size for the Next Completion

You can control the size of completions by using callbacks and the class member next_completion_size_in_rcb. It specifies the size in RCB units of the completion that follows this TLP.

It is used only in associated with the following callbacks:

- svt_pcie_target_app_callback::post_rx_tlp_get
- svt_pcie_target_app_callback::pre_tx_tlp_put

The next_completion_size_in_rcb attribute is ignored in all other uses. It is used in two cases:

- A memory read request received by the vip: it specifies the size in RCB of the first completion transmitted by the vip in response.
- A memory read completion transmitted by the vip, if there will be at least one subsequent completion for the same request: it specifies the size of the next completion transmitted by the VIP.

If the value is set to 1, the VIP will transmit only enough data to reach the first RCB. In that case, the length of the first completion could be as small as one dword.

To control completion size using this value, set it in either the pre_tx_tlp_put callback or the post_rx_tlp_get callback in the svt_pcie_target_app_callback class.

This value is ignored if:

- This TLP is not a memory read request received by the vip or a completion transmitted by the VIP in response to a memory read request.
- This TLP is the last completion of a read request.
- The value is not set during the specified callbacks.

This value is not intended to produce error scenarios and will also be ignored if the value is less than 1. For large values the completion size will be truncated to either the max payload size or the max read completion data size, whichever is smaller.

If the specified size is larger than the number of dwords remaining to finish the request, the next completion will include all remaining data (unless limited by the max payload size or max read completion data size).

17 Partition Compile and Precompiled IP

In design verification, every compilation and recompilation of the design and testbench contributes to the overall project schedule. A typical System-on-Chip (SoC) design may have one or more VIPs where changes are performed in the design or the testbench outside of VIPs. During the development cycle and the debug cycle, the complete design along with the VIP is recompiled, which leads to increased compilation time.

Verification Compiler offers the integration of VIPs with Partition Compile (PC) and Precompiled IP (PIP) flows. This integration offers a scalable compilation strategy that minimizes the VIP recompilations, and thus improves the compilation performance. This further reduces the overall time to market of a product during the development cycle and improves the productivity during the debug cycle.

The PC and PIP features in Verification Compiler provide the following solutions to optimize the compilation performance:

- The partition compile flow creates partitions (of module, testbench or package) for the design and recompiles only the changed or the modified partitions during the incremental compile.
- The PIP flow allows you to compile a self-contained functional unit separately in a design and a testbench. A shared object file and a debug database are generated for a self-contained functional unit. All of the generated shared object files and debug databases are integrated in the integration step to generate a simv executable. Only the required PIPs are recompiled with incremental changes in design or testbench.

For more information on the partition compile and Precompiled IP flows, see the VCS/VCS MX LCA Features Guide.

17.1 Implementing Partition Compile in Testbench

To effectively utilize the partition compile technology, PCIe VIP internally creates multiple sub-packages under existing top package, `svt_pcie_ovm_pkg`. The created multiple sub packages are compiled in parallel to improve the overall VIP compilation time. The compile time improvement depends on the number of cores selected for parallel execution. PCIe VIP sub-packages are mainly based on PCIe protocol layers like Transport, Data Link and Physical Layers along with separate package for coverage at each layer.

The partition compile feature is disabled by default. To enable the partition compile mode, you must perform the following functions:

- Include the file, `import_pcie_svt_ovm_pkgs.svi` to import all individual sub-packages.

- Set the compile macro, `SVT_PCIE_OPTIMIZED_COMPILE` in the compilation command.

17.1.1 High Level Architecture

The PCIe VIP creates multiple sub-packages for parallel compilation under existing PCIe OVM Package, `svt_pcie_ovm_pkg`. Following is the list of multiple sub-packages required to enable parallel compilation:

- `svt_pcie_common_ovm_pkg`
- `svt_pcie_tl_ovm_pkg` (TL Layer Tx/Rx)
- `svt_pcie_dl_ovm_pkg` (DL Layer TX/Rx)
- `svt_pcie_pl_pkg_common_ovm_pkg` (PL Layer Common)
- `svt_pcie_pl_ovm_pkg` (PL Layer Tx/Rx)
- `svt_pcie_tl_coverage_ovm_pkg` (TL Coverage)
- `svt_pcie_dl_coverage_ovm_pkg` (DL Coverage)
- `svt_pcie_pl_coverage_ovm_pkg` (PL Coverage)
- `svt_pcie_sequence_sequencer_collection_ovm_pkg` (Sequence Collection)

17.1.2 Guidelines for Partition Compile Usages

PCIe VIP support for partition compile feature is mostly backward compatible with the default VIP flow. Following are the scenarios where changes may be required to support partition compile flow:

1. **Avoid explicit file import from package.**

   A conflict can occur in scenarios, where class data type is referenced along with the existing PCIe package name using scope resolution. For example,

   ```
   import svt_pcie_ovm_pkg::svt_pcie_configuration;
   ```

   This existing class may have been moved to a new sub-package to support multiple partitions. Therefore, it is required to either change the package of explicit import of class

   ```
   `ifdef SVT_PCIE_OPTIMIZED_COMPILE
   import svt_pcie_common_ovm_pkg::svt_pcie_configuration;
   `elseif
   import svt_pcie_ovm_pkg::svt_pcie_configuration
   `endif
   ```

   Or, export the class file in the top package itself

   ```
   export svt_pcie_common_ovm_pkg::svt_pcie_configuration;
   ```

2. **Avoid accessing data class members using package name.**

   Example:
   `svt_pcie_ovm_pkg::svt_pcie_target_app_configuration::UNINIT_MEM_READ_RESP_BAAD;`

   This can cause an error because files might have been moved to a new package. If such access have been used at multiple places, then export the file in PCIe VIP top package, `svt_pcie_ovm_pkg`.

   Example:

   ```
   export svt_pcie_common_ovm_pkg::svt_pcie_target_app_configuration;
   ```

17.2 Use Model

You can use the three new simulation targets in the Makefiles of the VIP OVM examples to run the examples in the partition compile or the precompiled IP flow. In addition, Makefiles allow you to run the examples in back-to-back VIP configurations. The VIP OVM examples are located in the following directory:

$VC_HOME/examples/vl/vip/svt/vip_title/sverilog

For example,

/project/vc_install/examples/vl/vip/svt/pcie_svt/sverilog

Each VIP OVM example includes a configuration file called as the pc.optcfg file. This configuration file contains predefined partitions or precompiled IPs for the SystemVerilog packages that are used by VIP. The predefined partitions are created using the following heuristics:

- Separate partitions are created for packages that are common to multiple VIPs.
- The VIP level partitions are defined in a way that all of the partitions are compiled in the similar duration of time. This enables the optimal use of parallel compile with the -fastpartcomp option.

You can modify the pc.optcfg configuration file to include additional testbench or DUT level partitions.

17.2.1 Parallel Partition Compile

You must perform these steps to enable the parallel partition compilation flow with SVT PCIe VIP:

1. Provide the VCS compile argument:

   ```
   +define+SVT_PCIE_OPTIMIZED_COMPILE
   ```

2. Enable the partition compile flow using VCS partition compile options:

   ```
   -partcomp -fastpartcomp=j<N> +optconfigfile+pc.optcfg
   ```

3. Import all the packages or sub-packages at user testbench Top file. It is recommended to include methodology specific import file, say for example,

   ```
   `include "import_pcie_svt_ovm_pkgs.svi"
   ```

Use auto partition if you have less number of cores. When the number of available cores is less than 4, use the auto partitioning—that is, do not provide the `pc.optcfg` file with new packages because the partition compile on less cores with many partition degrades the compilation performance.

17.3 The “vcspcvlog” Simulator Target in Makefiles

The vcspcvlog simulator target in the Makefiles of the VIP OVM examples enables compilation of the examples in the two-step partition compile flow. The following partition compile options are used:

```
-partcomp +optconfigfile+pc.optcfg -fastpartcomp=j4 -lca
+define+SVT_PCIE_OPTIMIZED_COMPILE
```

One partition is created for each line specified in the pc.optcfg configuration file. The -fastpartcomp=j4 option enables parallel compilation of partitions on different cores of a multi-core machine. You can incorporate the partition compile options listed above into your existing vcs command line.

In the partition compile flow, changes in the testbench, VIP, or DUT source code trigger recompilation in only the required partitions. You must ensure that the Verification Compiler compilation database is not deleted between successive recompilations.

17.4 The “vcsmxpcvlog” Simulator Target in Makefiles

The vcsmxpcvlog simulator target in the Makefiles of the VIP OVM examples enables compilation of the examples in the three-step partition compile flow. The following partition compile options are used:

```
-partcomp +optconfigfile+pc.optcfg -fastpartcomp=j4 -lca
+define+SVT_PCIE_OPTIMIZED_COMPILE
```

There is no change in the vlogan commands. One partition is created for each line specified in the pc.optcfg configuration file. The -fastpartcomp=j4 option enables parallel compilation of partitions on different cores of a multi-core machine. You can incorporate the partition compile options listed above into your existing vcs command line.

In the partition compile flow, changes in the testbench, VIP, or DUT source code trigger recompilation only in the required partitions. You must ensure that the Verification Compiler compilation database is not deleted between successive recompilations.

17.5 The “vcsmxpipvlog” Simulator Target in Makefiles

The vcsmxpipvlog simulator target in the Makefiles of the VIP OVM examples enables compilation of the examples in the PIP flow. There is no change in the vlogan commands. One PIP compilation command with the -genip option is created for each line specified in the pc.optcfg configuration file. The -integ option is used in the integration step to generate the simv executable.

In the PIP flow, changes in the testbench, VIP, or DUT source code trigger recompilation in only the required PIPs. You must ensure that the Verification Compiler compilation database is not deleted between successive recompilations.

17.6 Precompiled IP Implementation in Testbenches with Verification IPs

You can use the Makefiles in the VIP OVM examples as a template to set up the partition compile or PIP flow in your design and verification environment by performing the following steps:

- Modify the pc.optcfg configuration file to include the user-defined partitions. The recommendations are as follows:
  - Create four to eight overall partitions (DUT and VIP combined).
  - Some VIP packages may include separate packages for transmitter and receiver VIPs. If only a transmitter or a receiver VIP is required, then the unused package can be removed from the configuration file.
  - Continue to use separate partitions for common packages, such as ovm_pkg and svt_ovm_pkg, as defined in the VIP configuration file.
- Incorporate the partition compile or precompiled IP command line options documented in previous sections or issued by the Makefile targets into the vcs command lines.

For more information on partition compile and precompiled IP options, such as, -sharedlib and -pcmakeprof, see the VCS/VCS MX LCA Features Guide.

17.7 Example

The following are the steps to integrate VIPs into the partition compile and PIP flows:

1. Once you set the VC_HOME variable, the VC_VIP_HOME variable is automatically set to the following location:

   ```
   $VC_HOME/vl
   ```

2. Check the available VIP examples using the following command:

   ```
   $VC_VIP_HOME/bin/dw_vip_setup -i home
   ```

3. Install the example.

   For example, to install the PCIe OVM Unified Example, use the following command:

   ```
   $VC_VIP_HOME/bin/dw_vip_setup -e pcie_svt/tb_pcie_svt_ovm_unified_vip_sys -svtb
   cd examples/sverilog/pcie_svt/tb_pcie_svt_ovm_unified_vip_sys
   ```

4. Run the tests present in the tests directory in the example.

   For example, to run the ts.base_test.sv test in the VCS two-step flow with partition compile, use the following command:

   ```
   gmake base_test USE_SIMULATOR=vcspcvlog
   ```

   To run the ts.base_test.sv test in the VCS UUM flow with partition compile, use the following command:

   ```
   gmake base_test USE_SIMULATOR=vcsmxpcvlog
   ```

   To run the ts.base_test.sv test in the VCS UUM flow with precompiled IP, use the following command:

   ```
   gmake base_test USE_SIMULATOR=vcsmxpipvlog
   ```

5. To modify or change the partitions, you must change the pc.optcfg file for the example.

18 Integrated Planning for VC VIP Coverage Analysis

To leverage the Verdi planning and management solutions, the VIP verification plans in the .xml format were required to be converted to the .hvp format manually. Now, VC VIPs provide an executable Verification Plan in the .hvp format together with the .xml format. You can now load the VIP verification plans easily to the Verdi verification and management solutions in a single step. Further, you can easily integrate these plans to the top level verification plan by a single click drag and drop.

