

# **VC Verification IP PCI Express UVM User Guide**

---

Version M-2017.03, March 2017



# Copyright Notice and Proprietary Information

© 2017 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

## Destination Control Statement

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

## Disclaimer

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

## Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <http://www.synopsys.com/Company/Pages/Trademarks.aspx>.

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

## Third-Party Links

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

Synopsys, Inc.  
700 E. Middlefield Road  
Mountain View, CA 94043  
[www.synopsys.com](http://www.synopsys.com)

# Contents

Preface .....	7
Chapter 1	
Introduction .....	11
1.1 Introduction .....	11
1.2 Prerequisites .....	12
1.3 Online Class Reference HTML Help .....	12
1.4 Product Overview .....	12
1.5 Key Features .....	12
1.6 Other Supported Features .....	13
1.6.1 Requester, Driver, and Completer Applications .....	13
1.6.2 Methodology Features .....	14
Chapter 2	
Installation and Setup .....	15
2.1 Verifying the Hardware Requirements .....	15
2.2 Verifying Software Requirements .....	16
2.2.1 Platform/OS and Simulator Software .....	16
2.2.2 Synopsys Common Licensing (SCL) Software .....	16
2.2.3 Other Third Party Software .....	16
2.3 Preparing for Installation .....	16
2.4 Downloading and Installing .....	17
2.4.1 Downloading From the Electronic Software Transfer (EST) System (Download Center) .....	17
2.4.2 Downloading Using FTP with a Web Browser .....	17
2.5 What's Next? .....	17
2.6 Licensing Information .....	18
2.6.1 Controlling License Usage .....	18
2.7 Environment Variable and Path Settings .....	19
2.7.1 Simulator-Specific Settings .....	19
2.8 Determining Your Model Version .....	19
2.9 Integrating a VC VIP into Your Testbench .....	19
2.9.1 Installing and Setting Up More than One VIP Protocol Suite .....	21
2.9.2 Updating an Existing Model .....	22
2.10 Including and Importing Model Files into Your Testbench .....	22
2.11 Compile-time and Runtime Options .....	23

## Chapter 3

Creating the PLI Object in 32-bit and 64-bit Simulation Modes .....	25
3.1 Compiling the Messaging PLI for the PCIe Model .....	25
3.2 Compiling the msglog.o and PLI Files .....	26

## Chapter 4

General Concepts .....	29
4.1 Introduction to UVM .....	29
4.2 Active and Passive Mode .....	29
4.3 PCIe UVM Interface .....	30
4.3.1 UVM Components of the PCIe Device Subenvironment .....	30
4.3.2 UVM Components of the PCIe MAC Agent .....	31
4.3.3 Configuration Data Objects .....	31
4.3.4 Status Data Objects .....	32
4.3.5 Sequence Item Data Objects .....	33
4.4 SVT Service Sequence/Sequencer .....	33
4.5 PCIe Gen3 Support .....	34
4.6 Compliance Patterns .....	34

## Chapter 5

Verification Features .....	37
5.1 The Transaction Logger .....	37
5.1.1 Printing TLP Payload Data to a Transaction Log File .....	38
5.1.2 Fields of the Transaction Log Header .....	38
5.2 The Symbol Logger .....	45
5.2.1 Fields of the Symbol Log Header .....	46
5.2.2 Synchronization of Simulation Time Between Transaction Log and Symbol Log .....	48
5.3 Using Native Protocol Analyzer for Debugging .....	50
5.3.1 Introduction .....	50
5.3.2 Enabling the Protocol Analyzer .....	50
5.3.3 Prerequisites .....	50
5.3.4 Compile-Time Options .....	51
5.3.5 Run Time Options .....	51
5.3.6 Invoking Protocol Analyzer .....	51
5.3.7 Limitations .....	52
5.4 Verification Planner .....	52
5.5 Global Shadow Memory .....	52
5.5.1 Global Shadow Memory Classes .....	53
5.5.2 Global Memory Examples .....	55
5.5.3 Multiple Global Shadows .....	56
5.5.4 Disabling Global Shadows .....	57
5.6 Target Memory .....	58
5.6.1 Ignoring Memory Ranges .....	60
5.7 Data Link Monitor .....	61

## Chapter 6

PCIe Verification Topologies .....	65
6.1 Introduction .....	65
6.2 Instantiation Models .....	66
6.3 General Steps In Picking an Instantiation Model .....	68

6.3.1 Flowcharts for Choosing an Instantiation Model .....	69
6.3.2 File Location and includes for Instantiation .....	71
6.4 SERDES Interface .....	72
6.5 PMA Interface .....	74
6.6 PIPE Interface, Phy VIP and MAC DUT .....	75
6.6.1 Using ifdefs with PIPE 4.0 To Select Additional Signals .....	77
6.6.2 Picking an Instantiation Model (SPIPE Options) .....	77
6.7 PIPE Interface, MAC VIP and PHY DUT (MPIPE) .....	82
6.8 PCIe Device and MAC Model Instantiation .....	87
6.8.1 Model Wire Interface Options .....	88
6.9 Configuring the PIPE Data Bus Width .....	88
6.9.1 PIPE 2.1/3 .....	88
6.9.2 PIPE4 and PIPE 4.2 .....	88
6.10 Compile-Time Parameter Settings .....	88
6.11 Instantiating Multiple-root Hierarchies .....	89
6.12 Model Configuration Overview .....	90
6.13 Turning Off Unused Lanes .....	90
6.14 PIPE CLK as Input to the SPIPE Interface .....	91
6.15 PIPE Coefficient Use .....	91
6.15.1 PHY PIPE GetLocalPresetCoefficients Interface ASCII Signals .....	93

## Chapter 7

Using the PCIe Verification IP .....	95
7.1 SystemVerilog UVM Example Testbenches .....	96
7.2 Installing and Running the Examples .....	97
7.2.1 Modifications to Run the Intermediate Example in Gen3 Using x8 with the PIPE Interface ..	98
7.3 Error Message Usage .....	99
7.4 Some Configuration Values to Set .....	100
7.5 UVM Reporting Levels .....	102
7.6 Controlling Verbosity From the Command Line .....	102
7.7 Resetting the PCIe VIP .....	103
7.8 Creating and Using Custom Applications .....	104
7.8.1 Setting Up Application ID Maps .....	104
7.8.2 Using Testbench Sequences to Emulate User Applications .....	105
7.8.3 Waiting for Completions .....	105
7.9 Backdoor Access to Completion Target Configuration Space .....	106
7.9.1 Setting up the Configuration Space for Backdoor Access .....	106
7.10 Setting VIP Lanes for Receiver Detect .....	109
7.11 Using ASCII Signals .....	109
7.11.1 Transaction Layer ASCII Signals .....	109
7.11.2 Data Link Layer ASCII Signals .....	110
7.11.3 Physical Layer ASCII Signals .....	110
7.12 Using the Ordering Application .....	111
7.12.1 Steps to Use the Ordering Application: .....	111
7.13 Using the reconfigure_via_task Call .....	112
7.14 Configuring Trace File Output .....	112
7.15 Setting Coefficient and Preset for Gen3 Equalization .....	113
7.15.1 Enabling Equalization .....	113
7.15.2 Specifying Coefficients, Presets, LF and FS Values .....	114
7.15.3 Preset and Coefficient Tuning Through Windowed Filtering .....	116

7.16	Target Application	118
7.17	Requester Application	120
7.18	What Are Blocking and Non-blocking Reads in PCIe SVT?	121
7.19	Using SKP Ordered Sets	122
7.20	Using Service Class Reset App	124
7.21	Using FLR	128
7.22	Programming Hints and Tips	130
7.22.1	PIPE Polarity	130
7.22.2	Address Translation Services	130
7.22.3	Calls For Analysis Port Set Up and Usage	130
7.22.4	Sequences and the <code>uvm_sequence :: get_response</code> Task	131
7.22.5	Setting the TH and PH Bits Using the Driver Application Class	132
7.22.6	Fast Link Training	132
7.22.7	When to Invoke Service Calls	132
7.22.8	Exceptions and Scrambler Control Bits	133
7.22.9	User TS1 Ordered Set Notes	133
7.23	PCIe VIP Bare COM Support	134
7.23.1	Background	134
7.23.2	Enabling VIP Bare COM transmission (to mimic the system scenario)	134
7.23.3	Enabling VIP Bare COM Reception	134
7.24	Up/Down Configure	135
7.25	Lane Reversal	135
7.26	Lane Reversal with Different Link Width Configurations	136
7.27	User-Supplied Memory Model Interface	138
7.28	External Clocking and Per Lane Clocking for Serial Interface	139
7.28.1	Enabling External Clocking and Per Lane Clocking Modes	139
7.29	Receiver Margining	141
7.29.1	RC (Upstream Component) Lane Margin of Remote PHY in Retimer Using CTRL-SKP (VIP Acting as MAC)	141
7.29.2	EP (Downstream Component) Lane Margin of Remote PHY in Retimer Using CTRL-SKP (VIP Acting as Retimer PHY)	142
7.29.3	RC (Upstream Component) or EP (Downstream Component) Lane Margin of Local PHY Using MBI (VIP Acting as MAC)	143
7.29.4	RC (Upstream Component) or EP (Downstream Component) Lane Margin of Local PHY Using MBI (VIP Acting as PHY)	144

## Chapter 8

PCIe Device Agent	145
8.1 Overview	145
8.2 Configuration	147
8.2.1 Initial Configuration	148
8.2.2 Dynamic Configuration	149
8.3 Status	151
8.4 Sequencers	152
8.4.1 Service Sequencers	153
8.4.2 Transaction Sequencers	155
8.4.3 Virtual Sequencer	155

## Chapter 9

PCIe Agent	161
9.1 Overview	161
9.2 Configuration	162
9.2.1 Initial Configuration	163
9.2.2 Dynamic Configuration (reconfiguration)	163
9.3 Status	163
9.4 Sequencers	163
9.4.1 Service Sequencers	164
9.4.2 Transaction Sequencers	165
9.4.3 Virtual Sequencer	166
Chapter 10	
Using the Transaction Layer	171
10.1 Transaction Layer	171
10.2 Transaction Layer Configuration	172
10.2.1 Verilog Configuration Parameters	175
10.3 Transaction Layer Sequencer and Sequences	176
10.4 Transaction Layer Callbacks and Exceptions	180
10.5 Transaction Layer Status	180
10.5.1 Determining if the Transaction Layer is Idle	180
10.6 Transaction Layer TLMs	181
10.7 Transaction Layer Verilog Interface	181
10.7.1 Transaction Layer Module IOs	181
Chapter 11	
Data Link Layer Features and Classes	183
11.1 Classes and Applications for Using the VIP's Data Link Layer	183
11.2 Additional Documentation on DL Programming Tasks	183
11.3 Class Elements of the Link Layer	184
11.4 Power Management	185
11.4.1 ASPM	185
11.4.2 L0s Entry	185
11.4.3 PM	186
11.4.4 VIP PM/ ASPM Checks	187
11.5 Component Class svt_pcie_dl	189
11.6 Configuration class svt_pcie_dl_configuration	192
11.6.1 Members and Features	192
11.7 Status class svt_pcie_dl_status	203
11.8 Service Class svt_pcie_dl_service	206
11.8.1 Members and Features	206
Chapter 12	
PHY Layer Features and Classes	209
12.1 Classes and Applications for Using the VIP's PHY Layer	209
12.2 Additional Documentation on PHY Programming Tasks	209
12.3 External Tx Bit Clk Use Model	209
12.4 Service Class svt_pcie_pl_service	211
12.5 UVM Component Class svt_pcie_pl	220
12.6 PHY Layer Configuration Class	221
Chapter 13	

Using the Driver Application .....	243
13.1 Introduction .....	243
13.2 Driver Application Configuration .....	244
13.3 Verilog Configuration Parameters and Tasks .....	247
13.3.1 Compile-time Verilog Configuration Parameters .....	247
13.3.2 Runtime Configuration Parameters .....	247
13.3.3 Runtime Verilog Tasks .....	248
13.4 Driver Application Sequencer and Sequences .....	248
13.4.1 Service Sequences .....	248
13.4.2 Transaction Sequences .....	250
13.5 Driver Application Callbacks and Exceptions .....	252
13.5.1 Transaction Layer Exceptions .....	252
13.6 Driver Status .....	252
13.6.1 Determining if the Driver Application is Idle .....	252
13.7 Driver Application Events .....	253
13.8 Driver Application TLMs .....	253
13.9 Driver Application Transactions .....	253
13.9.1 Blocking and Non-Blocking Transactions .....	254
Chapter 14	
Functional Coverage .....	255
14.1 Enabling Functional Coverage .....	255
14.2 Class Structure and Callbacks .....	255
14.3 Overriding the Default Coverage Class .....	256
14.3.1 Overriding With UVM .....	256
14.3.2 Overriding for SystemVerilog Users .....	256
14.4 Transaction Layer .....	257
14.4.1 Transaction Layer Functional Coverage .....	257
14.4.2 Transaction Layer Callbacks .....	258
14.5 Data Link Layer .....	259
14.5.1 Data Link Layer Functional Coverage .....	259
14.5.2 Link Layer Callbacks .....	276
14.6 Physical Layer .....	281
14.6.1 Physical Layer Functional Coverage .....	281
14.6.2 Physical Layer Callbacks .....	285
14.7 PIPE Interface .....	287
14.7.1 PIPE Functional Coverage .....	287
14.7.2 PIPE Interface Callbacks .....	287
Chapter 15	
M-PHY Adapter Layer .....	289
15.1 Overview .....	289
15.2 Configuration Classes .....	292
15.3 Signal Interfaces .....	293
15.4 Data Factory Objects .....	293
15.5 Exception List Factories .....	293
15.6 Error injection .....	293
15.7 Transactions .....	293
15.7.1 Data Transactions .....	294
15.7.2 Service Transactions .....	294



15.8 Using the M-PCIe Interface .....	295
15.9 Shared Status .....	295
15.10 Configuring the PCIe M-PHY Adapter for an RMMI Interface .....	296
15.10.1 Configuring the VIP for a DUT Subsystem That Does Not Include the Link .....	296
15.10.2 Configuring the VIP for a DUT Subsystem That Includes the Link .....	297
15.10.3 cfg Attribute Settings .....	297
15.10.4 Quick Discovery Configuration Mode .....	298
Chapter 16 .....	
Using Callbacks .....	299
16.1 Introduction .....	299
16.2 How Callbacks Are Used .....	299
16.2.1 Other Uses for Callbacks .....	300
16.2.2 Callback Hints .....	300
16.2.3 When Not to Use a Callback .....	301
16.3 Detailed Usage .....	301
16.3.1 A Basic Testcase .....	301
16.4 Advanced Topics in Callbacks .....	303
16.4.1 Exceptions – a “Delayed” Transaction Modification Request .....	303
16.4.2 Creating an Exception .....	303
16.4.3 Creating a Factory Exception .....	304
16.4.4 Error Injection Using Application Layer TX Callbacks .....	305
16.4.5 A More Comprehensive Example .....	308
16.5 SVT VIP Callbacks Reference .....	312
16.6 Transaction Layer Callbacks and Exceptions .....	315
16.6.1 Transaction Layer Exceptions .....	316
16.7 Datalink Layer Callbacks and Exceptions .....	316
16.7.1 Datalink Layer Exceptions .....	316
16.8 Physical Layer Callbacks and Exceptions .....	317
16.8.1 Physical Layer Exceptions .....	318
16.9 Controlling Completion Timing and Size Using Callbacks .....	319
16.9.1 Controlling Delay for the Current Completion .....	319
16.9.2 Controlling Delay for the Next Completion .....	319
16.9.3 Controlling Size for the Next Completion .....	320
Chapter 17 .....	
Partition Compile and Precompiled IP .....	323
17.1 Use Model .....	323
17.2 The “vcspcvlog” Simulator Target in Makefiles .....	324
17.3 The “vcsmxpcvlog” Simulator Target in Makefiles .....	324
17.4 The “vcsmxpcvlog” Simulator Target in Makefiles .....	324
17.5 Partition Compile and Precompiled IP Implementation in Testbenches with Verification IPs ..	325
17.6 Example .....	325
Chapter 18 .....	
Passive Monitor .....	327
18.1 Supported Protocol Features .....	328
18.2 Early Adopter (EA) Supported Features .....	328
18.3 Features Not Supported .....	328
18.4 UVM SVT Agent as Passive Monitor .....	328

18.5 Methodology Features .....	329
18.6 Usage .....	329
18.7 Types of Monitor Configuration Objects .....	331
18.8 Functional Coverage .....	331
18.8.1 LTSSM State Coverage .....	331
18.8.2 Error Check Coverage .....	331
18.8.3 Transaction Coverage .....	331
18.8.4 Coverage Callback Classes .....	331
18.8.5 Enabling Default Coverage .....	332
18.8.6 Coverage Shaping and Control .....	332
18.9 Interfaces and modports .....	332
18.10 Programming the Passive Monitor .....	334
18.10.1 Synopsys Passive Monitor Example .....	335
18.10.2 Configuring the Monitor in the Example Testbench .....	335
18.10.3 Callbacks .....	336
18.11 Configuration Space Tracking .....	337
18.11.1 Configuration Space Features .....	337
18.11.2 Programming Passive Monitor's Configuration Space .....	338
18.11.3 Passive Monitor Configuration Space Modes .....	340
18.11.4 Test Bench access to PCIe Passive Monitor Configuration Space. ....	341
18.11.5 Example Programing Monitor Configuration Space .....	343
18.11.6 In CFG_SPACE_ENUMERATION_UPDATE Mode .....	349
18.12 Equalization Support .....	349
18.12.1 Equalization Configuration Attributes .....	349
18.12.2 Equalization Status Attributes .....	351
Chapter 19 .....	
Integrated Planning for VC VIP Coverage Analysis .....	353
19.1 Use Model .....	353
Appendix A	
Protocol Checks .....	358
Appendix B	
PCIe PIE-8 Interface .....	361
B.1 Supported Interface Signals .....	361
B.2 Configuration Parameters .....	363
B.3 Status Class PIE8 Members .....	364
B.4 PHY PIE-8 ASCII Signals .....	365
B.5 PHY PIE-8 Internal Signals .....	365
B.6 PIE-8 Protocol Check "MSGCODEs" .....	366
Appendix C	
PCIe Compile-time Parameters .....	369
C.1 Model Parameters .....	369
C.2 Driver Application Parameters .....	370
C.3 Requester Parameters .....	370
C.4 Completion Target Parameters .....	371
C.5 Memory Target Parameters .....	372
C.6 Transaction Layer Parameters .....	372
C.7 Data Link Layer Parameters .....	373

C.8 Physical Layer Parameters .....	373
C.8.1 General Parameters .....	373
C.8.2 Physical Layer LTSSM-specific Parameters .....	373
C.8.3 Equalization Parameters .....	375
C.9 Physical Coding Sublayer (PCS) Parameters .....	375
C.10 Serializer/Deserializer (SERDES) Parameters .....	376
Appendix D	
Verilog Task/Parameter to SVT Class Mapping .....	377
D.1 Transaction Layer Verilog Tasks and Parameters to UVM Class Members Map .....	377
D.1.1 Transaction Layer Verilog Task to UVM Class Member Map .....	377
D.1.2 Transaction Layer Verilog Parameters to UVM Class Members Map .....	378
D.2 Data Link Layer Verilog Tasks and Parameters to UVM Class Member Maps .....	379
D.2.1 Data Link Layer Verilog Task to UVM Class Member Map .....	379
D.2.2 Data Link Layer Verilog Parameter to UVM Class Member Map .....	380
Appendix E	
ECN Support .....	383
Appendix F	
SolvNet PCIe VIP Articles .....	385
F.1 Transaction Layer .....	385
F.2 Data Link Layer .....	387
F.3 PHY Layer .....	388
F.4 Methodology, Testbench, and Debug .....	391
F.5 Miscellaneous .....	393
Appendix A	
Reporting Problems .....	395
A.1 Introduction .....	395
A.2 Debug Automation .....	395
A.3 Enabling and Specifying Debug Automation Features .....	395
A.4 Debug Automation Outputs .....	397
A.5 FSDB File Generation .....	398
A.5.1 VCS .....	398
A.5.2 Questa .....	398
A.5.3 Incisive .....	398
A.6 Initial Customer Information .....	398
A.7 Sending Debug Information to Synopsys .....	398
A.8 Limitations .....	399



# Preface

---

## About This Manual

This manual contains installation, setup, and usage material for SystemVerilog UVM users of the VC PCI Express, and is for design or verification engineers who want to verify PCIe operation using a UVM testbench written in SystemVerilog. Readers are assumed to be familiar with PCIe, object oriented programming (OOP), SystemVerilog, and Universal Verification Methodology (UVM) techniques.

## Manual Organization

The chapters of this databook are organized as follows:

- Chapter 1, [“Introduction”](#), introduces the VC PCI Express and its features.
- Chapter 2, [“Installation and Setup”](#), describes system requirements and provides instructions on how to install, configure, and begin using the VC PCI Express.
- 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 UVM 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 10, [“Using the Transaction Layer”](#), describes features of the Transaction Layer and how to use them.
- 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.
- Appendix A, [“Protocol Checks”](#), outlines the process for working through and reporting VC PCI Express issues.

- 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 G, “Reporting Problems”, outlines the process for working through and reporting VC PCI Express issues.

## Web Resources

- Documentation through SolvNet: <https://solvet.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:

- Open a Case through SolvNet.
  - Go to <https://onlinecase.synopsys.com/Support/OpenCase.aspx> and provide the requested information, including:
    - Product: **Verification IP**
    - Sub Product: **pcie\_svt**
    - Tool Version: **VIP\_version**
    - Fill in the remaining fields according to your environment and your issue.
  - If applicable, provide the information noted in Appendix G, “Reporting Problems” on page 367.
- Send an e-mail message to [support\\_center@synopsys.com](mailto:support_center@synopsys.com).
  - Include the Product name, Sub Product name, and Tool Version (as noted above) in your e-mail so it can be routed correctly.
  - If applicable, provide the information noted in Appendix G, “Reporting Problems” on page 367.
- Telephone your local support center.
  - North America:  
Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
  - All other countries:  
<http://www.synopsys.com/Support/GlobalSupportCenters>





# 1

## Introduction

---

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

This chapter discusses the following topics:

- [“Introduction”](#) on page 11
- [“Prerequisites”](#) on page 12
- [“Online Class Reference HTML Help”](#) on page 12
- [“Product Overview”](#) on page 12
- [“Other Supported Features”](#) on page 13

### 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 Universal Verification Methodology (UVM). This approach leverages advanced verification technologies and tools that provide:

- Protocol functionality and abstraction
- Constrained random verification
- Functional coverage
- Rapid creation of complex tests
- 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 UVM.

## 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_uvm_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 UVM VIP is a suite of UVM-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 UVM 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/ UVM 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 UVM 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
  - UVM 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 PCI Express. When you complete this checklist, the provided example testbench will be operational and the VC PCI Express will be ready to use.

The checklist consists of the following major steps:

1. [“Verifying the Hardware Requirements”](#) on page 15
2. [“Verifying Software Requirements”](#) on page 16
3. [“Preparing for Installation”](#) on page 16
4. [“Downloading and Installing”](#) on page 17
5. [“What’s Next?”](#) on page 17

This chapter contains the following additional topics:

- [“Licensing Information”](#) on page 18
- [“Environment Variable and Path Settings”](#) on page 19
- [“Determining Your Model Version”](#) on page 19
- [“Integrating a VC VIP into Your Testbench”](#) on page 19
- [“Including and Importing Model Files into Your Testbench”](#) on page 22
- [“Compile-time and Runtime Options”](#) on page 23

**Note**

If you encounter any problems with installing the VC PCI Express, see [“Customer Support”](#) on page 9.

## 2.1 Verifying the Hardware Requirements

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

- 400 MB available disk space for installation
- 1 GB available swap space

- 1 GB RAM (physical memory) recommended
- FTP anonymous access to ftp.synopsys.com (optional)

## 2.2 Verifying Software Requirements

The VC PCI Express is qualified for use with certain versions of platforms and simulators. This section lists software that the VC PCI Express 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 PCI Express Release Notes](#)*

### 2.2.2 Synopsys Common Licensing (SCL) Software

- The SCL software provides the licensing function for the VC PCI Express. Acquiring the SCL software is covered here in the installation instructions in "[Licensing Information](#)" on page 18.

### 2.2.3 Other Third Party Software

- **Adobe Acrobat:** VC PCI Express documents are available in Acrobat PDF files. You can get Adobe Acrobat Reader for free from <http://www.adobe.com>.
- **HTML browser:** VC PCI Express 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](mailto: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](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 SolvNet 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, refer to ["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](#)” on page 18
- “[Environment Variable and Path Settings](#)” on page 19
- “[Determining Your Model Version](#)” on page 19
- “[Integrating a VC VIP into Your Testbench](#)” on page 19

## 2.6 Licensing Information

The PCI Express 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-SOC-LIBRARY-SVT
- VIP-LIBRARY-SVT and DesignWare-Regression
- 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.

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](#)” on page 19.

### 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 PCI Express 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 UVM Users

1. UVM users must set the value of the `UVM_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 can be done in a run script or on the command line, as follows:

```
+define+UVM_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 UVM version 1.1b, 1.1c, or 1.1d, then you must define the `UVM_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 UVM automatic transaction begin and end event triggering feature. If `UVM_DISABLE_AUTO_ITEM_RECORDING` is not defined the VIP issues a fatal error.

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

When the `UVM_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 intermediate UVM example:  
% $DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -e  
    pcie_svt/tb_pcie_svt_uvm_intermediate_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_uvm_basic_sys  
usage: run_pcie_svt_uvm_basic_sys [-32] [-verbose] [-debug] [-waves] [-clean] [-  
nobuild] [-norun] [-pa] <scenario> <simulator>
```

where <scenario> is one of: all base\_completer\_callback\_pipe\_test  
 base\_completer\_callback\_pma\_test base\_completer\_callback\_serdes\_test base\_pipe\_test  
 base\_pma\_test base\_serdes\_test directed\_pipe\_test directed\_pma\_test  
 directed\_serdes\_test requester\_traffic\_pipe\_test

<simulator> is one of: vcsvlog vcsmxvlog mtivlog vcsmxpcvlog vcsmxpipvlog  
 ncvlog vcspcvlog vcsscvclog vcsvhdl ncmvlog

- 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|dve|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|dve|dump] [NOBUILD=1] [PA=1] [<scenario> ...]
Valid simulators are: vcsvlog vcsmxvlog mtivlog vcsmxpcvlog vcsmxpipvlog ncvlog
vcspcvlog vcsscvclog vcsvhdl ncmvlog
Valid scenarios are: all base_completer_callback_pipe_test
base_completer_callback_pma_test base_completer_callback_serdes_test base_pipe_test
base_pma_test base_serdes_test directed_pipe_test directed_pma_test
directed_serdes_test requester_traffic_pipe_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 an existing model, do the following:

1. Install the model to the same location as your other VIPs by setting the \$DESIGNWARE\_HOME environment variable.
2. Issue the following command using design\_dir as the location for your project directory.

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

3. You can also update your design\_dir by specifying the version number of the model.

```
% dw_vip_setup -path design_dir -add <model-name>_svt -v 3.50a model_vmt -v 3.50a
```

## 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. Following is a code list of the includes and imports for PCIe:

```
/* include uvm package before VIP includes, If not included elsewhere*/
#include "svt_pcie.uvm_pkg"

/** Include the PCIe SVT UVM package */
#include "svt_pcie.uvm.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 UVM Package */
import uvm_pkg::*;
`include "uvm_macros.svh"

/** Import the SVT UVM Package */
import svt_uvm_pkg::*;

/** 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+UVM_PACKER_MAX_BYTES=1500000
Optional: -timescale=1ns/1ps
Required: +define+SVT_PCIE_INCLUDE_USER_DEFINES
Required: +define+SYNOPSYS_SV
```

`vcs_run_options`

Required: `+UVM_TESTNAME=$scenario`

Note: “scenario” is the uvm 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:

- [“Compiling the Messaging PLI for the PCIe Model”](#) on page 25
- [“Compiling the msglog.o and PLI Files”](#) on page 26

### 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 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 veriusuer file, which registers the PLI information with the simulator.

#### a. For VCS|VCS MX:

```
${VCS_HOME}/bin/veriusuer_to_pli_tab -include ${VCS_HOME}/include  
${DESIGNWARE_HOME}/vip/svt/pcie_svt/latest/C/src/veriusuer.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_veriusuer.c  
-o dyn_veriusuer.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_veriusuer.c  
-o dyn_veriusuer.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\_veriusuer files to generate the PLI file.

#### a. For MTI:

Once the dyn\_veriusuer.o file is generated from step 3, run this compile:

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

#### b. For NC:

Once the dyn\_veriusuer.o file is generated from step 3, run this compile:

```
cc ${ldflags_dyn} -o dyn_ncvpli.so msglog.o dyn_veriusuer.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 UVM environment, and its user interface. This chapter discusses the following topics:

- [“Introduction to UVM”](#) on page 29
- [“Active and Passive Mode”](#) on page 29
- [“PCIe UVM Interface”](#) on page 30
- [“PCIe Gen3 Support”](#) on page 34

### 4.1 Introduction to UVM

UVM 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 UVM 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 UVM. For more information:

- For the IEEE SystemVerilog standard, see:
  - IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language
- For essential guides describing UVM as it is represented in SystemVerilog, along with a Class Reference, see:
  - *Universal Verification Methodology (UVM) 1.0 User’s Manual* at `$VCS_HOME/doc/UserGuide/pdf/uvm_users_guide_1.0.pdf`

### 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 UVM Interface

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

### **svt\_pcie\_device\_agent**

The **svt\_pcie\_device\_agent** object defines a UVM agent that contains the following `uvm_components` for PCIe applications:

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

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

### **svt\_pcie\_agent**

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

#### 4.3.1 UVM Components of the PCIe Device Subenvironment

- **svt\_pcie\_driver\_app** – A `uvm_component` object that implements the PCIe Driver application, which transmits PCIe packets to PCIe transaction layer.
- **svt\_pcie\_requester\_app** – A `uvm_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 `uvm_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 `uvm_component` object that supports the IO segment of the PCIe system.
- **svt\_pcie\_mem\_target** – A `uvm_component` object that supports the Memory segment of the PCIe system.

- **svt\_pcie\_cfg\_database** – A `uvm_component` object that supports the Configuration space of the PCIe system.
- **svt\_pcie\_global\_shadow** – A `uvm_component` object that implements the PCIe Device system IO, Memory and Configuration spaces.

### 4.3.2 UVM Components of the PCIe MAC Agent

- **svt\_pcie\_tl** – A `uvm_component` object that implements the PCIe Transaction Layer.
- **svt\_pcie\_dl** – A `uvm_component` object that implements the PCIe Link Layer.
- **svt\_pcie\_pl\_phy** – A `uvm_component` object that implements the PCIe Physical Layer.

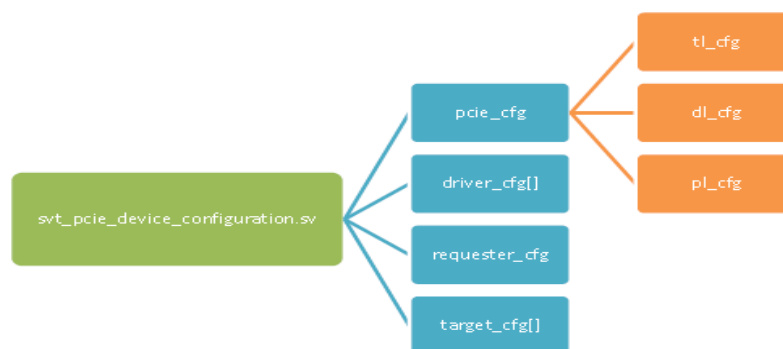
### 4.3.3 Configuration Data Objects

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**



In your environment you can access these variables from `pcie_env_cfg` `env_cfg` as shown below:

```
env_cfg.m_pcie_vip_cfg.pcie_cfg.tl_cfg.<=>;  
env_cfg.m_pcie_vip_cfg.pcie_cfg.dl_cfg.<=>;  
env_cfg.m_pcie_vip_cfg.pcie_cfg.pl_cfg.<=>;  
  
env_cfg.m_pcie_vip_cfg.driver_cfg[0].<=>;  
env_cfg.m_pcie_vip_cfg.requester_cfg.<=>;  
env_cfg.m_pcie_vip_cfg.target_cfg[0].<=>;
```

#### 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 UVM 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`

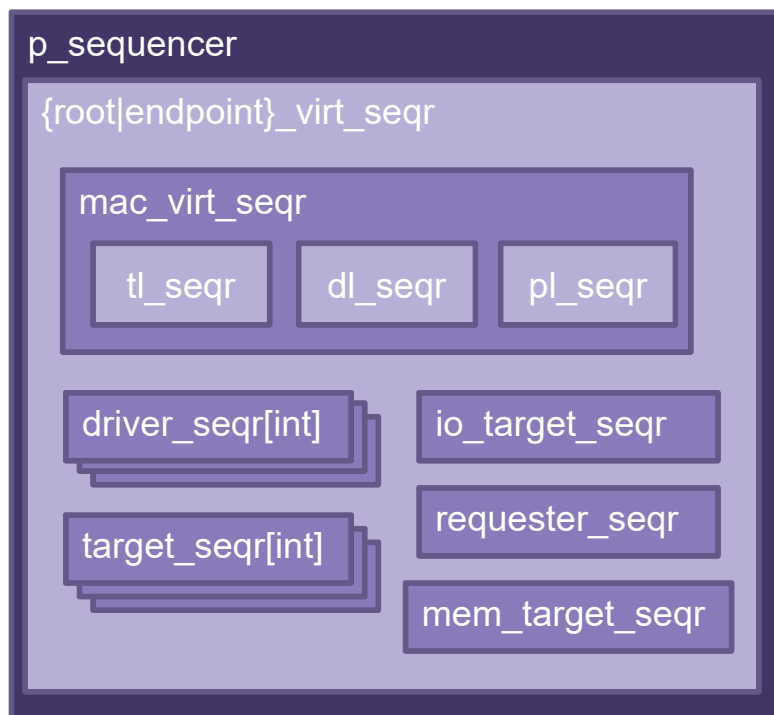
## 4.4 SVT Service Sequence/Sequencer

Each component in the agent has its own service sequencer. All SVT sequences are derived from `uvm_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:

```
`uvm_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”](#) on page 37
- [“The Symbol Logger”](#) on page 45
- [“Using Native Protocol Analyzer for Debugging”](#) on page 50
- [“Verification Planner”](#) on page 52
- [“Global Shadow Memory”](#) on page 52
- [“Target Memory”](#) on page 58
- [“Data Link Monitor”](#) on page 61

### 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 uvm_test;
. . .
virtual function void task build_phase(uvm_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
    . . .
endclass
```

```
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 intermediate 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 uvm_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 uvm_test;
...
  virtual function void build_phase(uvm_phase phase);
    `uvm_info("build_phase", "Entered...", UVM_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	<b>CfgRd0</b>	<b>0x0001/13</b>
vip0	159140.00	159160.00	T	<b>MRd32</b>	<b>0x0001/1f</b>

*DLLP*

Reporter	Start Time (ns)	End Time (ns)	Dir	TLP Type DLLP Type	R_ID/Tag   ST
vip0	133328.00	133328.00	R	<b>INITFC2_P_VCO</b>	
vip0	133772.00	133772.00	R	<b>ACK</b>	

**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 >	<b>Memory request:</b> This field represents the memory address.
< address >	<b>IO request:</b> This field represents the IO address.
BDF: < > R: < > or BDF: < > O: < >	<b>Configuration request:</b> "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 >	<b>Message request:</b> This field represent the message routing as defined in Table 2-18 of the PCIe Specification.
ID: < > Stat: < >	<b>Completion request:</b> "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   End Time   Dir   TLP Type   R\_ID/Tag   Seq   TC   T   P   I   R   N   Address

	(ns)	(ns)		DLLP Type	ST	Num	VC	H	H	D	O	S	Reg#/MsgRt/Cpl
										O			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	I D	R O	N S	Address Reg#/MsgRt/Cpl HdrFC DataFC
vip0	133596	133596	R	ACK		2							116 230
vip0	133124	133124	T	INITFC1_P_VCO									102 1024

**Field:** BE | ST

BC

MCode

**Description:**

This field has multiple representations of a TLP transaction.

TLP Field	Description
< byte enable >	<b>Memory request/IO request/Configuration request:</b> This field represents the byte enable.
< steering tag >	<b>Memory request:</b> When the TH field has a value of "1", this field represents the steering tag value.
BC: < >	<b>Completion request:</b> BC represents the byte count.
< message code >	<b>Message request:</b> 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	I D	R O	N S	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:

*MWrr32*

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 uvm_test;
    . . .
    virtual function void task build_phase(uvm_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

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

Symbol	Description
?	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.
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 [Table 5-2](#). Additional encodings for link operation at 8GT/s are listed in [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
End Time (ns)	TIME with "END" symbol

Example 1:

*Transaction Log*

```

vip0      16332.00  16336.00  T  INITFC2_P_VCO      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 capabilities. 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

You can enable FSDB as per your simulator's instructions. For VCS, you can specify the PLI files directly or use the FSDB specific switches.

Use any of the following options to enable Verdi PLI libraries:

```
-P ${NOVAS_HOME}/share/PLI/VCS/${novas_platform} /novas.tab ${NOVAS_HOME}/  
share/PLI/VCS/${novas_platform}/pli.a
```

OR

```
-debug_access+all
```

### 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
```



#### Note

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 and select the Protocol Analyzer option in the main window 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)

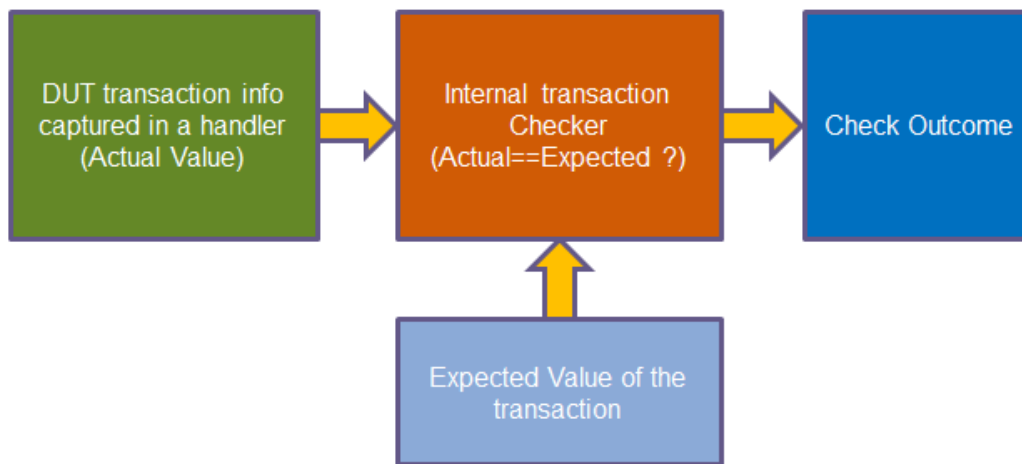
The VIP captures TLPs as they go on the wire and then are passed to the shadow's transaction handler which decodes all relevant transactions and updates the expected global shadow state. For example, outbound MemWrite TLPs are inspected and (if appropriate) written to the global shadow memory. This shadow 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 predictable 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 UVM 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 `uvm_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.
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_CAP )	Cfg Cap write.
READ_CFG_CAP( `SVT_PCIE_GLOBAL_SHADOW_SERVICE_READ_CFG_CAP )	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.

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

Member	Feature/Usage
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.
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;
....

`uvm_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;

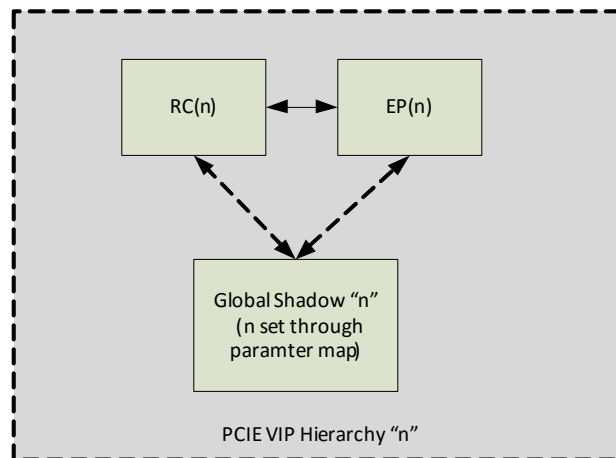
`uvm_config_db(mem_range_seq,p_sequencer.endpoint_virt_seqr.global_shadow,
{disp_stat_seq.service_type == svt_pcie_global_shadow_service::DISPLAY_STATS;} );

```

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

### 5.5.4 Disabling Global Shadows

Global shadow is a memory component used in PCIe SVT VIP to compare write data (MW<sub>r</sub> or IO<sub>W</sub><sub>r</sub> payload) with read data (CplD payload in response of MR<sub>d</sub> or IO<sub>R</sub><sub>d</sub>).

Global shadow component is instantiated in the testbench as shown in the following code snippet:

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

If you want to disable global shadow at run time, you must invoke the following function:

```
<device_cfg>.enable_all_global_shadow_vars(0);
```

Alternatively, you can set following configuration attributes per component to enable/disable shadow memory checking:

```
<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;
```

## 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”](#) on page 52 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.



**Table 5-6 Memory Target Service Types**

Service	Description
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 initial 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.

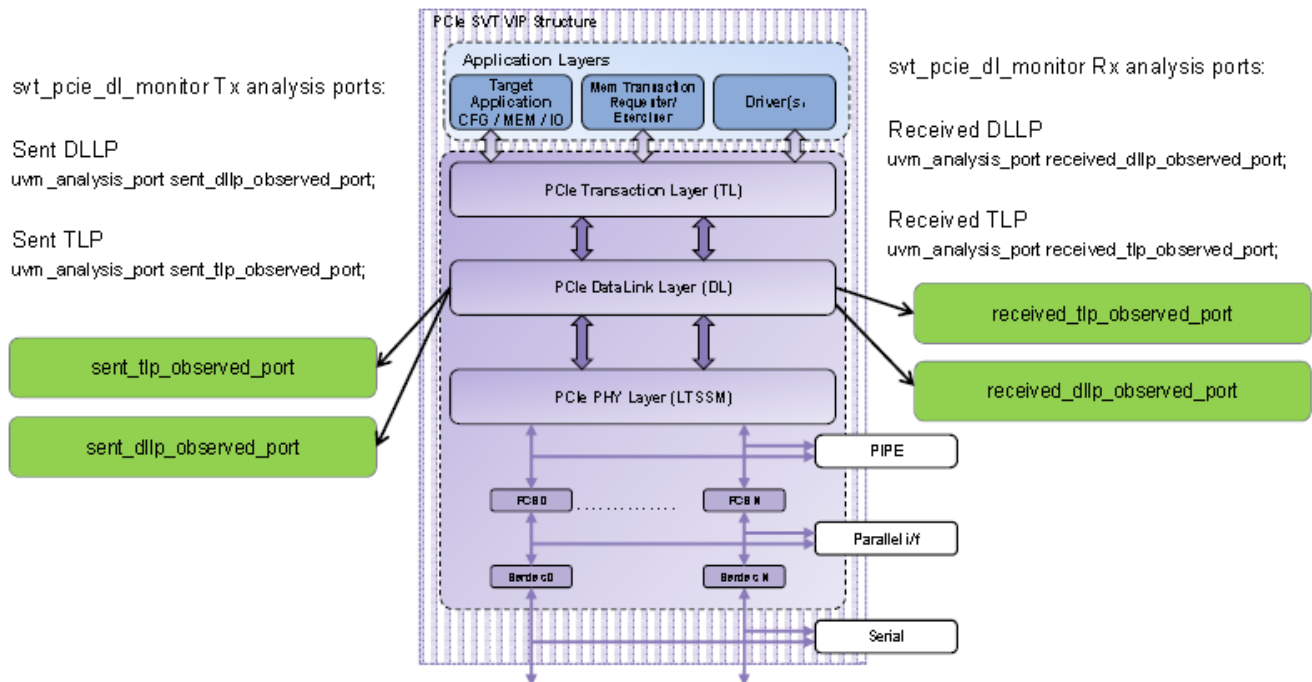
The configuration in the class `svt_pcie_target_app_configuration` for this is `uninit_mem_read_resp`, which can be set with various `UNINIT_MEM_READ_RESP_*` values.

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 and DLLPs.

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



The PCIe UVM 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:

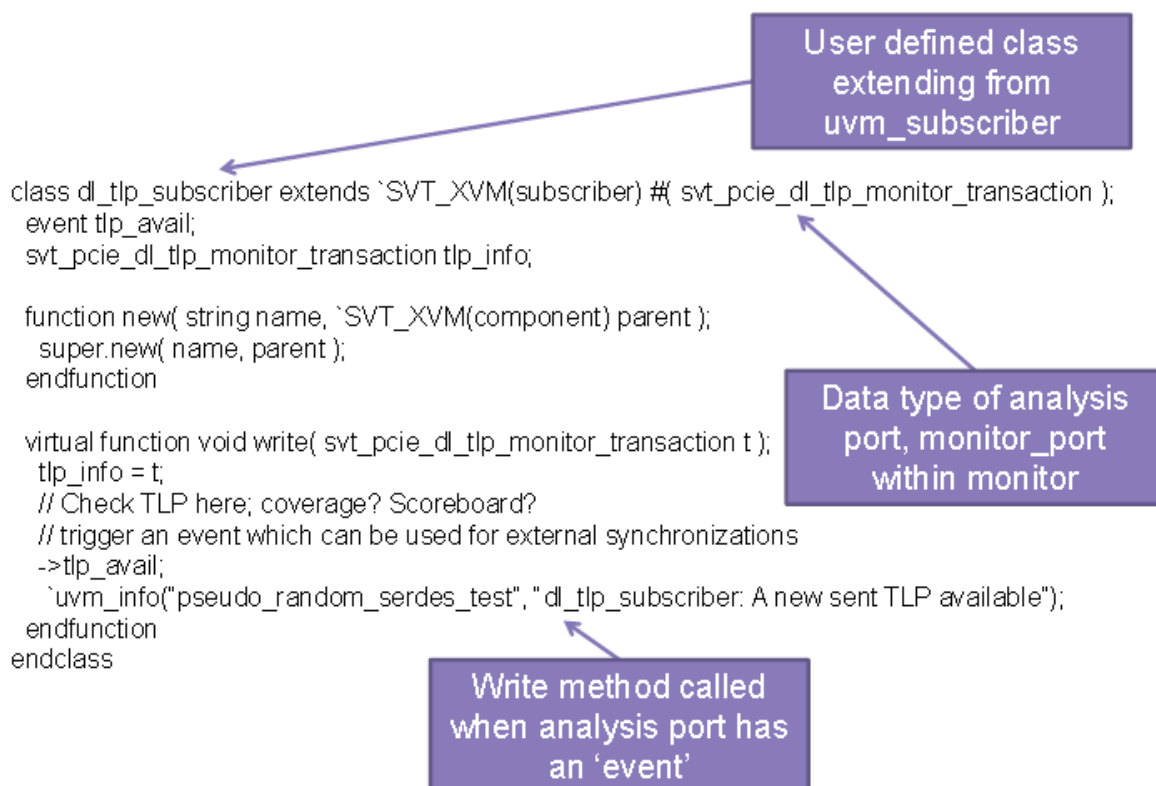
- [\\$DESIGNWARE\\_HOME/vip/svt/pcie\\_svt/latest/doc/pcie\\_svt\\_uvm\\_class\\_reference/html/class\\_svt\\_pcie\\_dl\\_configuration.html#item\\_received\\_tlp\\_interface\\_mode](#)
- [\\$DESIGNWARE\\_HOME/vip/svt/pcie\\_svt/latest/doc/pcie\\_svt\\_uvm\\_class\\_reference/html/class\\_svt\\_pcie\\_dl\\_configuration.html#item\\_sent\\_tlp\\_interface\\_mode](#)

Once enabled these ports can be used to subscribe to transaction being sent/received by the VIP model with the use of UVM 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 uvm\_subscriber extension  
Goal: uvm\_subscriber::write() will be called when the TLP is sent
5. Add a subscriber to your uvm\_test
  - a. Build phase : new your uvm\_subscriber class
  - b. Connect phase : connect to the uvm\_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 pseudo_random_serdes_test extends pcie_device_base_test ;
```

```
  dl_tlp_subscriber sent_tlp_subscriber;
```

```
  ...
```

```
  virtual function void build_phase(uvm_phase phase);
    `uvm_info("build_phase", "Entered...", UVM_LOW)
    super.build_phase(phase);
    sent_tlp_subscriber = new( "sent_tlp_subscriber", this );
    `uvm_info("build_phase", "Exiting...", UVM_LOW)
  endfunction: build_phase
```

```
  ....
```

```
  ...
```

```
  virtual function void connect_phase( uvm_phase phase );
    super.connect_phase( phase );
    env.endpoint.port.dl_monitor.sent_tlp_observed_port.connect( sent_tlp_subscriber.analysis_export );
  endfunction
```

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

from log....

Add subscriber to your  
uvm\_test  
implementation

Within the build\_phase,  
create an instance of  
your subscriber

"connect" to the  
monitor within the  
connect\_phase

Use analysis\_export to  
make the connection



## 6

# PCIe Verification Topologies

---

This chapter shows the supported interfaces of the PCIe VIP. It contains the following sections:

- [“Introduction” on page 65](#)
- [“General Steps In Picking an Instantiation Model” on page 68](#)
- [“SERDES Interface” on page 72](#)
- [“PMA Interface” on page 74](#)
- [“PIPE Interface, Phy VIP and MAC DUT” on page 75](#)
- [“PIPE Interface, MAC VIP and PHY DUT \(MPIPE\)” on page 82](#)
- [“PCIe Device and MAC Model Instantiation” on page 87](#)
- [“Configuring the PIPE Data Bus Width” on page 88](#)
- [“Compile-Time Parameter Settings” on page 88](#)
- [“Instantiating Multiple-root Hierarchies” on page 89](#)
- [“Model Configuration Overview” on page 90](#)
- [“Turning Off Unused Lanes” on page 90](#)
- [“PIPE CLK as Input to the SPIPE Interface” on page 91](#)
- [“PIPE Coefficient Use” on page 91](#)

## 6.1 Introduction

The PCI Express Transceiver VIP is a bus functional model that can generate and respond to PCI Express transactions. It can be used to verify PCI Express endpoint, switch, or root complex devices.

[Figure 6-1](#) shows all the supported layers in the PCIe VIP. The layers below the PHY layer will be supported based on the interface selected for the VIP. If the SERDES interface is selected, then the PCS and SERDES layers are included in the VIP along with the other three layers. If the Parallel interface is selected, then the PCS layer gets included along with the other three layers. For the PIPE interface, the VIP will contain the Transaction Layer, the Data Link Layer, and the PHY Layer.

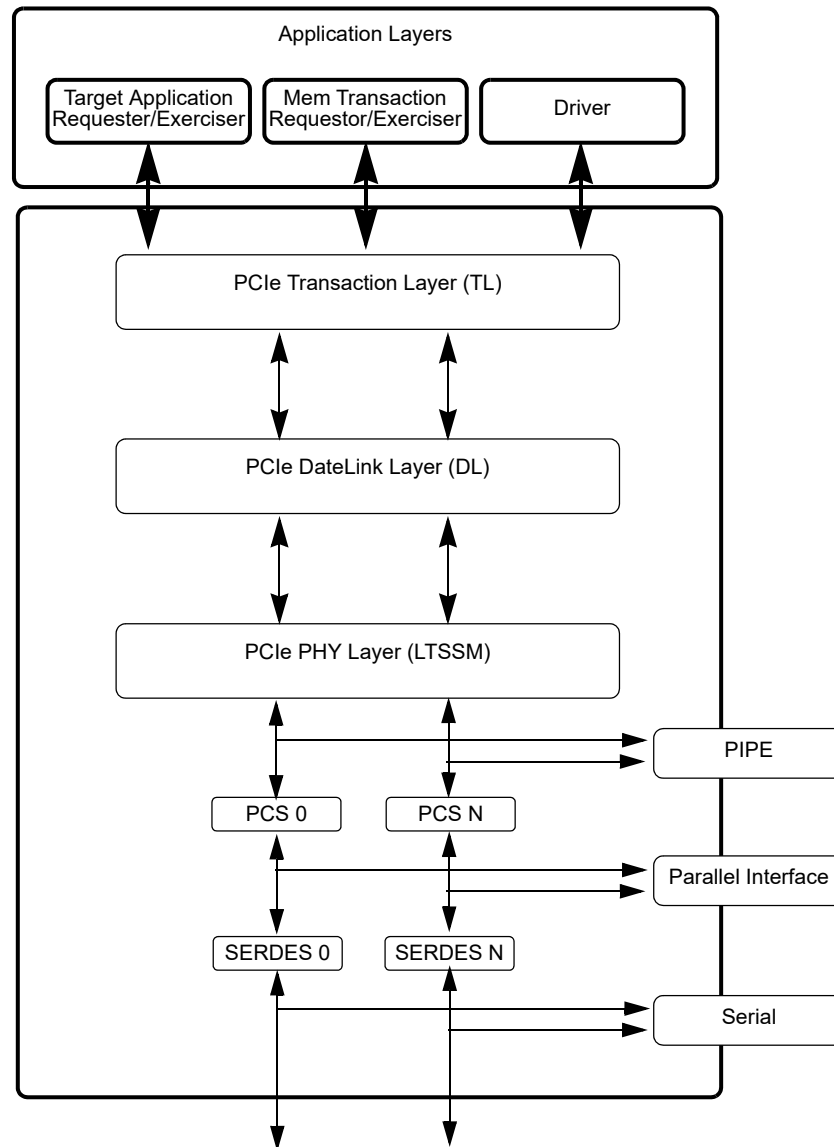


Figure 6-1 PCIe VIP Structure

## 6.2 Instantiation Models

The VIP is connected to a DUT via either a SERDES, 10-bit PMA, or PIPE (both “master” and “slave”) interface. The PIPE interface supports PIPE specification versions 2, 3 or 4 depending on the model used. Models are briefly described in [Table 6-1](#).



**Table 6-1 Types of Instantiation Models**

Model Names	Description
svt_pcie_device_agent_serdes_x16_8g_hdl.sv svt_pcie_device_agent_serdes_x16_hdl.sv svt_pcie_device_agent_serdes_x32_8g_hdl.sv svt_pcie_device_agent_serdes_x32_hdl.sv svt_pcie_device_agent_serdes_x4_8g_hdl.sv svt_pcie_device_agent_serdes_x4_hdl.sv svt_pcie_device_agent_serdes_x8_8g_hdl.sv svt_pcie_device_agent_serdes_x8_hdl.sv.	This is a full device model that connects via a SERDES interface and has support for a maximum of $n$ lanes (they do not all have to be used).
svt_pcie_device_agent_pma_x16_8g_hdl.sv svt_pcie_device_agent_pma_x16_hdl.sv svt_pcie_device_agent_pma_x32_8g_hdl.sv svt_pcie_device_agent_pma_x32_hdl.sv svt_pcie_device_agent_pma_x4_8g_hdl.sv svt_pcie_device_agent_pma_x4_hdl.sv svt_pcie_device_agent_pma_x8_8g_hdl.sv svt_pcie_device_agent_pma_x8_hdl.sv	This is a full device model that connects via 10bit PMA interface and has support for a maximum of $n$ lanes (they do not all have to be used).
svt_pcie_device_agent_mpipe_x16_8g_hdl.sv svt_pcie_device_agent_mpipe_x16_hdl.sv svt_pcie_device_agent_mpipe_x32_8g_hdl.sv svt_pcie_device_agent_mpipe_x32_hdl.sv svt_pcie_device_agent_mpipe_x4_8g_hdl.sv svt_pcie_device_agent_mpipe_x4_hdl.sv svt_pcie_device_agent_mpipe_x8_8g_hdl.sv svt_pcie_device_agent_mpipe_x8_hdl.sv	This is a full device (RC or endpoint) model that has a standard master PIPE interface to a DUT Phy.
svt_pcie_device_agent_spipe_x16_8g_hdl.sv svt_pcie_device_agent_spipe_x16_hdl.sv svt_pcie_device_agent_spipe_x32_8g_hdl.sv svt_pcie_device_agent_spipe_x32_hdl.sv svt_pcie_device_agent_spipe_x4_8g_hdl.sv svt_pcie_device_agent_spipe_x4_hdl.sv svt_pcie_device_agent_spipe_x8_8g_hdl.sv svt_pcie_device_agent_spipe_x8_hdl.sv	This is a full device (RC or endpoint) model that has a slave PIPE interface designed to connect directly to a DUT MAC master PIPE interface.
svt_mac_mpipe_xn_model	This is a MAC only model (no applications) for use in the compliance environment. It is also automatically instantiated within the full device model.
svt_mac_spipe_xn_model	This is a MAC only model (no applications) for use in the compliance environment. It is also automatically instantiated within the full device model.

**Table 6-1 Types of Instantiation Models (Continued)**

Model Names	Description
svt_phy_serdes_xn_model	This is a phy model used only in the compliance environment.
svt_pcie_device_agent_mpipe_rev4_2_pclk_input_x16_8g_hdl.sv svt_pcie_device_agent_mpipe_rev4_2_pclk_input_x32_8g_hdl.sv svt_pcie_device_agent_mpipe_rev4_2_pclk_input_x4_8g_hdl.sv svt_pcie_device_agent_mpipe_rev4_2_pclk_input_x8_8g_hdl.sv	This is a full device model that connects via a MPIPE interface and has support for a maximum of $n$ lanes (they do not all have to be used), and where the clk is an input. Following are current instantiation model files in SystemVerilog.
svt_pcie_device_agent_spipe_rev4_2_pclk_input_x32_8g_hdl.sv svt_pcie_device_agent_spipe_rev4_2_pclk_input_x4_8g_hdl.sv svt_pcie_device_agent_spipe_rev4_2_pclk_input_x8_8g_hdl.sv	This is a full device model that connects via a Slave Pipe interface and has support for a maximum of $n$ lanes (they do not all have to be used), and where the clk is an input. Following are current instantiation model files in SystemVerilog.

**Note**

Models supporting 8G have “\_8g” appended to the model name. 8G models should not be used without the corresponding Gen 3 license add-on.

## 6.3 General Steps In Picking an Instantiation Model

The first step in choosing an instantiation model is to answer a series of question/checklists as shown in the following table.

Note that due to the difference in data widths of PIPE and PMA Interfaces at 8/16G, the PCS may detect +/- 1 symbol skew even if serdes\_locked is asserted at the same time on all lanes."

**Table 6-2 Instantiation Model Checklist**

Type of Interface?	
	SERDES (serial) ** Very slow for simulation
	PMA (10bit)
	PIPE <ul style="list-style-type: none"> <li>• Master (SVT is MAC / DUT is Phy)</li> <li>• Slave (DUT is MAC / SVT is Phy)</li> </ul>
Speed?	
	Gen 1: 2.5G
	Gen 2: 5G
	Gen 3: 8G (requires optional license)

**Table 6-2**     **Instantiation Model Checklist (Continued)**

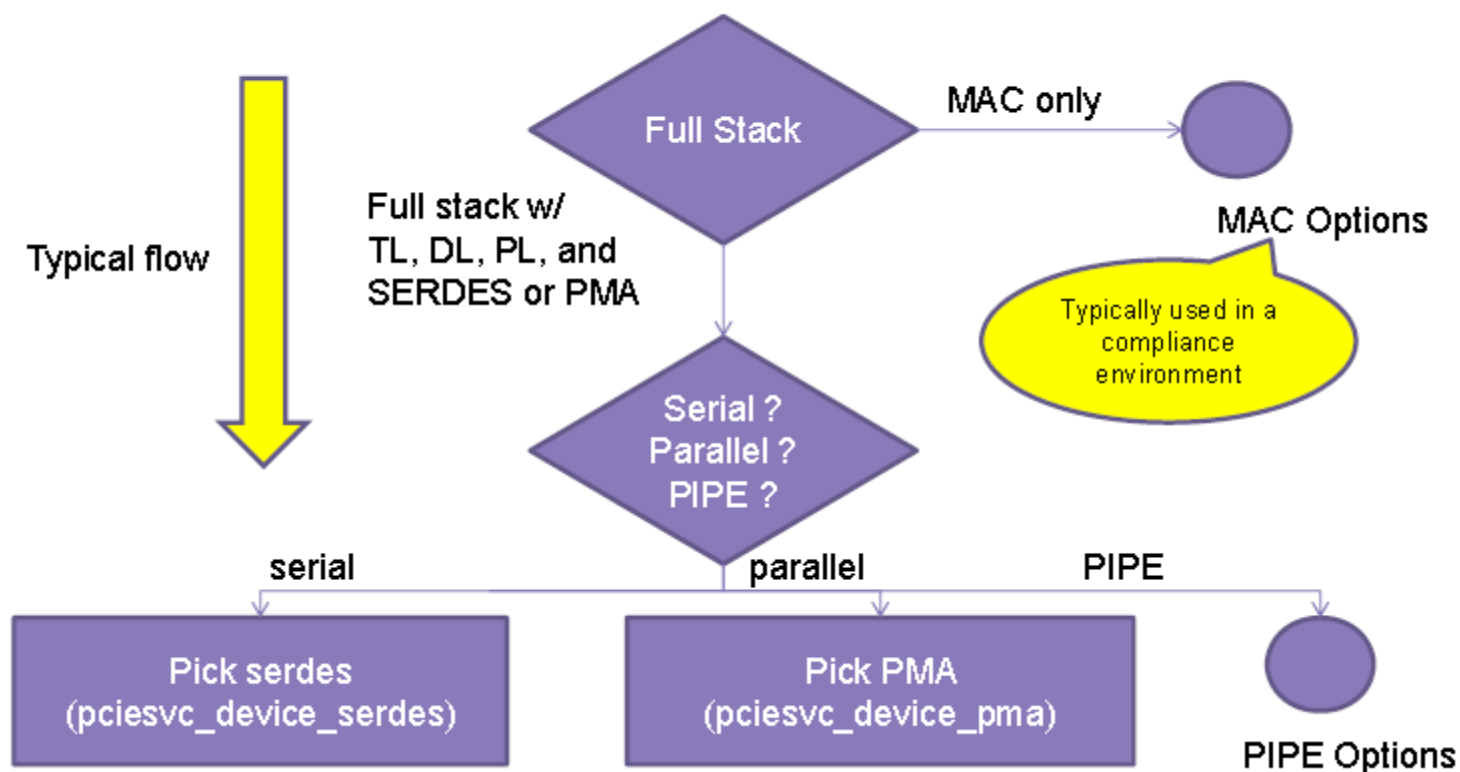
Link Width?	
	x4 model (x1, x2, x4)
	x8 model (x1 - x8)
	X32 model (x1 – x32)

### 6.3.1 Flowcharts for Choosing an Instantiation Model

Note, more details follow this overview section on each instantiation model.

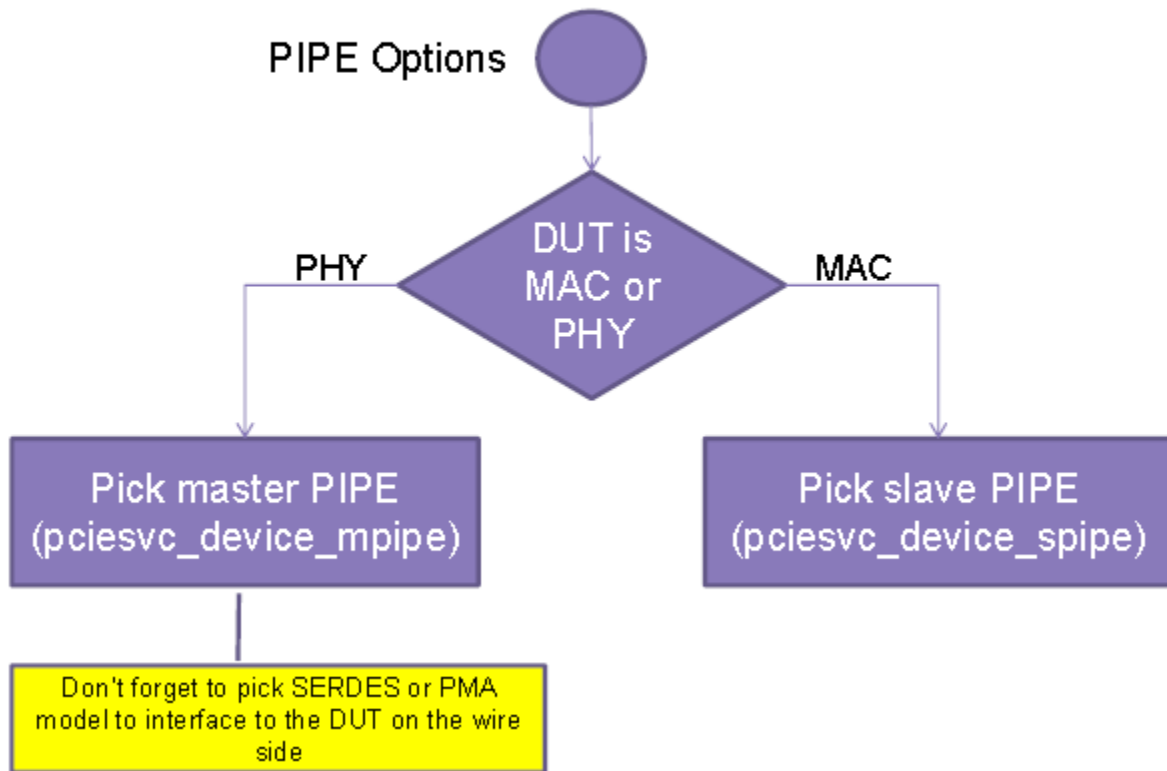
The following flowchart provides guidance on choosing the instantiation model for SERDES/PMA.

#### Picking an Instantiation Model: SERDES / PMA



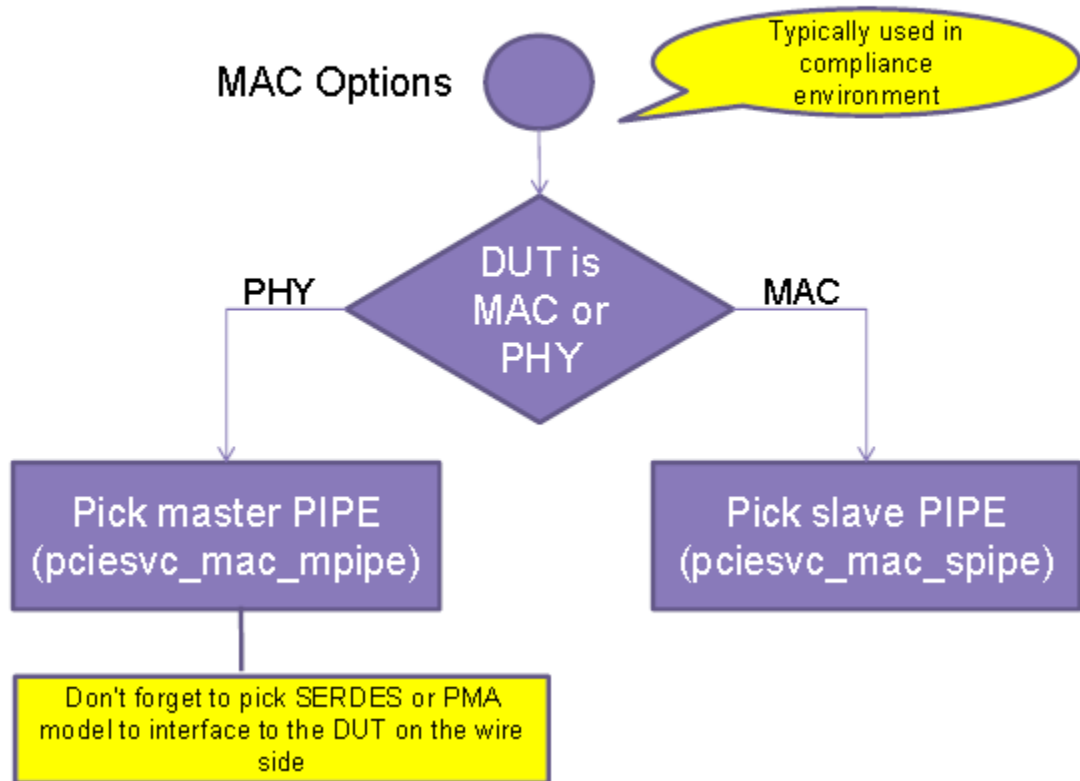
The following flow chart provides guidance on choosing your PIPE interface.

## Picking an Instantiation Model: PIPE Options



Use the following flowchart for guidance on choosing MAC Options:

## Picking an Instantiation Model: MAC Options



### 6.3.2 File Location and includes for Instantiation

The usage flow for instantiation follows:

1. Ensure the include path has the following directories:

"your\_design\_path"/src/sverilog/vcs

"your\_design\_path"/src/verilog/vcs

These directories are created by running `dw_setup_vip` when you install the models into a design directory.

2. Choose an appropriate instantiation model. Refer to [Table 6-1](#) on page 67 for a list of instantiation models.
3. Include the SystemVerilog model in your testbench.

The `hdl_interconnect` directory in our UVM examples show how to instantiate and connect `pcie_svt` models. The intermediate example has a directory called `hdl_interconnect` which has a copy of all the instantiation models used in the example. It is located at:

"your\_design\_path"/examples/sverilog/pcie\_svt/tb\_pcie\_svt\_uvm\_intermediate\_sys/  
hdl\_interconnect

The `hdl_interconnect` directory in the intermediate example has the following files to use for interconnect:

- `svt_pcie_8g_device_pipe_x4.sv`
- `svt_pcie_8g_device_pma_x4.sv`
- `svt_pcie_8g_device_serdes_x4.sv`
- `svt_pcie_device_pipe_x4.sv`
- `svt_pcie_device_pma_x4.sv`
- `svt_pcie_device_serdes_x4.sv`

The file we are interested in is:

`svt_pcie_8g_device_pipe_x4.sv`

The path to the `svt_pcie_device_agent_spipe_x4_8g_hdl` module is in the directory:

`"your_design_path"/src/sverilog/vcs.`

This directory `your_design_path"/src/sverilog/vcs` contains all the SystemVerilog instantiation files. Note that there are Verilog instantiation models in the directory `"your_design_path"/src/verilog/vcs.`

While the SystemVerilog instantiation models reference the `*.v` files, Synopsys recommends only use the `*sv` instantiation files. The Verilog `*.v` files are useful for the signal names used in connecting the model.

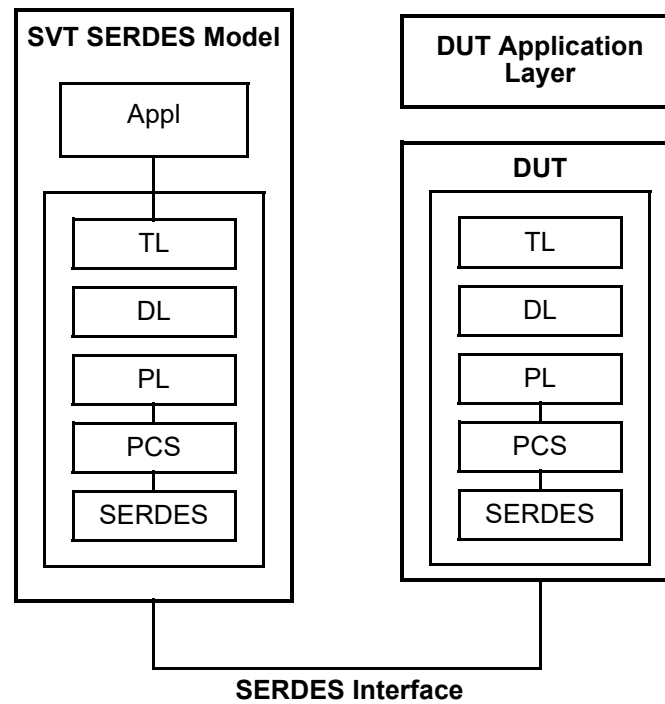
## 6.4 SERDES Interface

If the serial interface is used, the SERDES layer will be provided by the VIP. The instantiation model is named `svt_pcie_device_agent_serdes_x[4,8,16,32]_[8g]_hdl`.

The Serializer/Deserializer (SERDES) is used to convert the serial, differential bit stream into a stream of 10-bit bytes. In addition, it also does signal-level validation, receiver clock-recovery and bit alignment.

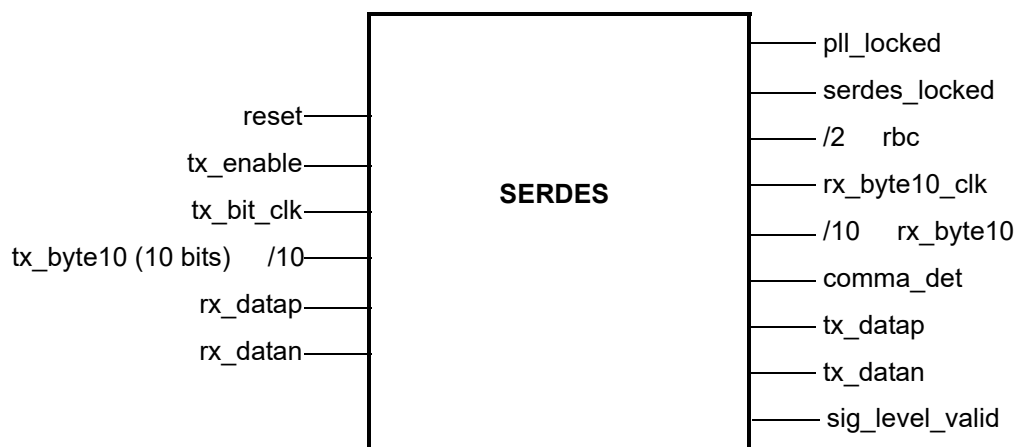
The SERDES model supplied is not an analog SERDES model, but a digital representation, and is not meant for verifying an analog SERDES design. It is provided so that Phy Layer functionality such as speed negotiation may be tested.

Figure 6-2 shows the layers supported by the PCIe VIP when using the serial interface along with the DUT serial interface.



**Figure 6-2** PCIe VIP serial interface layers

Figure 6-3 shows the SERDES module ports.



**Figure 6-3** SERDES module ports

**Table 6-3** svt\_pcie\_device\_agent\_model SERDES ports

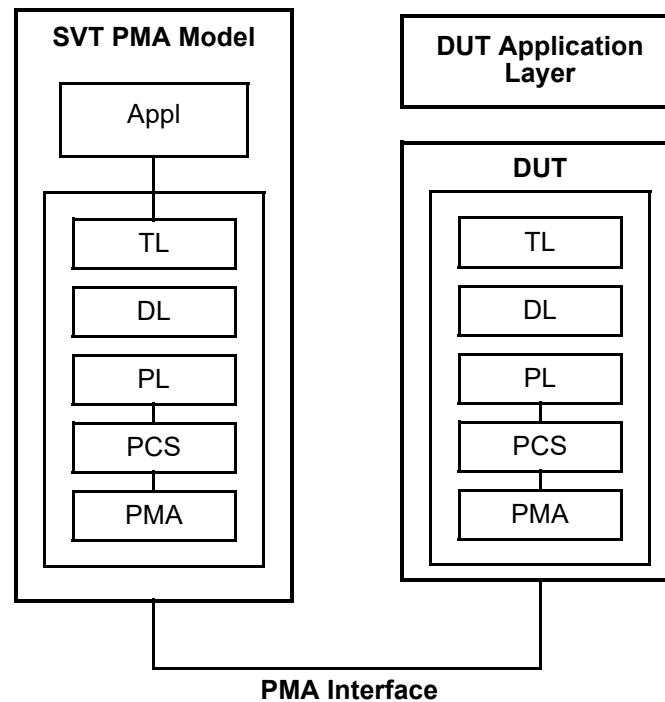
Model Interface Type	Port Name	I/O	Description
SERDES Interface.	Reset		Reset to the VIP model is active high. Reset must be deasserted at time 0 such that a posedge is seen by the VIP. Reset should be asserted at 1ns and held in reset for at least 100 ns. Once deasserted, it should remain deasserted for the remainder of the simulation.
	phyn_rx_datap		Serial datap in to VIP
	phyn_rx_datan		
	phyn_tx_datap		serial datap out of VIP
	phyn_tx_datan		

## 6.5 PMA Interface

The PCIe VIP supports a 10-bit parallel interface. The 10-bit quantity produced by the model is the result of 8b/10b encoding as defined by the PCI Express Base Specification. This is the data that would be then serialized and put onto the physical link. The 10-bit interface is fully compliant with the PCI Express protocol standard and can be used to verify PHY layers.

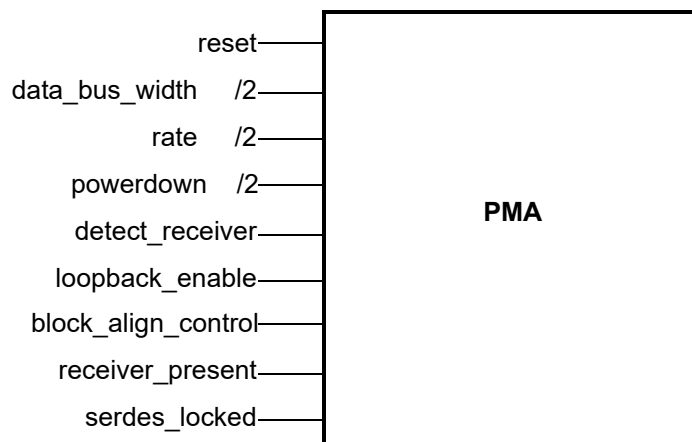
The Physical Coding Sublayer (PCS) checks for commas and performs symbol lock at 2.5GT/s, 5GT/s, and 8GT/s (Optional). It also does 8b/10b encoding at 2.5GT/s and 5GT/s and inserts data sync headers at 8GT/s. The PCS also contains the elastic buffer which inserts/deletes SKP symbols from skip ordered sets. The PCS task pair interface and module inputs/outputs comply with Intel's PIPE interface specification. [Figure 6-4](#) shows the layers supported by the PCIe VIP when using the PMA along with the DUT PMA interface.





**Figure 6-4** PCIe VIP PMA layers

The PMA module ports are shown in [Figure 6-5](#)



**Figure 6-5** PMA module ports

## 6.6 PIPE Interface, Phy VIP and MAC DUT

In addition, the transceiver model supports the PHY Interface for the PCI Express (PIPE) Architecture. This is an Intel specification that defines Media Access, Physical Coding, and Physical Media Attachment sublayers in the PCI Express PHY layer.

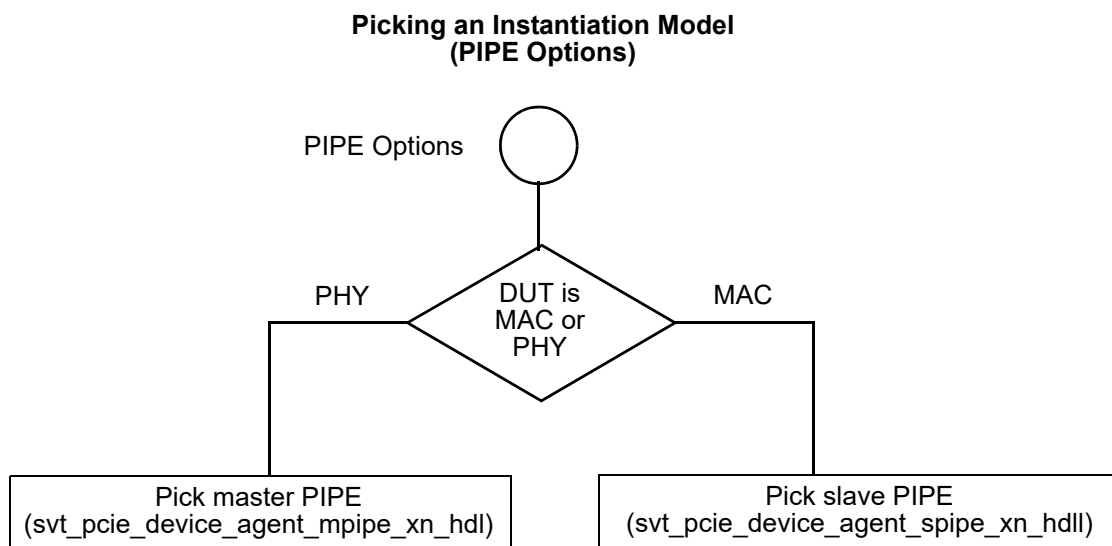
The term "PIPE" refers to the parallel interface between the Media Access and Physical Coding sublayers. Note the difference in interface/pin configurations between PIPE as shown in [Figure 6-3](#), and 10-bit interfaces. Models configured for a 10-bit interface ignore control pins, and use only 10 pins of the data lines txdata and rxdata. Contact Intel for instructions on obtaining the PHY Interface for the PCI Express™ Architecture specification.

The following versions of the PIPE interface are supported:

- 2.1
- 3.0
- 4.0

The model supports both 8-bit and 16-bit implementations of the PIPE, and can be used to support one of the following PIPE modes (see [Figure 6-6](#)):

- PHY PIPE (SPIPE) mode: In this mode, the model acts like a PIPE-compliant PHY and supports the complete interface.
- MAC PIPE (MPIPE) mode: In this mode, the model acts like a PIPE-compliant MAC and supports the complete interface.



**Figure 6-6** Instantiation model PIPE options

You set the PIPE version in the `svt_pcie_device_configuration` class. The settings are:

- For PIPE 2 (Gen2 version of the PCIe spec):  