Coverage results are annotated to the plan that helps to map the verification completeness on a feature by feature basis at the aggregate level.

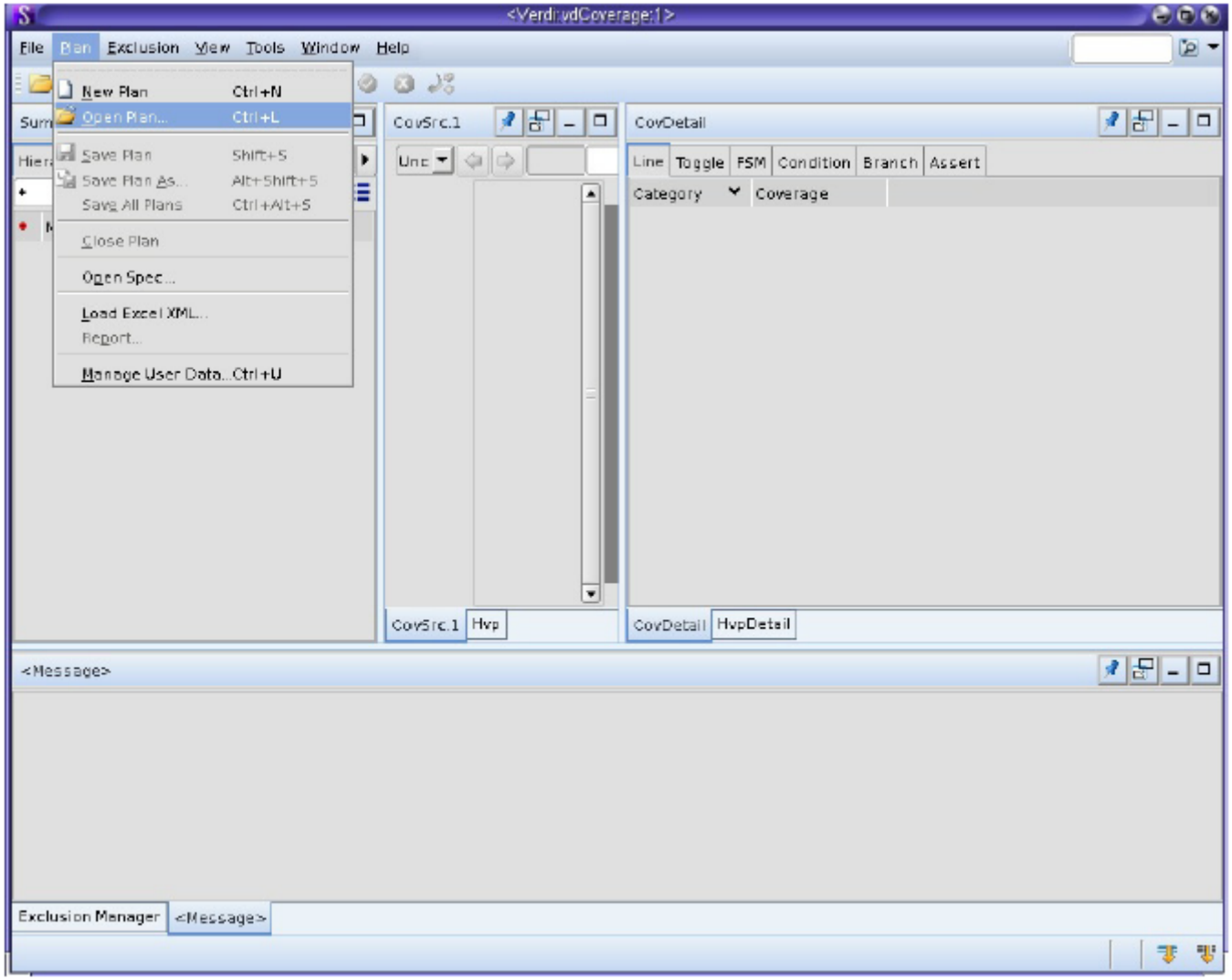
18.1 Use Model

The Verdi Coverage flow requires the .hvp files that capture the Verification Plan to be loaded on the Verdi Coverage Graphical User Interface (GUI), and the coverage annotated within the GUI.

To leverage the verification plan, perform the following steps:

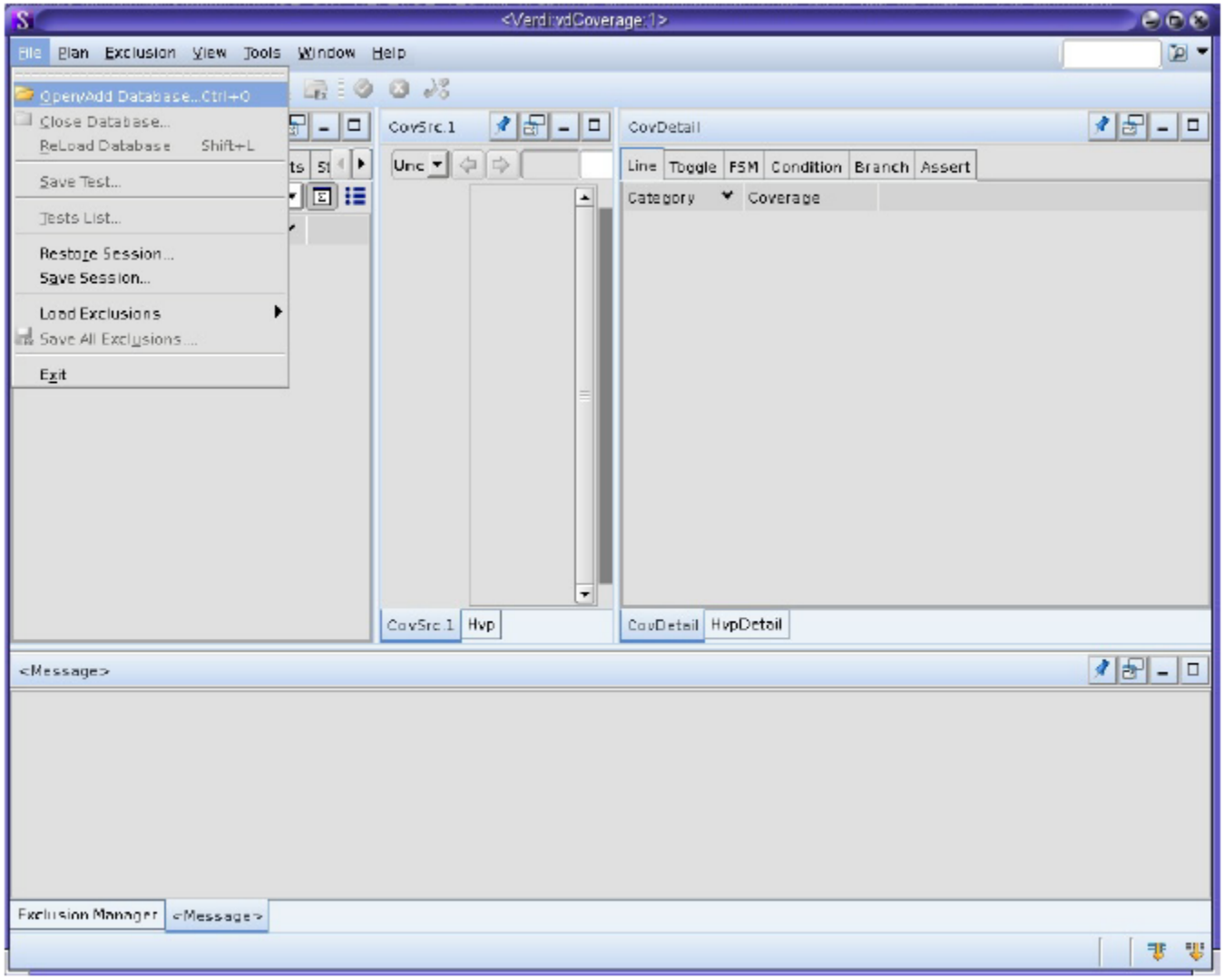
1. Navigate to the example directory using the following command:
   cd <unified_example_dir>
2. 2.Invoke the make command with the name of the test you want to run, as follows:
   gmake USE_SIMULATOR=vcsvlog <test_name>
3. Invoke the Verdi GUI using the following command:
   verdi –cov &
4. Copy the Verification Plans folder from the installation path to the current work area, as follows:
   cp $VC_VIP_HOME/vip/svt/pcie_svt/latest/doc/VerificationPlans .
5. Select the load plan from the Plan drop down menu as shown in the following illustration.

Figure 18-1 Open Plan



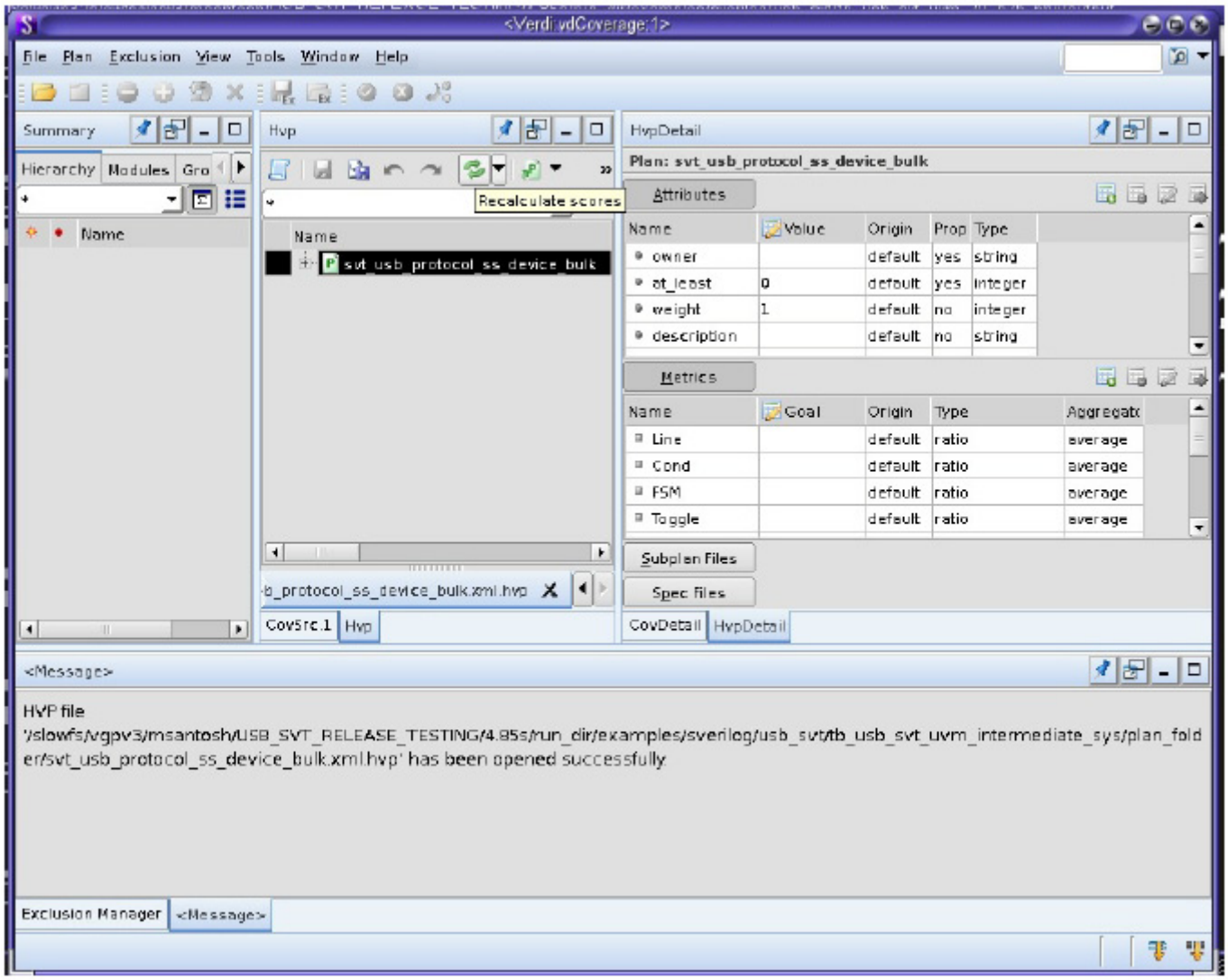
Load the coverage database (simvcssvlog.vdb) from the output folder in the current directory using the Verdi drop down menu, as shown in the following illustration.