```
svt_pcie_device_configuration::pipe_spec_ver==svt_pcie_device_configuration::PIPE_SPEC_VER_2
```
- For PIPE 3 or 4 (Gen 3 version of the PCIe spec):  

```
==svt_pcie_device_configuration::PIPE_SPEC_VER_4
```

The model may be configured to support the PIPE's interface as either a PIPE compliant PHY (PHY PIPE mode), or as a PIPE compliant MAC (MAC PIPE mode). The PHY PIPE mode and MAC PIPE mode are enabled by selecting the correct instantiation model, either the `svt_pcie_device_agent_mpipe` or the

svt\_pcie\_device\_agent\_spipe. This diagram should help to select the correct instantiation model once the DUT configuration is known.

### 6.6.1 Using ifdefs with PIPE 4.0 To Select Additional Signals

Two “`ifdef”s that are used to by the PIPE 4.0 (and above) instantiation models (“InstantiationModels/svt\_pcie\_device\_agent\_\*pipe\_x\*\_8g.sv”) to select additional interface signals for PIPE 4.2 and PIPE 4.3.. These “`ifdef”s are:

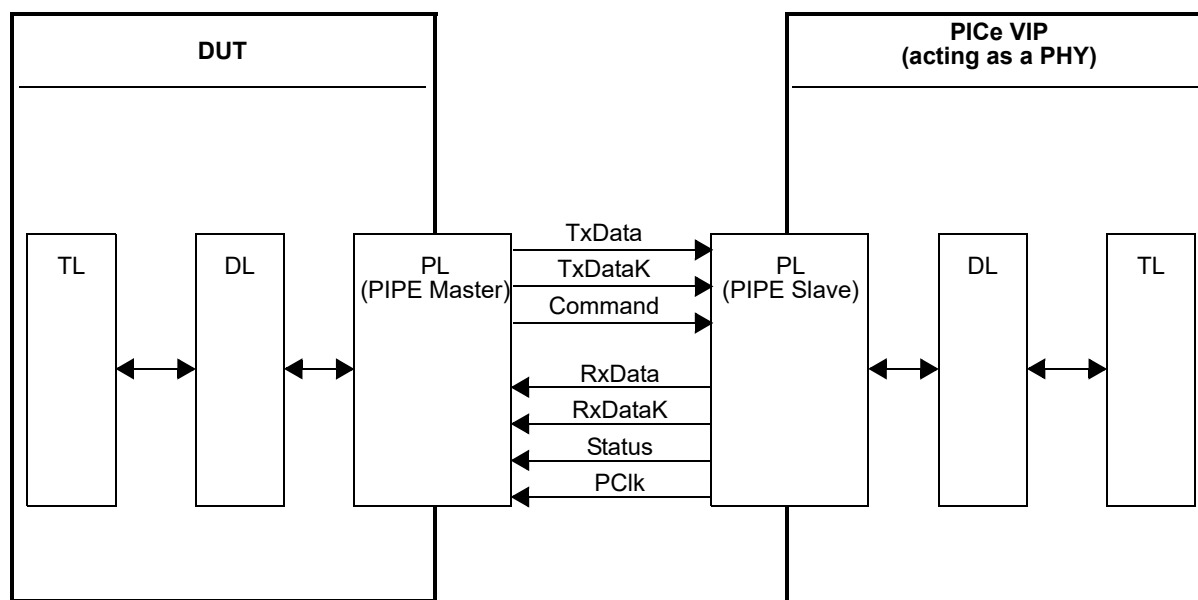
- **`ifdef PCIESVC\_PIPE\_SPEC\_VER\_GTR\_4\_0** This “define” selects the additional signals associated with PIPE 4.2 versus PIPE 4.0. Signals such as “rx\_eq\_in\_progress\_0” (through \_n), “lf\_0” (through \_n), “fs\_0” (through \_n). It is also used to determine the “PIPE\_SPEC\_VER” assigned to the “pl0” PHY instance in the “InstantiationModels/svt\_pcie\_device\_agent\_mpipe\_x\*\_8g.sv” models.
- **`ifdef PCIESVC\_PIPE\_SPEC\_VER\_4\_3** This “define” selects the additional signals associated with PIPE 4.3 versus PIPE 4.2. Signals such as “async\_power\_change\_ack” and “powerdown[3:0]” (versus [2:0]). It is also used with “`ifdef PCIESVC\_PIPE\_SPEC\_VER\_GTR\_4\_0” to determine whether “4.2” or 4.3” is assigned to “PIPE\_SPEC\_VER” that is then assigned to the “pl0” PHY instance in the “InstantiationModels/svt\_pcie\_device\_agent\_\*pipe\_x\*\_8g.sv” models.

As a result:

- PIPE 4.0 usage would have neither of the above two “`ifdef”s defined.
- PIPE 4.2 usage would have “PCIESVC\_PIPE\_SPEC\_VER\_GTR\_4\_0” defined.
- PIPE 4.3 usage would have “PCIESVC\_PIPE\_SPEC\_VER\_GTR\_4\_0” and “PCIESVC\_PIPE\_SPEC\_VER\_4\_3” defined.

### 6.6.2 Picking an Instantiation Model (SPIPE Options)

When enabled to act as a PHY, the model supports validation of the MAC interface to a user's DUT. This is shown in [Figure 6-7](#). Note on the right side of the diagram that the PCIe transceiver acts not only as the DUT's PHY, but that you can drive the PHY through the model, which also acts as endpoint connected to the PHY from a system point of view.



**Figure 6-7** PCIe transceiver acting as a DUT's PHY to test the MAC

**Table 6-4** svt\_pcie\_device\_agent\_model SPIPE ports

Model Interface Type	Port Name	I/O	Description
<p>“Slave” PIPE Version 3 Interface</p> <p>The signals prefixed “attached_” are relative to the DUT and are designed to connect up in a slave-like manner to a MAC DUT. PIPE 3 models do not have a _8g postfix in their file name.</p>	reset	I	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 at 1ns and held in reset for at least 100 ns. Once deasserted, it should remain deasserted for the remainder of the simulation.
	attached_pipe_reset_n	I	PIPE reset signal from the mac.
	pipe_clk	IO	The pipe_clk signal is an input signal only by using the the Define PCIESVC_PIPE_PCLK_AS_PHY_INPUT. Use the following to make it an input signal: +define+PCIESVC_PIPE_PCLK_AS_PHY_INPUT Otherwise by default the pipe_clk signal is an output of the PHY for use by the DUT mac.
	attached_powerdown[1:0]	I	Command from the DUT MAC to change power state of the phy.
	attached_rate[1:0]	I	Link signaling rate control from the DUT MAC. NOTE: Port width is 1 bit for PIPE2 and 2 bits for PIPE3.
	attached_txdetectrx	I	Command from the DUT mac to detect receiver.
	attached_block_align_controll	I	Controls slave block alignment. NOTE: PIPE3 and above.
	attached_phy_status[31:0]	O	Phy response to rate or power change. Bit 0 only used for PIPE2.
	attached_data_bus_width[1:0]	O	Defines the bus width of per lane data.
	attached_rx_data_n	O	Per lane receive data into the DUT. Supports 8, 16, and 32bit bus widths
	attached_rx_data_k_n	O	Per lane control indication. Supports 8, 16, and 32 bit bus widths.
	attached_rx_status_n[2:0]	O	Per lane receiver status. See PIPE spec for encodings.
	attached_rx_valid_n	O	Per lane li nd ica ti on that data coming into the DUT is valid and symbol lock has been achieved.
	attached_rx_elec_idle_n	I/O	Per lane indication that electrical idle is being received into the DUT

**Table 6-4 svt\_pcie\_device\_agent\_model SPIPE ports (Continued)**

Model Interface Type	Port Name	I/O	Description
“Slave” PIPE Version 3 Interface (continued)	attached_invert_rx_polarity_n	I	Per lane indication to perform polarity inversion on data into the DUT
	attached_tx_data_n[31:0]	I	Per lane transmit data out from the DUT.
	attached_tx_data_k_n[3:0]	I	Per lane transmit_control from the DUT.
	attached_tx_ei_code_n[31:0]	I	Per lane error injection code. Indicates to the phy that an EI was either injected, or phy needs to inject the EI indicated by the code. For VIP only. No connection to DUT.
	attached_tx_compliance_n	I	Per lane transmit compliance pattern enable from the DUT
	attached_tx_elect_idle_n	I	Per lane transmit electrical idle/transmit enable from the DUT Note: Any unused ports must be tied to a 1.
“Slave” PIPE Version 4 interface  The signals prefixed “attached_” are relative to the DUT and are designed to connect up in a slave like manner to a MAC DUT. The 8G models require a PIPE4 interface, hence all models in the InstantionModels directory with the _8g suffix use the PIPE4.	reset	I	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 at 1ns and held in reset for at least 100 ns. Once deasserted, it should remain deasserted for the remainder of the simulation.
	attached_pipe_reset_n	I	PIPE reset signal from the mac.
	pipe_clk	O	PIPE clock generated by the PIPE slave for use by the DUT mac.
	max_pclk	O	Maximum pclk rate for a given speed.
	attached_powerdown[2:0]	I	Command from the DUT MAC to change power state of the phy.
	attached_rate[1:0]	I	Link signaling rate control from the DUT MAC. NOTE: port width is 1 bit for PIPE2 and 2 bits for PIPE3.
	attached_pclk_rate[2:0]	I	PCLK rate requested by the mac.
	attached_txdetectrx	I	Command from the DUT mac to detect receiver.
	attached_block_align_control	I	Controls slave block alignment. NOTE: PIPE3 and above.
	attached_tx_margin[2:0]	I	Selects transmitter voltage levels.
	attached_tx_swing	I	Controls transmitter voltage swing level.
	attached_lf[5:0]	I	Provides the LF value advertised by the link partner.
	attached_fs[5:0]	I	Provides the full swing value advertised by the link partner.

**Table 6-4 svt\_pcie\_device\_agent\_model SPIPE ports (Continued)**

Model Interface Type	Port Name	I/O	Description
	attached_width[1:0]	I	Reflects the width of the data bus the mac wants to use.
	attached_rx_standby[31:0]	I	Controls whether the phy RX is active when in L0 or L0s.
	attached_phy_status[31:0]	O	Phy response to rate or power change. Bit 0 only used for PIPE2.
	attached_rx_standby_status[31:0]	O	Reflects the active/standby state of the rx receiver.
	attached_data_bus_width[1:0]	O	Defines the bus width of per lane data.
	attached_rx_data_n[	O	Per lane receive data into the DUT. Supports 8, 16, and 32bit bus widths.
	attached_rx_data_k_n[	O	Per lane control indication. Supports 8, 16, and 32 bit bus widths.
	attached_rx_status_n[[2:0]	O	Per lane receiver status. See PIPE spec for encodings.
	attached_rx_valid_n[	O	Per lane indication that data coming into the DUT is valid and symbol lock has been achieved.
	attached_rx_data_valid_n[	O	Per lane indication that data coming into the DUT is valid. Used primarily for rate matching. NOTE: PIPE3 only.
	attached_rx_elec_idle_n	IO	Per lane indication that electrical idle is being received into the DUT
	attached_rx_start_block_n[	O	Per lane indication that data transmitted by the DUT is the first byte of data in the block. NOTE: PIPE3 only.
	attached_rx_sync_header_n[[1:0]	O	Per lane block sync header. NOTE: PIPE3 only.
	attached_invert_rx_polarity_n[	I	Per lane indication to perform polarity inversion on data into the DUT.
	attached_tx_data_n[[31:0]	I	Per lane transmit data out from the DUT.
	attached_tx_data_k_n[[3:0]	I	Per lane transmit_control from the DUT.
	attached_tx_ei_code_n[[31:0]	I	Per lane error injection code. Indicates to the phy that an EI was either injected, or phy needs to inject the EI indicated by the code. For SVC only. No connection to DUT.
	attached_tx_compliance_n[	I	Per lane transmit compliance pattern enable from the DUT

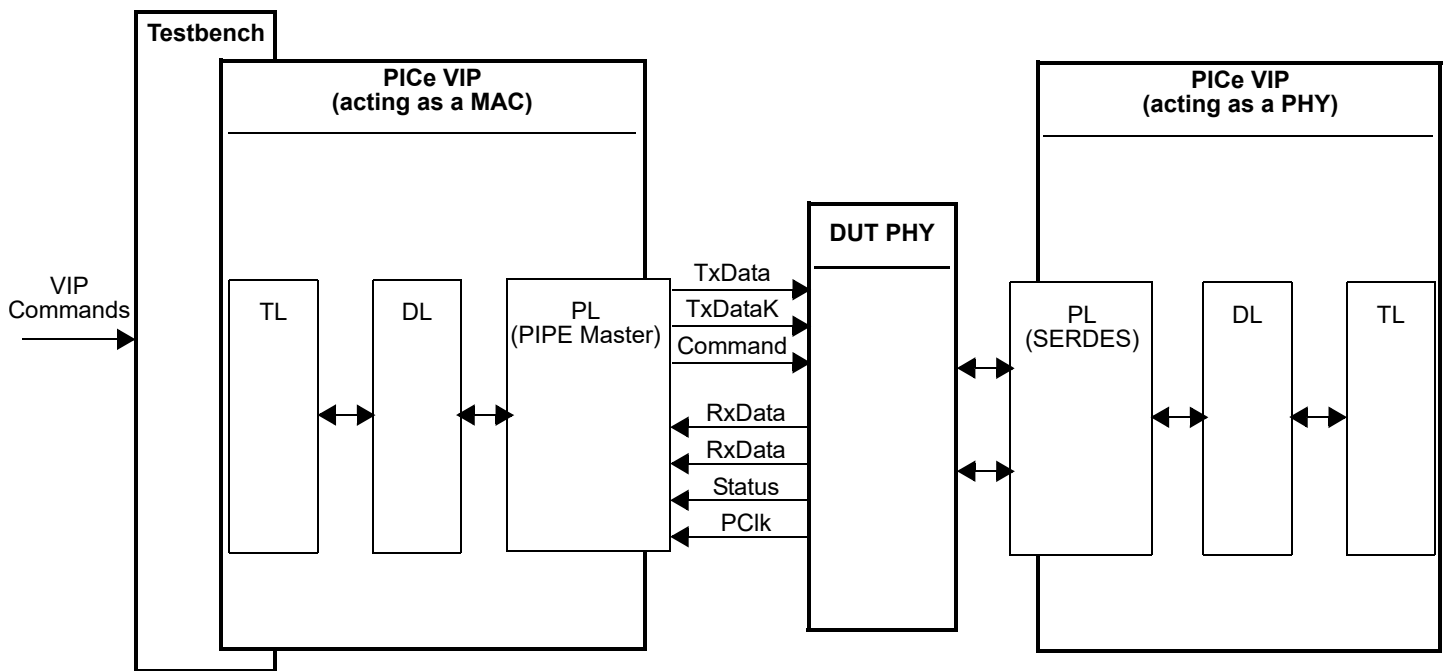
**Table 6-4** svt\_pcie\_device\_agent\_model SPIPE ports (Continued)

Model Interface Type	Port Name	I/O	Description
	attached_tx_elect_idle_n[	I	Per lane transmit electrical idle/transmit enable from the DUT. Note: Any unused ports must be tied to a 1.
	attached_tx_data_valid_n[	I	Per lane indication that data transmitted to the SVC is valid. Used primarily for rate matching. NOTE: PIPE3 only.
	attached_tx_start_block_n[	O	Per lane indication that data transmitted to the SVC is the first byte of data in the block. NOTE: PIPE3 only.
	attached_tx_sync_header_n[[1:0]		Per lane block sync header. NOTE: PIPE3 only.
	attached_local_tx_preset_coefficientsn [[17:0]		Coefficients returned from a coefficient lookup request.
	attached_link_eval_feedback_figure_of_meritn[7:0]	O	Figure of merit value from a link equalization evaluation request.
	attached_link_eval_feedback_direction_changen[5:0]	O	Coefficient direction feedback from rx link eval request.
	attached_local_fsm[5:0]	O	Provides the FS value for the phy.
	attached_local_lfn[5:0]	O	Provides the LF value for the phy.
	attached_local_tx_coefficients_validn	O	Indicates that the values in local_tx_preset_coefficients[] are valid.
	attached_tx_deemphn[17:0]	I	Selects transmitter deemphasis.
	attached_local_preset_indexn[3:0]	I	Index for the local phy preset coefficients requested by the mac.
	attached_rx_preset_hintn[2:0]	I	RX preset hint for the receiver.
	attached_get_local_preset_coefficientsn	I	Request a preset to coefficient mapping lookup.
	attached_rx_eq_evaln	I	Request from the mac for the receiver to perform an equalization evaluation.
	attached_invalid_requestn	I	Indicates the link eval feedback requested was out of range.

## 6.7 PIPE Interface, MAC VIP and PHY DUT (MPIPE)

When enabled to act as a MAC, the model supports validation of the PIPE or SERDES interface (encoding, serialization, and so on) to a user's PHY. This is shown in [Figure 6-8](#).





**Figure 6-8** PCIe transceiver acting as a DUT's MAC

**Table 6-5 svt\_pcie\_device\_agent\_model MPIPE ports**

Model Interface Type	Port Name	I/O	Description
<b>"Master" PIPE Version 3 Interface</b>  This type of interface is a standard PIPE interface to a phy. PIPE 3 models do not have a _8g postfix in their file names.	reset	I	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 at 1ns and held in reset for at least 100 ns. Once deasserted, it should remain deasserted for the remainder of the simulation.
	pipe_reset_n	O	PIPE reset signal to phy
	pipe_clk	I	PIPE clock from phy
	powerdown[1:0]	O	Command to change power state of the phy
	rate[1:0]	O	Link signaling rate control
	txdetectrx	O	Command to detect receiver
	phy_status[31:0]	I	Phy response to rate or power change. For PIPE2, only bit 0 is used.
	data_bus_width[1:0]	I	Indicates the per lane PIPE data bus width
	rx_data_n[31:0]	I	Per lane receive data into the VIP. Supports 8, 16, and 32bit bus widths
	rx_data_k_n[3:0]	I	Per lane control indication. Supports 8, 16, and 32 bit bus widths.
	rx_status_n[2:0]	I	Per lane receiver status. See PIPE spec for encodings. Note: Any unused ports must be tied to a 0.
	rx_valid_n	I	Per lane indication that data coming into the VIP is valid and symbol lock has been achieved.
	rx_data_valid_n	I	Per lane indication that data coming into the VIP is valid. Used primarily for rate matching. NOTE: PIPE3 only.

**Table 6-5 svt\_pcie\_device\_agent\_model MPIPE ports (Continued)**

Model Interface Type	Port Name	I/O	Description
	rx_elec_idle_n	I	Per lane indication that electrical idle is being received into the VIP Note: Any unused ports must be tied to a 1.
	invert_rx_polarity_n	O	Per lane indication to perform polarity inversion on data into the VIP
	tx_data_n[31:0]	O	Per lane transmit data out from the VIP
	tx_data_k_n[3:0]	O	Per lane transmit_control from the VIP
	tx_ei_code_n[31:0]	O	Per lane error injection code. Indicates to the phy that an EI was either injected, or phy needs to inject the EI indicated by the code. For VIP only. No connection to DUT.
	tx_compliance_n	O	Per lane transmit compliance pattern enable from the VIP
	tx_elect_idle_n	O	Per lane transmit electrical idle/transmit enable from the VIP
<b>"Master" PIPE Version 4 Interface</b>  This type of interface is a standard PIPE interface to a phy. The 8G models require a PIPE4 interface, hence all models in the InstantionModels directory with the _8g suffix use the PIPE4.	reset	I	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 at 1ns and held in reset for at least 100 ns. Once deasserted, it should remain deasserted for the remainder of the simulation.
	pipe_reset_n	O	PIPE reset signal to phy
	pipe_clk	I	PIPE clock from phy
	powerdown[2:0]	O	Command to change power state of the phy No vendor specific power levels are supported.
	rate[1:0]	O	Link signaling rate control
	pclk_rate[2:0]	O	Specifies the value of the PIPE clock.
	txdetectrx	O	Command to detect receiver
	block_align_control	O	Controls slave block alignment. NOTE: PIPE3 and above.
	tx_margin[2:0]	O	Selects transmitter voltage levels
	tx_swing	O	Controls transmitter voltage swing level.
	lf[5:0]	O	Provides the LF value advertised by the link partner.
	fs[5:0]	O	Provides the full swing value advertised by the link partner.

**Table 6-5 svt\_pcie\_device\_agent\_model MPIPE ports (Continued)**

Model Interface Type	Port Name	I/O	Description
	width[1:0]	O	Outputs the width of the per-lane data bus the mac is currently using.
	rx_standby[31:0]	O	Controls whether the phy RX is active when in L0 or L0s.
	phy_status[31:0]	I	Phy response to rate or power change. For PIPE2, only bit 0 is used.
	rx_standby_status[31:0]	I	Reflects the active/standby state of the rx receiver.
	data_bus_width[1:0]	I	Indicates the per lane PIPE data bus width.
	rx_data_n[31:0]	I	Per lane receive data into the SVC. Supports 8, 16, and 32bit bus widths.
	rx_data_k_n[3:0]	I	Per lane control indication. Supports 8, 16, and 32 bit bus widths.
	rx_status_n[2:0]	I	Per lane receiver status. See PIPE spec for encodings. Note: Any unused ports must be tied to a 0.
	rx_valid_n	I	Per lane indication that data coming into the SVC is valid and symbol lock has been achieved.
	rx_data_valid_n	I	Per lane indication that data coming into the SVC is valid. Used primarily for rate matching.
	rx_elec_idle_n	I	Per lane indication that electrical idle is being received into the SVC. Note: Any unused ports must be tied to a 1.
	rx_start_block_n	I	Per lane indication that electrical idle is being received into the SVC is the first byte of data in the block.
	rx_sync_header_n	I	Per lane block sync header.
	invert_rx_polarity_n	O	Per lane indication to perform polarity inversion on data into the SVC.
	tx_data_n[31:0]	O	Per lane transmit data out from the SVC.
	tx_data_k_n[3:0]	O	Per lane transmit_control from the SVC
	tx_ei_code_n[31:0]	O	Per lane error injection code. Indicates to the phy that an EI was either injected, or phy needs to inject the EI indicated by the code. For SVC only. No connection to DUT.
	tx_compliance_n	O	Per lane transmit compliance pattern enable from the SVC.

**Table 6-5 svt\_pcie\_device\_agent\_model MPIPE ports (Continued)**

Model Interface Type	Port Name	I/O	Description
	tx_elect_idle_n	O	Per lane transmit electrical idle/transmit enable from the SVC.
	tx_data_valid_n	O	Per lane indication that data transmitted by the SVC is valid. Used primarily for rate matching.
	tx_start_block_n	O	Per lane indication that data transmitted by the SVC is the first byte of data in the block.
	tx_sync_header_n[1:0]	O	Per lane block sync header.
	local_tx_preset_coefficientsn[17:0]	I	Coefficients returned from a coefficient lookup request.
	link_eval_feedback_figure_of_meritn[7:0]	I	Figure of merit value from a link equalization evaluation request.
	link_eval_feedback_direction_changen[5:0]	I	Coefficient direction feedback from rx link eval request.
	local_fsn[5:0]	I	Provides the FS value for the phy.
	local_lfn[5:0]	I	Provides the LF value for the phy.
	local_tx_coefficients_validn	I	Indicates that the values in local_tx_preset_coefficients[] are valid.
	tx_deemphn[17:0]	O	Selects transmitter deemphasis.
	local_preset_indexn[3:0]	O	Index for the local phy preset coefficients requested by the mac.
	rx_preset_hin[n][2:0]	O	RX preset hint for the receiver.
	get_local_preset_coefficientsn	O	Request a preset to coefficient mapping lookup.
	rx_eq_evaln	O	Request from the mac for the receiver to perform an equalization evaluation.
	invalid_requestn	O	Indicates the link eval feedback requested was out of range

## 6.8 PCIe Device and MAC Model Instantiation

The PCIe device model contains the connectivity, clocking, and basic configuration for the application layer, Transaction Layer, Data Link Layer, and Physical Layer. The MAC model is instantiated internally within the device model.

Generally, you should use the device model to take advantage of the Application Layer. MAC models do not include the application layer and its associated scoreboarding.

The device model is generic – it can be personalized to be either a Root Complex or an Endpoint.

## 6.8.1 Model Wire Interface Options

Table 6-3 defines the port list for each interface configuration of the `svt_pcie_device_agent_model` instantiation.



### Note

All unused input ports must be tied to 0, with the exception of `*_elec_idle_n`. Any unused `*_elec_idle_n` input ports must be tied to a 1.

## 6.9 Configuring the PIPE Data Bus Width

### 6.9.1 PIPE 2.1/3

In PIPE 2.1 and PIPE 3.0 spec versions, the MAC does not specify the per-lane width and rate of the PIPE clock. For `*mpipe*` models the width will take the width reflected on the `data_bus_signal`. See Table 7-3 for information about the `data_bus_signal`. For `*spipe*` models, the width must be configured for each supported speed. These settings are made in the `svt_pcie_pl_configuration` PL configuration class. The settings are:

- `pipe_width[0]`: Gen 1 data bus width
- `pipe_width[1]`: Gen 2 data bus width
- `pipe_width[2]`: Gen 3 data bus width

Supported widths are (`pipe_width_enum`): `PIPE_8_BITS`, `PIPE_16_BITS`, `PIPE_32_BITS`

### 6.9.2 PIPE4 and PIPE 4.2

With PIPE 4, the per-lane data bus width and rate of the PIPE clock are determined by the `pclk_rate` and width PIPE signals. For `*mpipe*` models the `pclk_rate` and width signals are set within the PL configuration class: `svt_pcie_pl_configuration`. The settings are:

- `pclk_rate[0]`: Gen 1 `pclk_rate`
- `pclk_rate[1]`: Gen 2 `pclk_rate`
- `pclk_rate[2]`: Gen 3 `pclk_rate`

Options are (`pclk_rate_enum`): `PCLK_67_5_MHZ`, `PCLK_125_MHZ`, `PCLK_250_MHZ`, `PCLK_500_MHZ`, `PCLK_1000_MHZ`

- `pipe_width[0]`: Gen 1 data bus width
- `pipe_width[1]`: Gen 2 data bus width
- `pipe_width[2]`: Gen 3 data bus width

Supported widths are (`pipe_width_enum`): `PIPE_8_BITS`, `PIPE_16_BITS`, `PIPE_32_BITS`

For `*spipe*` models connect the `pclk_rate` and width PIPE signals to the MAC.

Note: For SERDES or PMA models, do not set the `pclk_rate` or `pipe_width`.

## 6.10 Compile-Time Parameter Settings

Several parameters are set at the model. They typically 'trickle-down' to the individual layers.

Table 6-6 lists the Verilog parameters that you set at compile time. They are not runtime changeable. They are not accessible from the UVM environment.

**Table 6-6 Parameters set in the model**

Parameter Name	Type	Range	Default Value	Description
NUM_PMA_INTERFACE_BITS	Integer	10, 16, 32, 64, 128	10	Number of bits on the PMA interface
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		"pcie_svt_device_agent_x_x_yy_hdl" (Specific to each model).	String prefixed to messages to display in the output log.

**Example 6-1 Setting Verilog parameters at compile time**

```
// This is a PIPE SLAVE model, so inputs and outputs are reversed from a normal PIPE
pcie_svt_device_agent_spipe_x4_hdl #(.DISPLAY_NAME("root0."),
    .DEVICE_IS_ROOT(1)) root0(.reset
    (reset),
    .pipe_clk
    (pipe_clk),
    // shared signals
    .attached_pipe_reset_n
    (endpoint0_pipe_reset_n),
    // inputs
    ...
    .attached_tx_compliance_3
    (endpoint0_tx_compliance_3),
    .attached_tx_elec_idle_3
    (endpoint0_tx_elec_idle_3
);
```

## 6.11 Instantiating Multiple-root Hierarchies

If you need to instantiate multiple root hierarchies, then each one must be marked uniquely to distinguish it. This is done via the hierarchy\_number parameter. For each root (and associated endpoint) model you instantiate, you need to defparam the appropriate hierarchy number to that model. If you are using the Shadow Memory mechanism for doing automatic checking of data, you also need to defparam those shadows respectively.

Each hierarchy number should be unique. It is recommended that you make it match the instantiation number, for ease of reference. The HIEARCHY\_NUMBER is used in conjunction with the SHADOW\_MEMORY checking. For example, in the above instantiation, add the HIERARCHY\_NUMBER as follows:

```
pcie_svt_device_agent_spipe_x4_hdl #(.DISPLAY_NAME("root0."),
    .DEVICE_IS_ROOT(1), .HIEARCHY_NUMBER(1)) root0(.reset
```

## 6.12 Model Configuration Overview

Configuration tasks are used after reset is deasserted from the `pcie_svt_device_agent_model`.

Recommended tasks to be set prior to usage are described in [Table 6-7](#). The details of the task may be found in the respective section of the documentation.

**Table 6-7 Configuration tasks to set before usage**

Name	Layer
<code>svt_pcie_tl_configuration::init_[cpl/np/p]_[data/hdr]_tx_credits</code>	Transaction Layer
<code>svt_pcie_tl_service_set_vc_en_sequence</code>	Transaction Layer
<code>svt_pcie_tl_service::service_type_enum=SET_TRAFFIC_CLASS_MAP</code>	Transaction Layer
<code>svt_pcie_tl_service::service_type_enum=ADD_MEM_ADDR_APPL_ID_MAP_ENTRY</code>	Transaction Layer
<code>svt_pcie_tl_service::service_type_enum=ADD_IO_ADDR_APPL_ID_MAP_ENTRY</code>	Transaction Layer
<code>svt_pcie_dl_service_set_link_en_sequence</code>	Data Link Layer
<code>svt_pcie_pl_configuration::set_link_speed_values()</code>	Physical Layer
<code>svt_pcie_pl_configuration::set_link_width_values()</code>	Physical Layer

## 6.13 Turning Off Unused Lanes

When configured to operate in any lower link width configuration, the model asserts the `TxCmpliance` and `TxElecIdle` signals to turn OFF the unused lanes of the link. Use the following member for the turn off feature:

```
rand bit enable_pipe_reset_n_assertion_in_detect_quiet = 0;
```

This attribute controls automatic `pipe_resetn` assertion in `detect.quiet` to test the switching to the P1 method for lane turn off feature.



## 6.14 PIPE CLK as Input to the SPIPE Interface

With the 4.2 version of the PHY PIPE specification, Synopsys added support for using the `pipe_clk` as an input to the SPIPE model. To use the `pipe_clk` as an input, do the following.

1. Instantiate the correct model. Use one of these models depending on the number of lanes you require:
  - `svt_pcie_device_agent_spipe_rev4_2_pclk_input_x16_8g_hdl.sv`
  - `svt_pcie_device_agent_spipe_rev4_2_pclk_input_x32_8g_hdl.sv`
  - `svt_pcie_device_agent_spipe_rev4_2_pclk_input_x4_8g_hdl.sv`
  - `svt_pcie_device_agent_spipe_rev4_2_pclk_input_x8_8g_hdl.sv`
2. Define `PCIESVC_PIPE_PCLK_AS_PHY_INPUT`. Use:  
`+define+PCIESVC_PIPE_PCLK_AS_PHY_INPUT`

The previous models can be found in the following installation directory:

`/your_DW_install/vip/svt/pcie_svt/latest/verilog/src/vcs`

## 6.15 PIPE Coefficient Use

Before using the PIPE preset coefficient feature of the model, you must first enable the feature with the following configuration members of the `svt_pcie_pl_configuration` class. The configuration members are used for both SPIPE and MPIPE.

- **enable\_get\_local\_preset\_coefficients.** Before you can use the PIPE `GetLocalPresetCoefficients` interface of the PHY you must enable those signals using the `enable_get_local_preset_coefficients` member. If you have a SERDES interface, this parameter has no effect as this interface is only present in a PIPE model (version 4.0 or greater).
- **MPIPE model:** If the `enable_get_local_preset_coefficients` is a "1", then it enables the per-Lane `GetLocalPresetStateMachine` to drive the per-Lane `get_local_preset_coefficients/local_preset_index[3:0]` outputs and interpret the per-Lane `local_tx_preset_coefficients[17:0]/local_tx_coefficients_valid` inputs. If the `enable_get_local_preset_coefficients` is a "0", there will be no activity presented on the previous outputs nor any response to the previous inputs.
- **SPIPE model.** If the `enable_get_local_preset_coefficients` is a "1", it enables the per-Lane `"RespLocalTxPresetCoeffStateMachine"` to interpret a `get_local_preset_coefficients/local_preset_index[3:0]` input request and respond with a per-Lane `"local_tx_preset_coefficients[17:0]/local_tx_coefficients_valid"` output. If the `enable_get_local_preset_coefficients` is a "0", there will be no response to any per-Lane `get_local_preset_coefficients/local_preset_index[3:0]` input request.
- **enable\_get\_local\_preset\_coefficients\_checking.** When set to a "1", you enable the model to check the returned preset coefficient values from the PHY against the values in its "preset mapping table". The preset to coefficients mapping table is used to map received preset requests to coefficients for use in local transmitter settings.

**Table 6-8 PIPE GetLocalPresetCoefficients Signals**

Name	Direction	Active Level	Description
GetLocalPresetCoefficients	Input	High	<p>A MAC holds this signal high for one PCLK cycle requesting a preset to co-efficient mapping for the preset on LocalPresetIndex[3:0] to coefficients on LocalTxPresetCoefficient[17:0]</p> <p>Maximum Response time of PHY is 128 nSec.</p> <p>Note. A MAC can make this request any time after reset.</p> <p>Note. After a local preset coefficient request a MAC could assert GetLocalPresetCoefficients again as soon as the next PCLK after LocalTxCoefficientsValid deasserts.</p>
LocalPresetIndex[3:0]	Input	NA	<p>Index for local PHY preset coefficients requested by the MAC</p> <p>The preset index value is encoded as follows:</p> <p>0000b – Preset P0.  0001b – Preset P1.  0010b – Preset P2.  0011b – Preset P3.  0100b – Preset P4.  0101b – Preset P5.  0110b – Preset P6.  0111b – Preset P7.  1000b – Preset P8.  1001b – Preset P9.  1010b – Preset P10.  1011b – Reserved  1100b – Reserved  1101b – Reserved  1110b – Reserved  1111b – Reserved.</p>
LocalTxCoefficientsValid	Output	High	<p>A PHY holds this signal high for one PCLK cycle to indicate that the LocalTxPresetCoefficients[17:0] bus correctly represents the coefficients values for the preset on the LocalPresetIndex bus.</p>

**Table 6-8 PIPE GetLocalPresetCoefficients Signals (Continued)**

Name	Direction	Active Level	Description
LocalTxPresetCoefficients[17:0]	Output	NA	<p>These are the coefficients for the preset on the LocalPresetIndex[3:0] after a GetLocalPresetCoefficients request:</p> <p>[5:0] C-1</p> <p>[11:6] C0</p> <p>[17:12] C+1</p> <p>Valid on assertion of LocalTxCoefficientsValid.</p> <p>The MAC will reflect these coefficient values on the TxDeemph bus when MAC wishes to apply this preset.</p>

You fill in the preset mapping table using PHY layer configuration data member `preset_to_coefficients_mapping_table[16]`. Its declaration is as follows:

```
rand bit[17:0] preset_to_coefficients_mapping_table[16] = '{ 18'h0c900, 18'h0c900,
18'h0c900, 18'h0c900, 18'h0c900, 18'h0c900, 18'h0c900, 18'h0c900,
18'h0c900, 18'h0c900, 18'h0c900, 18'h0c900, 18'h0c900, 18'h0c900, 18'h0c900};
```

When the model is acting as an SPIPE, you must also set the min and max delay value from when the model will respond by asserting values on the `local_tx_preset_coefficients_<n>` set of signals. The parameters you must set are:

- **min\_spipe\_preset\_coefficients\_delay**. Default value is 4 time units. If performing as the PIPE slave for the GetLocalPresetCoefficients interface, this is the minimum number of “PCLK cycles that a ‘per Lane’ “Respond Local Tx Preset Coefficients” state machine will wait in the `RESP_LOCAL_TX_COEFF_DELAY` state before asserting the “LocalTxCoefficientsValid” and “LocalTxPresetCoefficients[17:0]” PIPE interface signals and proceeding to completion. Selection is random between `MIN_SPIPE_PRESET_COEFFICIENTS_DELAY` and `MAX_SPIPE_PRESET_COEFFICIENTS_DELAY`.
- **max\_spipe\_preset\_coefficients\_delay**. Default value is 8 time units. This is the “Max” value that is paired with `MIN_SPIPE_PRESET_COEFFICIENTS_DELAY` above.

**Attention**

The model generates a random value which is between the min and max values set by the previous configuration members.

### 6.15.1 PHY PIPE GetLocalPresetCoefficients Interface ASCII Signals

The following table shows ASCII signals for the PHY PIPE GetLocalPresetCoefficients interface.

**Table 6-9 ASCII Signal Names for PHY PIPE GetLocalPresetCoefficients Interface.**

Signal Name	Description
<code>ascii_pipe_lane0_get_coeff_state</code>	Current state of Lane 0's GetLocalPresetStateMachine state machine
<code>ascii_pipe_lane1_get_coeff_state</code>	Current state of Lane 1's GetLocalPresetStateMachine state machine

**Table 6-9** ASCII Signal Names for PHY PIPE GetLocalPresetCoefficients Interface. (Continued)

Signal Name	Description
ascii_pipe_lane30_get_coeff_state	Current state of Lane 30's GetLocalPresetStateMachine state machine
ascii_pipe_lane31_get_coeff_state	Current state of Lane 31's GetLocalPresetStateMachine state machine
ascii_pipe_lane0_coeff_response_state	Current state of Lane 0's RespLocalTxPresetCoeffStateMachine state machine
ascii_pipe_lane1_coeff_response_state	Current state of Lane 1's RespLocalTxPresetCoeffStateMachine state machine
ascii_pipe_lane30_coeff_response_state	Current state of Lane 30's RespLocalTxPresetCoeffStateMachine state machine
ascii_pipe_lane31_coeff_response_state	Current state of Lane 31's RespLocalTxPresetCoeffStateMachine state machine

## 7

## Using the PCIe Verification IP

---

This chapter discusses the following topics:


- [“SystemVerilog UVM Example Testbenches” on page 96](#)
- [“Installing and Running the Examples” on page 98](#)
- [“Messaging Usage” on page 91](#)
- [“Controlling Verbosity From the Command Line” on page 102](#)
- [“Some Configuration Values to Set” on page 101](#)
- [“UVM Reporting Levels” on page 102](#)
- [“Resetting the PCIe VIP” on page 104](#)
- [“Creating and Using Custom Applications” on page 105](#)
- [“Backdoor Access to Completion Target Configuration Space” on page 106](#)
- [“Using ASCII Signals” on page 110](#)
- [“Using the Ordering Application” on page 89](#)
- [“Using the reconfigure\\_via\\_task Call” on page 113](#)
- [“Configuring Trace File Output” on page 113](#)
- [“Setting Coefficient and Preset for Gen3 Equalization” on page 113](#)
- [“Target Application” on page 102](#)
- [“Requester Application” on page 121](#)
- [“What Are Blocking and Non-blocking Reads in PCIe SVT?” on page 121](#)
- [“Using Service Class Reset App” on page 124](#)
- [“Programming Hints and Tips” on page 130](#)
- [“PCIe VIP Bare COM Support” on page 134](#)
- [“Up/Down Configure” on page 135](#)

- “Lane Reversal” on page 135
- “Lane Reversal with Different Link Width Configurations” on page 136
- “User-Supplied Memory Model Interface” on page 138
- “External Clocking and Per Lane Clocking for Serial Interface” on page 139
- “Receiver Margining” on page 141


## 7.1 SystemVerilog UVM Example Testbenches

This section describes SystemVerilog UVM example testbenches that show general usage for various applications. A summary of the examples is listed in [Table 7-1](#)

**Table 7-1 SystemVerilog Example Summary**

Example Name	Level	Description
	Basic	<p>The example consists of the following:</p> <ul style="list-style-type: none"> <li>• A top-level module, which includes tests, instantiates interfaces, HDL interconnect wrapper, generates system clock, and runs the tests</li> <li>• A base test, which is extended to create a directed and a random test</li> <li>• The tests create a testbench environment, which in turn creates PCIe System Env</li> <li>• PCIe System Env is configured with Root Complex and one Endpoint</li> </ul> <p> <b>Note</b> PCIe UVM Basic example Quickstart is located at: <code>\$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/ examples/sverilog/tb_pcie_svt_uvm_basic/doc/ tb_pcie_svt_uvm_basic/</code></p>

**Table 7-1 SystemVerilog Example Summary (Continued)**

Example Name	Level	Description
tb_pcie_svt_uvm_intermediate_sys	Intermediate	<p>The example consists of the following:</p> <ul style="list-style-type: none"><li>• A top-level module, which includes tests, instantiates interfaces, HDL interconnect wrapper, generates system clock, and runs the tests</li><li>• A base test, which is extended to create a directed and a random test</li><li>• The tests create a testbench environment, which in turn creates PCIe System Env</li><li>• PCIe System Env is configured with one Root Complex and one Endpoint</li><li>• Shows how to reconfigure the PCIe link for speed and lane width</li><li>• Coverage generation</li></ul> <p> <b>Note</b> PCIe UVM Basic example Quickstart is located at: <i>\$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/examples/sverilog/tb_pcie_svt_uvm_intermediate/doc/tb_pcie_svt_uvm_intermediate/</i></p> <ul style="list-style-type: none"><li>• Functional coverage is enabled</li><li>• Protocol Analyzer is enabled</li><li>• Symbol and transaction log are enabled</li></ul>

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 basic example are:

ts.base\_completer\_callback\_pipe\_test.sv  
ts.base\_completer\_callback\_pma\_test.sv  
ts.base\_completer\_callback\_serdes\_test.sv  
ts.base\_pipe\_test.sv  
ts.base\_pma\_test.sv  
ts.base\_serdes\_test.sv  
ts.directed\_pipe\_test.sv  
ts.directed\_pma\_test.sv  
ts.directed\_serdes\_test.sv

The tests in the intermediate example are:

ts.base\_pipe\_test.sv  
ts.base\_pma\_test.sv

```
ts.base_serdes_test.sv
ts.directed_pipe_test.sv
ts.directed_pma_test.sv
ts.directed_serdes_test.sv
ts.base_pipe_8g_test.sv
ts.base_pma_8g_test.sv
ts.base_serdes_8g_test.sv
```

## 7.2 Installing and Running the Examples

Below are the steps for installing and running example `tb_pcie_svt_uvm_basic_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_uvm_basic_sys -svtb
```

This installs the example under:

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

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

- a. Use the Makefile:

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

- i. `ts.base_test.sv`
- ii. `ts.directed_test.sv`
- iii. `ts.random_wr_rd_test.sv`

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

```
gmake USE_SIMULATOR=vcsvlog directed_test WAVES=1
```

To see more options, invoke "gmake help".

- b. Use the sim script:

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

```
./run_pcie_svt_uvm_basic_sys -w random_wr_rd_test vcsvlog
```

To see more options, invoke " ./run\_pcie\_svt\_uvm\_basic\_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_uvm_basic_sys
/README
```

or

```
<design_dir>/examples/sverilog/pcie_svt/tb_pcie_svt_uvm_basic_sys/README
```



## 7.2.1 Modifications to Run the Intermediate Example in Gen3 Using x8 with the PIPE Interface

Use the following procedure to change the intermediate example files to use x8 and Gen3.

1. Go to `src/verilog/vcs/` directory.
2. Choose the desired model, for example: `svt_pcie_device_agent_s(m)pipe_x8_8g_hdl.sv` for x8 8G.
3. Go to `hdl_interconnect` directory.
4. Create a new file `svt_pcie_8g_device_pipe_x8.sv` and instantiate the above MPIPE/SPIPE Verilog modules and make connections by taking a reference of the existing x4 pipe connection.
5. Create a new top file `top.pcie_8g_device_pipe_x8.sv` which will have no change but the ``include` of the above `svt_pcie_8g_device_pipe_x8.sv`.
6. Modify the prescript in order to make `top.svt_pcie_device_pipe_x8.sv` as `top_test.sv`.

Alternatively, instead of modifying the prescript, you can overwrite the `svt_pcie_8g_device_pipe_x4.sv` with instantiation of `svt_pcie_device_agent_s(m)pipe_x8_8g_hdl.sv` and make the requisite connections

7. Modify the configuration attributes for the required configuration.

For example, the `link_width` needs to be changed when going from x4 to x8 in the following files:

`env/pcie_8g_device_test_base.sv`

`env/pcie_shared_cfg.sv`

In each of those files, change the argument from 4 to 8 for the `set_link_width_values()` task:

```
cust_cfg.root_cfg.pcie_cfg.pl_cfg.set_link_width_values(8);
```

## 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.

```
UVM_ERROR svt_err_check_stats.sv(817) @ 226895734.80 ps:
uvm_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:** "uvm\_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/UVM\_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:

```
UVM_INFO svt_err_check_stats.sv(817) @ 226895734.80 ps: uvm_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 UVM\_INFO verbosity settings of UVM\_LOW, UVM\_HIGH, and UVM\_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\_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. SetAllocatedCredits
  - 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 uvm_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 uvm_test;
...
    virtual function void build_phase(uvm_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;
        ...

        `uvm_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 UVM Reporting Levels

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

- UVM\_NONE - Only print error messages
- UVM\_LOW - Print important messages that are not error messages
- UVM\_MEDIUM - Drivers and monitors print transactions they transmit and receive
- UVM\_HIGH - Print more detailed messages about component operation
- UVM\_FULL - Print internal debug messages

## 7.6 Controlling Verbosity From the Command Line

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

UVM Options are shown below.

**Table 7-2 UVM Verbosity Options**

UVM Option	Description
+UVM_VERBOSITY	setting for all components
+uvm_set_verbosity	Granular control by component and phase or time
+uvm_set_action	Action to take upon message (None, display, log, count, stop, exit, hook)
+uvm_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 UVM. The `uvm_cmdline_processor` looks for the `+UVM_VERBOSITY` option on the simulator command line, and will set the initial verbosity for all UVM components to the supplied level.

Examples:

```
// Display only UVM_FATAL, UVM_ERROR, UVM_WARNING
simv +UVM_TESTNAME=base_pipe_test +UVM_VERBOSITY=UVM_NONE
// Display all messages
simv +UVM_TESTNAME=base_pipe_test +UVM_VERBOSITY=UVM_FULL
```

### Per Component:

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

`+uvm_set_verbosity=<component_name>, <id>, <verbosity>, <phase_name>`

or

`+uvm_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 UVM_NONE except for the tl which is at UVM_LOW
simv +UVM_TESTNAME=base_pipe_test +UVM_VERBOSITY=UVM_NONE \
+uvm_set_verbosity=uvm_test_top.env.root.port0.tl0, _ALL_, UVM_LOW, time, 0
// Set all components to UVM_NONE except for root
simv +UVM_TESTNAME=base_pipe_test +UVM_VERBOSITY=UVM_NONE \
+uvm_set_verbosity=uvm_test_top.env.root.*", _ALL_, UVM_LOW, time, 0
```

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

Example:

```
// UVM_NONE for all with the exception that all instances of the dl are at UVM_LOW and
// all instances of the tl are at UVM_FULL
simv +vip_verbosity=svt_pcie_dl:UVM_LOW,svt_pcie_tl:UVM_FULL
```

### SVT Verbosity relationship to UVM Severity Levels

```

`define SVT_FATAL_VERBOSITY UVM_NONE
`define SVT_ERROR_VERBOSITY UVM_NONE
`define SVT_WARNING_VERBOSITY UVM_NONE
`define SVT_NORMAL_VERBOSITY UVM_LOW
`define SVT_TRACE_VERBOSITY UVM_MEDIUM
`define SVT_DEBUG_VERBOSITY UVM_HIGH
`define SVT_VERBOSE_VERBOSITY UVM_FULL

```

## 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;
`uvm_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`:

```
`uvm_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
`uvm_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;
`uvm_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:

```
`uvm_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 through 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_uvm_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_uvm_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
    `uvm_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
    `uvm_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;
    `uvm_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
    `uvm_send(cfg_database_seq);

```

```

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

`uvm_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;
`uvm_send(cfg_database_seq);
if(cfg_database_seq.command_status != `SVT_PCIE_CFG_DATABASE_STATUS_SUCCESSFUL)
  `uvm_error("body", $sformatf("Command status not SUCCESSFUL(0x%h) received
    0x%h", `SVT_PCIE_CFG_DATABASE_STATUS_SUCCESSFUL,
      cfg_database_seq.command_status))
  `uvm_info("body", $sformatf("Config_reg 0 is backdoor read data=0x%x",
    cfg_database_seq.dword_data), UVM_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.

**Table 7-4 Data Link Layer ASCII signals (Continued)**

Signal Name	Description
ascii_rx_dllp_credit_data_value	Received DLLP data credit value.
ascii_rx_dllp_credit_hdr_value	Received DLLP header credit value.
ascii_dlcsm_state	Data Link Control Management State Machine.
ascii_vc[0-7]_fcsn_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 [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_lanen_rx_data	Data received on lane <i>n</i> , where <i>n</i> is a number between 0 and 31.
ascii_lanen_tx_data	Data transmitted on lane <i>n</i> , where <i>n</i> is a number between 0 and 31.
ascii_pipe_lanen_rx_data;	Symbol data on received on PIPE lane <i>n</i> , where <i>n</i> is a number between 0 and 31.
ascii_pipe_lanen_tx_data	Symbol data on transmitted on PIPE lane <i>n</i> , where <i>n</i> 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 `uvm_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 `uvm_config_db` so it can be obtained by the application.

```
uvm_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`.

```
uvm_config_db #( svt_pcie_tlp_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 `uvm_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.

```
uvm_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
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**

Attribute	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.
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
<code>reject_preset_coefficient_request</code>	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
```

## 7.15.3 Preset and Coefficient Tuning Through Windowed Filtering

During `Recovery.Equalization` phases 2 and 3, the preset and coefficient of PHY transmitter are tuned for optimal signal integrity at receiver. PCIe specification describes the equalization process but does not specify the actual algorithm involving the preset and coefficient tuning. This section describes the supported optional algorithm to tune preset and coefficient through a library sequence.

**Note**

Currently, only the sequence operating on Endpoint VIP with master PIPE at Gen 3 is part of sequence library.

This section describes the preset and coefficient tuning algorithm from the perspective of VIP acting as upstream device with master PIPE interface, during `Recovery.Equalization` phase 2.

Figure of Merit (FOM) feedback mechanism is used for the process of fine tuning link partners PHY TX presets and Direction Change (DIR) feedback is used for fine tuning link partner's PHY TX coefficients. Both preset and coefficient tuning are options, MAC may choose to not to do any tuning or either one or both. If both preset and coefficient are to be tuned, preset tuning is done first followed by coefficient tuning.

**Figure of Merit (FOM)**

For preset selection through FOM feedback, MAC will send a set of user-programed presets to link partners PHY one at a time and will instruct its own PHY to evaluate incoming RX signal integrity. After PHY is done with evaluation of incoming signal, it will respond with a relative score for the preset sent, ranging from 0 to 255 through `LinkEvaluationFeedbackFigureMerit[7:0]` PIPE signal. Higher the FOM feedback value, better is the incoming RX signal integrity. Once MAC has evaluated all the programed presets, it will select the preset with the highest FOM feedback value.

**Direction Change (DIR)**

For coefficient tuning through DIR feedback, MAC sends coefficients (precursor, cursor and postcursor) for link partner PHY transmitter, then request its own PHY to evaluate the incoming RX signal integrity. After PHY is done with evaluating the incoming signal integrity, it will provide the feedback on `LinkEvaluationFeedbackDirectionChange[5:0]` PIPE signal and 2-bit feedback per coefficient. A value of 2'b00 indicates no further change is required for coefficient, a value of 2'b01 means increment the coefficient value by 1 and a value of 2'b10 means decrement value of coefficient by 1. Value of 2'b11 is reserved. Coefficient tuning is considered done if DIR feedback of zero (no change) is received on all lanes at the same time, this approach might take a long time, something may result in oscillatory behavior with settling down and DIR feedback does not converge on zero for all lanes at the same time. An alternate approach will be to use Windowed Filtering for DIR convergence.

**DIR Convergence Criteria of Windowed Filtering**

The simple convergence of DIR feedback is to get 2'b00 corresponding to precursor and postcursor coefficients on all relevant lanes, at times this can result in a scenario where the feedback oscillates between increment by 1 (2'b01) and decrement by 1 (2'b10) without settling down no change (2'b00) or DIR feedback of 2'b00 is not achieved for all lanes simultaneously. To overcome these issues, an alternate option of Windowed Filtering for DIR Convergence Criteria can be used.

Windowed Filtering for DIR Convergence Criteria involve requesting at least  $D$  number of coefficients, such that the maximum difference between the coefficients values in last  $D$  attempts are less than equal to a value  $A$ . The parameter  $D$  is Convergence Window Depth and  $A$  is Convergence Window Aperture.

**7.15.3.1 Equalization Windowing Endpoint (Upstream Port) Sequence**

The `svt_pcie_device_virtual_endpoint_mpipe_8g_equalization_seq` sequence will respond to equalization process initiated by downstream port. If the equalization process involves phase 2 and phase 3, in phase 2 it will provide means to fine tune downstream port PHY transmitter preset/coefficient and in Phase 3, it responds to downstream port's attempts to fine tune upstream port PHY's transmitter preset/coefficients.

This sequence performs equalization through Endpoint with MPIPE at Gen3 speed. The sequence takes control during equalization phase 2 when Endpoint (upstream port) is the equalization master. Sequence can be programmed to tune preset, coefficients or both. Endpoint in MPIPE mode in equalization phase 2 sends presets and coefficients to link partner PHY and after getting acknowledgment from link partner, instructs Endpoint PHY to do evaluation of preset/coefficients (by asserting `rx_eq_eval`). Endpoint in MPIPE mode in equalization phase 3 receives the preset/coefficients from link partner and passes it on to Endpoint PHY through MPIPE signals (`TxDemph`).

For preset tuning in phase 2, sequence can be programmed to send multiple presets, sequence keeps track of all the preset and FOM feedback they received. After sending all the presets, sequence picks the preset with highest FOM.

For coefficients tuning, sequence can be programmed through `dir_convergence_mode_8g`, either use Windowed Filtering method or feedback of zero on all lanes.

Coefficients tuning with feedback of zero on all lanes, sequence will calculate next set of coefficients based on DIR feedback, sends the new coefficients until DIR feedback precursor and postcursor coefficients are zero.

In Windowed Filtering convergence mode, sequence must be programmed with `dir_convergence_window_depth_8g` and `dir_convergence_window_aperture_8g`.

- `dir_convergence_window_depth_8g` is the minimum number of coefficient sets to be tried.
- `dir_convergence_window_aperture_8g` is the acceptable difference between the minimum and maximum value of coefficient in last  $D$  tries, where  $D$  is `dir_convergence_window_depth_8g`.

Coefficient tuning during Windowed Filtering is considered done if the delta between the minimum and maximum coefficient is equal to or less than `dir_convergence_window_aperture_8g` for `dir_convergence_window_depth_8g`.



#### Note

This sequence will run only on upstream port (Endpoint) agent with MPIPE interface, otherwise results in fatal error.

**Table 7-9 Controls**

Control	Description
rand bit <code>tune_remote_tx_preset_8g</code>	If set, Endpoint will request preset in <code>preset_vector_8g</code> and find the best one through FOM feedback process.
rand bit [15:0] <code>preset_vector_8g</code>	List of presets to try. Each bit corresponds to a respective preset encoding, —that is, if <code>preset_vector_8g [0]</code> is set transmitter preset encoding "0000" will be requested. Same preset will be tried on all lanes as per the implemented algorithm.
rand bit <code>tune_remote_tx_coefficient_8g</code>	If set, sequence will do coefficient tuning in equalization phase 2.
rand int unsigned <code>max_iteration_for_coeff_convergence_8g</code>	Maximum number of iteration sequence will try for coefficient tuning process to converge. If coefficient tuning process does not converge even after reaching the maximum number of iterations, sequence will exit coefficient tuning with a warning.

**Table 7-9 Controls**

Control	Description
rand bit dir_convergence_mode_8g	Coefficient tuning convergence mode <ul style="list-style-type: none"><li>0 - Feedback of zero on all lanes</li><li>1 - Windowed filtering.</li></ul>
rand int unsigned dir_convergence_window_depth_8g	The minimum number of coefficient sets iteration to be tried before checking if convergence criteria is met. Only used if dir_convergence_mode_8g is windowed filtering.
rand int unsigned dir_convergence_window_aperture_8g	The acceptable difference between the minimum and maximum value of coefficient in last <i>N</i> iteration, when <i>N</i> is dir_convergence_window_depth_8g. Only used if dir_convergence_mode_8g is windowed filtering.

## 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 requests 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 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_uvm_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_put", $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

// UVM 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(uvm_phase phase);
        super.end_of_elaboration_phase(phase);

        set_ep_target_cb = new("set_ep_target_cb");
```

```
uvm_callbacks#(svt_pcie_target_app,svt_pcie_target_app_callback)::add(env.endpoint.target[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 of 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:
    `uvm_do_on_with(cfg_rd_rq,p_sequencer.root_virt_seqr.driver_transaction_seqr[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:

```
`uvm_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 7-10 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.
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			

**Table 7-10 SKP Ordered Set Configuration Members (Continued)**

Type	Range	Default	Description
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 7-11](#) on page 125 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 7-11 Service Class App Sequence Resets**

Reset Sequence for Application	Description
svt_pcie_requester_app_service_reset_app_sequence	<ul style="list-style-type: none"><li>• This sequence implements ResetApp</li><li>• RESET_APP is a reset for the Driver</li><li>• 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.</li><li>• It is not necessary to call Reset App at the beginning of simulation</li></ul>
svt_pcie_io_target_service_reset_app_sequence	<ul style="list-style-type: none"><li>• This sequence implements Reset App</li><li>• RESET_APP resets the application back to its initial state. All data will be lost.</li><li>• It is not necessary to call this at start of sim.</li></ul>

**Table 7-11 Service Class App Sequence Resets (Continued)**

Reset Sequence for Application	Description
svt_pcie_mem_target_service_reset_app_sequence	<ul style="list-style-type: none"> <li>This sequence implements Reset App</li> <li>RESET_APP resets the memory target back to its initial state. All data will be lost.</li> <li>It is not necessary to call this at start of sim</li> </ul>
svt_pcie_requester_app_service_reset_app_sequence	<ul style="list-style-type: none"> <li>This sequence implements Reset App</li> <li>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.</li> <li>It is not necessary to call RESET_APP at the start of simulation.</li> </ul>
svt_pcie_target_app_service_reset_app_sequence	<ul style="list-style-type: none"> <li>This sequence implements ResetApp</li> <li>RESET_APP resets the target application. All outstanding requests are dropped and will not be completed.</li> <li>It is not necessary to call RESET_APP at the start of simulation.</li> </ul>

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_uvm_do(link_up_seq);

    // Add memory ranges to Requester application

    `svt_uvm_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_uvm_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_uvm_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_uvm_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_uvm_do_on(reset_target_app_seq,p_sequencer.root_virt_seqr.target_seqr[0]);
*/

->seq_to_test1;
uvm_report_info( "TEST", "Triggering seq_to_test1 event.", UVM_NONE );

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

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

->seq_to_test2;
uvm_report_info( "TEST", "Triggering seq_to_test2 event.", UVM_NONE );

`svt_uvm_do_on(device_reset_vseq, p_sequencer.endpoint_virt_seqr);

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

`svt_uvm_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_uvm_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_uvm_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_uvm_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_uvm_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_uvm_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_uvm_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_uvm_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_uvm_do_on(reset_target_app_seq,p_sequencer.root_virt_seqr.target_seqr[0]);
    */

    ->seq_to_test1;
    uvm_report_info( "TEST", "Triggering seq_to_test1 event.", UVM_NONE );

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

`svt_uvm_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_uvm_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_uvm_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_uvm_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_uvm_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_uvm_do_on(req_disp_stats_seq,p_sequencer.endpoint_virt_seqr.requester_seqr);
`svt_uvm_do_on(req_clr_stats_seq,p_sequencer.endpoint_virt_seqr.requester_seqr);
`svt_uvm_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:

```

`uvm_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, through the custom application interface. 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 given request has been translated via the ATS protocol. It is up to the end user's application or test to utilize the AT bit field in the support of ATS.

For additional information, [“Creating and Using Custom Applications”](#) on page 105.

### 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 the 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 shows 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 `uvm_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 `uvm_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 `uvm_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 its 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.
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 want 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.

```
`uvm_info(get_type_name(), $sformatf("Begin process set_tx_ts1_pattern_seq 2nd  
time\n"), UVM_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 COMs*, all the receiving model's `ordered_set_checker` modules requires message suppression using the `MSGCODE_PCIE SVC_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\_configuration 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\_configuration: link\_width\_value)
- supported\_link\_width\_vector\_value (svt\_pcie\_pl\_configuration: supported\_link\_width\_vector\_value)
- expected\_link\_width\_value (svt\_pcie\_pl\_configuration: expected\_link\_width\_value)
  - The [link\\_width\\_value \(svt\\_pcie\\_pl\\_configuration: link\\_width\\_value\)](#) must be same as DUT link width value.
  - The [supported\\_link\\_width\\_vector\\_value \(svt\\_pcie\\_pl\\_configuration: supported\\_link\\_width\\_vector\\_value\)](#) 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

**Table 7-13 Controls for Supported Link Width Vector**

Link Width	supported_link_width_vector
4	32'h07
8	32'h0F
12	32'h1F
16	32'h3F;
32	32'h7F;

- [expected\\_link\\_width\\_value \(svt\\_pcie\\_pl\\_configuration: expected\\_link\\_width\\_value\)](#): 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_configuration` 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(uvm_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\_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.

## 7.29 Receiver Margining

The following steps illustrate the flow of Receiver Margining. Steps 1–3 are only done when accessing remote Receiver in Retimer.

1. Obtain the capabilities of the Receiver.
2. Obtain the maximum number of lanes to margin simultaneously.
3. Set Error Count Limit (if needed).
4. Step Margin Timing/Voltage (Clear Error Log).
5. Go to normal settings.

### 7.29.1 RC (Upstream Component) Lane Margin of Remote PHY in Retimer Using CTRL-SKP (VIP Acting as MAC)

1. Configure device with values for Rx Margin attributes.
2. Wait for RC to reach 16G speeds and then enter L0 with data stream turned on.
3. Issue service request to send REPORT\_MARGIN\_CONTROL\_CAPABILITIES command using CTRL\_SKP\_CMD.
4. Wait for RC to receive a CTRL-SKP with margin\_type set to 3'b001 and receiver\_number matching requested address using attributes in the svt\_pcie\_ctrl\_skp\_receiver\_status class.

5. Then wait for the reception of a reflected `No Command`. There is no need to issue this command using the service request since the VIP will automatically send a `No Command` if no other command is issued.
6. If required, issue additional `REPORT` commands one at a time and wait for their reply using the above method to request additional configuration values from port that is going to be margined.
7. Issue service request to send `SET_ERROR_COUNT_LIMIT` command using `CTRL_SKP_CMD`.
8. Wait for RC to receive a CTRL-SKP with `margin_type` set to `3'b010` and `receiver_number` matching requested address and expected returned error count limit using attributes in the `svt_pcie_ctrl_skp_receiver_status` class.
9. Wait for the reception of a reflected `No Command`. There is no need to issue this command using the service request since the VIP will automatically send a `No Command` if no other command is issued.
10. Issue service request to send `STEP_MARGIN_TO_TIMING_OFFSET` or `STEP_MARGIN_TO_VOLTAGE_OFFSET` command with appropriate `PAYLOAD_VALUES` using `CTRL_SKP_CMD`.
11. Continue monitoring the status of the received CTRL-SKP OS's for margin status and error count and take the appropriate action based upon the received status. The `svt_pcie_ctrl_skp_receiver_status` gets updated every time the VIP receives a new CTRL-SKP OS and the testbench can monitor the reception by waiting for changes to the `num_ctrl_skp_received`.
12. If you want to run longer, issue service request to clear error log if need be during Step operation.
13. Terminate Step Margin operation by sending service request to `GO_TO_NORMAL_SETTING`.
14. Go back and issue a new Step Margin request or finish margining.

## 7.29.2 EP (Downstream Component) Lane Margin of Remote PHY in Retimer Using CTRL-SKP (VIP Acting as Retimer PHY)

1. Configure CTRL-SKP response attributes for each Retimer device the VIP is modeling in addition to configuring the CTRL-SKP delay (`margin_response_time`).
2. When a Rx Margin report command is received via a CTRL-SKP OS to a receiver address that the device supports, respond back in the next CTRL-SKP with the appropriate configuration attribute based upon receiver number. This includes request for maximum number of lanes supported.
3. When a set error limit command is received, update the internal error count limit register and send back a response in the next CTRL-SKP OS.
4. When a Step Margin Timing or Voltage command is received, start internal Rx margining processing, respond back with CTRL-SKP indicating execution status of Setup in progress.
5. After configured delay, switch to sending CTRL-SKP responses indicating Margining in progress.
6. Issue service request `INCREMENT_RX_MARGIN_BIT_ERROR_COUNT` to device to indicate that a bit error has occurred. Continue sending bit error service requests whenever you want to simulate a bit error failure. When the received error count reaches the internal error limit, change `Execution` status in the CTRL-SKP response to `Too Many Errors`.
7. If a CTRL-SKP with clear error log is received, reset the internal error count if `Execution` status is still margining in progress.
8. When a CTRL-SKP is received with a `Go to Normal settings` command, terminate the internal margining process.

### 7.29.3 RC (Upstream Component) or EP (Downstream Component) Lane Margin of Local PHY Using MBI (VIP Acting as MAC)

There are two different methods for generating MBI requests when the VIP is modeling the MAC side of the Message Bus Interface (MBI).

The first method uses the same service requests as the remote PHY CTRL-SKP service requests by mapping RC CTRL-SKP requests of type (`CLEAR_ERROR_LOG`, `STEP_MARGIN_TO_VOLTAGE_OFFSET`, `STEP_MARGIN_TO_TIMING_OFFSET` and `GO_TO_NORMAL_SETTINGS`) with the `receiver_number` set to 3'b010, or EP with `receiver_number` set to 3'b110 to corresponding MBI requests. The CTRL-SKP will not be updated and the device will continue to send CTRL-SKP with No Command.

The second method uses a more fine-grained command specific to the MBI using a new service request type `MBI_CMD`. This service request can be used to send `NOP`, `WRITE_UNCOMMITTED`, `WRITE_COMMITED` and `WRITE_ACK` commands.

#### 7.29.3.1 Generating MBI Requests Using CTRL-SKP

Perform the following steps to generate MBI requests using CTRL-SKP:

1. Wait for RC to reach 16G speeds and then enter L0 with data stream turned on.
2. Issue service request to send `STEP_MARGIN_TO_TIMING_OFFSET` or `STEP_MARGIN_TO_VOLTAGE_OFFSET` command with appropriate `PAYLOAD_VALUES` using `CTRL_SKP_CMD`.
3. Continue monitoring the status of the internal `Rx_Margin_Status` registers to detect changes to execution status, sample count and error count and take appropriate action based upon the received status changes. The internal registers get updated every time a `WRITE_COMMITED` to the status register is detected on the MBI from the PHY.
4. If you want to run longer, issue service request to `CLEAR_ERROR_LOG` if need be during Step operation.
5. Terminate Step Margin operation by sending service request to `GO_TO_NORMAL_SETTING`.
6. Go back and issue a new Step Margin request or finish margining.

#### 7.29.3.2 Generating MBI Requests Using MBI\_CMD

Perform the following steps to generate MBI requests using `MBI_CMD`:

1. Wait for RC to reach 16G speeds and then enter L0 with data stream turned on.
2. Issue service request to send `WRITE_UNCOMMITTED` and/or `WRITE_COMMITED` commands with appropriate `MBI_REGISTER` and `MBI_DATA` values using `MBI_CMD`.
3. Continue monitoring the status of the internal `Rx_Margin_Status` registers to detect changes to execution status, sample count and error count and take appropriate action based upon the received status changes. The internal registers get updated every time a `WRITE_COMMITED` to the status register is detected on the MBI from the PHY.
4. If you want to run longer, issue service request to `WRITE` command to the appropriate control register with the Error Count Reset bit set if need be during Step operation.
5. Terminate Step Margin operation by sending service request `WRITE` to `Rx_Margin_Control_0` with the Start Margin bit set to 0.
6. Go back and issue a new Step Margin request or finish margining.

#### 7.29.4 RC (Upstream Component) or EP (Downstream Component) Lane Margin of Local PHY Using MBI (VIP Acting as PHY)

1. Configure MBI response delay (`write_ack_delay`).
2. Wait for a received MBI request to `Rx_Margin_Control_0` to indicate start of margin operation (Voltage or Timing).
3. Issue service request `INCREMENT_RX_MARGIN_SAMPLE_COUNT` and `INCREMENT_RX_MARGIN_BIT_ERROR_COUNT` to device to indicate a change in the corresponding internal counter. Keep sending service requests whenever the testbench wants to simulate a change to the internal register. Each update to the internal counter would have the VIP generate MBI writes to the appropriate status register in the MAC.
4. If a `WRITE` to `Rx_Margin_Control_0` either Error Count Reset or Sample Count Reset is seen the appropriate counter is cleared.
5. When a `WRITE` to `Rx_Margin_Control_0` with Start Margin bit set to 0 is detected, the internal margining process is terminated.

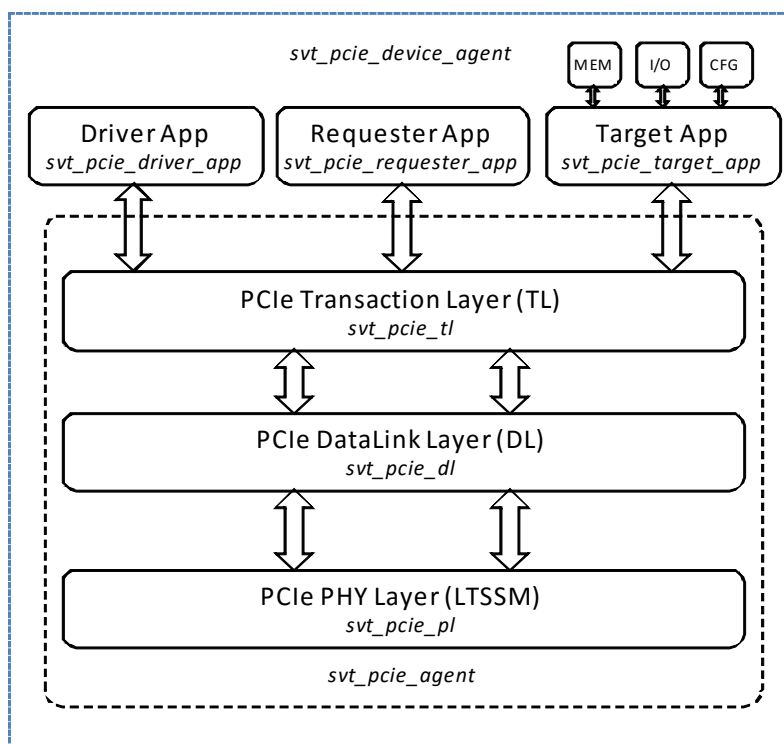
## 8

# PCIe Device Agent

## 8.1 Overview

The PCIe UVM 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 UVM 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 UVM 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”](#) on page 161.



## 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. The following table contains the attributes of this class.

**Table 8-1 Device Configuration Members**

Type	Member	Description
<code>svt_pcie_driver_app_configuration</code>	<code>driver_cfg [ int ]</code>	Holds the configuration attributes for the Driver application.
bit	<code>is_active = 1;</code>	Specifies if the agent is an active or passive component. Allowed values are: 1, Configures component in active mode. Enables sequencer and driver. 0:=, Configures component in passive mode. Enables only the monitor in the agent. Configuration type: Static
string	<code>model_instance_scope</code>	This is the full hierarchical path name to the instance of the Verilog instantiation model. The programming of this variable binds the PCIe UVM Agent with its Verilog counterpart.
<code>pipe_spec_ver_enum</code>	<code>pipe_spec_ver</code>	PCIe PIPE specification version number. Allowed values are: <code>svt_pcie_device_configuration::PIPE_SPEC_VER_2</code> , configures the PCIe PIPE specification version number to 2.0 <code>svt_pcie_device_configuration::PIPE_SPEC_VER_4</code> , configures the PCIe PIPE specification version number to 4.0 <code>svt_pcie_device_configuration::PIPE_SPEC_VER_4_2</code> , configures the PCIe PIPE specification version number to 4.2 <code>svt_pcie_device_configuration::PIPE_SPEC_VER_4_3</code> , configures the PCIe PIPE specification version number to 4.3

**Table 8-1 Device Configuration Members**

Type	Member	Description
pcie_spec_ver_enum	pcie_spec_ver	Specifies the PCIe specification version number. Allowed values are: svt_pcie_device_configuration::PCIE_SPEC_VER_1_1, configures the PCIe specification to version 1.1 svt_pcie_device_configuration::PCIE_SPEC_VER_2_0, configures the PCIe specification to version 2.0 svt_pcie_device_configuration::PCIE_SPEC_VER_2_1, configures the PCIe specification to version 2.1 svt_pcie_device_configuration::PCIE_SPEC_VER_3_0, configures the PCIe specification to version 3.0 svt_pcie_device_configuration::PCIE_SPEC_VER_3_1, configures the PCIe specification to version 3.1 svt_pcie_device_configuration::PCIE_SPEC_VER_4_0, configures the PCIe specification to version 4.0
svt_pcie_configuration	pcie_cfg	Holds the configuration attributes for the PCIe agent.
svt_pcie_requester_app_configuration	requester_cfg	Holds the configuration attributes for the Requester application.
svt_pcie_target_app_configuration	target_cfg [ int ]	Holds the configuration attributes for the Target application.

**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 instantiated and connected to the DUT. This configuration parameter binds the UVM 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 (`uvm_config_db`) class defined in UVM. The PCIe Device Agent—the recipient of the configuration object has a `uvm_config_db::get()` defined within the `build_phase()` and so the parent test/environment has to specify a `uvm_config_db::set()` in its `build_phase()`. Before calling the `uvm_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_basic_env extends uvm_env;
...
svt_pcie_device_agent root;
svt_pcie_device_configuration root_cfg
...
function void build_phase( uvm_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 UVM 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 uvm_config_db::set() to set the configuration of the UVM 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
    uvm_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 `uvm_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\\_uvm\\_class\\_reference/html/class\\_svt\\_pcie\\_device\\_configuration.html](#)

## 8.2.2 Dynamic Configuration

The configuration set during the build of the UVM 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 `uvm_config_db::set()` that is issued before calling `refresh_cfg()`.

- 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 `uvm_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 uvm_test;
  `uvm_component_utils(pcie_pipe_speed)
  ...
  task run_phase (uvm_phase phase);
    svt_configuration temp_cfg = null;
    svt_pcie_device_configuration new_cfg = null;

    super.run_phase(phase);
    ...
    // Assumptions:
    // The UVM test has an instance of UVM environment named 'env'
    // The UVM 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 uvm_test;
  `uvm_component_utils(pcie_pipe_speed)
  ...
  task run_phase (uvm_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 UVM test has an instance of UVM environment named 'env'
    // The UVM 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 uvm_config_db::set() to set the new configuration.
    uvm_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. [Table 8-2](#) contains status class members. 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` instantiated as `status`. The members of class `svt_pcie_device_status` are listed in the following table.

**Table 8-2 Status Class Members**

Class Objects	Member	Description
svt_pcie_driver_app_status	driver_status [\$]	Status of Driver application
svt_pcie_io_target_status	io_target_status	Status of IO Target application
svt_pcie_mem_target_status	mem_target_status	Status of Mem Target application
svt_pcie_status	pcie_status	Status of MAC, that is 3 layer stack Transaction layer, Data link layer and Physical layer.
svt_pcie_requester_app_status	requester_status	Status of Requester application
svt_pcie_target_app_status	target_status [\$]	Status of Target application

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 uvm_test;
  `uvm_component_utils(pcie_pipe_speed)
  ...
  task run_phase (uvm_phase phase);
    svt_pcie_device_status root_dev_status;
    super.run_phase(phase);
    ...
    // Assumptions:
    // The UVM test has an instance of UVM environment named 'env'
    // The UVM 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.

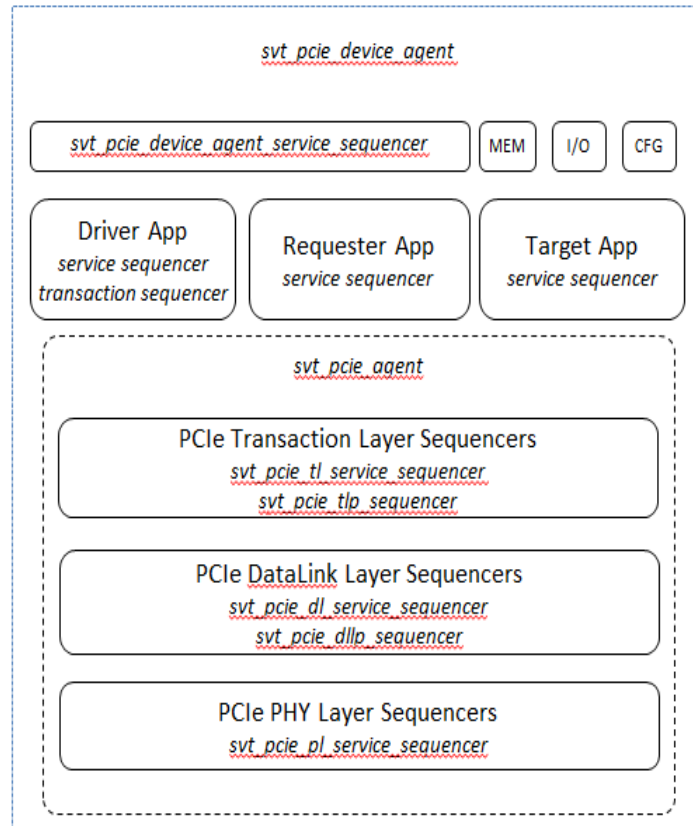
    // Do other things
  endtask
endclass
```

## 8.4 Sequencers

The PCIe Device Agent class (`svt_pcie_device_agent`) has eight UVM sequencer objects to schedule

transaction requests or service requests on any of its subcomponent Drivers. A UVM 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 UVM 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 UVM 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 UVM 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) uvm\_seq\_port of UVM 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 UVM 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 UVM 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 UVM 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 UVM 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 UVM 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 UVM 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 uvm_test;
  `uvm_component_utils(my_pcie_test)
  ...
  task run_phase (uvm_phase phase);
    svt_pcie_driver_app_service_wait_until_idle_sequence drv_app_serv_seq;
    ...
    // Assumptions:
    // The UVM test has an instance of UVM environment named 'env'
    // The UVM 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 UVM driver class svt\_pcie\_driver\_app.