Figure 18-2 Verdi GUI



Click Recalculate button to annotate the coverage results, as shown in the following illustration.

Figure 18-3 Recalculate



A Protocol Checks

A number of automatic protocol checks are built into the PCIe VIP, to test for compliance with the PCIe specification. The HTML class reference documentation includes the protocol checks of the following groups:

Active component protocol check groups:

- ACTIVE_PL_LANE_ENDEC
- ACTIVE_PL_LANE_OS
- ACTIVE_PL_LANE_PCS
- ACTIVE_PL_PIE8
- ACTIVE_PL_PIPE
- ACTIVE_PL_LANE_SERDES
- ACTIVE_PL
- ACTIVE_DL
- ACTIVE_TL
- ACTIVE_TARGET_APP
- ACTIVE_REQUESTER_APP
- ACTIVE_DRIVER_APP

Passive component protocol check groups:

- PASSIVE_PL_PIPE
- PASSIVE_PL
- PASSIVE_DL
- PASSIVE_TL

For check description and PCIe specification version, see “Protocol Checks” tab in the HTML class reference documentation available at the following locations:

- PCIe SVT

$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_ovm_class_reference/html/protocolChecks.html

- MPHY SVT

  $DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/mpcie_svt_ovm_class_reference/html/protocolChecks.html

B PCIe PIE-8 Interface

The PIE-8 specification is the "*PHY Interface Extensions Supporting 8GT/s PCIe*" (Revision 2.02 dated October 1, 2014). It is an extension of the PIPE 2.0 specification that supports 8.0GT/s and 16.0GT/s data rates and equalization at those rates.

Attention Synopsys supports only that portion of this PIE-8 specification as it relates to passing information to and from the PHY for equalization control and response.

B.1 Supported Interface Signals

Table B-1 lists the PIE-8 signals implemented to support equalization functionality.

Table B-1 PIE-8 Control/Response Signals

Name	Direction	Active Level	Description
MacDataEn	Input	High	Used for 8.0 GT/s and 16.0 GT/s equalization and setting Lane numbers. When '1', a transfer to the PHY is in-progress. One signal per Lane.
MacData[5;0]	Input	N/A	Used with MACDataEn for 8.0 GT/s and 16.0 GT/s equalization and setting Lane numbers. Control and data for equalization control and setting Lane numbers. One signal per Lane.
PhyDataEn	Ouput	High	Used for 8.0 GT/s and 16.0 GT/s equalization. When '1', a transfer to the MAC is in-progress. One signal per Lane.
PhyData[5:0]	Ouput	N/A	Used with PHYDataEn for 8.0 GT/s and 16.0 GT/s equalization. Control and data for equalization control. One signal per Lane.

All other signals are the same as the PIPE 4.3 specification (PCLK as PHY output) with the exception of those signals specified in the following table.

Table B-2 PIPE 4.3 Equalization Signals Not Used

Name	Spipe Dir.	Lane 0 Mpipe / Spipe Name	Mpipe / Spipe Task
GetLocalPresetCoefficients			
	Input	get_local_preset_coefficients_0 / attached_get_local_preset_coefficients_0	PipeSerdesEqControl / PipeSlaveEqControl
LocalTxCoefficientsValid			
	Output	local_tx_coefficients_valid_0 / attached_local_tx_coefficients_valid_0	PipeSerdesEqControl / PipeSlaveEqControl
LocalPresetIndex[3:0]			
	Input	local_preset_index_0 / attached_local_preset_index_0	PipeSerdesEqControl / PipeSlaveEqControl
LocalTxPresetCoefficients			
[17:0]	Output	local_tx_preset_coefficients_0 / attached_local_tx_preset_coefficients_0	PipeSerdesEqControl / PipeSlaveEqControl
LocalFS[5:0]			
	Output	local_fs_0 / attached_local_fs_0	PipeSerdesEqControl / PipeSlaveEqControl
LocalLF[5:0]			
	Output	local_lf_0 / attached_local_lf_0	PipeSerdesEqControl / PipeSlaveEqControl
RxEqEval			
	Input	rx_eq_eval_0 / attached_rx_eq_eval_0	PipeSerdesEqControl / PipeSlaveControl
InvalidRequest			
	Input	invalid_request_0 / attached_invalid_request_0	PipeSerdesEqControl / PipeSlaveEqControl
TxDeemph[17:0]			
	Input	tx_deemph_0 / attached_tx_deemph_0	PipeSerdesEqControl / PipeSlaveEqControl
FS[5:0]			
	Input	fs_0 / attached_fs_0	PipeSerdesEqControl / No connection

Table B-2 PIPE 4.3 Equalization Signals Not Used (Continued)

Name	Spipe Dir.	Lane 0 Mpipe / Spipe Name	Mpipe / Spipe Task
LF[5:0]			
	Input	lf_0 / attached_lf_0	PipeSerdesEqControl / No connection
RxPresetHint[2:0]			
	Input	rx_preset_hint_0 / attached_rx_preset_hint_0	PipeSerdesEqControl / PipeSlaveEqControl
LinkEvaluationFeedbackFigureMerit[7:0]			
	Output	link_eval_feedback_figure_of_merit_0 / attached_link_eval_feedback_figure_of_merit_0	PipeSerdesEqControl / PipeSlaveEqControl
LinkEvaluationFeedbackDirectionChange [5:0]			
	Output	link_eval_feedback_direction_change_0 / attached_link_eval_feedback_direction_change_0	PipeSerdesEqControl / PipeSlaveEqControl
RxEqInProgress			
	Input	rx_eq_in_progress_0 / attached_rx_eq_in_progress_0	PipeSerdesEqControl / PipeSlaveControl

B.2 Configuration Parameters

The following table, Table B-3 lists configuration members for setting the PIE8 Interface in the class svt_pcie_pl_configuration. For additional information for the PHY layer configuration consult the HTML Reference documentation.

Table B-3 PIE-8 Parameters

Configuration Name	Description
pie8_mode_en	
	Enable PIE-8 mode (PHY Interface Extensions Supporting 8GT/s PCIe): Enables the PIE-8 mode state machines (either as MAC master1dor PHY Slave based on is_pie8_mode_master = SVT_PCIE_IS_PIE8_MODE_MASTER_DEFAULT;)
is_pie8_mode_master	
	Indicates whether the component is PIE8 master or PIE8 slave. When set to 1, acts as PIE8 master. When set to 0, acts as PIE8 slave.

Table B-3 PIE-8 Parameters (Continued)

Configuration Name	Description
pie8_phy_delay_to_tx_cmd_out_min	
	If performing as a PHY slave for the PIE-8 interface, This is the minimum number of PClk cycles that the PIE-8 slave state machine will wait in the PHY_RX_WAIT_TX_PHY_RESP state before asserting PHYDataEn signal and proceeding toward completion.
pie8_phy_delay_to_tx_cmd_out_max	
	If performing as a PHY slave for the PIE-8 interface, This is the maximum number of PClk cycles that the PIE-8 slave state machine will wait in the PHY_RX_WAIT_TX_PHY_RESP state before asserting PHYDataEn signal and proceeding toward completion.
pie8_enable_equalization_checks	
	When set to a 1, it enables the PIE-8 checks in the PIE8 PHY state machine. The checks performed are enabled individually as defined by pie8_enable_equalization_checks = SVT_PCIE_PIE8_ENABLE_EQUALIZATION_CHECKS_DEFAULT;.
pie8_equalization_check_vector	
	Enable PIE-8 checks when pie8_enable_equalization_checks is set to 1.
pie8_max_mac_wait_delay_to_phy_dataen_timeout	
	The maximum time (in ns) that a lane\u2019s Pie8MacStateMachine will wait in its TX_WAIT_RX_PHY_RESP state for the PHYDataEn signal to be received.

B.3 Status Class PIE8 Members

The following table, Table B-4 lists all the status members for the PIE8 interface in the class svt_pcie_pl_status.

Table B-4 PIE8 Members in Class svt_pcie_pl_status

PIE8 Member	Description
last_pie8_mac_state	Last state of the PIE-8 MAC master state machine. See list below for states.
last_pie8_phy_state	Last state of the PIE-8 PHY slave state machine. See list below for states
pie8_mac_state	Current state of the PIE-8 MAC master state machine. See list below for states.
pie8_phy_state	Current state of the PIE-8 PHY slave state machine. See list below for states.

These are the possible PIE8 PHY states:

- PHY_RX_IDLE
- PHY_RX_DATA
- PHY_RX_WAIT_TX_PHY_RESP
- PHY_TX_CMD_OUT

- PHY_TX_DATA
- PHY_WAIT_EVAL_RESP
- PHY_WAIT_MAC_DATA_EN_DROP

These are the possible PIE8 MAC states

- MAC_TX_IDLE
- MAC_TX_CMD_OUT
- MAC_TX_DATA(`SVT_PCIE_STATE_PIE8_MAC_TX_DATA)
- MAC_TX_WAIT_RX_PHY_RESP
- MAC_RX_DATA
- MAC_WAIT_PHY_DATA_EN_DROP

B.4 PHY PIE-8 ASCII Signals

The following table lists the ASCII signals on the PIE-8 PHY.

Table B-5 PHY PIE-8 ASCII Signals

Signal Name	Description
ascii_pie8_lane<n>_mac_state	<n> = 0 to 31. Current state of Lane <n>'s MAC PIE-8 Master state machine
ascii_pie8_lane<n>_phy_state	<n> = 0 to 31. Current state of Lane <n>'s MAC PIE-8 Slave state machine

B.5 PHY PIE-8 Internal Signals

The following signals may be helpful in debugging DUT issues (only one of these machines will be active in a given "root" or "endpoint" model). They are per lane and for both the MAC and PHY.

Table B-6 MAC Internal PIE-8 "per Lane" Signals

Signal Name	Description
pie8_mac_state	Current state of a Lane's MAC PIE-8 Master state machine
last_pie8_mac_state	Previous state of a Lane's MAC PIE-8 Master state machine
pie8_cycle_en	"Per bit" start of a Lane's MAC PIE-8 Master state machine
pie8_command	Current active command of a Lane's MAC PIE-8 Master state machine (if pie8_cycle_en[lane_num]" is a "1").
pie8_command_done	Last command of a Lane's MAC PIE-8 Master state machine has completed.
pie8_num_tx_data	Number of data cycles to be transmitted by a Lane's MAC PIE-8 Master state machine. Loaded when command starts and pre-decremented as data is transmitted.