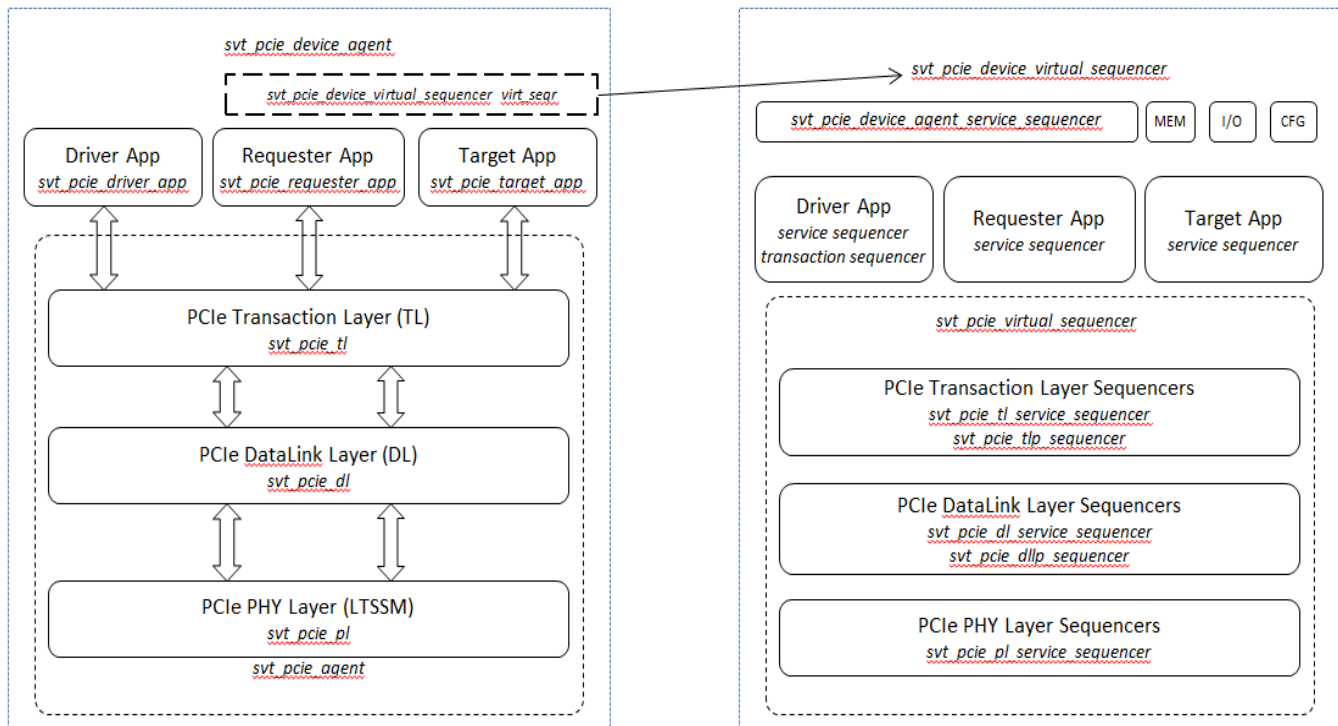
[Example 8-6](#) shows how you can send a transaction request to the driver using the driver application transaction sequencer.

### Example 8-6

```
class my_pcie_test extends uvm_test;
  `uvm_component_utils(my_pcie_test)
  ...
  task run_phase (uvm_phase phase);
    svt_pcie_driver_app_transaction_mem_write_sequence mem_wr_seq;
    ...
    // Assumptions:
    // The UVM test has an instance of UVM environment named 'env'
    // The UVM 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 instantiated 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 UVM drivers/agents as sub-components, it has the UVM sequencer of the corresponding UVM 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 UVM 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 uvm_test;
  `uvm_component_utils(my_pcie_test)
  ...
  task run_phase (uvm_phase phase);
    svt_pcie_driver_app_service_wait_until_idle_sequence drv_app_serv_seq;
    ...
    // Assumptions:
    // The UVM test has an instance of UVM environment named 'env'
    // The UVM 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 uvm_test;
  `uvm_component_utils(my_pcie_test)
  ...
  task run_phase (uvm_phase phase);
    svt_pcie_driver_app_transaction_mem_write_sequence mem_wr_seq;
    ...
    // Assumptions:
    // The UVM test has an instance of UVM environment named 'env'
    // The UVM 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 uvm_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 uvm_env;
  svt_pcie_device_agent root;
  usb_device_agent usb_dev;
  my_system_virtual_sequencer sys_virt_seqr;

  ...
endclass
```

```

function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    ...
    this.sys_virt_seqr = my_system_virtual_sequencer::create("sys_virt_seqr", this);

endfunction

function void connect_phase(uvm_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 UVM 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 uvm_test;
    `uvm_component_utils(my_pcie_test)
    ...
    task run_phase (uvm_phase phase);
        svt_pcie_driver_app_transaction_mem_write_sequeunce 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_seq.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

```



### 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.



# 9

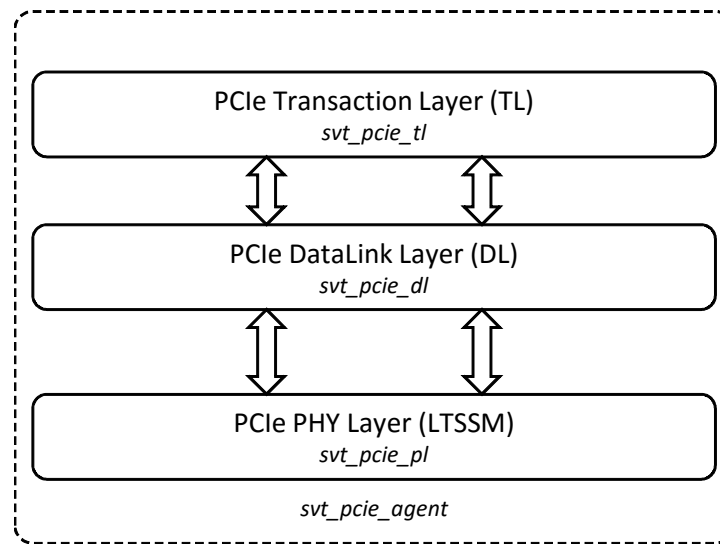
## PCIe Agent

---

### 9.1 Overview

The PCIe Agent encapsulates the UVM 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 instantiated 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 4-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\\_uvm\\_class\\_reference/html/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 `uvm_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

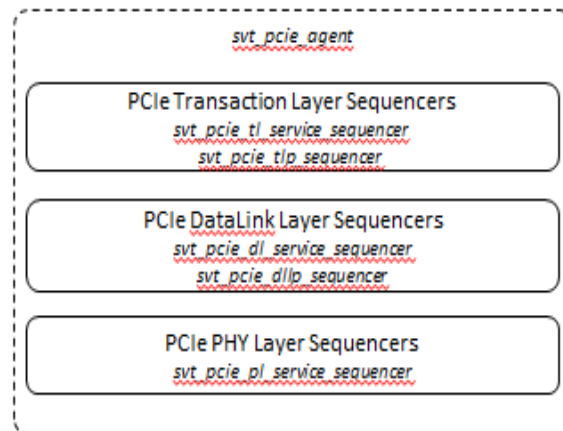
The PCIe Agent has a set of state values representing the status of its constituent blocks at any given time in the test simulation. And these state values are encapsulated within the `svt_pcie_status` class. The `svt_pcie_device_status` class has an object of type `svt_pcie_status` instantiated as `pcie_status`. The members of class `svt_pcie_status` are listed in the table below.

Class Objects	Member	Description
<code>svt_pcie_config_space_status</code>	<code>config_space_status</code>	Status class for tracking observed PCIe configuration space transactions
<code>svt_pcie_dl_status</code>	<code>dl_status</code>	Status of Link layer
<code>svt_pcie_pl_status</code>	<code>pl_status</code>	Status of Physical layer
<code>svt_pcie_tl_status</code>	<code>tl_status</code>	Status of Transaction layer

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 UVM sequencer objects to schedule transaction requests or service requests on any of its subcomponent drivers. A UVM 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 UVM 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 UVM 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 UVM 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 UVM 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 UVM 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 uvm_test;
  `uvm_component_utils(my_pcie_test)
  ...
  task run_phase (uvm_phase phase);
    svt_pcie_tl_service_set_tc_map_sequence tl_serv_seq;
    ...
    // Assumptions:
    // The UVM test has an instance of UVM environment named 'env'
  endtask
endclass
  
```

```

// The UVM 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 UVM 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 UVM 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 uvm_test;
  `uvm_component_utils(my_pcie_test)
  ...
  task run_phase (uvm_phase phase);
    svt_pcie_tlp_mem_request_sequence mem_tlp_seq;
    ...
    // Assumptions:
    // The UVM test has an instance of UVM environment named 'env'
    // The UVM environment has an instance of the PCIe device agent named 'root'

```

```

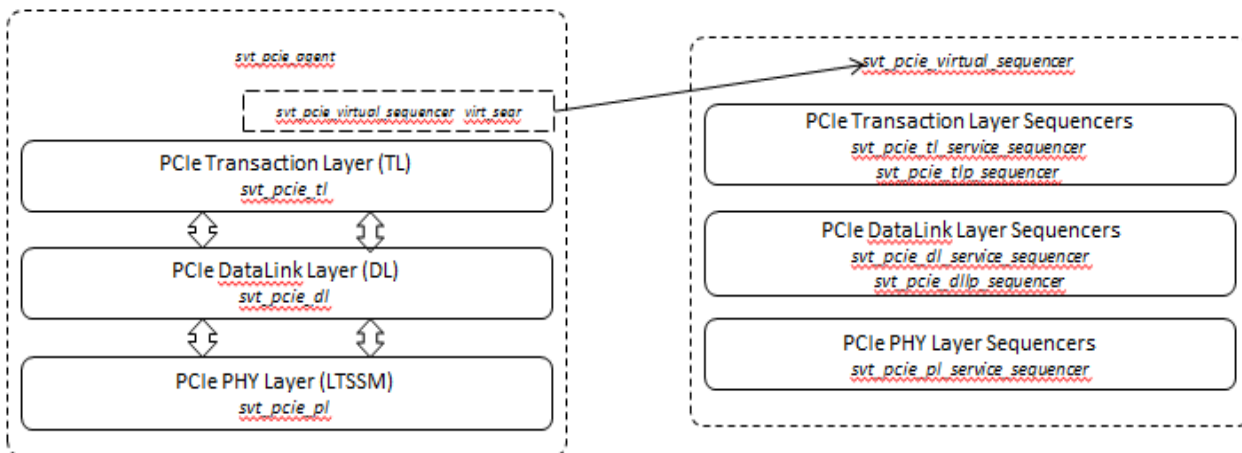
// VIP's DL is enabled.
// LTSSM is in L0
mem_tlp_seq = new();
mem_tlp_seq.randomize with {
    address == 'h8000;
    length == 2;
    requester_id == 'h100;
    tlp_type == svt_pcie_tlp::MEM_REQ;
    fmt == svt_pcie_tlp::WITH_DATA_3_DWORD;
    // Program other header fields.
    ...
    foreach(payload[i])
        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` instantiated 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 UVM drivers/agents as subcomponents, it has the UVM sequencer of the corresponding UVM 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 uvm_test;
    `uvm_component_utils(my_pcie_test)
    ...

```

```
task run_phase (uvm_phase phase);
    svt_pcie_tl_service_set_tc_map_sequence tl_serv_seq;
    ...
    // Assumptions:
    // The UVM test has an instance of UVM environment named 'env'
    // The UVM 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 uvm_test;
    `uvm_component_utils(my_pcie_test)
    ...
    task run_phase (uvm_phase phase);
        svt_pcie_tlp_mem_request_sequence mem_tlp_seq;
        ...
        // Assumptions:
        // The UVM test has an instance of UVM environment named 'env'
        // The UVM 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;
        }
    endtask
endclass
```

```

    };
    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 uvm_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 uvm_env;
svt_pcie_device_agent root;
usb_device_agent usb_dev;
my_system_virtual_sequencer sys_virt_seqr;

...

function void build_phase(uvm_phase phase);
super.build_phase(phase);

...
this.sys_virt_seqr = my_system_virtual_sequencer::create("sys_virt_seqr", this);
endfunction

function void connect_phase(uvm_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 uvm_test;
`uvm_component_utils(my_pcie_test)
...
task run_phase (uvm_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 `uvm_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 (see [“Transaction Layer Callbacks and Exceptions”](#)), with exception capability (see [“Transaction Layer Exceptions”](#)), as well as a status object [“Transaction Layer Status”](#).



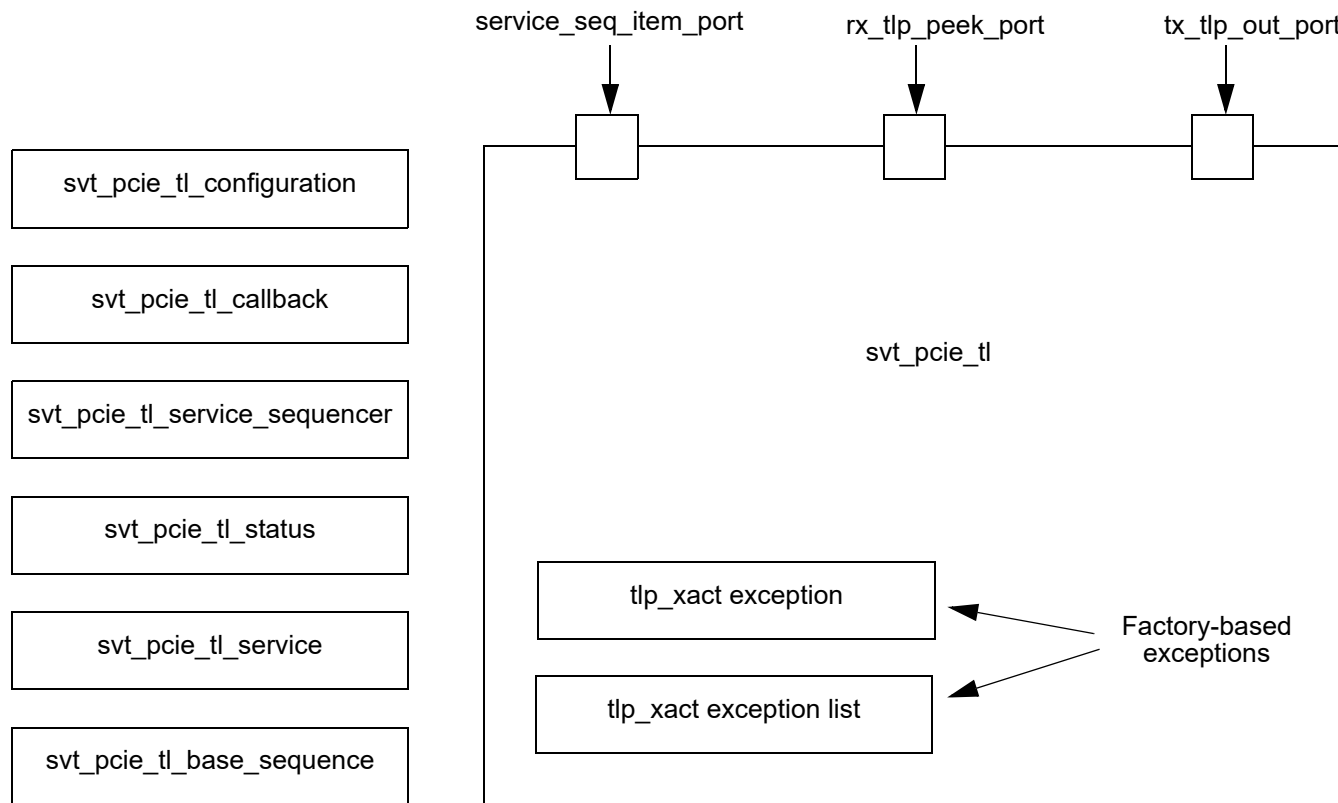
#### Attention

Note, the descriptions for the classes and applications show the most important and often used members/features. Consult the [PCIe UVM 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.

- [“SolvNet PCIe VIP Articles”](#) on page 385.
- [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`*

Members of the `svt_pcie_device_configuration` class are listed in [Table 10-1](#).

**Table 10-1 Transaction Layer configuration members**

<p>svt_pcie_tl_configuration::auto_enable_vc0_at_startup</p> <p>Type: rand bit</p> <p>Range: 0-1</p> <p>Default: 1</p> <p>Description:</p> <p>VC0 automatically initializes when enabled.</p>
<p>svt_pcie_tl_configuration::credit_starvation_timeout_ns</p> <p>Type: rand int unsigned</p> <p>Range: 0+ ns</p> <p>Default: 10000ns</p> <p>Description:</p> <p>If a VC is credit starved, i.e. TLP transmission is gated, for too long, a warning is issue. Credits should be returned in a timely fashion. Set to 0 to disable check.</p>
<p>svt_pcie_tl_configuration::default_route_at_appl_id</p> <p>Type: rand int unsigned</p> <p>Default: 0</p> <p>Description:</p> <p>Route address translation (AT).</p>
<p>svt_pcie_tl_configuration::default_route_cfg_type0_appl_id</p> <p>Type: rand int unsigned</p> <p>Default: 0</p> <p>Description:</p> <p>Default application ID to use.</p>
<p>svt_pcie_tl_configuration::default_route_cfg_type1_appl_id</p> <p>Type: rand int unsigned</p> <p>Default: 0</p> <p>Description:</p> <p>Default application ID to use.</p>
<p>svt_pcie_tl_configuration::default_route_io_appl_id</p> <p>Type: rand int</p> <p>Default: 0</p> <p>Description:</p> <p>Default application ID to use.</p>
<p>svt_pcie_tl_configuration::default_route_mem_appl_id</p> <p>Type: rand int unsigned</p> <p>Default: 0</p> <p>Description:</p> <p>Default application ID to use.</p>

**Table 10-1 Transaction Layer configuration members (Continued)**

svt_pcie_tl_configuration::default_route_msg_appl_id Type: rand int unsigned Default: 0 Description: Application ID.
svt_pcie_tl_configuration::enable_route_at_to_function Type: rand bit Default: 0 Description: Routing to function address translation (AT).
svt_pcie_tl_configuration::enable_route_cfg_type0_to_function Type: rand bit Default: 0 Description: Enable routing type 0 configuration requests.
svt_pcie_tl_configuration::enable_route_cfg_type1_to_function Type: rand bit Default: 0 Description: Enable routing type1 configuration requests.
svt_pcie_tl_configuration::enable_route_io_to_function Type: rand bit Default: 0 Description: Enable routing I/O requests.
svt_pcie_tl_configuration::enable_route_mem_to_function Type: rand bit Default: 0 Description: Enable routing memory requests.
svt_pcie_tl_configuration::enable_route_msg_to_function Type: rand bit Default: 0 Description: Enable routing message requests.
svt_pcie_tl_configuration::init_cpl[np p]_data_tx_credits Type: rand int unsigned [8] Range: 0-4096 Default: 1025 Description: Set initial data credits.

**Table 10-1 Transaction Layer configuration members (Continued)**

svt_pcie_tl_configuration::init_[cpl np p]_hdr_tx_credits Type: rand int unsigned [8] Range: 0-255 Default: 101 Description: Set initial header credits.
svt_pcie_tl_configuration::max_vc[0-7]_[p np cpl]_updatefc_delay Type: rand int unsigned Range: 0+ ns Default: 1 ns Description: Maximum flow control (FC) delay.
svt_pcie_tl_configuration::min_vc[0-7]_[p np cpl]_updatefc_delay Type: rand int unsigned Range: 0+ ns Default: 100 ns Description: Minimum flow control (FC) delay
svt_pcie_tl_configuration::remote_extended_tag_field_enabled Type: rand int unsigned Range: 0-1 Default: 0 Description: Enable remote extended tag.
svt_pcie_tl_configuration::remote_max_read_request_size Type: rand int unsigned Range: 0-4096 Default: 512 Description: Remote max read request size.

## 10.2.1 Verilog Configuration Parameters

Not all configuration items are currently controlled from within the UVM 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 10-2](#).

**Table 10-2 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 10-3](#).

**Table 10-3 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*

**Note**

The nonvirtual 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 nonvirtual 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;  
...  
`uvm_do_on (fc_seq, p_sequencer.root_virt_seqr.pcie_virt_seqr.tl_seqr);  
...
```

Transaction Layer service sequences are listed in [Table 10-4](#).

**Table 10-4 Transaction Layer service sequences**

<code>svt_pcie_dl_service_set_link_en_sequence</code> Attribute: rand bit enable Range: 0-1 Description: Service enables the Data Link Layer. The Data Link Layer will automatically initiate VC0 InitFC process when enabled.
<code>svt_pcie_tl_service_base_sequence</code> Description: Base sequence for the TL; all other sequences extend from this sequence.
<code>svt_pcie_tl_service_check_final_credits_sequence</code> Description: 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. This task should be called at the end of every test. Any lost credits will be flagged.

**Table 10-4 Transaction Layer service sequences (Continued)**

svt_pcie_tl_service_clr_stats_sequence Description: Clears all stats in Transaction Layer.
svt_pcie_tl_service_disp_stats_sequence Description: Displays all stats in Transaction Layer.
svt_pcie_tl_service_null_sequence Description: Null sequence used to turn off the default sequence; typically used with UVM 1.0
svt_pcie_tl_service_set_vc_en_sequence Attribute: rand bit vc_enable Attribute: rand bit [2:0] vc_num Description: Enable and disable virtual channel. Call for each VC to enable. VC0 is enabled by default. vc_enable = 1 to enable, 0 to disable. vc_num is between 0 and 7.

In addition to the sequence library shown in [Table 10-4](#), 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 10-5](#).

**Table 10-5 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.			



**Table 10-5 Service type enumerated parameters (Continued)**

	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.
<b>ADD_AT_ADDR_APPL_ID_MAP_ENTRY</b> 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.
<b>ADD_CFG_BDF_APPL_ID_MAP_ENTRY</b> 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.
<b>ADD_RID_APPL_ID_MAP_ENTRY</b> 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.
<b>ADD_RID_MSG_CODE_APPL_ID_MAP_ENTRY</b> 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.
<b>DISPLAY_MEM_ADDR_APPL_ID_MAP</b> Displays all memory addressses.			
<b>DISPLAY_IO_ADDR_APPL_ID_MAP</b> Displays all I/O address to application ID map entries.			
<b>DISPLAY_AT_ADDR_APPL_ID_MAP</b> Displays all memory address to application ID map entries that require address translation.			

**Table 10-5 Service type enumerated parameters (Continued)**

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.			
	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.			
 <b>Note</b> 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 <a href="#">“Determining if the Transaction Layer is Idle”</a> .			
	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 10-5](#).



#### 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 `uvm_blocking_put_port` and a `uvm_blocking_peek_imp`, which are `rx_tlp_out_put` and `rx_tlp_peek_port`, respectively. You can connect to those ports for access to all received TLPs.

Additionally, there is a `uvm_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 10-6](#) 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 10-6 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 `uvm_component` which implements the PCIe Data Link Layer. This class is included in the `svt_pcie_agent`. When you instantiate the PCIe UVM 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 UVM HTML Class Reference](#) for a complete listing of the members and their data types.

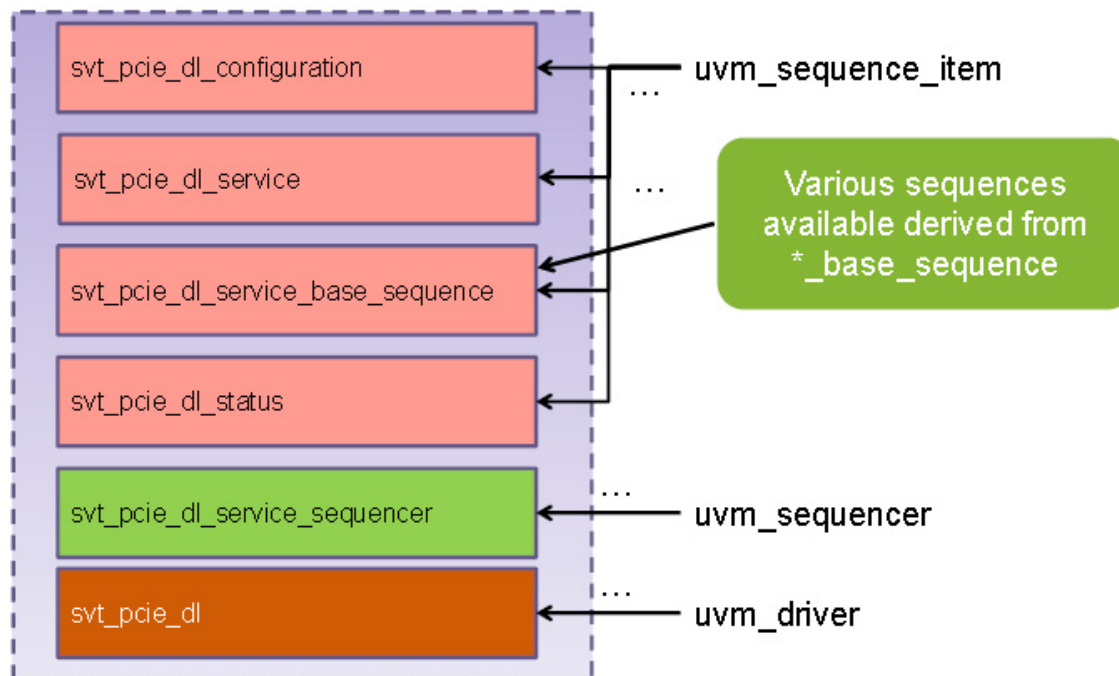
### 11.2 Additional Documentation on DL Programming Tasks

Consult the following to get information on DL related programming tasks.

- [“SolvNet PCIe VIP Articles”](#) on page 385.
- [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: ASPM 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 * InitiateASPMLOsEntry task.

Table 11-2 DL Service Requests

Member	Description
INITIATE_ASPM_L0S_ENTRY	Initiates VIP to enter ASPM Tx L0s low power state.
INITIATE_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 TLPs 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 UVM\_WARNING.

**Table 11-8 Pm/ASPM Checks**

Member	Description
MSGCODE_PCIESVC_DL_ASPM_L1_HANDSHAKE_TIMEOUT	<ul style="list-style-type: none"> <li>• f DUT fails to respond to ASPM request handshake for this:</li> <li>• # symbols, NOTICE will be issued</li> <li>• and ASPM entry will be aborted. aspm_timeout_cnt_limit = `SVT_PCIE_ASPM_TIMEOUT_CNT_LIMIT_DEFAULT;</li> </ul>
MSGCODE_PCIESVC_DL_ASPM_L1_1_HANDSHAKE_TIMEOUT	
MSGCODE_PCIESVC_DL_ASPM_L1_2_HANDSHAKE_TIMEOUT	
MSGCODE_PCIESVC_DL_PM_L1_HANDSHAKE_TIMEOUT	<ul style="list-style-type: none"> <li>• If DUT fails to respond to PM request handshake for this:</li> <li>• # symbols, NOTICE will be issued and</li> <li>• * PM entry will be aborted. */ rand int unsigned pm_timeout_cnt_limit = `SVT_PCIE_PM_TIMEOUT_CNT_LIMIT_DEFAULT;</li> </ul>
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 handshake 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 handshake 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 handshake is complete. Spec 5.3.2.3

## 11.5 Component Class svt\_pcie\_dl

The UVM 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_callback(...)	Called from the SVC Link Layer to report an outbound TLP for callback.
get_cfg t(...)	Overrides the base method to generate an error.
reconfigure t(...)	Overrides the base method to generate an error.
set_err_check t(...)	Used to set the err_check object and to fill in all of the local checks.
<b>Following are derived from various base classes for port tracking and error injection</b>	
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

The following table lists configuration members under these major functional areas:

- [Flow/Credit Management](#)
- [Replay Ack/Nak](#)
- [DLL Packet Control](#)
- [Power Management](#)

#### 11.6.1.1 Flow/Credit Management

Following are some tasks using Flow/Credit Management configuration:

- Set initial credits
- Control response to credit requests

**Table 11-10 Flow and Credit Configuration Features**

Member	Feature
<code>initfc_timeout_ns</code>	Maximum time between receiving all 3 types (P, NP, CPL) of InitFC DLLPs. SVC will issue NOTICE if timeout value is exceeded.
<code>max_initfc_delay</code>	Maximum interval between InitFC DLLPs. Used to model delays to due media access. Value should be relatively small.
<code>max_updatefc_delay</code>	Maximum interval between UpdateFC DLLPs. Used to model delays to due media access. Value should be relatively small.
<code>min_initfc_delay</code>	Minimum interval between InitFC DLLPs. Used to model delays to due media access. Value should 0 be relatively small.
<code>min_num_tx_initfc1_cpl</code>	Minimum number of InitFC1-CPL DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC1-CPL DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays.
<code>min_num_tx_initfc1_np</code>	Minimum number of InitFC1-NP DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC1-NP DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays.

**Table 11-10 Flow and Credit Configuration Features (Continued)**

Member	Feature
min_num_tx_initfc1_	Minimum number of InitFC1-P DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC1-P DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays.
min_num_tx_initfc2_cpl	Minimum number of InitFC2-CPL DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC2-CPL DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays. Setting all to 0 will allow the VIP to send only 1 iniFC2-P/NP/CPL in FC_INIT2 state.
min_num_tx_initfc2_np	Minimum number of InitFC2-NP DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC2-NP DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays. Setting all to 0 will allow the VIP to send only 1 iniFC2-P/NP/CPL in FC_INIT2 state.
min_num_tx_initfc2_p	Minimum number of InitFC2-P DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC2-P DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays. Setting all to 0 will allow the VIP to send only 1 iniFC2-P/NP/CPL in FC_INIT2 state.
updatefc_timeout_ns	Maximum time between receiving UpdateFC DLLPs. Timeout is tracked per VC/Type. SVC will issue NOTICE if timeout value is exceeded.
min_updatefc_delay	Minimum interval between UpdateFC DLLPs. Used to model delays to due media access. Value should be relatively small.
vc0_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
vc1_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.

**Table 11-10 Flow and Credit Configuration Features (Continued)**

Member	Feature
vc2_updatefc_interval_ns	
vc3_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
vc4_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
vc5_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
vc6_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
vc7_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
tx_fc_init_completed_by_tlp	Enables sending a TLP or an UpdateFC DLLP to complete flow control initialization instead of sending and INITFC2 DLLP. A TLP must be queue in order for initialization to complete. The probability of one of these is controlled by the percentage. The weights control the relative likelihood of one over the other.
tx_fc_init_completed_by_updatefc	Enables sending a TLP or an UpdateFC DLLP to complete flow control initialization instead of sending and INITFC2 DLLP. A TLP must be queue in order for initialization to complete. The probability of one of these is controlled by the percentage. The weights control the relative likelihood of one over the other. ;
tx_fc_init_completed_percentage	Enables sending a TLP or an UpdateFC DLLP to complete flow control initialization instead of sending and INITFC2 DLLP. A TLP must be queue in order for initialization to complete. The probability of one of these is controlled by the percentage. The weights control the relative likelihood of one over the other.



**Table 11-10 Flow and Credit Configuration Features (Continued)**

Member	Feature
initfc_timeout_ns	Maximum time between receiving all 3 types (P, NP, CPL) of InitFC DLLPs. SVC will issue NOTICE if timeout value is exceeded.
max_initfc_delay	Maximum interval between InitFC DLLPs. Used to model delays to due media access. Value should be relatively small.
max_updatefc_delay	Maximum interval between UpdateFC DLLPs. Used to model delays to due media access. Value should be relatively small.
min_initfc_delay	Minimum interval between InitFC DLLPs. Used to model delays to due media access. Value should 0 be relatively small.
min_num_tx_initfc1_cpl	Minimum number of InitFC1-CPL DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC1-CPL DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays.
min_num_tx_initfc1_np	Minimum number of InitFC1-NP DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC1-NP DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays.
min_num_tx_initfc1_p	Minimum number of InitFC1-P DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC1-P DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays.
min_num_tx_initfc2_cpl	Minimum number of InitFC2-CPL DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC2-CPL DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays. Setting all to 0 will allow the VIP to send only 1 iniFC2-P/NP/CPL in FC_INIT2 state.
min_num_tx_initfc2_np	Minimum number of InitFC2-NP DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC2-NP DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays. Setting all to 0 will allow the VIP to send only 1 iniFC2-P/NP/CPL in FC_INIT2 state.

**Table 11-10 Flow and Credit Configuration Features (Continued)**

Member	Feature
min_num_tx_initfc2_p	Minimum number of InitFC2-P DLLPs that must be transmitted in FC_INIT1 state before transitioning to FC_INIT2 state. The VIP might send more InitFC2-P DLLPs in FC_INIT1 state depending on reception of InitFC-P/NP/CPL DLLPs per spec. Commonly set to a value greater than the default to mimic real life processing delays. Setting all to 0 will allow the VIP to send only 1 iniFC2-P/NP/CPL in FC_INIT2 state.
updatefc_timeout_ns	Maximum time between receiving UpdateFC DLLPs. Timeout is tracked per VC/Type. SVC will issue NOTICE if timeout value is exceeded.
min_updatefc_delay	Minimum interval between UpdateFC DLLPs. Used to model delays to due media access. Value should be relatively small.
vc0_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
vc1_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
vc2_updatefc_interval_ns	
vc3_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
vc4_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
vc5_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.

**Table 11-10 Flow and Credit Configuration Features (Continued)**

Member	Feature
vc6_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
vc7_updatefc_interval_ns	If credits for a VC have not been sent to the link partner for this interval, all 3 types of credits will be schedule for transmission. Setting to 0 disables scheduling credits. This should be set to 0 for most testing as periodic updates could mask credits being lost.
tx_fc_init_completed_by_tlp	Enables sending a TLP or an UpdateFC DLLP to complete flow control initialization instead of sending and INITFC2 DLLP. A TLP must be queue in order for initialization to complete. The probability of one of these is controlled by the percentage. The weights control the relative likelihood of one over the other.
tx_fc_init_completed_by_updatefc	Enables sending a TLP or an UpdateFC DLLP to complete flow control initialization instead of sending and INITFC2 DLLP. A TLP must be queue in order for initialization to complete. The probability of one of these is controlled by the percentage. The weights control the relative likelihood of one over the other. ;
tx_fc_init_completed_percentage	Enables sending a TLP or an UpdateFC DLLP to complete flow control initialization instead of sending and INITFC2 DLLP. A TLP must be queue in order for initialization to complete. The probability of one of these is controlled by the percentage. The weights control the relative likelihood of one over the other.
rx_credit_latency_in_symbols	The variable represents maximum latency until a received credit is passed to the TL for processing. The latency is in symbol times.

### 11.6.1.2 Replay Ack/Nak

**Table 11-11 Replay Ack/Nak Configuration Members and Features**

Member	Feature
max_payload_size	
attached_ack_nak_latency_tolerance_x1	Variance in number of symbols in which ACK may be received. This should account for clock domain crossings, TLP processing variances, etc. Each link width can have a different value.
attached_ack_nak_latency_tolerance_x12	Variance in number of symbols in which ACK may be received. This should account for clock domain crossings, TLP processing variances, etc. Each link width can have a different value.

**Table 11-11 Replay Ack/Nak Configuration Members and Features (Continued)**

Member	Feature
attached_ack_nak_latency_tolerance_x16	Variance in number of symbols in which ACK may be received. This should account for clock domain crossings, TLP processing variances, etc. Each link width can have a different value.
attached_ack_nak_latency_tolerance_x2	Variance in number of symbols in which ACK may be received. This should account for clock domain crossings, TLP processing variances, etc. Each link width can have a different value.
attached_ack_nak_latency_tolerance_x32	Variance in number of symbols in which ACK may be received. This should account for clock domain crossings, TLP processing variances, etc. Each link width can have a different value.
attached_ack_nak_latency_tolerance_x4	Variance in number of symbols in which ACK may be received. This should account for clock domain crossings, TLP processing variances, etc. Each link width can have a different value.
attached_ack_nak_latency_tolerance_x8	Variance in number of symbols in which ACK may be received. This should account for clock domain crossings, TLP processing variances, etc. Each link width can have a different value.
attached_internal_delay_2_5g	Internal delay used in “Ack Transmit Latency Limit” equation to check attached replay timer and AckNak latency timer at 2.5G. Accounts for packet processing delay and transmission latency.
attached_internal_delay_5g	Internal delay used in “Ack Transmit Latency Limit” equation to check attached replay timer and AckNak latency timer at 5G. Accounts for packet processing delay and transmission latency.
attached_internal_delay_8g	Internal delay used in “Ack Transmit Latency Limit” equation to check attached replay timer and AckNak latency timer at 8G. Accounts for packet processing delay and transmission latency.
attached_replay_timeout	Length of the attached replay timer in symbols. Used to check the DUT’s replay timer. Setting value = 0 will enable automatic updates if MAX_PAYLOAD_SIZE_VAR, link_width, or speed change.
attached_replay_timeout_tolerance_x1	Number of symbols in which retry may be received after a timeout. This should account for clock domain crossings, ACK processing variances, etc. Each link width can have a different value.
attached_replay_timeout_tolerance_x12	Number of symbols in which retry may be received after a timeout. This should account for clock domain crossings, ACK processing variances, etc. Each link width can have a different value.
attached_replay_timeout_tolerance_x16	Number of symbols in which retry may be received after a timeout. This should account for clock domain crossings, ACK processing variances, etc. Each link width can have a different value.

**Table 11-11 Replay Ack/Nak Configuration Members and Features (Continued)**

Member	Feature
attached_replay_timeout_tolerance_x2	Number of symbols in which retry may be received after a timeout. This should account for clock domain crossings, ACK processing variances, etc. Each link width can have a different value.
attached_replay_timeout_tolerance_x32	Number of symbols in which retry may be received after a timeout. This should account for clock domain crossings, ACK processing variances, etc. Each link width can have a different value.
attached_replay_timeout_tolerance_x4	Number of symbols in which retry may be received after a timeout. This should account for clock domain crossings, ACK processing variances, etc. Each link width can have a different value.
attached_replay_timeout_tolerance_x8	Number of symbols in which retry may be received after a timeout. This should account for clock domain crossings, ACK processing variances, etc. Each link width can have a different value.
max_nak_latency	The maximum latency before a NAK is issued
max_ack_nak_latency	Max length of the ACK/NAK latency timer in symbols. If called, value is sticky. Setting to value = 0 will enable automatic updates.
max_attached_ack_nak_latency	Max length of the attached ACK/NAK latency timer in symbols. Used to check DUT's AckNak latency timer. If called, value is sticky. Setting value = 0 will enable automatic updates if MAX_PAYLOAD_SIZE_VAR or link_width change.
max_attached_nak_latency	Max NAK latency limit in symbols. Used to check DUT's NAK latency. If called, value is sticky. Setting to value = 0 will enable automatic updates.
max_num_replays ;	Maximum number of replays before the link gives up on transmitting a TLP
internal_delay_2_5g	Internal delay used in "Ack Transmit Latency Limit" equation for replay timer and AckNak latency timer at 2.5G.
internal_delay_5g	Internal delay used in "Ack Transmit Latency Limit" equation for replay timer and AckNak latency timer at 5G.
internal_delay_8g	Internal delay used in "Ack Transmit Latency Limit" equation for replay timer and AckNak latency timer at 8G.
min_ack_nak_latency	Min Length of the ACK/NAK latency timer in symbols.
min_attached_ack_nak_latency	Min length of the attached ACK/NAK latency timer in symbols. Default value = 0.
min_nak_latency	Minimum latency until a NAK is issued

**Table 11-11 Replay Ack/Nak Configuration Members and Features (Continued)**

Member	Feature
min_pm_ack_latency	The variable represents minimum latency until PM_REQUEST_ACK is issued.
replay_timeout	Length of the attached replay timer in symbols. Used to check the DUT's replay timer. Setting value = 0 will enable automatic updates if MAX_PAYLOAD_SIZE_VAR, link_width, or speed change.

**11.6.1.2.1 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
```

### 11.6.1.3 DLL Packet Control

Table 11-12

Member	Feature
enable_ei_tx_tlp_on_retry	The variable enables error injection on retry frames.
enable_tx_tlp_reporting	If enabled, transmitted TLPs will be reported to the global shadow memory.
initial_receive_sequence_value	The first TLP sent must have this sequence number
initial_transmit_sequence_value	The first TLP will be transmitted with this value. Used to check rollover at 4096
ltr_l1_2_threshold_scale	The variable represents LTR Threshold scale for L1.2 substate.
ltr_l1_2_threshold_value	The variable represents LTR Threshold value for L1.2 substate.
max_size_packet	Maximum possible header size plus maximum possible payload size.
max_header_size	Maximum possible header size plus maximum possible prefixes.
max_num_prefixes	Maximum possible prefixes.
max_tx_ipg	Max gap between packets (TLPs and/or DLLPs) as measured in symbol times.
max_tx_nullified_tlp_len	Maximum nullified TLP length in dwords.
min_tx_ipg	Min gap between packets (TLPs and/or DLLPs) as measured in symbol times. 0 = packets can be packed in same symbol. 1 = packets will always start on lane 0 in symbol following END/EDB.
min_tx_nullified_tlp_len	Minimum nullified TLP length in dwords.
model_instance_scope	This is the full hierarchical path name to the instance of the SVC model which the driver model is instantiated in. The path name is concatenated with the name of this component passed to the constructor to generate the lookup string used to find the SV API instance.
received_dllp_interface_mode	Select DLLPs available at Receive DLLP analysis port. DLLPs are filtered based on the bits enabled in received_dllp_interface_mode bit vector.
received_tlp_interface_mode	Select TLPs available at Receive TLP analysis port. TLPs are filtered based on the bits enabled in received_tlp_interface_mode bit vector
sent_dllp_interface_mode	Select DLLPs available at Sent DLLP analysis port. DLLPs are filtered based on the bits enabled in sent_dllp_interface_mode bit vector

Table 11-12

Member	Feature
sent_tlp_interface_mode	Select TLPs available at Sent TLP analysis port. TLPs are filtered based on the bits enabled in sent_tlp_interface_mode bit vector
tx_nullified_tlp_hdr0_value	The default automatically generated nullified TLP is a MsgD, fatal error message. If the DUT forwards this TLP, it should be flagged by upper layers.
tx_nullified_tlp_hdr1_value	The default automatically generated nullified TLP is a MsgD, fatal error message. If the DUT forwards this TLP, it should be flagged by upper layers.
tx_nullified_tlp_hdr2_value	The default automatically generated nullified TLP is a MsgD, fatal error message. If the DUT forwards this TLP, it should be flagged by upper layers.
tx_nullified_tlp_hdr3_value	The default automatically generated nullified TLP is a MsgD, fatal error message. If the DUT forwards this TLP, it should be flagged by upper layers.