Table B-6 MAC Internal PIE-8 "per Lane" Signals

Signal Name	Description
pie8_phy_rx_control	First datum received by a Lane's MAC PIE-8 Master state machine when the "PhyDataEn" for that lane is first asserted. Valid only when pie8_cycle_en[lane_num]" is a "1".
pie8_exp_num_rx_data	Number of data cycles to be received by a Lane's MAC PIE-8 Master state machine. Will start out as greater than 0 for MAC commands that expect a PHY response. Loaded when "PhyDataEn" for that lane is first asserted and pre-decremented as data is received. Valid only when pie8_cycle_en[lane_num]" is a "1".

Table B-7 PHY Internal PIE-8 "per Lane" Signals

Signal Name	Description
pie8_phy_state	Current state of a Lane's PHY PIE-8 Slave state machine
last_pie8_phy_state	Previous state of a Lane's PHY PIE-8 Slave state machine
pie8_num_tx_data	Number of data cycles to be transmitted by a Lane's MAC PIE-8 Slave state machine. Loaded when command starts and pre-decremented as data is transmitted.
pie8_exp_num_rx_data	Number of data cycles to be received by a Lane's PHY PIE-8 Slave state machine from the MAC. Will start out as greater than 0 for MAC commands that expect a PHY response. Loaded when "MACDataEn" for that lane is first asserted and pre-decremented as data is received.

B.6 PIE-8 Protocol Check "MSGCODEs"

The following table list the "MSGCODEs" associated with PIE-8 protocol and data checking.

Table B-8 PIE-8 MSGCODES

Signal Name	Description
MSGCODE_PCIESVC_PIE8_PHY_RX_PRESET_ MISCOMPARE	
	The "Pie8PhyStateMachine" checks that the "preset" it received from the MAC in the "Set Initial Presets" command is the one expected based on what was saved by the VIP in its respective "upstream_preset_reg" or "downstream_preset_reg" at the 8.0 GT/s data rate.
MSGCODE_PCIESVC_PIE8_PHY_RX_PRESET_ MISCOMPARE_16G	
	The "Pie8PhyStateMachine" checks that the "preset" it received from the MAC in the "Set Initial Presets" command is the one expected based on what was saved by the VIP in its respective "upstream_preset_reg_16g" or "downstream_preset_reg_16g" at the 16.0 GT/s data rate.
MSGCODE_PCIESVC_PIE8_PHY_RX_PRESET_ HINT_MISCOMPARE	

Table B-8 PIE-8 MSGCODES (Continued)

Signal Name	Description
	The "Pie8PhyStateMachine" checks that the "preset hint" it received from the MAC in the "Set Initial Presets" command is the one expected based on what was saved by the VIP in its respective "upstream_preset_hint_reg" or "downstream_preset_hint_reg" at the 8.0 GT/s data rate.
MSGCODE_PCIESVC_PIE8_PHY_RX_PRESET_ HINT_MISCOMPARE_16G	
	The "Pie8PhyStateMachine" checks that the "preset hint" it received from the MAC in the "Set Initial Presets" command is the one expected based on what was saved by the VIP in its respective "upstream_preset_hint_reg_16g" or "downstream_preset_hint_reg_16g" at the 16.0 GT/s data rate.
MSGCODE_PCIESVC_PIE8_PHY_RX_PRESET_ TABLE_ENTRY_INVALID	
	The "Pie8PhyStateMachine" checks that the "preset" it received from the MAC in the "Request Local Preset" command has a valid entry (checks the "VALID" bit - not the coefficient rules) in its respective "upstream_tx_preset_coefficient_mapping_table" or "downstream_tx_preset_coefficient_mapping_table" at the 8.0 GT/s data rate.
MSGCODE_PCIESVC_PIE8_PHY_RX_PRESET_ TABLE_ ENTRY_INVALID_16G	
	The "Pie8PhyStateMachine" checks that the "preset" it received from the MAC in the "Request Local Preset" command has a valid entry (checks the "VALID" bit - not the coefficient rules) in its respective "upstream_tx_preset_coefficient_mapping_table_16g" or "downstream_tx_preset_coefficient_mapping_table_16g" at the 16.0 GT/s data rate.
MSGCODE_PCIESVC_PIE8_PHY_RX_MAC_DATA_ LESS_THAN_EXP	
	The "Pie8PhyStateMachine" checks in its "PHY_RX_DATA" state that it receives the expected number of datums from the MAC for any command that receives data (more than just the "control" byte). If "MACDataEn" de-asserts before the pie8_exp_num_rx_data" for the Lane reaches "0", this check will fire.
MSGCODE_PCIESVC_PIE8_PHY_RX_INVALID_ MAC_DATA_EN	
	The "Pie8PhyStateMachine" checks in its "PHY_RX_WAIT_TX_PHY_RESP" state to see if the "MACDataEn" is driven to a "1" again. If it is, this check will fire.
MSGCODE_PCIESVC_PIE8_PHY_RX_ UNSUPPORTED_MAC_CONTROL	
	The "Pie8PhyStateMachine" checks in its "PHY_RX_IDLE" state whether the "control" value sent when "MACDataEn" is driven to a "1" initially is a valid "command". If it is a "reserved" or "unsupported" command (such as "Set Lane Number"), this check will fire.
MSGCODE_PCIESVC_PIE8_PHY_MAC_DATA_ EN_RESTARTED	
	The "Pie8PhyStateMachine" checks in its "PHY_TX_CMD_OUT", "PHY_TX_DATA" and "PHY_WAIT_EVAL_RESP" states to see if the "MACDataEn" is driven to a "1" again. If it is, this check will fire.
MSGCODE_PCIESVC_PIE8_MAC_ UNSUPPORTED_COMMAND	

Table B-8 PIE-8 MSGCODES (Continued)

Signal Name	Description
	The "Pie8MacStateMachine" checks in its "MAC_TX_IDLE" state whether its "per Lane" command ("pie8_command[lane_num]") that it received from the LTSSM when its "pie8_command[lane_num]" bit was asserted is a valid PIE-8 MAC "Information Transfer". If it is not a valid MAC "Information Transfer", this check will fire and the state machine will remain in the "MAC_TX_IDLE" state, clearing the offending command.
MSGCODE_PCIESVC_PIE8_MAC_RX_DATA_ LESS_THAN_EXP	
	The "Pie8MacStateMachine" checks in its "MAC_RX_DATA" state whether its "per Lane" expected number of datums for the currently active command (pie8_exp_num_rx_data[lane_num]) have been received (reached "0") before the "PHYDataEn" is de-asserted. If "PHYDataEn" is de-asserted before this value reaches "0", this check will fire. This means that the PHY did not send all the data required for the command sent to it.
MSGCODE_PCIESVC_PIE8_MAC_RX_DATA_ MORE_THAN_EXP	
	The "Pie8MacStateMachine" checks in its "MAC_WAIT_PHY_DATA_EN_DROP" state that the "PHYDataEn" is de-asserted. If the "PHYDataEn" remains asserted, this check will fire. This means that the PHY is sending too much data for the command sent to it.
MSGCODE_PCIESVC_PIE8_MAC_WAIT_PHY_ DATAEN_TIMEOUT	
	The "Pie8MacStateMachine" checks in its "MAC_TX_WAIT_RX_PHY_RESP" state that the "PHYDataEn" is asserted before the timeout defined by the "PIE9_MAX_MAC_WAIT_DELAY_TO_PHY_DATAEN_TIMEOUT_VAR" (in nS). If "PHYDataEn" is not asserted within this timeout value, the MAC state machine will fire this check, clear the current command and return to its "MAC_TX_IDLE" state. This would be the result of the PHY not responding in time to the MAC "Information Transfer" command.

C PCIe Compile-time Parameters

Parameters that must be set before compilation are listed in the following sections:

- Model Parameters
- Driver Application Parameters
- Requester Parameters
- Completion Target Parameters
- Memory Target Parameters
- Transaction Layer Parameters
- Data Link Layer Parameters
- Physical Layer Parameters
- Physical Coding Sublayer (PCS) Parameters
- Serializer/Deserializer (SERDES) Parameters

C.1 Model Parameters

Several parameters are set at the model. They typically 'trickle-down' to the individual layers. Table C-1 lists those parameters.

Table C-1 Parameters set in the model

Parameter Name	Type	Range	Default Value	Description
DEVICE_IS_ROOT	Integer	0-1	(Per model)	This value is '1' if the particular model is for a root, else it is '0'.
NUM_PMA_INTERFACE_BITS	Integer	10, 16, 32, 64, 128	10	Number of bits on the PMA interface

Table C-1 Parameters set in the model (Continued)

PCIE_SPEC_VER	Real	1.1, 2.0, 2.1, 3.0	PCIE_SPEC_VER_3_0	See Include/pciesvc_parms.v: PCIE_SPEC_VER_* Note: Please set this here, not in the individual layers.
HIERARCHY_NUMBER	Integer	0 - large value	0	The per-root hierarchy number – these start at 0 and count upwards. Set this to the root hierarchy that this model belongs to.
DISPLAY_NAME	String		"svt_pcie_device_agent_xx_yy_hdl_model_zz." (Specific to each model).	String prefixed to messages to display in the output log.

C.2 Driver Application Parameters

Driver parameters are listed in Table C-2.

Table C-2 Driver parameters

Parameter Name	Type	Range	Default	Description
MAX_NUM_TAGS				
	Integer	1-256	32	Maximum number of tags that can be used. If greater than 32 it is assumed that the extended tag bits are legal to use.
CMB_TABLE_SIZE				
	Integer	16-256	256	Size of the command management block, which is used to track pending and outstanding transactions.
DISPLAY_NAME				
	String		"pciesvc_driver"	Default display name for the driver.

C.3 Requester Parameters

Requester parameters are listed in Table C-3.

Table C-3 Requester parameters

Parameter Name	Type	Range	Default	Description
MAX_NUM_TAGS				
	Integer	1-256	32	Maximum number of outstanding tags. Note that this must be smaller than the command management block table size (see CMB_TABLE_SIZE parameter below.)
NUM_MEM_ADDR_SEGMENTS				
	Integer	16-4096	10	Number of unique randomization segments – each of which is a min/max memory address range.

Table C-3 Requester parameters (Continued)

CMB_TABLE_SIZE				
	Integer	16-256	64 entries	Size of the command management block table, which is used to track pending and outstanding transactions.
DISCARD_COMPLETIONS				
	Integer	01	0	When set to a 1, the driver will immediately upon reception of a completion discard the status of completed commands. Completion status cannot be checked in this mode. Users should only set this to 1 if they will not be checking completion results.
DISPLAY_NAME				
	String		pciesvc_ requester	Default prefix in $msglog calls.

C.4 Completion Target Parameters

Completion target parameters are listed in Table C-4.

Table C-4 Completion target parameters

Parameter Name	Type	Range	Default	Description
MEM_WRITE_NOTIFICATION_FIFO_SIZE				
	Integer	16-4096	64 entries	Number of queued memory-write notifications that can be queued up.
CMB_TABLE_SIZE				
	Integer	16-256	64 entries	Size of the command management block table, which is used to track pending and outstanding transactions.
DISPLAY_NAME				
	String		“pciesvc_ target”	Default prefix in $msglog calls.
PERCENTAGE_CORRUPT_TLP_DIGEST				
	Integer		0	Percentage of outbound transactions with a TLP digest (ECRC), that have a corrupt tlp digest (ECRC). Expected Response. DUT should drop the TLP and verification of the generation and transmission of the ERR_MSG is left up to the user.

C.5 Memory Target Parameters

Memory target parameters are listed in Table C-5.

Table C-5 Memory target parameters

Parameter Name	Type	Range	Default	Description
PAGE_SIZE_IN_DWORDS				
	Integer	8-4096	64	Large page size, in units of dwords. Any transfer of this size (or larger) will allocate pages of this size.
SMALL_PAGE_SIZE_IN_DWORDS				
	Integer	8-256	8	Small page size, in units of dwords. Any transfer of this size (or larger, but less than the above PAGE_SIZE_IN_DWORDS) will allocate pages of this size. Any requests smaller than this will use individual Dwords.
NUM_64_BIT_PAGES				
	Integer	1024-16k	4096	Number of the 64 or 32-bit pages initialized at startup. If the application attempts to allocate more pages than initialized, the allocation will fail, and the user should up the number of pages. Each Large, Small or Dword uses a single page entry.
NUM_32_BIT_PAGES				
	Integer	1024-16k	4096	
NUM_ATTR_TBL_ENTRIES				
	Integer	1-1024	64	The number of attribute table entries. This defines how many memory ranges are available (see the Add/RemoveMemRange task calls below).
DISPLAY_NAME				
	String		“memory_target0.”	Default prefix in $msglog calls.

C.6 Transaction Layer Parameters

Transaction Layer parameters are listed in Table C-6.

Table C-6 Transaction Layer parameters

Parameter Name	Type	Range	Default Value	Description
DEFAULT_ROUTE_AT_APPL_ID				
	Integer	0 - large value	0	Default Application ID to route Address Translation requests to.
NUM_APPL_ID				
	Integer	8-128	8	Max number of unique Application IDs. IDs assigned to applications must be less than this value.

Table C-6 Transaction Layer parameters (Continued)

RID_APPLID_TABLE_SIZE				
	Integer	4-4096	64	Number of unique RID to Appl_id map entries.
RID_MSGCODE_APPLID_TABLE_SIZE				
	Integer	4-4096	64	Number of unique {RID,msgcode} to Appl_id map entries.
MEM_ADDR_ADDPLID_TABLE_SIZE				
	Integer	4-4096	64	Number of unique Mem Address to Appl_id map entries.
IO_ADDR_ADDPLID_TABLE_SIZE				
	Integer	4-4096	64	Number of unique I/O Address to Appl_id map entries.
AT_ADDR_ADDPLID_TABLE_SIZE				
	Integer	4-4096	64	Number of unique AT Address to Appl_id map entries.
IS_TX_DOWNSTREAM				
	Integer	0-1	0	Stack direction. Used for TLP header checking/routing.
IS_SWITCH				
	Integer	0-1	0	Is this stack part of a switch? Used for TLP header checking/routing.

C.7 Data Link Layer Parameters

Data Link Layer parameters are listed in Table C-7.

Table C-7 Data Link Layer parameters

Parameter Name	Type	Range	Default	Description
MAX_NUM_RETRY_BUFFER_DWORDS				
	Integer	16-2^16	4096	Maximum number of dwords the retry buffer can hold before it backpressures the Transaction Layer.

C.8 Physical Layer Parameters

C.8.1 General Parameters

General Physical Layer parameters are listed in Table C-8.

Table C-8 Physical Layer general parameters

Parameter Name	Type	Range	Default	Description
DISPLAY_NAME				
	String		pciesvc_pl0	Default display name for the Physical Layer.

C.8.2 Physical Layer LTSSM-specific Parameters

Physical Layer LTSSM-specific parameters are listed in Table C-9.

Table C-9 LTSSM-specific parameters

Parameter Name	Type	Range	Default	Description
IS_TX_DOWNSTREAM				
	Integer	0-1	0	Indicates whether the transmitter is downstream or not. This affects link training behavior.
IS_PIPE_MASTER				
	Integer	0-1	1	When set to a 1, the VIP will behave as a pipe master. When set to 0, the VIP will behave as a pipe slave.

C.8.3 Equalization Parameters

Equalization parameters are listed in Table C-10.

Table C-10 Equalization parameters

Parameter Name	Type	Range	Default	Description
FULL_SWING				
			6'd48	Default fs value port will advertise in outgoing EQTS. It is recommended to use the SetTxTS1FSLF task to change this value rather than using a defparam.
LOW_FREQUENCY_COEFFICIENT				
			?'d3	Default LF value port will advertise in outgoing EQTS. It is recommended to use the SetTxTS1FSLF task to change this value rather than using a defparam.
PRECURSOR_COEFFICIENT				
			3	Default precursor coefficient value that is loaded into all preset settings. Users should use the task SetEQPreset2CoeffTable to change preset to coefficient mappings.
CURSOR_COEFFICIENT				
			5	Default cursor coefficient value that is loaded into all preset settings. Users should use the task SetEQPreset2CoeffTable to change preset to coefficient mappings.
POSTCURSOR_COEFFICIENT				
			6	Default postcursor coefficient value that is loaded into all preset settings. Users should use the task SetEQPreset2CoeffTable to change preset to coefficient mappings.

C.9 Physical Coding Sublayer (PCS) Parameters

PCS parameters are listed in Table C-11.

Table C-11 PCS parameters

Parameter Name	Type	Range	Default	Description
ENABLE_RX_ELASTIC_BUFFER				
	Integer	0-1	1	Enables the receive elastic buffer. This is not necessary for most simulations.
NUM_PMA_INTERFACE_BITS				
	Integer	10, 16, 32, 64, 128	10	Number of bits on the PMA interface. NOTE: please set this in the model.
DISPLAY_NAME				
	String		pciesvc_pcs	Default display name for msglog statements.

C.10 Serializer/Deserializer (SERDES) Parameters

Parameters that affect the behavior of the SERDES are listed in Table C-12.

Table C-12 SERDES parameters

Parameter Name	Type	Range	Default	Description
COMMA_SYNC_COUNT				
	Integer	0 - large value	0	Number of aligned commas before SERDES lock declared. Not used in PCIE.
BIT_SYNC_COUNT				
	Integer	10 - large value	1000	Number of min bit periods seen before PLL lock declared. Must be larger than maximum time of OOB active burst.
CLK_TOLERANCE				
	Integer	Any value	0.0001	Tolerance in ns. Equivalent to 100 PPM SAS requirement. This should not be changed as it affects SSC clock tracking.
COMMA_P				
	Integer	Any 10-bit value	10'b0011 111010	Definition of positive COMMA character.
COMMA_N				
	Integer	Any 10-bit value	10'b1100 000101	Definition of negative COMMA character.
DISPLAY_NAME				
	String		pciesvc_ serdes.	String prefixed to messages to display in the output log.

D Verilog Task/Parameter to SVT Class Mapping

This appendix provides mapping from SVC Verilog or SVT Verilog tasks and parameters to SVT OVM class members, for users who are migrating from the SVC or SVT Verilog PCIe VIP to the OVM PCIe.

This appendix contains the following sections:

- Transaction Layer Verilog Tasks and Parameters to OVM Class Members Map
- Data Link Layer Verilog Tasks and Parameters to OVM Class Member Maps

D.1 Transaction Layer Verilog Tasks and Parameters to OVM Class Members Map

This section contains the following tables:

- Transaction Layer Verilog Task to OVM Class Member Map
- Transaction Layer Verilog Parameters to OVM Class Members Map

D.1.1 Transaction Layer Verilog Task to OVM Class Member Map

Transaction Layer Verilog tasks are mapped to SVT class members in Table D-1, listed alphabetically by Verilog task.

Table D-1 Map of Transaction Layer Verilog tasks to OVM class members

Verilog Task	OVM Class Member
AddATAddrApplIdMapEntry()	svt_pcie_tl_service::service_type =
AddCfgBDFApplIdMapEntry()	svt_pcie_tl_service::service_type
AddIOAddrApplIdMapEntry()	svt_pcie_tl_service::service_type
AddMemAddrApplIdMapEntry()	svt_pcie_tl_service::service_type
AddRequesterIdApplIdMapEntry()	svt_pcie_tl_service::service_type
AddRIdMsgCodeApplIdMapEntry()	svt_pcie_tl_service::service_type

Table D-1 Map of Transaction Layer Verilog tasks to OVM class members (Continued)

Verilog Task	OVM Class Member
CheckFinalCredits()	svt_pcie_tl_service_check_final_credits_sequence
ClearStats()	svt_pcie_tl_service::service_type
DisplayATAddrApplidMap()	svt_pcie_tl_service::service_type
DisplayCfgBDFApplidMap()	svt_pcie_tl_service::service_type
DisplayIOAddrApplidMap()	svt_pcie_tl_service::service_type
DisplayMemAddrApplidMap()	svt_pcie_tl_service::service_type
DDisplayRidApplidMap()	svt_pcie_tl_service::service_type
DisplayRidMsgCodeApplidMap()	svt_pcie_tl_service::service_type
DisplayStats()	svt_pcie_tl_service::service_type
IsTransactionLayerIdle()	svt_pcie_tl_service::service_type
IsVcInitFinished()	svt_pcie_tl_status::vc_initialized
QueryCreditCounts()	svt_pcie_tl_status::credits_allocated::credit_limit::credits_consumed:: credits_received::init_credits_allocated::init_credit_limit
QueryRxCreditsAvailable()	svt_pcie_tl_status::rx_credits_available[48]
QueryTxCreditsAvailable()	svt_pcie_tl_status::tx_credits_available[48]
SetVcEnable()	svt_pcie_tl_service::service_type

D.1.2 Transaction Layer Verilog Parameters to OVM Class Members Map

Transaction Layer Verilog parameters are mapped to OVM class members in Table D-2, listed alphabetically by Verilog parameter.

Table D-2 Map of Transaction Layer class members to parameters

Verilog Parameter	OVM Class Member
AUTO_ENABLE_VC0_AT_STARTUP	svt_pcie_tl_configuration::auto_enable_vc0_at_startup
CREDIT_STARVATION_TIMEOUT_NS	svt_pcie_tl_configuration::credit_starvation_timeout_ns
DEFAULT_ROUTE_AT_APPL_ID	svt_pcie_tl_configuration::default_route_at_appl_id
DEFAULT_ROUTE_CFG_TYPE0_APPL_ID	svt_pcie_tl_configuration::default_route_cfg_type0_appl_id
DEFAULT_ROUTE_CFG_TYPE1_APPL_ID	svt_pcie_tl_configuration::default_route_cfg_type1_appl_id
DEFAULT_ROUTE_IO_APPL_ID	svt_pcie_tl_configuration::default_route_at_appl_id
DEFAULT_ROUTE_MEM_APPL_ID	svt_pcie_tl_configuration::default_route_mem_appl_id
DEFAULT_ROUTE_MSG_APPL_ID	svt_pcie_tl_configuration::default_route_msg_appl_id
ENABLE_ROUTE_AT_TO_FUNCTION	svt_pcie_tl_configuration::enable_route_at_to_function
ENABLE_ROUTE_CFG_TYPE0_TO_FUNCTION	svt_pcie_tl_configuration::enable_route_cfg_type0_to_function

Table D-2 Map of Transaction Layer class members to parameters (Continued)

Verilog Parameter	OVM Class Member
ENABLE_ROUTE_CFG_TYPE1_TO_FUNCTION	svt_pcie_tl_configuration::enable_route_cfg_type1_to_function
ENABLE_ROUTE_IO_TO_FUNCTION	svt_pcie_tl_configuration::enable_route_io_to_function
ENABLE_ROUTE_MEM_TO_FUNCTION	svt_pcie_tl_configuration::enable_route_mem_to_function
ENABLE_ROUTE_MSG_TO_FUNCTION	svt_pcie_tl_configuration::enable_route_msg_to_function
MAX_VC[0-7]_[P/NP/CPL]_UPDATEFC_DELAY	svt_pcie_tl_configuration::max_vc[0-7]_[p|np|cpl]_updatefc_delay
MIN_VC[0-7]_[P/NP/CPL]_UPDATEFC_DELAY	svt_pcie_tl_configuration::min_vc[0-7]_[p|np|cpl]_updatefc_delay
NUM_VC_[P/NPCPL]_NIT_[HDRDATA]_CREDITS	svt_pcie_tl_configuration::init_[cpl|np|p]_data_tx_credits
NUM_VC_[P/NPCPL]_NIT_[HDRDATA]_CREDITS	svt_pcie_tl_configuration::init_[cpl|np|p]_hdr_tx_credits
REMOTE_EXTENDED_TAG_FIELD_ENABLED	svt_pcie_tl_configuration::remote_extended_tag_field_enabled
REMOTE_MAX_PAYLOAD_SIZE	remote_max_payload_size::remote_max_payload_size
REMOTE_MAX_READ_REQUEST_SIZE	svt_pcie_tl_configuration::remote_max_read_request_size

D.2 Data Link Layer Verilog Tasks and Parameters to OVM Class Member Maps

- "Data Link Layer Verilog Task to OVM Class Member Map"
- "Data Link Layer Verilog Parameter to OVM Class Member Map"

D.2.1 Data Link Layer Verilog Task to OVM Class Member Map

Data Link Layer Verilog tasks are mapped to OVM class members in Table D-3, listed alphabetically by Verilog task.