#### 11.6.1.4 Power Management

Table 11-13 Power Management and Features

Members	Features
aspm_timeout_cnt_limit	If 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	If DUT fails to respond to ASPM request handshake for this # symbols, NOTICE will be issued and ASPM entry will be aborted.
enable_aspm_l0s_entry	Enable ASPM L0S entry.
enable_aspm_l1_1_entry	The variable enables ASPM L1.1 entry.
enable_aspm_l1_2_entry	The variable enables ASPM L1.2 entry.
enable_aspm_l1_entry	Enable ASPM L1S entry.
enable_pm_l1_1_entry	The variable enables PM L1.1 entry.
enable_pm_l1_2_entry	The variable enables PM L1.2 entry.
max_tx_pm_ipg	Max number of symbol times between PM DLLPs.
min_tx_pm_ipg	Min number of symbol times between PM DLLPs.
max_pm_ack_latency	The variable represents maximum latency until PM_REQUEST_ACK is issued.



## 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.

Consult the following table for all the status and stastics you can gather from the class svt\_pcie\_dl\_status.

**Table 11-14 Status Members and Features**

Status Member	Feature
dl_link_up	Indicates DL Up status.
is_idle	Indicates whether the data link layer is idle
num_ack_received num_ack_sent	Number of ACK DLLPs received and sent.
num_vendor_dllp_received num_vendor_dllp_sent	Number of Vendor DLLPs sent and received.
num_bad_retries_sent	Number of retry TLPs sent with error.
num_bad_tlp_received num_bad_tlp_sent	Number of TLPs received with error.
num_dllp_received num_dllp_sent	Number of DLLPs received and sent.
num_duplicate_tlp_received	Number of duplicate TLPs received.
num_ei_rx_replay_count_failure	Number of replay count failure errors injected.
num_ei_rx_tlp_nak_good_tlp num_ei_rx_tlp_withhold_ack_nak	Number of NAK good TLP errors injected. Number of ACK withheld errors injected.
num_ei_tx_alignment_2_sdp_per_symbol num_ei_tx_alignment_2_stp_per_symbol num_ei_tx_alignment_sdp_packet_gap num_ei_tx_alignment_sdp_wrong_lane num_ei_tx_alignment_stp_packet_gap num_ei_tx_alignment_stp_wrong_lane	Not supported.

**Table 11-14 Status Members and Features (Continued)**

Status Member	Feature
num_ei_tx_dllp_8g_corrupt_header	Number of Tx DLLP 8G corrupt header errors injected.
num_ei_tx_dllp_bit_flip	Number of Tx DLLP bit flip errors injected.
num_ei_tx_dllp_code_violation	Number of Tx DLLP code violation errors injected.
num_ei_tx_dllp_corrupt_crc	Number of Tx DLLPs corrupt CRC errors injected.
num_ei_tx_dllp_corrupt_disparity	Number of Tx DLLPs with corrupt disparity errors injected.
num_ei_tx_dllp_duplicate_ack	Number of Tx DLLP duplicate ACK errors injected.
num_ei_tx_dllp_missing_end	Number of Tx DLLP missing end errors injected.
num_ei_tx_dllp_missing_start	Number of Tx DLLP missing start errors injected.
num_ei_tx_dllp_rsvd_non_zero	Number of Tx DLLPs with non-zero reserved bits injected.
num_ei_tx_dllp_scrambler_error	Number of Tx DLLP scrambler errors injected.
num_ei_tx_dllp_unknown_type	Number of Tx DLLPs with unknown type codes injected.
num_ei_tx_tlp_8g_corrupt_header_crc	Number of Tx TLP 8G corrupted header CRC errors injected.
num_ei_tx_tlp_bit_flip	Number of Tx TLPs with bit flip errors injected.
num_ei_tx_tlp_code_violation	Number of Tx TLPs with code violations injected.
num_ei_tx_tlp_corrupt_disparity	Number of Tx TLPs with disparity errors injected.
num_ei_tx_tlp_corrupt_lcrc	Number of Tx TLP corrupt LCRC errors injected.
num_ei_tx_tlp_duplicate_seq_num	Number of Tx TLP duplicate sequence number errors injected.
num_ei_tx_tlp_illegal_seq_num	Number of Tx TLP illegal sequence number errors injected.
num_ei_tx_tlp_missing_end	Number of Tx TLP with missing end errors injected.
num_ei_tx_tlp_missing_start	Number of Tx TLP with missing start errors injected.
num_ei_tx_tlp_nullified	Number of nullified Tx TLPs injected.
num_ei_tx_tlp_nullified_corrupt_lcrc	Number of nullified Tx TLPs with corrupted LCRC injected.
num_ei_tx_tlp_nullified_good_lcrc	Number of nullified Tx TLPs with good LCRC injected.
num_ei_tx_tlp_scrambler_error	Number of Tx TLP scrambler errors injected.
num_good_dllp_received	Number of DLLPs received and sent without error.
num_good_dllp_sent	
num_good_retries_sent	Number of retry TLPs sent without error.
num_good_tlp_received	Number of TLPs received and sent without error.
num_good_tlp_sent	
num_nak_received	Number of ACK DLLPs received and sent.
num_nak_sent	
num_nullified_tlp_received	Number of nullified TLPs received.
um_out_of_order_tlp_received	Number of out-of-order TLPs received.
num_phy_link_down_events	Number of Phy Link Down events
num_phy_retrain	Number of times the link was retrained to due the replay attempt counter rolling over.

**Table 11-14 Status Members and Features (Continued)**

Status Member	Feature
num_retries_due_to_nak num_retries_due_to_replay_timeout num_retries_sent	Number of retried packets due to a NAK. Number of retried packets due to replay timeout. Number of packets that had to be retransmitted.
num_rx_dllp_missing_end num_rx_tlp_missing_end	Number of DLLPs received with missing ENDs. Number of TLPs received with missing ENDs.
num_tlp_received num_tlp_sent	Number of TLPs received and sent.
num_vc<0-7>_cpl_initfc1_received num_vc<0-7>_cpl_initfc1_sent num_vc<0-7>_cpl_initfc2_received num_vc<0-7>_cpl_initfc2_sent num_vc<0-7>_cpl_updatefc_received num_vc<0-7>_cpl_updatefc_sent num_vc<0-7>_initfc1_received num_vc<0-7>_initfc1_sent num_vc<0-7>_initfc2_received num_vc<0-7>_initfc2_sent num_vc<0-7>_np_initfc1_received num_vc<0-7>_np_initfc1_sent num_vc<0-7>_np_initfc2_received num_vc<0-7>_np_initfc2_sent num_vc<0-7>_np_updatefc_received num_vc<0-7>_np_updatefc_sent num_vc<0-7>_p_initfc1_received num_vc<0-7>_p_initfc1_sent num_vc<0-7>_p_initfc2_received num_vc<0-7>_p_initfc2_sent num_vc<0-7>_p_updatefc_received num_vc<0-7>_p_updatefc_sent num_vc<0-7>_updatefc_received num_vc<0-7>_updatefc_sent	The number of sent and received credits on Virtual Channels zero-to-seven: <ul style="list-style-type: none"> <li>• CPL UpdateFC DLLPs</li> <li>• InitFC1/2 DLLPs</li> <li>• NP UpdateFC DLLPs</li> <li>• P, NP, or CPL InitFC1 DLLPs</li> </ul>

## 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

### 11.8.1 Members and Features

The following table lists major members of the `svt_pcie_dl_service` and the features they support.

**Table 11-15 Frequently Used Members and Features of Service Class `svt_pcie_dl_service`**

Member	Feature/Use
<code>ack_factor</code>	Updates the AckFactor table entries listed in specification.
<code>enable</code>	Set to allow DLCMSM to transition out of Disabled state.
<code>fc_type</code>	FC type of INITFC frames to be gated.
<code>gate_enable</code>	Gating enable/disable
<code>status</code>	Status information about the current processing state
<code>rate</code>	Store the link speed used during the updating of the AckFactor table entries listed in specifications.
The following define the elements of the <code>service_type_enum</code>	
<code>SET_LINK_ENABLE(0)</code>	Enables Data Link layer. When enabled, data link layer automatically initiates the VC0 InitFC process
<code>DISPLAY_ATTACHED_REPLAY_TIMER_TOLERANCES(1)</code>	Displays replay timer checker tolerances for link widths of 1, 2,4,8,12,16, and 32.
<code>DISPLAY_ATTACHED_ACK_NAK_LATENCY_TOLERANCES(2)</code>	Displays AckNak Latency checker tolerances for link widths of 1, 2,4,8,12,16, and 32.

**Table 11-15 Frequently Used Members and Features of Service Class svt\_pcie\_dl\_service (Continued)**

Member	Feature/Use
DISPLAY_STATS(3)	Displays link layer statistics.
CLEAR_STATS(4)	Resets statistics counters.
INITIATE_ASPM_L0S_ENTRY(5)	Initiates VIP to enter ASPM Tx L0s low power state.
INITIATE_ASPM_L1_ENTRY(6)	Initiates VIP to enter ASPM L1 low power state.
INITIATE_PM_L1_ENTRY(7)	Initiates VIP to enter PM L1 low power state.
INITIATE_PM_L23_ENTRY(8)	Initiates VIP to enter PM L2/L3 low power states.
INITIATE_ASPM_EXIT(9)	Initiates VIP to transition back to L0 from ASPM low power state.
INITIATE_PM_EXIT(10)	Initiates VIP to transition back to L0 from PM low power state.
GATE_TX_INITFC1(11)	Gates the Transmitted INITFC1 frames
GATE_TX_INITFC2(12)	Gates the transmitted INITFC2 frames
GATE_TX_ACKNAK(13)	Gates the Transmitted ACK and NAK frames
SET_ACK_FACTOR(14)	Sets ACK Factor value



# 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”](#) on page 211.
- [“UVM Component Class svt\\_pcie\\_pl”](#) on page 177
- [“PHY Layer Configuration Class”](#) on page 221

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 UVM 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.

- [“SolvNet PCIe VIP Articles”](#) on page 385.
- [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. <a href="#">Table 12-2</a> on page 213 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 <a href="#">Table 12-3</a> on page 213 for a listing of hot reset modes.
internal_condition	Type of internal condition. Refer to <a href="#">Table 12-4</a> on page 214 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. Consult <a href="#">Table 12-5</a> on page 214 for a complete listing of all the service requests supported by the VIP.
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.
pipe_get_local_preset_coefficients	Enables MPIPE VIP to assert <code>GetLocalPresetCoefficients</code> signal for one PCLK.

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 inactive 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

The following table shows the service type calls available through the “[service\\_type](#)” member.

**Table 12-5 Service Requests**

Requested Service	Description
SET_PHY_ENABLE(0)	When clear, LTSSM will attempt to enter the DISABLED state through the exchange of TS.
INITIATE_LINK_WIDTH_CHANGE(1)	Directs the LTSSM to attempt to upconfigure/ downconfigure the link width to the highest allowed link width in the supported link width vector field. Can only be called in L0. Calls to this task outside of L0 will be ignored.
INITIATE_RETRAIN_LINK(2)	Direct the LTSSM to Recovery. This service task has same functionality as setting retrain_link bit in PCIE Cfg's Link Control Register.
INITIATE_SPEED_CHANGE(3)	Sets the directed_speed_change variable in the LTSSM, causing the LTSSM to enter recovery and attempt a speed change. Should only be called in L0. Calls to this task outside of L0 will be ignored.
INITIATE_LOOPBACK(4)	Instructs the VIP to enter loopback as a loopback master.
CONFIGURE_LANE(5)	This service task associates an advertised link number with a physical link. The default behavior is to associate physical lane 0 with advertised lane 0, physical lane 1 will advertise itself as lane 1 and so on. Example- CONFIGURE_LANE(0,1,1) will instruct the LTSSM have physical lane 0 advertise itself as lane 1 in all TS1/ TS2 ordered sets. This is intended to be used only for LTSSM testing, the LTSSM will not support arbitrary lane N to N mapping
SET_USER_TX_TS1(6)	Defines a user training set and optionally an error injection for a given lane

**Table 12-5 Service Requests (Continued)**

Requested Service	Description
SET_USER_TX_TS1_PATTERN(7)	When enabled, the LTSSM will send a random burst of user training sets. The size of the burst is set by the min/max arguments. In between the burst of user training sets the LTSSM will send normal TS1. The number of normal TS1 is defined by the min/max spacing arguments.
SET_EI_TX_TS1_PATTERN(8)	Defines a constrained random burst of TS1 ordered sets with errors.
SET_USER_TX_TS2(9)	Defines a user training set and optionally an error injection for a given lane.
SET_USER_TX_TS2_PATTERN(10)	When enabled, the LTSSM will send a random burst of user training sets. The size of the burst is set by the min/max arguments. In between the burst of user training sets the LTSSM will send normal TS2. The number of normal TS2 is defined by the min/max spacing arguments.
SET_EI_TX_TS2_PATTERN(11)	Defines a constrained random burst of TS2 ordered sets with errors.
REQUEST_HOT_PLUG_MODE(12)	Allows users to simulate unplugging of VIP from the link. During a hot plug event all the transmitters are shut off and the LTSSM will go back to Detect. The VIP will not respond until it is plugged back in. In a passive VIP, LTSSM goes to 'Searching...' state and waits until it detects electrical idle on its RX and TX paths. While in 'Searching...' state monitor does not evaluate any protocol check. As soon as electrical idle is entered, LTSSM goes to Detect state and resumes normal operation.
PERFORM_EQUALIZATION(13)	Sets perform_equalization variable. This is used by LTSSM to determine if equalization should be performed during 8 GT/s speed negotiation or otherwise
REQUEST_EQUALIZATION(14)	Sets request_equalization variable. This is used by LTSSM to determine if equalization should be requested while in 8GT/s speeds.
QUEUE_EQ_FIGURE_MERIT_RESPONSE(15)	Sets a Figure Of Merit response from equalization evaluation. Response ranges from 0 to 255. If no responses are queued then the default value of 255 is returned as a figure of merit value. This is applicable only when VIP operates in PHY PIPE mode.
QUEUE_EQ_DIRECTION_CHANGE_RESPONSE(16)	Sets the direction change response from equalization evaluation. If no responses are queued, the default response used is no change on any of the coefficients.

**Table 12-5 Service Requests (Continued)**

Requested Service	Description
QUEUE_EQ_TX_REQUEST_PRESET_COEFF(17)	Queue either a Preset or a coefficient request. VIP sends this request in TS1 at 8GT/s in Recovery.Equalization. As long as a request is queued, VIP will issue a new request when the previous request completes. Phase will complete if no more requests are queued and transition criteria have been met.
SET_HOT_RESET_MODE(18)	Allows users to request the LTSSM to enter/exit hot reset.
SET_REJECT_COEFFICIENT_PRESET_REQUESTS(19)	Controls whether incoming requests during equalization are automatically rejected or accepted
SET_EI_TX_PHY_DATA_PATTERN(20)	When enabled, the phy will periodically inject errors into the data stream.
ASSERT_CLKREQ_N(21)	Controls whether clkreq_n is asserted or not. If not asserted the soft pullup will drive it to 'b1
MON_INITIATE_L1_2_EXIT(22)	Directs monitor LTSSM to exit from L1.2.Exit and transition to L1.Idle. It is applicable to a passive monitor instance monitoring either Endpoint or Root Complex. If LTSSM is already in L1.2.Exit state, then LTSSM transitions from this state to L1.Idle as soon as it is directed. If LTSSM is not in L1.2.Exit, then the service request is not effective immediately, but when LTSSM enters L1.2.Exit next time, LTSSM transitions from this state to L1.Idle immediately as it has been already directed to do so.
MON_INITIATE_L2_EXIT(23)	Directs monitor LTSSM to exit from L2.Idle state and transition to Detect.Quiet. It is applicable only to a passive monitor instance monitoring Root Complex. If LTSSM is already in L2.Idle state then LTSSM transitions from this state to Detect.Quiet as soon as it is directed. If LTSSM is not already in L2.Idle state then the service request is not effective immediately, but when LTSSM enters L2.Idle next time, LTSSM transitions from L2.Idle state to Detect.Quiet immediately as it has been already directed to do so.
MON_GOTO_L2_TRANSMIT_WAKE(24)	Directs monitor LTSSM to transition to L2.Transmit.Wake from L2.Idle. It is applicable only to a passive monitor instance monitoring Endpoint. If LTSSM is already in L2.Idle state, then LTSSM transitions from this state to L2.Transmit.Wake as soon as it is directed. If LTSSM is not in L2.Idle, then the service request is not effective immediately, but when LTSSM enters L2.Idle next time, LTSSM transitions from this state to L2.Transmit.Wake immediately as it has been already directed to do so.

**Table 12-5 Service Requests (Continued)**

Requested Service	Description
SET_TX_IDLE_DATA_CORRUPTION(25)	Set automatic corruption of TX Idle Data. Corruption of Idle Data is controlled by percentage. * If Idle Data is selected, actual byte to corrupt is based on weightings. * Byte corruption weightings available: * invalid_data_byte[0] (all speeds) * corrupt_disparity_byte[0] (2.5-5G only) * invalid_codeword_byte[0] (2.5-5G only) * Idle Data corruption is enabled via lane mask.
SET_TX_EIOS_CORRUPTION(26)	Set automatic corruption of TX EIOS. Corruption of EIOS is controlled by percentage. * If EIOS is selected, actual byte to corrupt is based on weightings. * Byte corruption weightings available: * invalid_data_byte[0-15] (all speeds) * corrupt_disparity_byte[0-3] (2.5-5G only) * invalid_codeword_byte[0-3] (2.5-5G only) * EIOS corruption is enabled via lane mask.
SET_TX_EIEOS_CORRUPTION(27)	Set automatic corruption of TX EIEOS. Corruption of EIEOS is controlled by percentage. * If EIEOS is selected, actual byte to corrupt is based on weightings. * Corruption mechanism selected by weightings: * invalid_data_byte[0-15] (all speeds) * corrupt_disparity_byte[0-15] (2.5-5G only) * invalid_codeword_byte[0-15] (2.5-5G only) * EIEOS corruption is enabled via lane mask.
SET_TX_FTS_CORRUPTION(28)	Set automatic corruption of TX FTS. Corruption of FTS is controlled by percentage. * If FTS is selected, actual byte to corrupt is based on weightings. * Corruption mechanism selected by weightings: * invalid_data_byte[0-15] (all speeds) * corrupt_disparity_byte[0-3] (2.5-5G only) * invalid_codeword_byte[0-3] (2.5-5G only) * FTS corruption is enabled via lane mask.
SET_TX_SKP_CORRUPTION(29)	Set automatic corruption of TX SKP. Corruption of SKP is controlled by percentage. * If SKP is selected, actual byte to corrupt is based on weightings. * Corruption mechanism selected by weightings: * invalid_data_byte[0-15] (all speeds) * corrupt_disparity_byte[0-3] (2.5-5G only) * invalid_codeword_byte[0-3] (2.5-5G only) * SKP corruption is enabled via lane mask.
SET_TX_SDS_CORRUPTION(30)	Set automatic corruption of TX SDS. Corruption of SDS is controlled by percentage. * If SDS is selected, actual byte to corrupt is based on weightings. * Corruption mechanism selected by weightings: * invalid_data_byte[0-15] (8G only) * SDS corruption is enabled via lane mask.

Table 12-5 Service Requests (Continued)

Requested Service	Description
SET_TX_EDS_CORRUPTION(31)	Set automatic corruption of TX EDS. CCorruption of EDS is controlled by percentage. * If EDS is selected, actual byte to corrupt is based on weightings. * Corruption mechanism selected by weightings: * invalid_data_byte[0-3] (8G only).
MON_INITIATE_DISABLED_EXIT(32)	Directs monitor LTSSM to exit from Disabled state and transition to Detect.Quiet. It is applicable only to a passive monitor instance monitoring Root Complex. If LTSSM is already in Disabled state then LTSSM transitions from this state to Disabled as soon as it is directed. If LTSSM is not already in Disabled state then the service request is not effective immediately, but when LTSSM enters Disabled next time, LTSSM transitions from Disabled state to Detect.Quiet immediately as it has been already directed to do so.
MON_SET_INTERNAL_CONDITION(33)	Sets internal conditions for monitor. The condition represents monitor's internal status * which monitor uses for tracking link activities. This service request should not be used * during normal operations. It is intended to be used in situations where the monitor cannot * track the LTSSM correctly due to observation delays. This service request can be used to * provide hints to the monitor that will allow tracking to continue.
MON_GET_INTERNAL_CONDITION(34)	Returns monitor's internal condition's current value. This is intended to be used * as a debug feature.
MON_INITIATE_LINK_WIDTH_SIZING(35)	Request the monitor to enter CONFIGURATION_LINKWIDTH_START as soon as it goes * to RECOVERY_IDLE.
MON_INITIATE_HOT_RESET_EXIT(36)	Directs monitor LTSSM to exit from Hot Reset state and transition to Detect.Quiet. It is applicable only to a passive monitor instance monitoring Root Complex. If LTSSM is already in Hot Reset state then LTSSM transitions from this state to Hot Reset as soon as it is directed. If LTSSM is not already in Hot Reset state then the service request is not effective immediately, but when LTSSM enters Hot Reset next time, LTSSM transitions from Hot Reset state to Detect.Quiet immediately as it has been already directed to do so.



**Table 12-5 Service Requests (Continued)**

Requested Service	Description
MON_INITIATE_LINK_SPEED_NEGOTIATION(37)	Directs monitor LTSSM to enter Recovery.Rcvrlock from L0 state. It also sets directed_speed_change_variable to 1'b1. If LTSSM is already in L0 state then LTSSM transitions from this state to Recovery as soon as it is directed. If LTSSM is not already in L0 state then the service request is not effective immediately, but when LTSSM enters L0 next time, LTSSM transitions from L0 state to Recovery.Rcvrlock immediately as it has been already directed to do so.
MON_GOTO_POLLING_COMPLIANCE(38)	Directs monitor LTSSM to transition to Polling.Compliance from Polling.Active. It is applicable only to a passive monitor instance. If LTSSM is already in Polling.Active state, then LTSSM transitions from this state to Polling.Compliance as soon as it is directed. If LTSSM is not in Polling.Active, then the service request is not effective immediately, but when LTSSM enters Polling.Active next time, LTSSM transitions from this state to Polling.Compliance immediately as it has been already directed to do so.
MON_INITIATE_POLLING_COMPLIANCE_EXIT(39)	Directs monitor LTSSM to exit from Polling.Compliance state and transition to Detect.Quiet. It is applicable only to a passive monitor instance. If LTSSM is already in Polling.Compliance state then LTSSM transitions from this state to Detect.Quiet as soon as it is directed. If LTSSM is not already in Polling.Compliance state then the service request is not effective immediately, but when LTSSM enters Polling.Compliance next time, LTSSM transitions from Polling.Compliance state to Detect.Quiet immediately as it has been already directed to do so.
SET_LANE_TX_DEEMPH(40)	Drives the tx_deemph pins on the pipe bus to the specified value at 2.5G and 5G.
QUEUE_PIE8_EQ_TX_REQUEST_PRESET_COEFF(41)	Queue either a Preset or a coefficient request in PIE8 mode. VIP sends this request in TS1 at 8GT/s in Recovery.Equalization. As long as a request is queued, VIP will issue a new request when the previous request completes. Phase will complete if no more requests are queued and transition criteria have been met.

**Table 12-5 Service Requests (Continued)**

Requested Service	Description
FORCE_LANE_TX_ELEC_IDLE(42)	Forces the specified lanes to shutdown and transmit electrical idle. corrupt_lane_mask[31:0] is a bit-mapped vector where corrupt_lane_mask[0] control physical lane 0 and corrupt_lane_mask[31] control physical lane 31. When corrupt_lane_mask[n] is 1'b1, the lane is turned OFF by forcing electrical idle on the lane. When corrupt_lane_mask[n] is 1'b0, the lane is turned back ON if it was forced to enter electrical idle earlier. When corrupt_lane_mask[n] is 1'b0 and if the lane was not forced to enter electrical idle earlier, then this exception doesn't have any effect on lane n. Lanes are not turned OFF immediately at the same symbol time, but they are shut down or forced to electrical idle after a programmable latency. The latency is programmed using an attribute namely svt_pcie_pl_configuration :: tx_shutdown_latency.

## 12.5 UVM Component Class svt\_pcie\_pl

This class is UVM 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 UVM phases.

**Table 12-6 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	Deprecated.
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.

**Table 12-6 Class svt\_pcie\_pl (Continued)**

Member	Description
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.
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

**Table 12-7 PHY Layer Functions and Configuration Members**

Member	Description
<b>Functions</b>	
enable_autonomous_16g_equalization	When set to 1, VIP maintains default behavior of carrying out autonomous 16G equalization procedure. When set to 0, the VIP does not carry out autonomous 16G equalization procedure. NOTE: When set to 0, user must call PERFORM_EQUALIZATION service sequence from the VIP (Downstream Port) to execute software based 16G equalization procedure.
get_expected_link_speed_value	This function returns the expected link speed value.
get_expected_link_width_value	This function returns the current expected link width. NOTE: This function does not return the deferred expected link width value. Please see the NOTE for set_link_width_values function which describes when the link width settings are deferred.
get_link_width_value	This function returns the current maximum link width setting supported by VIP. NOTE: This function does not return the deferred maximum link width value. Please see the NOTE for set_link_width_values function which describes when the link width settings are deferred.
get_supported_link_speeds_value	This function returns the supported link speeds.

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
get_supported_link_width_vector_value	This function returns the current supported link width vector. NOTE: This function does not return the deferred supported link width vector value. Please see the NOTE for set_link_width_values function which describes when the link width settings are deferred
get_target_link_speed_value	This function returns the target link speed value.
is_link_speed_settings_valid	Checks to see that link speed settings follow PCIe protocol requirements.
get_supported_link_width_vector_value	This function returns the current supported link width vector. NOTE: This function does not return the deferred supported link width vector value. Please see the NOTE for set_link_width_values function which describes when the link width settings are deferred.
get_target_link_speed_value	This function returns the target link speed value.
is_link_speed_settings_valid	Checks to see that link speed settings follow PCIe protocol requirements.
is_link_width_settings_valid	Checks to see that link width settings follow PCIe protocol requirements.
set_link_speed_values	<p>Sets the values of the link speed settings of VIP. NOTE: Link speed settings should not be reconfigured while VIP is in LTSSM Configuration.Complete state. VIP issues a warning when attempted to reconfigure while LTSSM is in Configuration.Complete state and reconfigured values are ignored.</p> <ul style="list-style-type: none"> <li>supported_link_speeds_value - It represents all the possible link speeds which are supported during link retraining. If link training is bypassed, the highest supported speed is always used.</li> <li>target_link_speed_value - Sets the desired speed of operation for a downstream port. For an upstream port the target link speed is not applicable. When unset, the target link speed is automatically updated to the highest supported link speed for a downstream port.</li> <li>expected_link_speed_value - The expected negotiated link speed value. The expected link speed check is performed only after LTSSM is in L0 state and a downstream port has transmitted a SDP packet. When unset, it is automatically updated to be same as the target link speed.</li> </ul>

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
set_link_width_values	<p>Sets the values of the link width settings of VIP. NOTE: Link width settings are applicable instantly when VIP LTSSM is not in L0 state. If link width settings are reconfigured while VIP LTSSM is in L0 state, then link width settings are deferred until LTSSM transitions out of L0 state. The deferred values can be reconfigured again to override initial deferred values of link width settings until LTSSM continues to stay in L0 state.</p> <ul style="list-style-type: none"> <li>link_width_value - The maximum link width the LTSSM is allowed to negotiate. When link training is bypassed, VIP uses maximum link width as negotiated link width.</li> <li>supported_link_width_vector_value - All the possible link widths a VIP can support from 1 upto link_width value. When unset in the function call, it prompts VIP to set supported_link_width_vector to support all the possible link widths from 1 upto link_width value.</li> <li>expected_link_width_value - The expected negotiated link width value. This value cannot be greater than the maximum link width value as represented by link_width variable. When unset in the function call, it prompts VIP to set expected_link_width value same as the value of link_width argument.</li> </ul>
<b>Attributes</b>	
allow_l0_exit_truncate_tx_packet	When set the LTSSM will exit L0 only when the transmitter is idle and not sending a TLP/DLLP during when receiving a compliance pattern
assert_turn_off_signaling_continuously	When the mpipe decides to use signaling to turn off lanes that aren't part of the link, setting this to a 1 makes the mpipe drive the turn off signaling continuously. A value of zero will cause the turn off signaling to occur for one pclk cycle.
attached_fs[32]	Programs the attached_fs value driven by VIP in PIPE mode for 8g when the VIP is the MAC.
attached_fs_16g[32]	Programs the attached_fs value driven by VIP in PIPE mode for 16g when the VIP is the MAC.
attached_lf[32]	Programs the attached_lf value driven by VIP in PIPE mode for 8g when the VIP is the MAC.
attached_lf_16g[32]	Programs the attached_lf value driven by VIP in PIPE mode for 16g when the VIP is the MAC.
configuration_complete_timeout_ns	Timeout value in the Configuraton.Complete state
configuration_idle_timeout_ns	Timeout value in the Configuration.Idle state.
configuration_lanenum_wait_timeout_ns	Timeout value in the Configuration.:Lanenum.Wait state.
configuration_linkwidth_accept_timeout_ns	Timeout value in the Configuration.Linkwidth state.
configuration_linkwidth_start_timeout_ns	Timeout value in the Configuration.Linkwidth state.

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
detect_active_timeout_ns	Timeout interval for the Detect.Active state.
detect_active_txdetectrx_spacing_ns	Spacing between receiver detects in detect.active when the first receiver detect does not return a receiver present on all configured lanes.
detect_quiet_speed_change_ns	If the LTSSM enters Detect.Quiet at a speed other than 2.5G, this is the time the LTSSM will state in Detect.Quiet to complete a speed change back to 2.5G regardless of whether data is being received.
detect_quiet_timeout_ns	Timeout interval for Detect.Quiet state.
disable_scrambling	When set to 1'b1 scrambling will be disabled for all speeds, and the LTSSM will set the disable scrambling bit on all transmitted TS1 and TS2 ordered sets. NOTE. This is applicable only for active VIP.
disabled_timeout_ns	Time before a downstream component will exit the Disabled state if no EIOS is detected
downstream_lanes_recovery_eq_phase1_timeout_ns	Timeout in NS for downstream lanes in Recovery.Equalization.Phase1.
downstream_lanes_recovery_eq_phase2_timeout_ns	Timeout in NS for downstream lanes in Recovery.Equalization.Phase2.
downstream_lanes_recovery_eq_phase3_timeout_ns	Timeout in NS for downstream lanes in Recovery.Equalization.Phase3.
downstream_preset_value	Downstream Preset value. Applicable only for devices using downstream ports at 8G.
downstream_preset_value_16g[32]	Downstream Preset value. Applicable only for devices using downstream ports at 16G.
downstream_receiver_preset_hint[32]	The variable is applicable to a device with downstream ports only. Downstream ports advertise this value in the transmitted EQ TS1s in equalization phase 0 at 8G.
downstream_receiver_preset_hint_16g[32]	The variable is applicable to a device with downstream ports only. Downstream ports advertise this value in the transmitted EQ TS1s in equalization phase 0 at 16G.
dut_receiver_present	For SPIPE, serdes and PMA models, this indicates which lanes the VIP will see as having a receiver present when the receiver detect is done in detect.active. Each bit corresponds to a lane with bit 0 corresponding to lane 0, bit 1 to lane 1 and so on. Not valid for MPIPE models-- MPIPE will use the receiver detect mechanism defined in the PIPE interface

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
enable_auto_directed_speed_change	This attribute when set to 1'b1 enables automatic speed change when advertised supported speed is greater than 2.5G on transmitted and received TS Ordered Sets.
enable_equalization_coefficients_checks	Enables equalization coefficients check.
enable_equalization_verification_mode	Enables equalization verification mode.
enable_extended_synch	When set, the transmitter for each lane must transmit 1024 TS1 ordered sets before transitioning from Recovery.RcvrLock to Recover.RcvrConfig.
enable_l1_clk_power_management	When set to 0, the LTSSM will keep clkreq# asserted in the L1 low power state. When set to a 1, the LTSSM will deassert clkreq# when entering the L1 low power state. Note that per the pipe spec if clkreq# is deasserted then the LTSSM should transition to P2 instead of P1. This control is valid only when L1 substate support is not enabled.
enable_ltssm_transition_when_compliance_bit_set	When set to 0 the LTSSM will behave normally and transition out of polling.active regardless of whether or not the compliance receive bit is set. When set to 1 (default) the LTSSM will remain in polling.active to test the case where when the DUT is receiving the compliance receive bit in polling.active it should wait until polling.active timeout and then go to polling.compliance.
enable_per_lane_spipe_phystatus_rand_delay	When set to a 1 the SPIPE VIP will randomize per lane phy phystatus asserted back to the DUT to signal completion of power state changes, speed changes, receiver detect, etc. This control has no effect for PIPE 2 models and MPIPE models.
enable_pipe_reset_n_assertion_in_detect_quiet	This attribute controls automatic pipe_resetn assertion in detect.quiet to test the switching to P1 method for lane turn off feature.
enable_random_data_on_turn_off_lane	When set, the VIP will send random data on turned off lanes for mpipe.
enable_relaxed_skp_checking	When set, the SVC will not flag skp ordered sets received with greater or less than 3 skp symbols. This should only be set when there are repeaters or switches between the root and endpoint.
enable_rxeqeval_default_settings_vector	This variable controls which phase or phases the device will request an evaluation on the settings that exist on entry to equalization. For RC: Setting bit 0 enables an eq eval during phase 1. Setting bit 1 enables an EQ eval during phase 3. For EP: Setting bit 0 enables an eq eval during phase 0. Setting bit 1 enables an EQ eval during phase 2. If both bits 0 and 1 are set, then an eq eval will be performed in both phase0/2 (EP) or phase 1/3(RC).

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
enable_upconfigure_support	Enables support for upconfiguration.
enter_compliance	This attribute represents enter_compliance register field.
eq_rx_reset_eieos_interval	During equalization phase 2 and 3, when the reset eieos interval bit is set on incoming TS1 this value controls the interval at which EIEOS are received.
expected_preset_to_coefficients_mapping_entry_enable	A bit mapped variable where bit N corresponds to entry N in expected_preset_to_coefficients_mapping_table. When bit is set to 1'b1, it indicates that corresponding entry in expected_preset_to_coefficients_mapping_table is valid and VIP should check incoming TS1s to show the correct coefficient values when a particular preset has been requested. For 8G speed only.
expected_preset_to_coefficients_mapping_entry_enable_16g	A bit mapped variable where bit N corresponds to entry N in expected_preset_to_coefficients_mapping_table. When bit is set to 1'b1, it indicates that corresponding entry in expected_preset_to_coefficients_mapping_table is valid and VIP should check incoming TS1s to show the correct coefficient values when a particular preset has been requested. For 16G speed only.
expected_preset_to_coefficients_mapping_table[16]	Preset to coefficients mapping table used to verify DUT's preset to coefficient mappings. The table should be programmed exactly like the DUT's preset table. For a passive VIP, the table should be programmed exactly like the monitored device's preset table. The table is indexed by a preset value. Mapping table stores coefficients values packed in the following format. { postcursor_coeff, cursor_coeff, precursor_coeff } Default value = { 6'h0c, 6'h24, 6'h00 } == 18'h0c900. For 8G only.
expected_preset_to_coefficients_mapping_table_16g[16]	Preset to coefficients mapping table used to verify DUT's preset to coefficient mappings. The table should be programmed exactly like the DUT's preset table. For a passive VIP, the table should be programmed exactly like the monitored device's preset table. The table is indexed by a preset value. Mapping table stores coefficients values packed in the following format. { postcursor_coeff, cursor_coeff, precursor_coeff } Default value = { 6'h0c, 6'h24, 6'h00 } == 18'h0c900. For 16G only.
fs_value[32]	Represents FS value advertised by VIP in TS1s during 8g equalization phase 1.
fs_value_16g[32]	Represents FS value advertised by VIP in TS1s during 16g equalization phase 1.
get_local_preset_coefficients_timeout_ns	Represents timeout in NS at which PIPE MAC times out waiting for PIPE PHY to respond to preset to coefficients mapping request.



**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
highest_enabled_equalization_phase	Specifies highest enabled equalization phase. When set to 1, enables equalization phase0 and phae 1. When set to 3, enables equalization phases 0, 1, 2, and 3.
hot_reset_mode	This attribute is deprecated. To control Hot Reset mode, use the SET_HOT_RESET_MODE service type in the svt_pcie_pl_service class.
hot_reset_timeout_ns	Timeout value in Hot_Reset state.
idle_to_rlock_transitioned_variable	This attribute represents idle_to_rlock_transitioned_variable LTSSM variable.
inferred_electrical_idle_recovery_rcvrclk_8g_time_ns	The amount of time in recovery.rcvrlock at 8G before the LTSSM will infer elec idle of no TS1 are received.
inferred_electrical_idle_skp_time_ns	The amount of time in L0 before the LTSSM will infer electrical idle if no SKP ordered sets are received.
invert_tx_polarity	The variable 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.
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
lane_reversal_mode	This attribute controls lane reversal mode
lf_value[32]	Represents LF value advertised by VIP in TS1s during 8g equalization phase 1.
lf_value_16g[32]	Represents LF value advertised by VIP in TS1s during 16g equalization phase 1.
link_number	This attribute represents link number used during link training.
local_fs[32]	Programs the local_fs value driven by VIP in PHY PIPE mode for 8g.
local_fs_16g[32]	Programs the local_fs value driven by VIP in PHY PIPE mode for 16g.
local_lf[32]	Programs the local_lf value driven by VIP in PHY PIPE mode for 8g.
local_lf_16g[32]	Programs the local_lf value driven by VIP in PHY PIPE mode for 16g.
loopback_exit_elec_idle_timeout_ns	Amount of time the LTSSM transmit electrical idle in Loopback.exit before entering detect.