Table D-3 Map of Data Link Layer Verilog tasks to OVM class members

Verilog Task	OVM Class Member
ClearStats	svt_pcie_dl_service_clr_stats_sequence
DisplayAttachedAckNakLatencyTolerances	svt_pcie_dl_configuration::min_ack_nak_latency
DisplayAttachedReplayTimeoutTolerances	svt_pcie_dl_service::DISPLAY_ATTACHED_REPLAY_TIMER_TOLERANCES
DisplayStats	svt_pcie_dl_service_disp_stats_sequence
InitiateASPMExit	svt_pcie_dl_configuration::min_ack_nak_latency
InitiateASPML0sEntry	svt_pcie_dl_configuration::min_ack_nak_latency
InitiateASPML1Entry	svt_pcie_dl_configuration::min_ack_nak_latency
InitiatePMExit	svt_pcie_dl_configuration::min_ack_nak_latency
InitiatePML1Entry	svt_pcie_dl_configuration::min_ack_nak_latency
InitiatePML23Entry	svt_pcie_dl_configuration::min_ack_nak_latency
IsDataLinkIdle	svt_pcie_dl_service_disp_stats_sequence

Table D-3 Map of Data Link Layer Verilog tasks to OVM class members (Continued)

Verilog Task	OVM Class Member
ReceivedDLLP	svt_pcie_dl::received_tlp_observed_port
ReceiveVendorDLLP	svt_pcie_dllp_vendor_defined_sequence, svt_pcie_dllp_vendor_defined_exception_sequence
SentDLLP	svt_pcie_dl::sent_dllp_observed_port
SentTLP	svt_pcie_dl::sent_tlp_observed_port
SetAttachedAckNakLatencyTolerance	svt_pcie_dl_configuration::attached_ack_nak_latency_tolerance_x1
SetAttachedReplayTimeout	svt_pcie_dl_configuration::attached_replay_timeout
SetAttachedReplayTimeoutTolerance	svt_pcie_dl_configuration:attached_replay_timeout_tolerance_xn where n is 1,2,4,8,12,16,32.
SetLinkEnable()	svt_pcie_dl_service_set_link_en_sequence::enable
SetMaxAckNakLatency	svt_pcie_dl_configuration::max_ack_nak_latency
SetMaxAttachedAckNakLatency	svt_pcie_dl_configuration::max_attached_nak_latency
SetMaxAttachedNakLatency	svt_pcie_dl_configuration::min_ack_nak_latency
SetMinAckNakLatency	svt_pcie_dl_configuration::min_ack_nak_latency
SetMinAttachedAckNakLatency	svt_pcie_dl_configuration::min_ack_nak_latency
SetReplayTimeout	svt_pcie_dl_configuration::replay_timeout
TransmitUserDLLP	svt_pcie_dllp_vendor_defined_sequence, svt_pcie_dllp_vendor_defined_exception_sequence

D.2.2 Data Link Layer Verilog Parameter to OVM Class Member Map

Data Link Layer Verilog parameters are mapped to OVM class members in Table D-4, listed alphabetically by Verilog parameter.

Table D-4 Map of Data Link Layer class members to tasks

Verilog Parameter	OVM Class Member
ASPM_TIMEOUT_CNT_LIMIT	svt_pcie_dl_configuration::aspm_timeout_cnt_limit
ATTACHED_INTERNAL_DELAY_2_5G	svt_pcie_dl_configuration::attached_internal_delay_2_5g
ATTACHED_INTERNAL_DELAY_5G	svt_pcie_dl_configuration::attached_internal_delay_5g
ENABLE_ASPM_L1_1_ENTRY	svt_pcie_dl_configuration
ENABLE_ASPM_L1_2_ENTRY	svt_pcie_dl_configuration
ENABLE_ASPM_L1_ENTRY	svt_pcie_dl_configuration::enable_aspm_l1_entry
ENABLE_EI_TX_TLP_ON_RETRY	svt_pcie_dl_configuration
ENABLE_PM_L1_1_ENTRY	svt_pcie_dl_configuration
ENABLE_PM_L1_2_ENTRY	svt_pcie_dl_configuration
ENABLE_TRANSACTION_LOG	svt_pcie_dl_configuration

Table D-4 Map of Data Link Layer class members to tasks (Continued)

Verilog Parameter	OVM Class Member
ENABLE_TX_TLP_REPORTING	svt_pcie_dl_configuration
INITFC_TIMEOUT_NS	svt_pcie_dl_configuration::min_ack_nak_latency
INITIAL_RECEIVE_SEQUENCE_VALUE	svt_pcie_dl_configuration::initial_receive_sequence_value
INITIAL_TRANSMIT_SEQUENCE_VALUE	svt_pcie_dl_configuration::initial_transmit_sequence_value
InitiateASPMExit	svt_pcie_dl_configuration::min_ack_nak_latency
InitiateASPML0sEntry	svt_pcie_dl_configuration::max_ack_nak_latency
InitiateASPML1Entry	svt_pcie_dl_configuration::min_ack_nak_latency
InitiatePMExit	svt_pcie_dl_configuration::min_ack_nak_latency
InitiatePML1Entry	svt_pcie_dl_configuration::min_ack_nak_latency
InitiatePML23Entry	svt_pcie_dl_configuration::min_ack_nak_latency
INTERNAL_DELAY_2_5G	svt_pcie_dl_configuration::internal_delay_2_5g
INTERNAL_DELAY_5G	svt_pcie_dl_configuration::internal_delay_5g
L0S_IDLE_TIMER_LIMIT_NS	svt_pcie_dl_configuration::l0s_idle_timer_limit_ns
LTR_L1_2_THRESHOLD_SCALE	svt_pcie_dl_configuration
LTR_L1_2_THRESHOLD_VALUE	svt_pcie_dl_configuration
MAX_INITFC_DELAY	svt_pcie_dl_configuration::min_ack_nak_latency
MAX_NAK_LATENCY	svt_pcie_dl_configuration::min_ack_nak_latency
MAX_NUM_REPLAYS	svt_pcie_dl_configuration::min_ack_nak_latency
MAX_PAYLOAD_SIZE	svt_pcie_dl_configuration::min_ack_nak_latency
MAX_TX_IPG	svt_pcie_dl_configuration::min_ack_nak_latency
MAX_TX_NULLIFIED_TLP_LEN	svt_pcie_dl_configuration::max_tx_nullified_tlp_len
MAX_UPDATEFC_DELAY	svt_pcie_dl_configuration::min_ack_nak_latency
MIN_INITFC_DELAY	svt_pcie_dl_configuration::min_ack_nak_latency
MIN_NAK_LATENCY	svt_pcie_dl_configuration::min_ack_nak_latency
MIN_TX_IPG	svt_pcie_dl_configuration::min_ack_nak_latency
MIN_TX_NULLIFIED_TLP_LEN	svt_pcie_dl_configuration::min_tx_nullified_tlp_len
MIN_UPDATEFC_DELAY	svt_pcie_dl_configuration::min_ack_nak_latency
PCIE_SPEC_VER	svt_pcie_dl_configuration::min_ack_nak_latency
PERCENTAGE_TX_TLP_INSTEAD_OF_INITFC2	svt_pcie_dl_configuration::tx_fc_init_completed_percentage
PM_TIMEOUT_CNT_LIMIT	svt_pcie_dl_configuration::pm_timeout_cnt_limit
RECEIVED_DLLP_INTERFACE_MODE	svt_pcie_dl_configuration::received_dllp_interface_mode
RECEIVED_TLP_INTERFACE_MODE	svt_pcie_dl_configuration::received_tlp_interface_mode
SENT_DLLP_INTERFACE_MODE	svt_pcie_dl_configuration::sent_tlp_interface_mode

Table D-4 Map of Data Link Layer class members to tasks (Continued)

Verilog Parameter	OVM Class Member
SENT_TLP_INTERFACE_MODE	svt_pcie_dl_configuration::sent_tlp_interface_mode
TX_NULLIFIED_TLP_HDR0_VALUE	svt_pcie_dl_configuration::tx_nullified_tlp_hdr0_value
TX_NULLIFIED_TLP_HDR1_VALUE	svt_pcie_dl_configuration::tx_nullified_tlp_hdr1_value
TX_NULLIFIED_TLP_HDR2_VALUE	svt_pcie_dl_configuration::tx_nullified_tlp_hdr2_value
TX_NULLIFIED_TLP_HDR3_VALUE	svt_pcie_dl_configuration::tx_nullified_tlp_hdr3_value
UPDATEFC_TIMEOUT_NS	svt_pcie_dl_configuration::min_ack_nak_latency
VC[0-7]_UPDATEFC_INTERVAL_NS	svt_pcie_dl_configuration::vc[0:7]_updatefc_interval_ns

E ECN Support

Engineering Change Notices (ECNs) that have been implemented in the PCIe VIP are listed in Table E-1.

Table E-1 ECNs implemented in the PCIe VIP

ECN	Spec Version	Comments
L1 sub-states	3	Supported
OBFF	3	Supported
LTR	3	Supported
SR-IOV	2.1	Supported
ARI Capability	2.1	Supported
DPC	3	Supported
TLP Prefixes	2.1	Supported
FLR	2.1	Supported EP
ASPM Optionality	2.1	Supported
TLP Processing Hints	2.1	Supported
Extended Tag Enable	2.1	Supported
ID Based Ordering	2.1	Supported
Atomic Operations	2.1	Supported
ARI Capability	2.1	Supported
Address Translation Serv	2.1	Supported

F SolvNetPlus PCIe VIP Articles

The following table lists and links to all the SolvNetPlus articles published on the PCIe VIP. The articles are organized by the following categories:

- Transaction Layer
- Data Link Layer
- PHY Layer
- Methodology, Testbench, and Debug

F.1 Transaction Layer

Title:VC VIP: Getting a Handle to Packets with Errors off a PCIe TLP Analysis Port

https://solvnet.synopsys.com/retrieve/2240203.html

Title:VC VIP: Generating MSG TLPs with the PCIe VIP

https://solvnet.synopsys.com/retrieve/2096908.html

Title:VC VIP: Configuring the PCIe VIP for Sending Packets Back to Back

https://solvnet.synopsys.com/retrieve/2060809.html

Title:VC VIP: Setting Read Completion Boundaries in the PCIe VIP

https://solvnet.synopsys.com/retrieve/1904746.html

Title:VC VIP: Creating Out of Order Completions with the OVM PCIe VIP

https://solvnet.synopsys.com/retrieve/1894083.html

Title:VC VIP: Varying Completion Response Latencies in the PCIe VIP

https://solvnet.synopsys.com/retrieve/1878299.html

Title:VC VIP PCIe: Traffic Class and Virtual Channels

https://solvnet.synopsys.com/retrieve/1822575.html

Title:PCIe SVT VIP: Memory Read Transactions Causing an Uninitialized Memory Error

https://solvnet.synopsys.com/retrieve/1777647.html

Title:PCIe SVT svt_pcie_driver_app_transaction transaction_type
https://solvnet.synopsys.com/retrieve/1653603.html
Title:PCIE SVT VIP: svt_pcie_driver_app_transaction, Config Read
https://solvnet.synopsys.com/retrieve/1623967.html
Title:PCIE SVT VIP : svt_pcie_driver_app_transaction : Config Write
https://solvnet.synopsys.com/retrieve/1623926.html
Title:PCIE SVT: Setting the EP Bit in a Driver Application Transaction for MemRd TLPs
https://solvnet.synopsys.com/retrieve/1598307.html
Title:PCIE SVT: Create out of order completions (CPL/CPLDs) of TLP packets
https://solvnet.synopsys.com/retrieve/1567155.html
Title:VC VIP: Updating/Setting a PCIe TLP Digest Field While Corrupting ECRC
https://solvnet.synopsys.com/retrieve/2298977.html
Title:VC VIP: Updating TLP Reserved Fields During a PCIe CFG Request
https://solvnet.synopsys.com/retrieve/2298994.html
Title: VC VIP: Controlling the Vendor ID Field of a Msg TLP in PCIe
https://solvnet.synopsys.com/retrieve/2335176.html
Title:VC VIP: Guidelines for Using max_read_cpl_data_size_in_bytes in PCIe Target App Configuration
https://solvnet.synopsys.com/retrieve/2333847.html
Title:VC VIP: Corrupting FMT to Cause Malformed Packets in the PCIe VIP Model
https://solvnet.synopsys.com/retrieve/2315907.html
Title:VC VIP: Emulating a PCIe Completion Timeout Error Using the PCIe VIP
https://solvnet.synopsys.com/retrieve/2327562.html
Title:VC VIP: Sending Different Size Completion Packets
https://solvnet.synopsys.com/retrieve/2359341.html
Title: VC VIP: Number of Fast Training Sequence (N_FTS) Settings in PCIe
https://solvnet.synopsys.com/retrieve/2369299.html
Title: VC VIP: Generating PCIe Error Poisoned Completions
https://solvnet.synopsys.com/retrieve/2368065.html
Title: VC VIP: Probably Cause for the PCIe VIP Not to Release Posted Credits
https://solvnet.synopsys.com/retrieve/2367808.html
Title: VC VIP: Avoiding PCIe DRIVER_COMMAND_TIMEOUT Errors (Verilog)
https://solvnet.synopsys.com/retrieve/2349570.html
Title: VC VIP: Setting the EP Bit in Completion TLPs Generated by a PCIe VIP
https://solvnet.synopsys.com/retrieve/2363332.html
Title: VC VIP: Sending Different Size Completion Packets
https://solvnet.synopsys.com/retrieve/2359341.html

Title: VC VIP: Resolving "MemRd accessed unitialized memory" Error Message in the PCIe VIP

https://solvnet.synopsys.com/retrieve/2359411.html

Title: VC VIP: Controlling the Number of Completions Sent in Response to a Read Request in the PCIe VIP

https://solvnet.synopsys.com/retrieve/2350962.html

Title: VC VIP: Setting the VIP to Expect Cfg Read to Have a Competion as Config Retry Status

https://solvnet.synopsys.com/retrieve/2339492.html

Title: VC VIP: Configuring the PCIe VIP to Not Expect a Completion for a Read Request

https://solvnet.synopsys.com/retrieve/2351152.html

Title: VC VIP: Understanding How the PCIe Model Manages the Transmission Order of TLPs

https://solvnet.synopsys.com/retrieve/2327901.html

Title: VC VIP: Preventing the Gerneration of ECRCs in PCIe

https://solvnet.synopsys.com/retrieve/2339416.html

Title: VC VIP: Creating Spurious Completion TLPs in the PCIe Model

https://solvnet.synopsys.com/retrieve/2317888.html

Title: VC VIP: Usage Examples for svt_pcie_driver_app_cfg_request_sequence (PCIe)

https://solvnet.synopsys.com/retrieve/2327932.html

Title: VC VIP: Controlling the Vendor ID Field of a Msg TLP in PCIe

https://solvnet.synopsys.com/retrieve/2335176.html

Title: VC VIP: Guidelines for Using max_read_cpl_data_size_in_bytes in PCIe Target App Configuration

https://solvnet.synopsys.com/retrieve/2333847.html

Title: VC VIP: Corrupting FMT to Cause Malformed Packets in the PCIe VIP Model

https://solvnet.synopsys.com/retrieve/2315907.html

Title: VC VIP: Emulating a PCIe Completion Timeout Error Using the PCIe VIP

https://solvnet.synopsys.com/retrieve/2327562.html

F.2 Data Link Layer

Title:VC VIP: Using Link Layer Callbacks of the Verilog Version of the PCIe VIP model

https://solvnet.synopsys.com/retrieve/2235954.html

Title:VC VIP: Calculating Delays for Sending Queued Packets on the PCIe VIP Link

https://solvnet.synopsys.com/retrieve/2113953.html

Title:VC VIP: PCIe Data Link Layer Enable Sequence

https://solvnet.synopsys.com/retrieve/2088989.html

Title:VC VIP: Data Link Layer Packet Generation Using the PCIe VIP

https://solvnet.synopsys.com/retrieve/2082099.html

Title:VC VIP: PCIe VIP Retry Support for CFG TLPs

https://solvnet.synopsys.com/retrieve/2061781.html

Title:VC VIP: Setting the PCIe VIP to Expect CRS Status on Received Completions

https://solvnet.synopsys.com/retrieve/2061608.html

Title:VC VIP: Resolving Why the PCIe Model Does Not Enter L1 Through ASPM

https://solvnet.synopsys.com/retrieve/2061376.html

Title:VC VIP: Setting the PCIe VIP FC Credit Starvation Timeout

https://solvnet.synopsys.com/retrieve/2058852.html

Title:VC VIP PCIe: Setting the Response Timeout for a RC VIP to Resend a RRAP Packet

https://solvnet.synopsys.com/retrieve/2058367.html

Title:VC VIP: Application Routing with an Upstream Port in the PCIe VIP

https://solvnet.synopsys.com/retrieve/1909091.html

Title:VC VIP: Using the PCIe VIP to Block INITFCs and Then Sending Out User-Defined and/or Vendor Specific DDLPS

https://solvnet.synopsys.com/retrieve/1878756.html

Title:VC VIP: Programming the PCIe VIP to Send a Single ACK for Multiple TLP Packets

https://solvnet.synopsys.com/retrieve/1864906.html

Title:VC VIP: Notes about PCIe VIP Handling of Duplicate TLPs

https://solvnet.synopsys.com/retrieve/2263434.html

Title:VC VIP: Understanding PCIe EI_CODEs and Their Usage

https://solvnet.synopsys.com/retrieve/2261844.html

Title:VC VIP: PCIe Simulation Hangs with DL_Inactive in loopback.active

https://solvnet.synopsys.com/retrieve/2322915.html

Title: VC VIP: How to Achieve DUT Retry Buffer Testing in PCIe?

https://solvnet.synopsys.com/retrieve/2372765.html

Title: VC VIP: Instructing the PCIe To Not Send an ACK or NAK for a Received TLP

https://solvnet.synopsys.com/retrieve/2369278.html

Title: VC VIP: Making the PCIe VIP Wait Until initFC2 Packets Complete Before Sending TLPs

https://solvnet.synopsys.com/retrieve/2313684.html

Title: VC VIP: Understanding Why VIP Initiated ASPML1 Entry Might Not Be Working (PCIe)

https://solvnet.synopsys.com/retrieve/2317924.html

F.3 PHY Layer

Title:VC VIP: Example Showing Multiple Iterations on PCIe L1 Sub-States

https://solvnet.synopsys.com/retrieve/2119994.html

Title:VC VIP: Some Issues to Consider When Connecting the PCIe VIP SERDES to the Synopsys PCIe IIP.

https://solvnet.synopsys.com/retrieve/2060917.html

Title:VC VIP: Bringing Up a Link When lane0 is Disabled in the PCIe VIP

https://solvnet.synopsys.com/retrieve/2058463.html
Title:VC VIP: Critical Points about PCIe Link Width Settings
https://solvnet.synopsys.com/retrieve/2057833.html
Title:VC VIP: Enabling the PCIe VIP PHY
https://solvnet.synopsys.com/retrieve/2057812.html
Title:VC VIP: Corrupting the 'COM' Symbol in a Skip Ordered-Set Using the PCIe SVT VIP
https://solvnet.synopsys.com/retrieve/2024177.html
Title:VC VIP: PCIe Model Unable to Negotiate Gen3 Speed Connected to Third-Party PHY Model
https://solvnet.synopsys.com/retrieve/2024163.html
Title:VC VIP: How Does the PCIe VIP Define the txdetectrx Signal?
https://solvnet.synopsys.com/retrieve/1905804.html
Title:VC VIP: Setting the Target & Expected Link Speeds in the PCIe VIP
https://solvnet.synopsys.com/retrieve/1904896.html
Title:VC VIP: Testing the Taking Down of a Link Using the PCIe VIP
https://solvnet.synopsys.com/retrieve/1894725.html
Title:VC VIP: PCIe VIP Not Sending User TS1s
https://solvnet.synopsys.com/retrieve/1894555.html
Title:VC VIP: Setting Valid FS, LF and Equalization/Coefficient Values in the OVM PCIe VIP
https://solvnet.synopsys.com/retrieve/1894110.html
Title:VC VIP: Is There an Example for Generating Hot Reset in the PCIe VIP?
https://solvnet.synopsys.com/retrieve/1887516.html
Title:VC VIP: What Data Does the PCIe VIP Generate when the VIP is a Loopback Master?
https://solvnet.synopsys.com/retrieve/1887492.html
Title:VC VIP: Is There an Example for a PCIe Gen3 VIP with a SerDes Interface?
https://solvnet.synopsys.com/retrieve/1887424.html
Title:VC VIP PCIe: Lane Reversal Testing
https://solvnet.synopsys.com/retrieve/1831170.html
Title:VC VIP PCIe: Polarity Inversion
https://solvnet.synopsys.com/retrieve/1824528.html
Title:PCIe SVT VIP: Detecting if the VIP is Idle
https://solvnet.synopsys.com/retrieve/1624052.html
Title:PCIE SVC / SVT VIP : Initiating L1 Entry
https://solvnet.synopsys.com/retrieve/1545886.html
Title:PCIe SVT VIP: PIPE 4/4.2 Wiring for Gen1/2/3
https://solvnet.synopsys.com/retrieve/1520787.html

Title:Why is the Discovery pcie_svt VIP stuck in the CONFIGURATION_LINK_WIDTH_START state when link number is not zero?

https://solvnet.synopsys.com/retrieve/1483737.html

Title:Configuring the PCIe SVT SerDes VIP model to transmit 0's in its disabled state

https://solvnet.synopsys.com/retrieve/1477040.html

Title:VC VIP: Understanding PLL Recovery Reset Messages From the PCIe SVT VIP.