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
loopback_master_enter_compliance_tx_ts1_timeout_ns	Time before the loopback master enters loopback.active if no TS1s are received w/ the enter compliance bit set.
loopback_master_no_enter_compliance_tx_ts1_timeout_ns	Time before the loopback master enters detect if no TS1s are received w/ the enter compliance bit clear.
loopback_master_speed_change_timeout_ns	Speed change delay for loopback master.
loopback_slave_speed_change_timeout_ns	Speed change delay for loopback slave.
ltssm_transition_evaluation_delay_ns	Amount of time in NS the port will delay before transitioning to next state in the presence of errors.
max_configuration_lanenum_accept_timeout_ns	Time at which config.linkwidth.accept will make a decision on the linkwidth if errors are received.
max_eq_evaluation_delay[32]	Represents maximum time in NS to evaluate attached transmitter setting changes during equalization.
max_eq_preset_coeff_validation_delay[32]	Represents maximum amount of time VIP takes to either accept or reject a received preset/coefficient request.
max_num_pclk_cycles_to_assert_invalid_request	This variable controls the maximum number of cycles that InvalidRequest will be asserted upon detecting invalid direction change feedback from an EQ evaluation request.
max_num_rx_eieos_before_fts	Maximum number of EIEOS to receive before sending FTS.
max_num_tx_eieos_before_fts	Maximum number of EIEOS ordered sets transmitted before transmitting FTS. (5GT/s only).
max_num_tx_eios_before_l1_gen1	Maximum number EIOS to send to accept L1 for Gen1.
max_num_tx_eios_before_l1_gen2	Maximum number EIOS to send to accept L1 for Gen2.
max_num_tx_eios_before_l1_gen3	Maximum number EIOS to send to accept L1 for Gen3.
max_rx_eq_eval_delay[32]	Represents maximum amount of time in NS VIP in PHY PIPE mode takes to respond to RxEqEval request.
max_rx_lane_skew_16g	Maximum allowed lane skew before the SVC flags a skew violation on the receive side at 16G.
max_rx_lane_skew_2_5g	Maximum allowed lane skew before the SVC flags a skew violation on the receive side at 2.5G.
max_rx_lane_skew_5g	Maximum allowed lane skew before the SVC flags a skew violation on the receive side at 5G.
max_rx_lane_skew_8g	Maximum allowed lane skew before the SVC flags a skew violation on the receive side at 8G.
max_rx_skp_interval_in_blocks	Maximum number of blocks before the upper phy flags an error due to lack of a SKP ordered set (8GT/s).

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
max_rx_skp_interval_in_symbol_times	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).
max_spipe_phystatus_delay	Maximum number of PIPE CLK cycles the SVC will wait before asserting phystatus indicating completion of a phy function (for example, power state change or rate change). Valid only for SPIPE models.
max_spipe_phystatus_p2_exit_change_delay	Maximum number of PIPE CLK cycles that the SVC will keep phystatus asserted upon exiting from the P2 power state. Valid for SPIPE only.
max_spipe_preset_coefficients_delay	Represents a maximum delay time by PIPE PHY to respond to preset request by returning the coefficients request.
max_tx_lane_skew_16g	Sets the maximum lane skew between transmit lanes in symbol times at 16Gb/s. Skew is randomized on every link down event, speed change, and at initial startup.
max_tx_lane_skew_2_5g	Sets the maximum lane skew between transmit lanes in symbol times at 2.5Gb/s. Skew is randomized on every link down event, speed change, and at initial startup.
max_tx_lane_skew_5g	Sets the maximum lane skew between transmit lanes in symbol times at 5Gb/s. Skew is randomized on every link down event, speed change, and at initial startup.
max_tx_lane_skew_8g	Sets the maximum lane skew between transmit lanes in symbol times at 8Gb/s. Skew is randomized on every link down event, speed change, and at initial startup.
max_tx_skp_interval_in_blocks	Maximum number of blocks before the upper phy must schedule the insertion of a SKP ordered set (8GT/s).
max_tx_skp_interval_in_symbol_times	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).
max_tx_skp_symbols_in_ordered_set	Maximum number of SKP symbols in an ordered set at 2.5GT/s and 5GT/s.
max_tx_skp_symbols_in_ordered_set_8g	Maximum number of SKP symbols in an ordered set at 8GT/s. Acceptable values: 1 – 5.
max_t_common_mode_ns	Max time before transmitting TS2 after coming out of L1 substates.
max_t_l1_2_ns	Maximum time to wait in L1.2 when CLKREQ must remain inactive.
max_t_power_off_ns	Maximum time to wait in L1.2 after sampling CLKREQ deasserted to refclk turned off.

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
max_t_power_on_ns	Maximum time to wait in L1.2 after sampling CLKREQ asserted before driving interface.
min_configuration_lanenum_accept_timeout_ns	Time at which config.linkwidth.accept will be forced to make a decision on the linkwidth if errors are received.
min_eq_evaluation_delay[32]	Represents minimum time in NS to evaluate attached transmitter setting changes during equalization.
min_eq_preset_coeff_validation_delay[32]	Represents minimum amount of time VIP takes to either accept or reject a received preset/coefficient request.
min_l1_idle_time_ns	Minimum amount of time the LTSSM will remain in L1.Idle at speeds other than 2.5G even if directed to exit.
min_num_pclk_cycles_to_assert_invalid_request	This variable controls the minimum number of cycles that InvalidRequest will be asserted upon detecting invalid direction change feedback from an EQ evaluation request.
min_num_rx_eieos_before_fts	Minimum number of EIEOS to receive before sending FTS.
min_num_tx_eieos_before_fts	SVT_PCIE_MIN_NUM_TX_EIEOS_BEFORE_FTS_DEFAULT Minimum number of EIEOS ordered sets transmitted before transmitting FTS. (5GT/s only)
min_num_tx_eios_before_l1_gen1	Minimum num EIOS to send to accept L1 for Gen1.
min_num_tx_eios_before_l1_gen2	Minimum num EIOS to send to accept L1 for Gen2.
min_num_tx_eios_before_l1_gen3	Minimum num EIOS to send to accept L1 for Gen3
min_rx_eq_eval_delay[32]	Represents minimum amount of time in NS VIP in PHY PIPE mode takes to respond to RxEqEval request.
min_rx_skp_interval_in_blocks	Minimum number of blocks before the upper phy flags an error due to the lack of a SKP ordered set (8GT/s).
min_rx_skp_interval_in_symbol_times	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)
min_spipe_phystatus_delay	Minimum number of pipe clk cycles the SVC will wait before asserting phystatus indicating completion of a phy function (e.g. power state change or rate change). Valid only for SPIPE models.
min_spipe_phystatus_p2_exit_change_delay	Minimum number of pipe clk cycles that the SVC will keep phystatus asserted upon exiting from the P2 power state. Valid for SPIPE only.
min_spipe_preset_coefficients_delay	Represents a minimum delay time by PIPE PHY to respond to preset request by returning the coefficients request.

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
min_tx_lane_skew_16g	Sets the minimum lane skew between transmit lanes in symbol times at 16Gb/s. Skew is randomized on every link down event, speed change, and at initial startup.
min_tx_lane_skew_2_5g	Sets the minimum lane skew between transmit lanes in symbol times at 2.5Gb/s. Skew is randomized on every link down event, speed change, and at initial startup.
min_tx_lane_skew_5g	Sets the minimum lane skew between transmit lanes in symbol times at 5Gb/s. Skew is randomized on every link down event, speed change, and at initial startup.
min_tx_lane_skew_8g	Sets the minimum lane skew between transmit lanes in symbol times at 8Gb/s. Skew is randomized on every link down event, speed change, and at initial startup.
min_tx_skp_interval_in_blocks	Minimum number of blocks before the upper phy must schedule the insertion of a SKP ordered set (8GT/s).
min_tx_skp_interval_in_symbol_times	Minimum 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_symbols_in_ordered_set	Minimum number of SKP symbols in an ordered set at 2.5GT/s and 5GT/s.
min_tx_skp_symbols_in_ordered_set_8g	Minimum number of SKP symbols in an ordered set at 8GT/s. Acceptable values: 1 – 5.
min_t_common_mode_ns	Minimum time before transmitting TS2 after coming out of L1 substates.
min_t_l1_2_ns	Minimum time to wait in L1.2 when CLKREQ must remain inactive,
min_t_power_off_ns	Minimum time to wait in L1.2 after sampling CLKREQ deasserted to refclk turned off.
min_t_power_on_ns	Minimum time to wait in L1.2 after sampling CLKREQ asserted before driving interface.
model_instance_scope	This is the full hierarchical path name to the instance of the SVC model which the driver model is instantiated in. The path name is concatenated with the name of this component passed to the constructor to generate the lookup string used to find the SV API instance.
num_additional_ts_eq_before_trans_eq0	Number of additional TS1s to transmit before transitioning out of equalization phase 0.
num_additional_ts_eq_before_trans_eq1	Number of additional TS1s to transmit before transitioning out of equalization phase 1.

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
num_additional_ts_eq_before_trans_eq2	Number of additional TS1s to transmit before transitioning out of equalization phase 2.
num_additional_ts_eq_before_trans_eq3	Number of additional TS1s to transmit before transitioning out of equalization phase 3.
num_rx_fts_required_2_5g	The minimum number of FTS ordered sets required for a receiver to infer lock when exiting from electrical idle at 2.5GT/s.
num_rx_fts_required_5g	The minimum number of FTS ordered sets required for a receiver to infer lock when exiting from electrical idle at 5GT/s.
num_rx_fts_required_8g	The minimum number of FTS ordered sets required for a receiver to infer lock when exiting from electrical idle at 8GT/s.
num_rx_recovery_idle_data	Number of Idle data symbols received in Recovery.Idle before transitioning to L0. This attribute can be used to delay transition from Recovery.Idle to L0.
num_tx_fts_required_2_5g	When SKIP_INITIAL_LINK_TRAINING is set, this is the default N_FTS value which the LTSSM will use at 2.5G when exiting from P1. All subsequent exits will use a value obtained from training sets used during link training.
num_tx_fts_required_5g	When SKIP_INITIAL_LINK_TRAINING is set, this is the default N_FTS value which the LTSSM will use at 5G when exiting from P1. All subsequent exits will use a value obtained from training sets used during link training.
num_tx_fts_required_8g	When SKIP_INITIAL_LINK_TRAINING is set, this is the default N_FTS value which the LTSSM will use at 8G when exiting from P1. All subsequent exits will use a value obtained from training sets used during link training.
num_tx_recovery_idle_data	Number of Idle data symbols transmitted in Recovery.Idle before transitioning to L0. This attribute can be used to delay transition from Recovery.Idle to L0.
num_tx_ts1_in_polling_active	Sets the number of training sets the LTSSM must transmit in Polling.Active before exiting this state.
pclk_rate[4]	Specifies the PCLK rate to be used at various data rates. pclk_rate[0] represents the rate used at 2.5 GT/s data rate. pclk_rate[1] represents the rate used at 5.0 GT/s data rate. pclk_rate[2] represents the rate used at 8.0 GT/s data rate. pclk_rate[3] represents the rate used at 16.0 GT/s data rate.  Also refer to the description of 'pipe_width' which summarizes various PIPE configurations and 'pipe_width' and 'pclk_rate' settings to achieve these PIPE configurations.

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
pclk_turned_off_in_p2	This attribute controls whether MAC should turn off PCLK in P2 while running in PCLK as PHY input mode or not.
phystatus_timeout_ns	When SVC pl is a pipe master, this is the time the pl will wait for a PHYSTATUS to go high to acknowledge a receiver detect, a rate change, or powerdown change.
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's Pie8MacStateMachine will wait in its TX_WAIT_RX_PHY_RESP state for the PHYDataEn signal to be received.
pie8_mode_en	Enable PIE-8 mode (PHY Interface Extensions Supporting 8GT/s PCIe): Enables the PIE-8 mode state machines (either as "MAC master" or "PHY Slave" based on is_pie8_mode_master = SVT_PCIE_IS_PIE8_MODE_MASTER_DEFAULT;).
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_phy_delay_to_tx_cmd_out_minpie8_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.
x_margin[4]	This configuration sets the tx_margin field.. When set VIP(MAC) will transmit this value of the tx_margin on the PIPE interface. <ul style="list-style-type: none"> <li>• tx_margin[0] - tx_margin @ Gen1 speed.</li> <li>• tx_margin[1] - tx_margin @ Gen2 speed.</li> <li>• tx_margin[2] - tx_margin @ Gen3 speed.</li> <li>• tx_margin[3] - tx_margin @ Gen4 speed.</li> </ul>
tx_ui_skew_2_5g	Maximum unit interval skew for 2.5G Valid range = [0:9]. The skew is in terms of a reference lane which is selected by the VIP randomly and then VIP distributes UI skew on all other active lanes randomly(within [1:tx_ui_skew_2_5g]) The unit for tx_ui_skew_2_5g is bit_clk period(400ps) for 2.5G

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
tx_ui_skew_5g	Maximum unit interval skew for 5G Valid range = [0:9]. The skew is in terms of a reference lane which is selected by the VIP randomly and then VIP distributes UI skew on all other active lanes randomly (within [1:tx_ui_skew_5g]) The unit for tx_ui_skew_5g is bit_clk period (200ps) for 5G
tx_ui_skew_8g	Maximum unit interval skew for 8G Valid range = [0:8]. The skew is in terms of a reference lane which is selected by the VIP randomly and then VIP distributes UI skew on all other active lanes randomly (within [1:tx_ui_skew_8g]) The unit for tx_ui_skew_8g is bit_clk period (125ps) for 8G
tx_ui_skew_16g	Maximum unit interval skew for 16G Valid range = [0:8]. The skew is in terms of a reference lane which is selected by the VIP randomly and then VIP distributes UI skew on all other active lanes randomly (within [1:tx_ui_skew_8g]) The unit for tx_ui_skew_8g is bit_clk period (62.5ps) for 16G
ssc_mode [4]	Spread spectrum clocking mode for serial bit clock on the transmit path. It is only applicable for VIP running with serial interface. 0 (default) - SSC_MODE_DISABLED => no ssc profile. 1 - SSC_MODE_DOWN => down spread spectrum. 2 - SSC_MODE_CENTER => center spread spectrum. ssc_mode[0] - ssc_mode for Gen1 speed. ssc_mode[1] - ssc_mode for Gen2 speed. ssc_mode[2] - ssc_mode for Gen3 speed. ssc_mode[3] - ssc_mode for Gen4 speed.
ssc_max_spread [4]	This configuration represents the ssc spread value in ppm. When ssc_mode = SC_MODE_DISABLED, this configuration is not used. When ssc_mode = SC_MODE_DOWN, this configuration represents DOWN spread value in ppm. When ssc_mode = SC_MODE_CENTER, this configuration represents CENTER spread value in ppm. ssc_max_spread[0] - ssc_max_spread @ Gen1 speed. ssc_max_spread[1] - ssc_max_spread @ Gen2 speed. ssc_max_spread[2] - ssc_max_spread @ Gen3 speed. ssc_max_spread[3] - ssc_max_spread @ Gen4 speed. default is 16'd2300 for all speed if ssc_mode = SC_MODE_CENTER   SC_MODE_DOWN



**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
ssc_modulation_rate	This configuration represents the modulation rate for ssc clock 2'b00 - 30Khz. 2'b01 - 31Khz. 2'b10 - 32Khz. 2'b11 - 33Khz. default value is 33Khz.

Table 12-7 PHY Layer Functions and Configuration Members (Continued)

Member	Description
pipe_width[4]	<p>Specifies the PIPE width to be used at various data rates.  pipe_width[0] represents the width used at 2.5 GT/s data rate.  pipe_width[1] represents the width used at 5.0 GT/s data rate.  pipe_width[2] represents the width used at 8.0 GT/s data rate.  pipe_width[3] represents the width used at 16.0 GT/s data rate.  Summary of various PIPE configurations and settings of 'pipe_width' and 'pclk_rate' to achieve these configurations.</p> <ol style="list-style-type: none"> <li>1. Fixed width[8 bits] and variable PCLK PIPE configuration  [PIPE_8] pipe_width[0] = PIPE_8_BITS; pclk_rate[0] = PCLK_250_MHZ; -- Achieve 2.5 GT/s pipe_width[1] = PIPE_8_BITS; pclk_rate[0] = PCLK_500_MHZ; -- Achieves 5.0 GT/s pipe_width[2] = PIPE_8_BITS; pclk_rate[0] = PCLK_1000_MHZ; -- Achieves 8.0 GT/s pipe_width[3] = PIPE_8_BITS; pclk_rate[0] = PCLK_2000_MHZ; -- Achieves 16.0 GT/s</li> <li>2. Fixed width[16 bits] and variable PCLK PIPE configuration  [PIPE_16] pipe_width[0] = PIPE_16_BITS; pclk_rate[0] = PCLK_125_MHZ; -- Achieve 2.5 GT/s pipe_width[1] = PIPE_16_BITS; pclk_rate[0] = PCLK_250_MHZ; -- Achieves 5.0 GT/s pipe_width[2] = PIPE_16_BITS; pclk_rate[0] = PCLK_500_MHZ; -- Achieves 8.0 GT/s pipe_width[3] = PIPE_16_BITS; pclk_rate[0] = PCLK_1000_MHZ; -- Achieves 16.0 GT/s</li> <li>3. Fixed width[32 bits] and variable PCLK PIPE configuration  [PIPE_32] pipe_width[0] = PIPE_32_BITS; pclk_rate[0] = PCLK_62_5_MHZ; -- Achieve 2.5 GT/s pipe_width[1] = PIPE_32_BITS; pclk_rate[0] = PCLK_125_MHZ; -- Achieves 5.0 GT/s pipe_width[2] = PIPE_32_BITS; pclk_rate[0] = PCLK_250_MHZ; -- Achieves 8.0 GT/s pipe_width[3] = PIPE_32_BITS; pclk_rate[0] = PCLK_500_MHZ; -- Achieves 16.0 GT/s</li> <li>4. Fixed PCLK[250 MHz] and variable width PIPE configuration  [PIPE_250_MHZ] pipe_width[0] = PIPE_8_BITS; pclk_rate[0] = PCLK_250_MHZ; -- Achieve 2.5 GT/s pipe_width[1] = PIPE_16_BITS; pclk_rate[0] = PCLK_250_MHZ; -- Achieves 5.0 GT/s pipe_width[2] = PIPE_32_BITS; pclk_rate[0] = PCLK_250_MHZ; -- Achieves 8.0 GT/s pipe_width[3] = PIPE_32_BITS; pclk_rate[0] = PCLK_500_MHZ; -- Achieves 16.0 GT/s</li> </ol>
polling_active_timeout_ns	If at least one but not all unconfigured lanes detect a receiver, this configures the time between receiver detect sequences.
polling_configuration_timeout_ns	Timeout limit for the Polling.Configuration state.

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
preset_to_coefficients_mapping_entry_valid	A bit mapped variable where bit N corresponds to entry N in preset_to_coefficients_mapping_table. When bit is set to 1'b1, it indicates that corresponding entry in preset_to_coefficients_mapping_table is valid for 8G.
preset_to_coefficients_mapping_entry_valid_16g	A bit mapped variable where bit N corresponds to entry N in preset_to_coefficients_mapping_table. When bit is set to 1'b1, it indicates that corresponding entry in preset_to_coefficients_mapping_table is valid for 16g.
preset_to_coefficients_mapping_table[16]	Preset to coefficients mapping table is used to map received preset requests to coefficients for use in local transmitter settings. For a passive VIP, the table should be programmed exactly like preset table of the link partner of monitored device. The table is indexed by a preset value. Mapping table stores coefficients values packed in the following format. { postcursor_coeff, cursor_coeff, precursor_coeff } Default value = { 6'h0c, 6'h24, 6'h00 } == 18'h0c900. For 8G speed only.
preset_to_coefficients_mapping_table_16g[16]	Preset to coefficients mapping table is used to map received preset requests to coefficients for use in local transmitter settings. For a passive VIP, the table should be programmed exactly like preset table of the link partner of monitored device. The table is indexed by a preset value. Mapping table stores coefficients values packed in the following format. { postcursor_coeff, cursor_coeff, precursor_coeff } Default value = { 6'h0c, 6'h24, 6'h00 } == 18'h0c900. For 16G speed only.
quiesce_guarantee	The variable represents the quiesce_guarantee variable in the VIP and this variable is reflected in Symbol 6 bit 6 of TS2 OS transmitted at 8G in Recovery.RcvrCfg state.
recovery_idle_timeout_ns	Timeout value in Recovery.Idle state.
recovery_rcvrcfg_timeout_ns	Timeout value in Recovery.RcvrCfg.
recovery_rcvrlock_timeout_ns	Timeout in Recovery.RcvrLock state.
recovery_speed_electrical_idle_time_ns	Number of ns the transmitter is required to be transmitting electrical idle during a speed change once the receiver has detected/inferred electrical idle in the state RECOVERY_SPEED.
recovery_speed_timeout_ns	Timeout value in Recovery.Speed state.
reject_preset_coefficient_request	A bit mapped attribute where value of bit N applies to lane N. Setting a bit to 1'b1 forces that particular lane to reject all new requests. This bit is applicable only in equalization phase 2 for downstream ports and equalization phase 3 for upstream ports.

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
rx_dc_balance_mode_8g	This attribute controls the dc_balance field in 8g Rx TS Ordered Sets. If set to DC_BALANCE mode, the dc_balance field is set per spec when link is running at 8g. If set to TS_IDENTIFIER mode, dc_balance field is set to TS1/2 identifier similar to spec for 2.5g/5g TS1/2 Ordered Sets.
set_ts_compliance_receive	When set, outgoing training sets will have the compliance receive bit set.
srisc_mode_enabled	Used to enable/disable SRIS mode. NOTE. Currently this is only used to control transmission/reception of modified compliance pattern at 128/130b in SRIS and non SRIS modes. If set to "1" then VIP will transmit/receive SRIS mode modified compliance pattern else non SRIS one.
skip_initial_link_training	Allows link training to be completely skipped. The SVC will enter L0 once lane alignment has been achieved.
skip_polling_active	Allows Polling.Active to be skipped during link training. This helps speed up link training for cases not specifically testing the ltssm states.
spipe_receiver_present	For SPIPE only, this sets which lanes will return receiver present value when a receiver detect is requested by a DUT MAC. Each bit corresponds to a lane with bit zero corresponding to lane 0, bit 1 to lane 1 and so on. Not valid for serial, MPIPE and PMA models.
transmit_modified_compliance	When set, the SVC will transmit the modified compliance pattern upon entry into Polling.Compliance.
turn_off_unused_lanes_on_downconfigure	When the mpipe is doing a downconfigure, this controls whether or not the special 'turn off' signaling will be sent on unused lanes. This should only be done if the link doesn't support or won't be upconfigured later on.
tx_compliance_sos	Instructs the VIP to send 2 SKP ordered sets instead of 1 when in compliance mode.
tx_dc_balance_mode_8g	This attribute controls the dc_balance field in 8g Tx TS Ordered Sets. If set to DC_BALANCE mode, the dc_balance field is set per spec when link is running at 8g. If set to TS_IDENTIFIER mode, dc_balance field is set to TS1/2 identifier similar to spec for 2.5g/5g TS1/2 Ordered Sets.
tx_select_deemphasis	At 5G, the value of select_deemphasis.
tx_shutdown_latency	This attribute controls the latency to shutdown a lane.

**Table 12-7 PHY Layer Functions and Configuration Members (Continued)**

Member	Description
tx_ts1_reset_eieos_interval_count_bit	When transmitting TS1s at 8GT/s speeds, and this bit is set to 1'b1, it forces VIP to set reset_eieos_interval_count bit in transmitted TS1s. Each lane can be set individually, so to for example to set this bit on lanes 0-3 only, set this to 32'h0000_000f.
upstream_lanes_recovery_eq_phase0_timeout_ns	Timeout in NS for upstream lanes in Recovery.Equalization.Phase0.
upstream_lanes_recovery_eq_phase1_timeout_ns	Timeout in NS for upstream lanes in Recovery.Equalization.Phase1.
upstream_lanes_recovery_eq_phase2_timeout_ns	Timeout in NS for upstream lanes in Recovery.Equalization.Phase2.
upstream_lanes_recovery_eq_phase3_timeout_ns	Timeout in NS for upstream lanes in Recovery.Equalization.Phase3.
upstream_preset_value[32]	Upstream Preset value. Applicable only for devices using downstream ports at 8G.
upstream_preset_value_16g[32]	Upstream Preset value. Applicable only for devices using downstream ports at 16G.
upstream_receiver_preset_hint[32]	The variable is applicable to a device with downstream ports only. Downstream ports advertise this value in the transmitted EQ TS2s before equalization phase 0 at 8G.
upstream_receiver_preset_hint_16g[32]	The variable is applicable to a device with downstream ports only. Downstream ports advertise this value in the transmitted EQ TS2s before equalization phase 0 at 16G.
user_tx_ts1_lane_mask	A bitwise lane enable for user training sets. A 'b1 enables user training sets for a particular lane and 'b0 disables user training sets. Bit 0 of this vector corresponds to lane 0, bit 1 corresponds to lane 1 and so on. Refer to the user training set tasks section for more details.
user_tx_ts2_lane_mask	A bitwise lane enable for user training sets. A 'b1 enables user training sets for a particular lane and 'b0 disables user training sets. Bit 0 of this vector corresponds to lane 0, bit 1 corresponds to lane 1 and so on. Refer to the user training set tasks section for more details.
rx_standby_supported	This parameter is used to enable or disable requiring rx_standby to be asserted when rate, pclk_rate or width are changed in PIPE 4.0 and above models. When not supporting rx_standby, this parameter must be set to 0. The default value of this parameter is 1.

Table 12-7 PHY Layer Functions and Configuration Members (Continued)

Member	Description
<code>fixed_ppm_due_to_tx_rx_xo</code>	<p>This configuration represents fixed ppm value in tx serial bit clk which comes due to different tx/rx crystal oscillator. The unit of this attribute is <i>ppm</i> (parts per million).</p> <p>For example, -600 <i>ppm</i> would correspond to a bit clock period of .4002401441 ns <math>(1000000.0 * .4)/(1000000.0 - 600)</math> at Gen1 speed.</p>
<code>allow_rate_power_simultaneous_change</code>	<p>In PIPE Interface mode, setting this variable to 1 allows Rate and PowerDown to change simultaneously—that is, if set to 1 the PHY (SPIPE) will not consider this as an illegal behavior and will not issue any warning. The default value of this variable is set to 1.</p> <p>Note: Only utilized by the active component and in PIPE models.</p>
<code>enable_up_port_tx_8g_eq_ts2_with_transmitter_preset_in_sym7_rcvr_cfg</code>	<p>This configuration controls the enabling/disabling of sending 8G EQ TS2 by upstream port while in Recovery.Cfg. It also allows you to configure VIP to send user-defined transmitter preset values in Bit [6:3] of Sym7 of TS2 OS while in Recovery.Cfg. The description of 5 bits of this parameter is as follows:</p> <ul style="list-style-type: none"> <li>• Bit[4]: Controls the enabling/disabling of this feature. If set to 1, upstream port will send 8G EQ TS2 OS. If set to 0, upstream port will send normal TS2 identifier in SYM7.</li> <li>• Bit[3:0]: Contains the Transmitter Preset value to be transmitted by upstream port in Bit [6:3] of Sym7 of TS2 OS while in Recovery.Cfg.</li> </ul> <p>The default value of this variable is set to 1.</p>
<code>enable_powerdown_change_before_rate</code>	<p>In PIPE_INTERFACE mode, setting this variable to 1 will enable the MAC (MPIPE) to change PowerDown before Rate while entering L2.Idle or Detect LTSSM state.</p> <p>Note: Only utilized by the active component and in PIPE models. The default value of this variable is set to 0.</p>
<code>enable_additional_power_state_transition_during_l1ss_exit</code>	<p>Specifies whether to enable or disable additional power state transition in L1.0 as described by attribute <code>pipe_powerdown_state_for_l1_0</code> during L1ss exit. This is applicable to MPIPE mode for PIPE specification revisions 4.0 or higher. The default value of this variable is set to 0.</p>

Table 12-7 PHY Layer Functions and Configuration Members (Continued)

Member	Description
<code>use_dynamic_local_fs_lf_values</code>	<p>Specifies the configuration to drive FS/LF fields in TS1 ordered sets by VIP for MPIPE.</p> <ul style="list-style-type: none"> <li>When set to 1 and dynamic preset coefficient updates are enabled (<code>enable_get_local_preset_coefficients</code> set to 1), the VIP will use <code>LocalFS</code> and <code>LocalLF</code> signals driven by PHY to drive FS and LF fields in TS1 ordered sets for MPIPE.</li> <li>When set to 1 and dynamic preset coefficient updates are disabled (<code>enable_get_local_preset_coefficients</code> set to 0), the VIP will use <code>fs_value/lf_value</code> when rate is 8 GT/s or <code>fs_value_16g/lf_value_16g</code> when rate is 16 GT/s to drive FS and LF fields in TS1 ordered sets for MPIPE.</li> <li>When set to 0, the VIP will use <code>fs_value/lf_value</code> when rate is 8 GT/s or <code>fs_value_16g/lf_value_16g</code> when rate is 16 GT/s to drive FS and LF fields in TS1 ordered sets for MPIPE.</li> </ul> <p>The default value of this attribute is set to 1 and would be applicable only when dynamic preset coefficient updates are enabled.</p>
<code>use_dynamic_local_preset_coefficients</code>	<p>Specifies the configuration to drive coefficients in TS1 ordered sets by VIP for MPIPE for a preset request from the link partner.</p> <ul style="list-style-type: none"> <li>When set to 1, the VIP will use <code>LocalTxPresetCoefficients</code> signal driven by PHY to drive coefficients in TS1 ordered sets for MPIPE to a preset request from the link partner.</li> <li>When set to 0, the VIP will use preset mapping tables as indicated by <code>preset_to_coefficients_mapping_table</code> when rate is 8 GT/s or <code>preset_to_coefficients_mapping_table_16g</code> when rate is 16 GT/s to drive coefficients in TS1 ordered sets for MPIPE to a preset request from the link partner.</li> </ul> <p>This attribute is applicable only when dynamic preset coefficient updates are enabled (<code>enable_get_local_preset_coefficients</code> set to 1). Default value of this attribute is set to 0.</p>
<code>compliance_speed_change_timeout_ns</code>	<p>Specifies the maximum amount of time in ns the VIP will remain in electrical idle when changing speed while in Polling Compliance state prior to sending compliance pattern. If randomized, the variable will resolve to a value within the range specified by the constraint <code>#valid_ranges</code>.</p>
<code>rx_modified_compliance_pattern_lock_bit</code>	<p>Specifies the value of the pattern lock bit in modified compliance pattern on each lane. The default value is set to 0.</p>
<code>re_request_setting_on_ts1_after_invalid_request</code>	<p>Supports re-request of last accepted preset/coefficient setting after <code>InvalidRequest</code>. Default value is set to 0.</p>







# 13

## Using the Driver Application

---

### 13.1 Introduction

The Driver application is implemented as a `uvm_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](#).

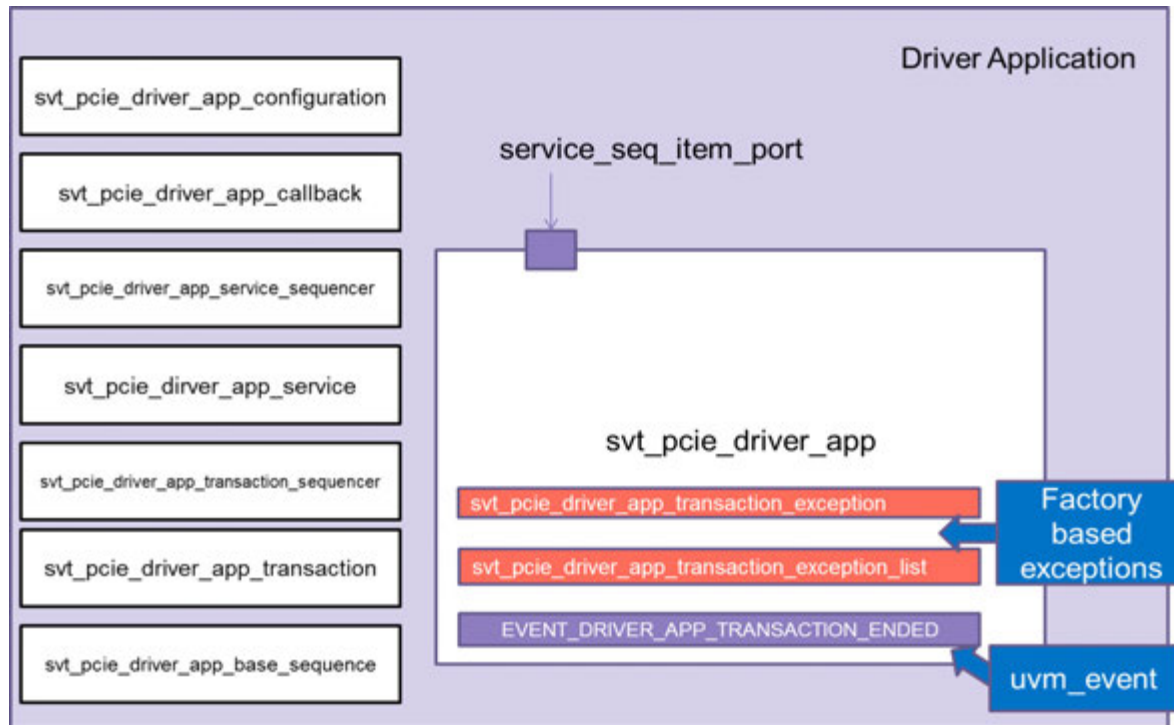


Figure 13-1 PCIe Driver application components

## 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]*

Table 13-1 shows the `svt_pcie_driver_app_configuration` items.

**Table 13-1 Driver Application Configuration Members**

<p>svt_pcie_driver_app_configuration::application_number</p> <p>Type: rand int unsigned</p> <p>Default: 1</p> <p>Description:</p> <p>Application number for the Driver application.</p>
<p>svt_pcie_driver_app_configuration::completion_timeout_ns</p> <p>Type: rand int unsigned</p> <p>Default: 50,000</p> <p>Description:</p> <p>Completion timeout value. The request times out after the time as specified by this variable if completions are not received for the request.</p>
<p>svt_pcie_driver_app_configuration::display_outstanding_commands_period</p> <p>Type: rand int unsigned</p> <p>Default: 50000</p> <p>Description:</p> <p>The display_outstanding_commands_period variable specifies the time after which the Driver application displays the outstanding/pending transactions.</p>
<p>svt_pcie_driver_app_configuration::enable_shadow_memory_checking</p> <p>Type: rand bit</p> <p>Default: 1</p> <p>Description:</p> <p>When set to 1, read transactions are checked against the data from Global Shadow application. When set to 0, read transactions are not checked against the data from Global Shadow application.</p>
<p>svt_pcie_driver_app_configuration::enable_tx_tlp_reporting</p> <p>Type: rand bit</p> <p>Default: 0</p> <p>Description:</p> <p>When set to 1, transmitted TLPs are reported to Global Shadow application. When set to 0, transmitted TLPs are not reported to Global Shadow application.</p>
<p>svt_pcie_driver_app_configuration::max_time_to_next_transition</p> <p>Type: rand int unsigned</p> <p>Default: 0</p> <p>Description:</p> <p>Maximum time between transactions before the driver will transmit another queued request.</p>
<p>svt_pcie_driver_app_configuration::min_time_to_next_transition</p> <p>Type: rand int unsigned</p> <p>Default: 0</p> <p>Description:</p> <p>Minimum time between transactions before the driver will transmit another queued request.</p>

**Table 13-1 Driver Application Configuration Members (Continued)**

svt_pcie_driver_app_configuration::model_instance_scope Type: string Description: The full hierarchical path name to the instance of the model in which the driver model is instantiated. The path name is concatenated with the name of this component passed to the constructor to generate the lookup string used to find the SV API instance. The model_instance_scope value should not be changed at this level: it is set at the agent level and propagates to all sub-levels.
svt_pcie_driver_app_configuration::percentage_use_tlp_digest Type: rand int unsigned Default: 0 Description: Percentage probability of the TD bit being set, indicating that the packet has a TLP digest.
svt_pcie_driver_app_configuration::read_completion_boundary_in_bytes Type: rand int unsigned Default: 64 Description: The variable read_completion_boundary_in_bytes specifies the RCB value. The Driver application checks all the received completions against this boundary.
svt_pcie_driver_app_configuration::requester_id Type: rand bit [15:0] Default: 32h'000_0001 Description: Requester ID
svt_pcie_driver_app_transaction::register_number Type: rand bit [9:0] Default: 'h0 Description The register_number variable represents concatenation of Extended Register Number and Register Number field in configuration packets. register_number = 0 => [Device ID   Vendor ID] PCIe configuration space Byte offset 00h register_number = 1 => [Status   Command ] => PCIe configuration space Byte offset 04h Note: most firmware use convention register numbers 0,1,2,3,4,5,6,7 as byte offset numbers as defined in the specification.
svt_pcie_driver_app_configuration::set_ido_field Type: rand bit Default: 0 Description: Sets the IDO bit in all outgoing TLPs when applicable.
svt_pcie_driver_app_configuration::set_no_snoop_field Type: rand bit Default: 0 Description: Sets the No Snoop bit in all outgoing TLPs when applicable.

**Table 13-1 Driver Application Configuration Members (Continued)**

svt_pcie_driver_app_configuration::set_relaxed_ordering_field Type: rand bit Default: 0 Description: Sets the Relaxed Ordering bit in all outgoing TLPs when applicable.
svt_pcie_driver_app_configuration::use_internal_data_buffers Type: rand bit Default: 0 Description: When set to 1, internal data buffers are used for the transactions. When set to 0, data buffers should be provided by user.
svt_pcie_driver_app_configuration:: use_ordered_tags Type: rand bit Default: 0 Description: When this is set, the driver will issue tags in ascending numerical order rather than picking a random value.

## 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 13-2](#).

**Table 13-2 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 13-3](#).

**Table 13-3 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 13-4](#).

**Table 13-4 Driver application runtime Verilog tasks**

Parameter Name	Arguments	I/O	Description
AddTLPPrefix	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.
Adds the prefix or prefixes contained in prefix_array to the next queued command.			

## 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]*



### Note

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;  
...  
`uvm_do_on (wait_seq, p_sequencer.root_virt_seqr.driver_seqr[0]);  
...
```

The service sequencer is described in [Table 13-5](#).

The service sequences are listed in [Table 13-6](#).

**Table 13-5** Driver application service sequencer

<b>svt_pcie_driver_app_service_sequencer</b> 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 uvm_sequencer to the driver if the agent is configured as UVM_ACTIVE.
---

**Table 13-6** Driver application service sequences

<b>svt_pcie_driver_app_service_base_sequence</b> This sequence is the base class for the svt_pcie_driver_app_service sequence. All the other sequences are extended from this sequence.
<b>svt_pcie_driver_app_service_is_trans_complete_sequence</b> 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.
<b>svt_pcie_driver_app_service_null_sequence</b> This sequence implements Null Traffic. This class creates a null sequence that can be associated with a sequencer but generates no traffic.
<b>svt_pcie_driver_app_service_wait_for_compl_sequence</b> 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.
<b>svt_pcie_driver_app_service_wait_until_idle_sequence</b> 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 13-7](#).

**Table 13-7 Transaction fields**

Transaction Type	address	length	first_dw_be	last_dw_be	traffic_class	address_translation	ep	payload	exception_list	block	Config only		Message only			Responses only		
											cfg_type	register_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 13-8](#).

The transaction sequences are listed in [Table 13-9](#).



#### Note

Transaction sequences for a given type of transaction use the fields shown in [Table 13-7](#).



**Table 13-8 Driver application transaction sequencer**

<b>svt_pcie_driver_app_transaction_sequencer</b> 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 uvm_sequencer to the Driver if the agent is configured as UVM_ACTIVE.
---

**Table 13-9 Driver application transaction sequences**

<b>svt_pcie_driver_app_transaction_atomicop_fetchadd_sequence</b> This sequence generates an atomic operation fetchadd sequence.
<b>svt_pcie_driver_app_transaction_atomicop_cas_sequence</b> This sequence generates an atomic operation CAS (compare and swap) request.
<b>svt_pcie_driver_app_transaction_atomicop_swap_sequence</b> This sequence generates an atomic operation swap request.
<b>svt_pcie_driver_app_transaction_base_sequence</b> 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
<b>svt_pcie_driver_app_transaction_cfg_rd_sequence</b> This sequence generates a configuration read request.
<b>svt_pcie_driver_app_transaction_cfg_wr_sequence</b> This sequence generates a configuration write request.
<b>svt_pcie_driver_app_transaction_io_rd_sequence</b> This sequence generates an I/O read request.
<b>svt_pcie_driver_app_transaction_io_wr_sequence</b> This sequence generates an I/O write request.
<b>svt_pcie_driver_app_transaction_mem_rd_sequence</b> This sequence generates a memory read request.
<b>svt_pcie_driver_app_transaction_mem_wr_sequence</b> This sequence generates a memory write request.
<b>svt_pcie_driver_app_transaction_null_sequence</b> This sequence implements Null Traffic. This class creates a null sequence which can be associated with a sequencer but generates no traffic.
<b>svt_pcie_driver_app_transaction_vendor_msg_0_sequence</b> This sequence generates a vendor-defined message type 0 request.

**Table 13-9 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 does not provide status.

### 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;
...
`uvm_do_on (wait_seq, p_sequencer.root_virt_seqr. driver_seqr[0]);
...
```

## 13.7 Driver Application Events

The driver provides an `uvm_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 `uvm_seq_item_pull_port`, `service_seq_item_port` is used for service requests.