https://solvnet.synopsys.com/retrieve/1476941.html

Title:MSI/MSI-X with the PCIe SVT VIP

https://solvnet.synopsys.com/retrieve/1450929.html

Title:Discovery PCIe SVT: Selecting PIPE clock rate and width

https://solvnet.synopsys.com/retrieve/1440701.html

Title:VC VIP: PCIE Redo-Equalization in the L0 state

https://solvnet.synopsys.com/retrieve/2261869.html

Title:VC VIP: Reconfiguring the Link Width of the PCIe Model During Run Time

https://solvnet.synopsys.com/retrieve/2263409.html

Title:VC VIP: Configuring SKP Symbols in a SKP Order Set in the PCIe VIP Model

https://solvnet.synopsys.com/retrieve/2298918.html

Title:VC VIP: Keeping the PCIe Model in the L0S State

https://solvnet.synopsys.com/retrieve/2327637.html

Title:VC VIP: Avoiding FIFO Overflow Errors from the PCIe VIP

https://solvnet.synopsys.com/retrieve/2322974.html

Title:VC VIP: Avoiding FIFO Underflow Errors from PCIe VIP

https://solvnet.synopsys.com/retrieve/2322996.html

Title:VC VIP: PCIe VIP Issues RxStandby Error Messages in Recovery.Speed

https://solvnet.synopsys.com/retrieve/2302038.html

Title:VC VIP: Programming a Mid-Sim Reset in PCIe VIP

https://solvnet.synopsys.com/retrieve/2357728.html

Title:VC VIP: PCIe OVM Attributes to Change to Support a 500MHZ PCLK Frequency in Gen3

https://solvnet.synopsys.com/retrieve/2359311.html

Title: VC VIP: Illegal Pipe Interface Request in PCI Express

https://solvnet.synopsys.com/retrieve/2367786.html

Title: VC VIP: PCIe OVM Attributes to Change to Support a 500MHZ PCLK Frequency in Gen3

https://solvnet.synopsys.com/retrieve/2359311.html

Title: VC VIP: Blocking and Transmitting SKP Ordered Sets in the PCIe VIP

https://solvnet.synopsys.com/retrieve/2335767.html

Title: VC VIP: Updating Symbols During User Defined TS1 OS Using the PCIe VIP

https://solvnet.synopsys.com/retrieve/2343320.html

Title: VC VIP: Disabling Scrambling in the PCIE SVT VIP

https://solvnet.synopsys.com/retrieve/2349545.html

Title: VC VIP: Using Real Values for LTSSM Timeouts in the PCIe Model

https://solvnet.synopsys.com/retrieve/2317906.html

Title: VC VIP: Enabling eq_eval During RECOVERY_EQUALIZATION in the PCIe SVT Model

https://solvnet.synopsys.com/retrieve/2343288.html

Title: VC VIP: Configuring SKP Symbols in a SKP Order Set in the PCIe VIP Model

https://solvnet.synopsys.com/retrieve/2298918.html

Title: VC VIP: PCIe Simulation Hangs with DL_Inactive in loopback.active

https://solvnet.synopsys.com/retrieve/2322915.html

Title: VC VIP: Keeping the PCIe Model in the L0S State

https://solvnet.synopsys.com/retrieve/2327637.html

Title: VC VIP: PCIe VIP Issues RxStandby Error Messages in Recovery.Speed

https://solvnet.synopsys.com/retrieve/2302038.html

F.4 Methodology, Testbench, and Debug

Title:VC VIP: PCIe VIP Compile and Work Around for VCS's -debug_acc+pp+dmptf+thread -debug_region=cell+encrypt Switch

https://solvnet.synopsys.com/retrieve/2246895.html

Title:VC VIP: Understanding Why the PCIe VIP Issues Null Object Acess (NOA) Errors After Initial Model Reset

https://solvnet.synopsys.com/retrieve/2061817.html

Title:VC VIP: Writing to PCIe Configuration Space Using Backdoor Methods

https://solvnet.synopsys.com/retrieve/2061679.html

Title:VC VIP: Flagging OVM_ERROR Data Mismatch Errors Using the PCIe VIP

https://solvnet.synopsys.com/retrieve/2061428.html

Title:VC VIP: Disabling Shadow Memory Checking in the PCIe VIP

https://solvnet.synopsys.com/retrieve/2061397.html

Title:VC VIP: Identifying a Solution When the PCIe VIP Cannot Obtain a License

https://solvnet.synopsys.com/retrieve/2061282.html

Title:VC VIP: Injecting Errors Randomly Using the PCIe VIP

https://solvnet.synopsys.com/retrieve/2061209.html

Title:VC VIP: PCIe Configuration and Scoreboarding

https://solvnet.synopsys.com/retrieve/2087662.html

Title:VC VIP: Configuring the PCIe Transaction Log to Show TLP Payload Data

https://solvnet.synopsys.com/retrieve/2060836.html

Title:VC VIP: Specifying PCIe Transaction and Symbol Log Name and Path
https://solvnet.synopsys.com/retrieve/2058441.html
Title:VC VIP: Fixing PCIe Fatal Error -- Exceeding Number of 32-bt Pages
https://solvnet.synopsys.com/retrieve/2057854.html
Title:VC VIP: Using the PCIe SPEC_VER Setting
https://solvnet.synopsys.com/retrieve/2057792.html
Title:VC VIP: What is the Purpose of the PCIe VIP's Active/ Passive Setting?
https://solvnet.synopsys.com/retrieve/2057771.html
Title:All VC VIPs: Model Not Installing into DESIGNWARE_HOME
https://solvnet.synopsys.com/retrieve/1920276.html
Title:VC VIP: VCS Compilation Flows with the PCIe VIP
https://solvnet.synopsys.com/retrieve/1914814.html
Title:VC VIP: Displaying All Payload Data in the PCIe VIP Transaction Log File
https://solvnet.synopsys.com/retrieve/1905837.html
Title:VC VIP: Resolving a FATAL Message When an EP PCIe VIP Negotiates GEN3 With a RC DUT
https://solvnet.synopsys.com/retrieve/1905783.html
Title:VC VIP: Setting the Number of DWORD's Printed in a PCIe VIP Transaction Log
https://solvnet.synopsys.com/retrieve/1904812.html
Title:VC VIP: Message Demotion is Not Working as Expected in the Pcie VIP
https://solvnet.synopsys.com/retrieve/1894682.html
Title:VC VIP: Configuring the PCIe SVT VIP in EP Mode to Support More Than One Function
https://solvnet.synopsys.com/retrieve/1894507.html
Title:VC VIP: Configuring the PCIe SVT VIP in EP Mode to Support More Than One Function
https://solvnet.synopsys.com/retrieve/1894507.html
Title:VC VIP: Disabling Shadow Memory Checking in the PCIe VIP
https://solvnet.synopsys.com/retrieve/1887564.html
Title:VC VIP: Understanding Why Data is Not Written to PCIe VIP Memory
https://solvnet.synopsys.com/retrieve/1887540.html
Title:VC VIP : Reducing License Checkout Time for the PCIe VIP
https://solvnet.synopsys.com/retrieve/1827384.html
Title:VC VIP: Disabling a Simulation with PCIe SVT VIP
https://solvnet.synopsys.com/retrieve/1801304.html
Title:VC VIP: Working Around OVM_DISABLE_ITEM_RECORDING in PCIe Testbenches
https://solvnet.synopsys.com/retrieve/1801125.html
Title:PCIe SVT: Caution with DISPLAY_NAME not matching instance name
https://solvnet.synopsys.com/retrieve/1624024.html

Title:PCIE SVT VIP : DISPLAY_NAME vs. instance name
https://solvnet.synopsys.com/retrieve/1623995.html
Title:Handling DUT port reset with the PCIe SVT VIP
https://solvnet.synopsys.com/retrieve/1598388.html
Title:PCIe SVT: Setting Up and Showcasing Global Shadow
https://solvnet.synopsys.com/retrieve/1588012.html
Title:What is causing the Null object access [NoA] error in my pcie_svt OVM test bench?
https://solvnet.synopsys.com/retrieve/1432988.html
Title:VC VIP: PCIe VIP Encryption Preventing VCS from Collecting Code Coverage
https://solvnet.synopsys.com/retrieve/2266126.html
Title:VC VIP: Preloading Memory in the PCIe Model
https://solvnet.synopsys.com/retrieve/2261886.html
Title:VC VIP: Viewing PCIe SVT Functional Coverage Information
https://solvnet.synopsys.com/retrieve/2284910.html
TitleVC VIP: Getting a Fatal Error Stating 8G is Not Supported
https://solvnet.synopsys.com/retrieve/2272176.html
Title:VC VIP: Interfacing a PCIe OVM Agent to a DUT via model_instance_scope
https://solvnet.synopsys.com/retrieve/2305315.html
Title:VC VIP: Resolving Errors Related to the pli.tab and msglog.o Files in the PCIe VIP
https://solvnet.synopsys.com/retrieve/2334587.html
Title:VC VIP: Resolving "Undefined System Task Call " Error Message in the PCIe VIP
https://solvnet.synopsys.com/retrieve/2359377.html

F.5 Miscellaneous

Title: VC VIP: Sending Payload Over PCIe Serial Link (Big Endian or Little Endian)
https://solvnet.synopsys.com/retrieve/2322544.html
Title: VC VIP: Creating Filters To Exclude Non-Relevant Coverage Reports in the PCIe VIP
https://solvnet.synopsys.com/retrieve/2369320.html
Title: VC VIP: Avoiding FIFO Underflow Errors from the PCIe VIP
https://solvnet.synopsys.com/retrieve/2322996.html
Title: VC VIP: Does the PCIe VIP Support D States?
https://solvnet.synopsys.com/retrieve/2326026.html
Title: VC VIP: Programming a Mid-Sim Reset in PCIe VIP
https://solvnet.synopsys.com/retrieve/2357728.html
Title: VC VIP: Resolving Errors Related to the pli.tab and msglog.o Files in the PCIe VIP
https://solvnet.synopsys.com/retrieve/2334587.html

Title: VC VIP: Resolving "Undefined System Task Call " Error Message in the PCIe VIP

https://solvnet.synopsys.com/retrieve/2359377.html

Title: VC VIP: Options for Controlling PCIe Logging Verbosity from the Command Line

https://solvnet.synopsys.com/retrieve/2357749.html

Title: VC VIP: Addressing Timescale Issues in the PCIe VIP

https://solvnet.synopsys.com/retrieve/2298935.html

Title: VC VIP: Using the PIPE InvalidRequest Signal in the PCIe VIP

https://solvnet.synopsys.com/retrieve/2350924.html

Title: VC VIP: Understanding OVM Warning Message Regarding Shadow Memory Configuration in the PCIeVIP

https://solvnet.synopsys.com/retrieve/2337518.html

Title: VC VIP: Setting the Version of Your PCIe VIP PIPE Interface

https://solvnet.synopsys.com/retrieve/2328799.html

Title: VC VIP: Resolving Memory Allocation Issues in the PCIe SVT VIP

https://solvnet.synopsys.com/retrieve/2343641.html

Title: Configuring the ExpertIO Verilog SVC PCIe VIP for 8-bits PIPE

https://solvnet.synopsys.com/retrieve/2331593.html

Title: VC VIP: Initializing PCIe Memory Space with Random Data

https://solvnet.synopsys.com/retrieve/2327880.html

Title: VC VIP: Avoiding FIFO Overflow Errors from the PCIe VIP

https://solvnet.synopsys.com/retrieve/2322974.html

Title: VC VIP: Locating and Using PCIe SVT Instantiation Files

https://solvnet.synopsys.com/retrieve/2299296.html

G Reporting Problems

G.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.

G.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* steps. You can use the generated debug data for your own debugging, or as debug data you would send to Synopsys support.

G.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 G-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 OVM_HIGH
- starting at time zero (default) until the end of the simulation (default):

```
+svt_debug_opts=inst:/.*endpoint.*/,verbosity:OVM_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 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> DEBUG_OPTS=1 PA=FSDB
```

The 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`.

G.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).

G.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.

G.5.1 VCS

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

```
-debug_access
```

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*.

G.5.2 Questa

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

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

G.5.3 Incisive

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

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

G.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.

G.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.

For SVT, you must set the verbosity to UVM_HIGH/OVM_HIGH.

5. The case submittal tool will display options on how to send the file to Synopsys.

G.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.