## 13.9 Driver Application Transactions

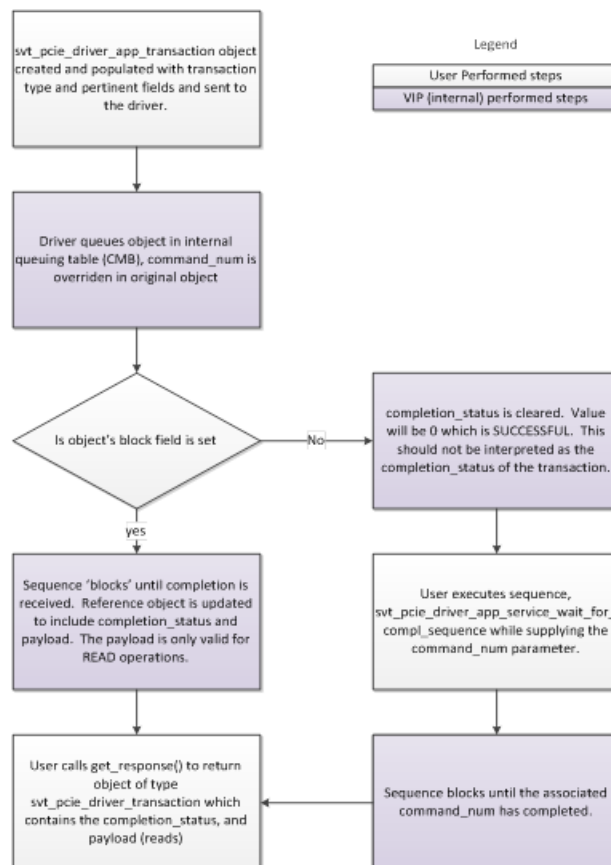
When utilizing the driver and its corresponding sequences for bus transactions, you may want to retrieve the completion status and optional data from reads. This section explains how this can be done in both blocking and non-blocking order.

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;
`uvm_create(write_tran);
// populate read_tran
read_tran.block = 0;
`uvm_send(read_tran)
get_response(read_tran);
`uvm_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.

- [“Enabling Functional Coverage”](#) on page 255
- [“Class Structure and Callbacks”](#) on page 255
- [“Overriding the Default Coverage Class”](#) on page 256
- [“Transaction Layer”](#) on page 257
- [“Data Link Layer”](#) on page 259
- [“Physical Layer”](#) on page 281
- [“PIPE Interface”](#) on page 287

### 14.1 Enabling Functional Coverage

To enable function coverage define the macro 'SVT\_PCIE\_INCLUDE\_AC\_COVERAGE' at compile time, and set the following variables in the svt\_pcie\_configuration class:

```
enable_cov = 4'b1111; // Bitwise enable
```

In the enable\_cov variable, bit 0 enables PIPE related functional coverage, and bits 1, 2, and 3 enable functional coverage for the Physical Layer, Data Link Layer, and Transaction Layers respectively.

### 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 UVM

UVM 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 14-1](#). These tasks may also be called through the SystemVerilog API.

**Table 14-1 Transaction override functions**

Function Name	Arguments	Layer
<code>SetTransactionCoverageClass</code>	<code>new_class</code> – handle to the new coverage class. The new class must be derived from <code>pciesvc_tl_fc_data</code> or <code>pciesvc_tl_fc_coverage</code> .	<code>SVC_PATH.port0.tl0</code>

**Table 14-1 Transaction override functions (Continued)**

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 14-2 lists the covergroups, coverpoints and bins present in the Transaction Layer coverage class.

**Table 14-2 Covergroups, coverpoints and bins in the Transaction Layer coverage class**

Covergroup	Coverpoints	Bins
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

**Table 14-2 Covergroups, coverpoints and bins in the Transaction Layer coverage class (Continued)**

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 14-3](#).

**Table 14-3 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 14-4](#) 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 14-4 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class**

Covergroup	Coverpoint	Bins	Comment
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	

**Table 14-4 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)**

	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	
	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	

**Table 14-4 Covergroups, coverpoints and bins in the Data Link Layer layer coverage class (Continued)**

	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 14-4 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 14-4 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 14-4 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 NAK	ACK/NAK DLLPs
	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 14-4 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 14-4 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 14-4 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 14-4 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 14-4 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 14-4 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 14-4 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 14-4 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_corrupt_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 14-4 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 14-4 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 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
	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 14-4 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 14-4 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 14-5](#).

**Table 14-5 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 14-5 Data Link Layer callbacks (Continued)**

UpdateTxTLP  Called after the link sends the last byte of a TLP.	tlp_fmt[2:0]	1	Format field
	tlp_type[4:0]	1	Type field
	tc[2:0]	1	Traffic class
	th	1	Transaction hint
	td	1	TLP Digest bit
	ep	1	Error/Poison bit
	attr_id_order	1	ID Order attribute bit
	attr_relax_order	1	Relaxed order attribute bit
	attr_no_snoop	1	No snoop attribute attribute
	at[1:0]	1	address translation
	length[9:0]	1	length field
	ecrc[31:0]	1	ECRC(digest) value
	lcrc[31:0]	1	Link CRC value
	sequence_num (int)	1	Sequence number of the TLP
	requester_id[15:0]	1	Requester ID field
	tag[7:0]	1	Tag value
	first_dw_be[3:0]	1	First DW byte enable field.
	last_dw_be[3:0]	1	Last DW byte enable field.
	address[63:0]	1	Address field.

**Table 14-5 Data Link Layer callbacks (Continued)**

	ph[1:0]	l	Processing hint
	bus_num[7:0]	l	bus number
	device_num[2:0]	l	device number
	function_num[2:0]	l	function number
	register_num[9:0]	l	Combines reg and ext_reg field for config TLPs
	message_code[7:0]	l	Message code field
	message_dword2[31:0]	l	2 <sup>nd</sup> dword of message TLP. This would be used for vendor specific messages.
	message_dword3[31:0]	l	3 <sup>rd</sup> dword of message TLP.
	completer_id[15:0]	l	Completer ID field
	completion_status[2:0]	l	Completion status
	bcm	l	Byte count modified field
	byte_count[11:0]	l	Byte count field
	lower_address[6:0]	l	Lower address field.
	steering_tag[7:0]	l	Steering tag.
	payload_data (dynamic array)	l	Payload data, if any present.
	ei_code [31:0]	l	Error injection code associated with the TLP.
UpdateRxTLP  Called after the link receives the last byte of a TLP.	tlp_fmt[2:0]	l	Format field
	tlp_type[4:0]	l	Type field
	tc[2:0]	l	Traffic class
	th	l	Transaction hint
	td	l	TLP Digest bit
	ep	l	Error/Poison bit
	attr_id_order	l	ID Order attribute bit
	attr_relax_order	l	Relaxed order attribute bit
	attr_no_snoop	l	No snoop attribute attribute

**Table 14-5 Data Link Layer callbacks (Continued)**

	at[1:0]	l	address translation
	length[9:0]	l	length field
	ecrc[31:0]	l	ECRC(digest) value
	lcrc[31:0]	l	Link CRC value
	sequence_num (int)	l	Sequence number of the TLP
	requester_id[15:0]	l	Requester ID field
	tag[7:0]	l	Tag value
	first_dw_be[3:0]	l	First DW byte enable field.
	last_dw_be[3:0]	l	Last DW byte enable field.
	address[63:0]	l	Address field.
	ph[1:0]	l	Processing hint
	bus_num[7:0]	l	bus number
	device_num[2:0]	l	device number
	function_num[2:0]	l	function number
	register_num[9:0]	l	Combines reg and ext_reg field for config TLPs
	message_code[7:0]	l	Message code field
	message_dword2[31:0]	l	2 <sup>nd</sup> dword of message TLP. This would be used for vendor specific messages.
	message_dword3[31:0]	l	3 <sup>rd</sup> dword of message TLP.
	completer_id[15:0]	l	Completer ID field
	completion_status[2:0]	l	Completion status
	bcm	l	Byte count modified field
	byte_count[11:0]	l	Byte count field
	lower_address[6:0]	l	Lower address field.
	steering_tag[7:0]	l	Steering tag.
	payload_data (dynamic array)	l	Payload data, if any present.
UpdateTxlp Called every time a new packet starts transmission.	ipg(int)	l	Number of bytes between current packet and previously transmitted packet.
UpdateRxlp Called on the start of a new received packet.	ipg(int)	l	Number of bytes between current packet and previously received packet

**Table 14-5 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 14-6](#).

**Table 14-6** Covergroups, 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 14-6 Covergroups, 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_configuration_laneenum_wait	
	configuration_linkwidth_accept_to_detect_quiet	
	configuration_laneenum_accept_to_configuration_complete	



**Table 14-6 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_rcv_rlock	
	recovery_rcvrlock_to_recovery_equalization_phase0*	
	recovery_rcvrlock_to_recovery_equalization_phase1*	
	recovery_rcvrlock_to_recovery_rcv_rcfg	
	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 14-6 Covergoups, coverpoints and bins in the Physical Layer coverage class (Continued)**

		recovery_equalization_phase2_to_recovery_speed*	
		recovery_equalization_phase2_to_recovery_equalization_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_I0	
		recovery_idle_to_detect_quiet	
		recovery_idle_to_recovery_rcvrlock	
		I0_to_recovery_rcvrlock	
		I0_to_I1_entry	
		I1_entry_to_I1_idle	
		I1_entry_to_recovery_rcvrlock	
		I1_idle_to_I1_1_idle*	

**Table 14-6 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_active 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_entry_to_l0s_idle l0s_idle_to_l0s_fts l0s_fts_to_l0	L0s substate transitions for the transmit side
cg_rx_l0s_substate_transitions	cp_rx_l0s_substate_transitions	l0_to_l0s_entry l0s_entry_to_l0s_idle l0s_idle_to_l0s_fts l0s_fts_to_l0 l0s_fts_to_recovery_rcvrlock	L0s substate transitions for the receive side
*Exist for 8G models only			

## 14.6.2 Physical Layer Callbacks

Callbacks in the Physical Layer are defined in [Table 14-7](#).

**Table 14-7 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 14-8](#).

**Table 14-8 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 14-9](#).

**Table 14-9 PIPE callbacks**

Function Name	Arguments	I/O	Values
UpdateRate  Called every time the pipe signal rate changes.	rate [1:0]	I	Pipe rate value
UpdatePowerDown  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 14-9 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)	l	lane number to be updated
	tx_data[31:0]	l	transmit data
	tx_data_k[3:0]	l	transmit data control bits
	tx_compliance	l	transmit compliance
	tx_data_valid*	l	data_valid
	tx_start_block*	l	start block
	tx_sync_header*	l	transmit sync header
	tx_elec_idle	l	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]	l	receive data
	rx_data_k[3:0]	l	receive data control bits
	rx_status[1:0]	l	receive status
	rx_valid	l	receive data valid
	rx_elec_idle	l	receive electrical idle
	rx_data_valid*	l	receive data valid
	rx_start_block*	l	received start of a new block
	rx_sync_header*	l	value of sync header
	invert_rx_polarity	l	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:

- “Overview” on page 289
- “Configuration Classes” on page 292
- “Configuration Classes” on page 292
- “Signal Interfaces” on page 293
- “Data Factory Objects” on page 293
- “Exception List Factories” on page 293
- “Error injection” on page 293
- “Transactions” on page 293
- “Using the M-PCIe Interface” on page 295
- “Shared Status” on page 295
- “Configuring the PCIe M-PHY Adapter for an RMMI Interface” on page 296

The PCIe VIP implements the M-PHY interface through a `uvm_component` named `svt_pcie_mphy_adapter`. The PCIe VIP M-PHY Adapter component object extends from the `uvm_component` class.

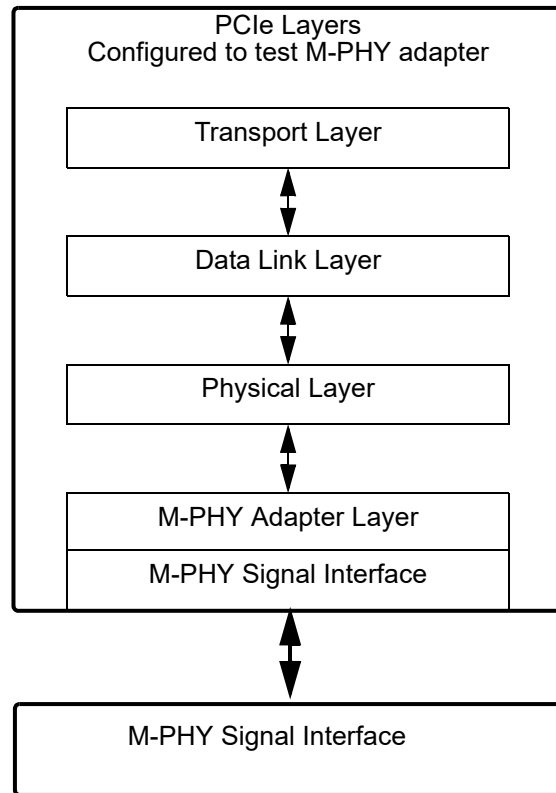
When an M-PCIe interface is being simulated, the M-PHY Adapter layer rather than the Physical layer models the interface to a DUT. The PCIe VIP M-PHY Adapter component provides the features defined in the M-PCIe ECN specification related to the PHY Adapter (PA) block and instances the necessary M-PHY VIP components.



#### Note

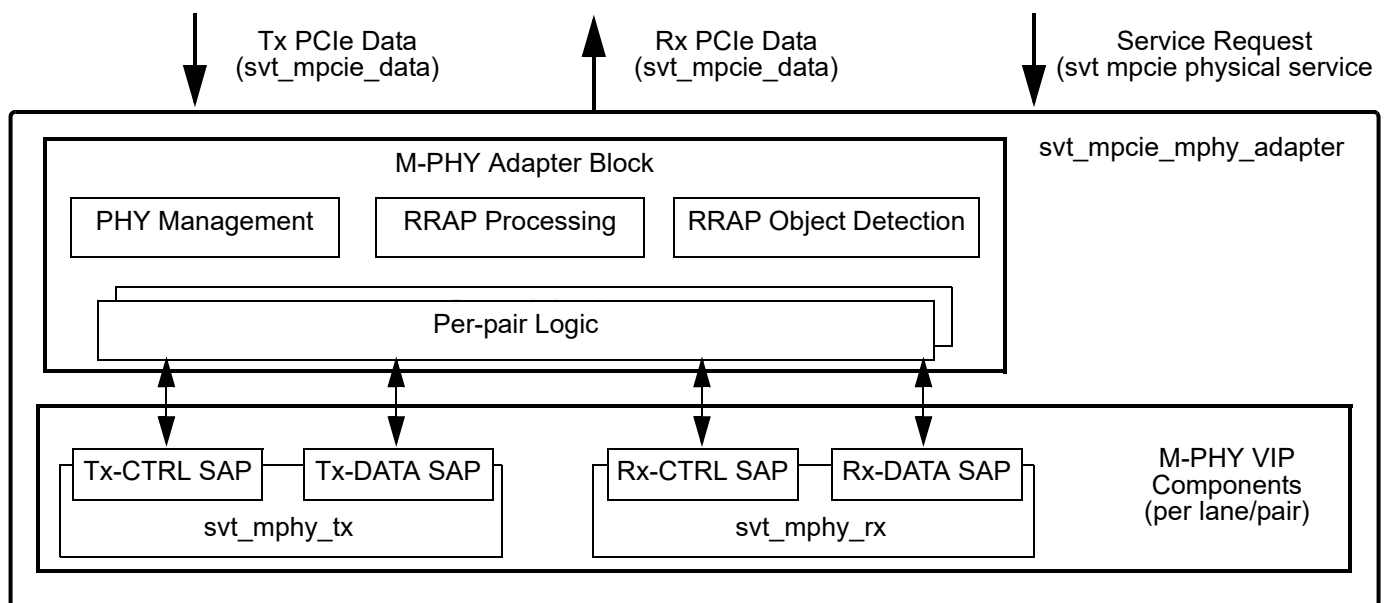
The VIP-MPCIE-OPT-SVT license key is required to run the M-PCIe VIP.

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 CTRL 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](#)" on page 294 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.

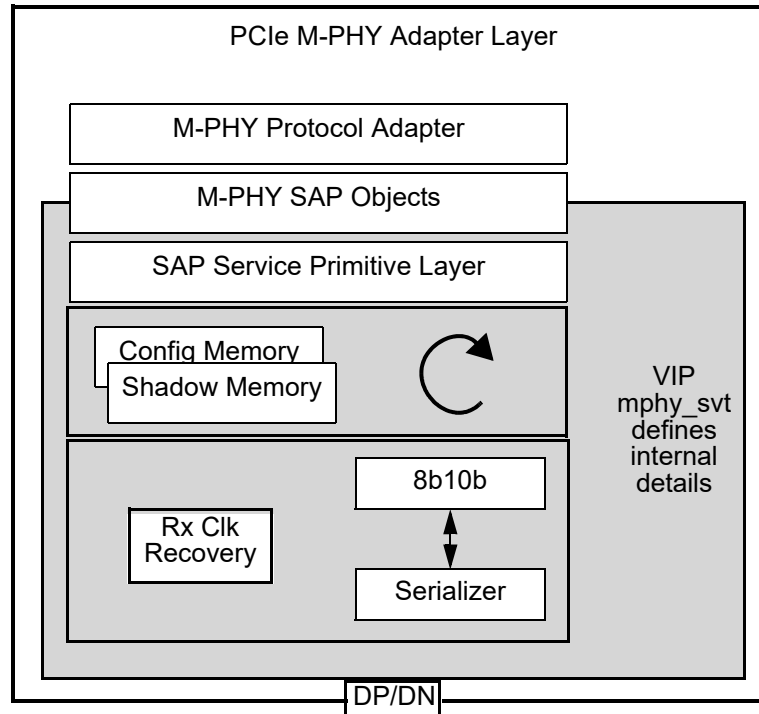


Figure 15-3 Overview of M-PHY Adapter Layer

## 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 the Hibern8 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_rmmtx_dut_phy_if`
- `svt_mphy_rmmrx_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_rmmtx_dut_controller_if`
- `svt_mphy_rmmrx_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 UVM, 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 reconfiguring.
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 is 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	<code>discovery_quick_configuration</code>
bit	<code>host_execution</code>
bit	<code>disable_correct_spec_rules</code>

Random VIP configuration values:

rand bit [7:0]	<code>c_refclk_tx_hs_g1_sync_length_control;</code>
rand bit [7:0]	<code>c_refclk_tx_hs_g2_sync_length_control;</code>
rand bit [7:0]	<code>c_refclk_tx_hs_g3_sync_length_control;</code>
rand bit [7:0]	<code>nc_refclk_tx_hs_g1_sync_length_control;</code>
rand bit [7:0]	<code>nc_refclk_tx_hs_g2_sync_length_control;</code>
rand bit [7:0]	<code>nc_refclk_tx_hs_g3_sync_length_control;</code>
rand bit [3:0]	<code>link_tx_hs_g1_prepare_length_control;</code>
rand bit [3:0]	<code>link_tx_hs_g2_prepare_length_control;</code>
rand bit [3:0]	<code>link_tx_hs_g3_prepare_length_control;</code>
rand bit [7:0]	<code>link_tx_hibern8_time;</code>
rand bit [3:0]	<code>link_tx_min_activation_time;</code>

The `mpcie_device_system_test_sequence_collection.sv::mpcie_device_system_rmmt_startup_sequence` adds control to enable the execution of the `svt_pcie_mphy_adapter_discovery_sequence` on both the root and endpoint agents.

bit	<code>discovery_quick_configuration</code>
-----	--

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 transaction, 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:

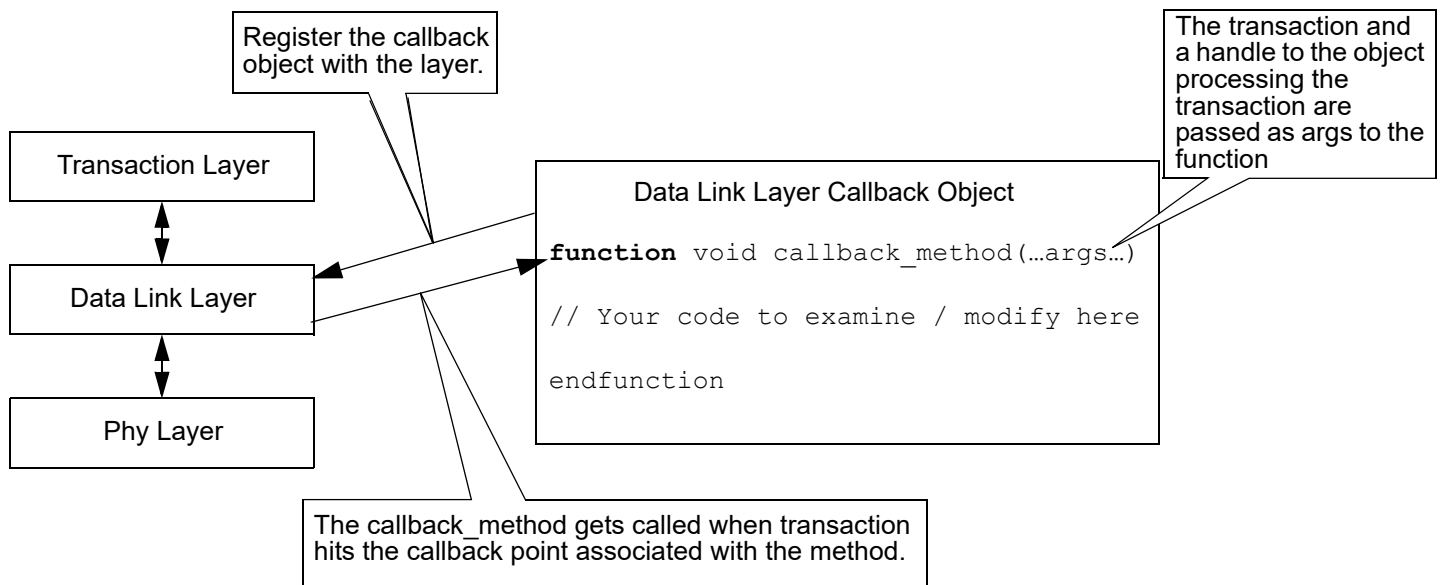
1. Identify which callback you need to use.
2. Sub-class the appropriate data type and implement the appropriate callback method with a user-defined action each time the callback is made.
3. Create an object of this type and add it to the queue of callback objects on the appropriate VIP instance.

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, UVM analysis ports are preferred for statistics, coverage, scoreboarding, and so on. However UVM 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 UVM 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
    `uvm_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",uvm_component parent=null);
    super.new(name,parent);
endfunction : new

// Configure/build various components:
virtual function void build_phase(uvm_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(uvm_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:
    uvm_callbacks#(svt_pcie_dl,svt_pcie_dl_callback)::add(env.endpoint.port.dl,
        dl_tlp_cb) ;
endfunction : end_of_elaboration_phase

task run_phase( uvm_phase phase );
    // In this case, there is not much to do in this phase
    `uvm_info(get_full_name(), "Running test dl_tlp_basic_callback_test .\n",
        UVM_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:
    `uvm_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);

`uvm_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), UVM_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();
uvm_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 `uvm_callbacks()` and `add()` to include that callback object in the list of callback objects. This, of course, is standard UVM. Some minor additions are available.

### 16.4.4.2 Methods to Modify the Transaction

There are three ways to modify 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 previously 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 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.

**Note**

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 `uvm_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:
  `uvm_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", uvm_component parent=null);
    super.new(name,parent);
  endfunction : new

  // build_phase: To build various components of class:
  virtual function void build_phase(uvm_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:
uvm_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();
uvm_report_cb::add(null, err_catcher);
endfunction : build_phase

function void end_of_elaboration_phase(uvm_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:
    uvm_callbacks#(svt_pcie_dl,svt_pcie_dl_callback)::add(env.endpoint.port.dl,
        dl_tlp_cb);
    `uvm_info(get_full_name(), "Exiting...", UVM_HIGH);
endfunction : end_of_elaboration_phase

task run_phase( uvm_phase phase );
    `uvm_info(get_full_name(), "Running test dl_tlp_example_callback .\n",
        UVM_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 sv_t_pcie_dl_callback;
    // Factory registration:
    `uvm_object_utils(tx_dl_tlp_callback)
    // `svt_uvm_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);
    `uvm_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),
            UVM_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 UVM_WARNING and UVM_ERROR messages:
class dl_tlp_err_catcher extends uvm_report_catcher;
    // Constructor:
    function new(string name="error catcher");
        super.new(name);
    endfunction

    // catch(): This function is where we handle the actual UVM 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() == UVM_ERROR) begin
            if ((uvm_is_match("*Received TLP with invalid seq num*", get_message()))
                begin
                    set_severity(UVM_INFO);
                end
            end
        else if (get_severity() == UVM_WARNING) begin
            if ((uvm_is_match("*Received TLP with bad LCRC*", get_message())) begin
                set_severity(UVM_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. <b>NOTE: Currently not implemented</b>
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. <b>NOTE: Currently not implemented</b>
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. <b>NOTE: Currently not implemented</b>
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. <b>NOTE: Currently not implemented</b>

**Table 16-1 Supported PCIe callbacks (Continued)**

pre_tx_tlp_put	svt_pcie_target_app target_app	I	The component object of the calling layer.
Target Application	svt_pcie_tlp transaction	I	The transaction to be examined/modified.
Called by the component just after scheduling a TLP transaction for transmission on the link, just prior to framing.	bit drop	ref	When set, the transaction is dropped prior to framing and is not transmitted.
post_rx_tlp_get	svt_pcie_target_app target_app	I	The component object of the calling layer.
Target Application	svt_pcie_tlp transaction	I	The transaction to be examined/modified.
Called by the component after receiving a TLP transaction from the link.	bit drop	ref	When set, the transaction is dropped without any further processing.
transaction_ended	svt_pcie_driver_app driver	I	The component object issuing the callback
Driver Application	svt_pcie_driver_app_transaction transaction	I	The transaction to be examined/modified.
Called by the component when the transaction is complete			
tx_ts_os_started	svt_pcie_pl pl	I	The component object issuing this callback.
Phy Layer	svt_pcie_os transaction	I	The transaction to be examined/modified.
Called by the component after building a TS OS transaction.			
tx_os_started	svt_pcie_pl pl	I	The component object issuing this callback.
Phy Layer	svt_pcie_os transaction	I	The transaction to be examined/modified.
Called by the component after building a non-TS OS Transaction			
pre_symbol_out_put	svt_pcie_pl pl	I	The component object issuing this callback.
Phy Layer	svt_pcie_symbol symbols[]	I	The array of symbols to be examined/modified.
Called by the component at every symbol time after gathering all the symbols to be transmitted on the link			

**Table 16-1 Supported PCIe callbacks (Continued)**

symbol_stripe_ended	svt_pcie_pl pl	1	The component object issuing this callback.
Phy Layer	svt_pcie_symbol_stripe symbol_stripe	1	The symbol stripe object to be examined.
Called by the component when the symbol stripe has been completed			

**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_tlp_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 t1 , svt_pcie_tlp tlp ,
        ref bit drop );
endfunction;
virtual function void post_seq_item_get(svt_pcie_tl t1, 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 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
<code>post_tlp_framed_in_get</code>	RX	Callback issued by the component after recognizing a TLP transaction received immediately from a link
<code>pre_tlp_framed_out_put</code>	TX	Callback issued by the component after scheduling a TLP transaction for transmission on the link, just prior to framing
<code>rx_dllp_started</code>	RX	Callback issued by the component after receiving User DLLP transaction from an input port
<code>tx_dllp_started</code>	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 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 12-4](#) 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
<code>post_tlp_framed_in_get</code>	<code>svt_pcie_dl_tlp</code>	<code>svt_pcie_dl_tlp_exception_list</code>	<code>svt_pcie_dl_tlp_exception</code>
<code>pre_tlp_framed_out_put</code>	<code>svt_pcie_dl_tlp</code>	<code>svt_pcie_dl_tlp_exception_list</code>	<code>svt_pcie_dl_tlp_exception</code>
<code>rx_dllp_started</code>	<code>svt_pcie_dllp</code>	<code>svt_pcie_dllp_exception_list</code>	<code>svt_pcie_dllp_exception</code>
<code>tx_dllp_started</code>	<code>svt_pcie_dllp</code>	<code>svt_pcie_dllp_exception_list</code>	<code>svt_pcie_dllp_exception</code>

## 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);
    virtual function void pre_pipe_data_out_put(svt_pcie_pl pl, svt_pcie_pipe_data
                                                    pipe_data);
    virtual function void post_pipe_data_in_get(svt_pcie_pl pl, svt_pcie_pipe_data
                                                    pipe_data);
endclass
```

**Table 16-6 Physical Layer Callbacks**

Callback	VIP Direction	Behavior
<code>pre_symbol_out_put</code>	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 multi-lane configures the symbols are striped per lane into <code>symbols[]</code> .
<code>tx_os_started</code>	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

**Table 16-6 Physical Layer Callbacks**

Callback	VIP Direction	Behavior
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
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.
pre_pipe_data_out_put	TX	<p>Callback issued by the component just before every <code>posedge pclk</code> after gathering all the <code>pipe_data</code> signals to be transmitted on the PCIe link. This is the last chance to force any pipe data signals before it goes on the wire.</p> <p>Note: MAC must be turned-off for this callback to work so that all the data is driven by the PIPE callback only. Service request <code>HOT_PLUG_UNPLUG</code> can be enabled to turn-off the MAC.</p>
post_pipe_data_in_get	RX	<p>Callback issued by the component after every <code>posedge</code> of <code>pipe_clk</code> after gathering all the pipe data signals that were received on the PIPE bus. Used solely to report values of the per-lane datapath signals received off the PIPE bus. Exceptions are not supported by this callback.</p> <p>Note: MAC must be turned-off as users are responsible for handling/demoting any errors that may occur if the MAC is turned on. Service request <code>HOT_PLUG_UNPLUG</code> can be enabled to turn-off the MAC.</p>

### 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
pre_pipe_data_out_put	svt_pcie_pipe_data_pipe	svt_pcie_pipe_data_exception_list	svt_pcie_pipe_data_exception
post_pipe_data_in_get	svt_pcie_pipe_data_pipe	Not Applicable	Not Applicable

## 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 Use Model

You can use the three new simulation targets in the Makefiles of the VIP UVM 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 UVM 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 UVM 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 The “vcspcvlog” Simulator Target in Makefiles

The `vcspcvlog` simulator target in the Makefiles of the VIP UVM 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
```

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.

## 17.3 The “vcsmxpcvlog” Simulator Target in Makefiles

The `vcsmxpcvlog` simulator target in the Makefiles of the VIP UVM 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
```

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.

## 17.4 The “vcsmxpipvlog” Simulator Target in Makefiles

The `vcsmxpipvlog` simulator target in the Makefiles of the VIP UVM 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.

## 17.5 Partition Compile and Precompiled IP Implementation in Testbenches with Verification IPs

You can use the Makefiles in the VIP UVM 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 `uvm_pkg` and `svt_uvm_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.

## 17.6 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/v1
```

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 DDR UVM Basic Example, use the following command:

```
$VC_VIP_HOME/bin/dw_vip_setup -e pcie_svt/tb_pcie_svt_uvm_basic_sys -svtb  
cd examples/sverilog/pcie_svt/tb_pcie_svt_uvm_basic_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

## Passive Monitor

---

The Synopsys PCIe Passive Monitor Verification IP supports verification of designs that include interfaces implementing the PCIe Base Specification version 3.0a. This document describes the use of PCIe Passive Monitor VIP in testbenches that comply with the SystemVerilog Universal Verification Methodology (UVM).

This approach leverages advanced verification technologies and tools that provide:

- ❖ Protocol functionality and abstraction
- ❖ Constrained random verification
- ❖ Functional coverage
- ❖ Rapid creation of complex tests
- ❖ 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

This document assumes that you are familiar with PCIe, object oriented programming, SystemVerilog, and UVM. For the Synopsys PCIe VIP class reference HTML documentation, see:

[\\$DESIGNWARE\\_HOME/vip/svt/pcie\\_svt/latest/doc/pcie\\_svt\\_uvm\\_public/html/index.html](#)

The PCIe Passive Monitor VIP is a standalone UVM-based verification component that can optionally be used in conjunction with the PCIe active component UVM VIP. Both are compatible for use with SystemVerilog-Compliant testbenches. The PCIe VIP active component simulates PCIe transactions through its active agent, while the passive component (monitor) receives transactions from both directions on the PCIe link.

## 18.1 Supported Protocol Features

The PCIe Passive Monitor VIP currently supports the following protocol features:

- ❖ PCIe Gen 1/2/3
- ❖ PIPE (up to version 4.2)
- ❖ Serial Interfaces
- ❖ One to 32 lanes
- ❖ Tracking of LTSSM states for devices on both sides of the link including Equalization and Low Power States
- ❖ Configuration Space Tracking
- ❖ Atomic Ops
- ❖ Callbacks at PL/DL/TL for Test Bench scoreboard support
- ❖ Mid Simulation Reset

## 18.2 Early Adopter (EA) Supported Features

The following features are at an Early Adopter level:

- ❖ Gen 4 (0.5 specification version)
- ❖ Functional Coverage

## 18.3 Features Not Supported

The following verification features are not supported:

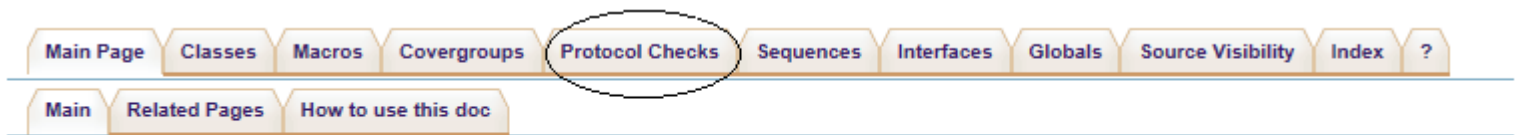
- ❖ Monitor issues `ltssm_advertised_16g_data_rate_check` protocol violation when active devices start 16G speed negotiation when 8G equalization stops at equalization phase 1. Issue will be fixed in next release.
- ❖ Monitor symbol log file displays blank columns for all those lanes which are not part of current active lanes. Issue will be fixed in next release.
- ❖ Passive Monitor VIP does not support OVM or VMM
- ❖ Loopback, Compliance Pattern, lane to lane skew, linkwidth upconfigure and downconfigure, lane reversal and reversed polarity are only partially tested.
- ❖ ECNs and features listed as optional in the spec other than those explicitly listed as supported in the User Guide are not supported.
- ❖ PHY DUT Monitoring.

## 18.4 UVM SVT Agent as Passive Monitor

Every UVM SVT agent can be made into a passive monitor. The PCIe agent follows UVM interface standards. The passive agent provides checking of the PCIe protocol at each of the transaction layers: Transaction Layer (TL), Data Link Layer (DL) and Physical Layer (PL) as well as checks specific to the LTSSM. To see a complete list of checks available go to:

[\\$DESIGNWARE\\_HOME/vip/svt/pcie\\_svt/latest/doc/pcie\\_svt\\_uvm\\_public/html/index.html](#)

And click on the 'Protocol Checks' tab.



## PCIe SVT UVM main page

In addition, the monitor provides a transaction log file and functional coverage for the LTSSM state transitions. Coverage information (PASS/FAIL) is also provided for each of the error checks.

The PCIe monitor supports a non-standard symbol log that can be enabled by setting the 'enable\_symbol\_logging' property in the svt\_pcie\_pl\_configuration class to '1'. The format of this symbol log mirrors the symbol log format defined by the PCIe active component and contains consolidated symbol lane information with additional annotations about synch headers in 128b/130b, and information about when the monitor detected speed changes, up-configure/down-sizing and lane reversal. Refer to “[Verification Features](#)” on page 37 for all log file formats.

The monitor only checks the device that is connected to Tx side.

Put a PCIe SVT UVM agent into passive mode with the following lines:

```
is_active = 0 ;  
monitor_enabled = 1;
```

NOTE. The PCIe Device agent cannot be used in the combined mode where active and passive features are supported in a single instance.

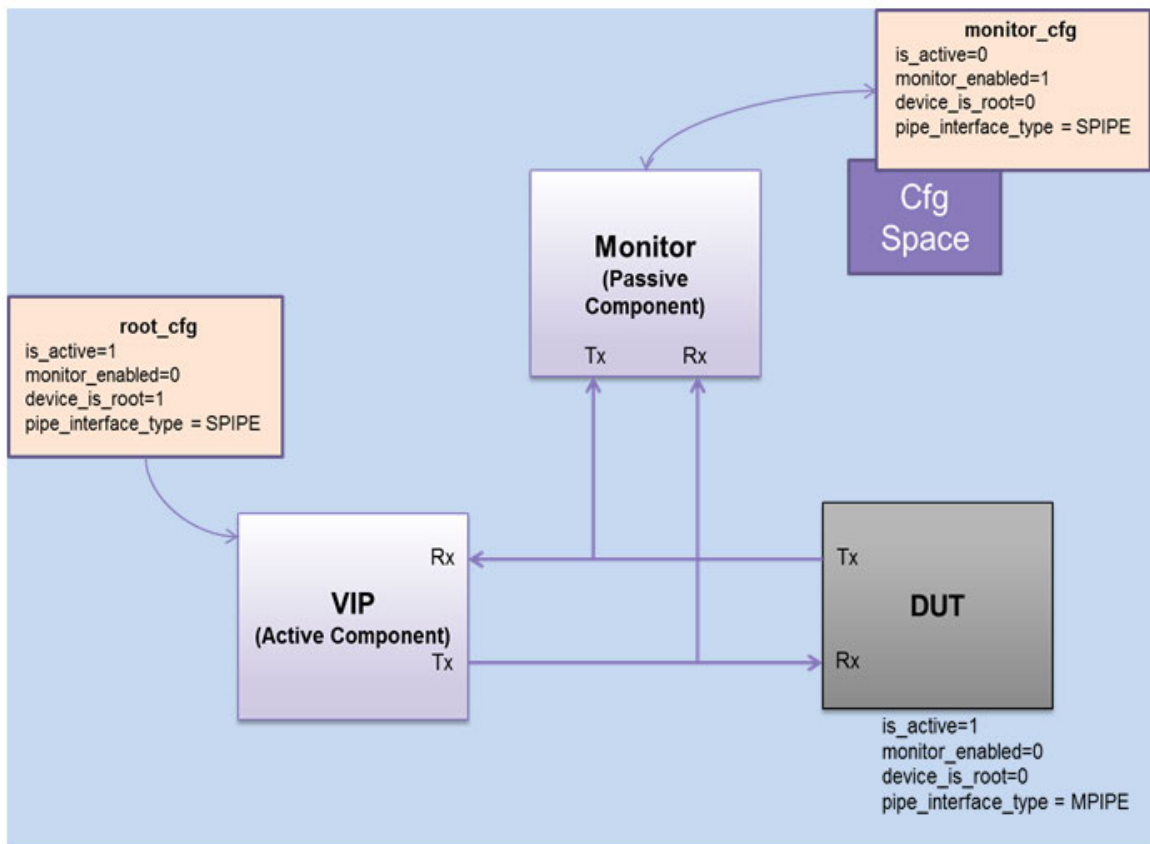
## 18.5 Methodology Features

The PCIe Passive Monitor VIP currently supports the following methodology functions:

- ❖ Standalone Agent instantiation
- ❖ Analysis ports at each protocol layer for connecting to scoreboard.
- ❖ Callbacks for LTSSM Coverage
- ❖ Transaction Log File

## 18.6 Usage

[Figure 18-1](#) on page 330 illustrates the connection and setup with a Root Complex and as a stand alone agent.

**Figure 18-1 Monitor as a Stand Alone Agent**

Note the following about [Figure 18-1](#):

- ❖ The Monitor as a standalone agent takes has its own config object as shown in the diagram above as 'monitor\_cfg'. This is an instance of `svt_pcie_configuration`, distinct from the instance used by the active component.
- ❖ The monitor must be programmed to match the position of the DUT or 'monitored device'. Specifically, the 'device\_is\_root' parameter of the `monitor_cfg` object should be set to '0' when the monitored device is an endpoint, and '1' when the monitored device is a root complex or downstream port of a switch device.
- ❖ The interface type of the monitor should match that of the monitored device. The Tx signals of the monitored device connect to the "Tx signals" of the passive monitor. The monitor does not drive these signals.
- ❖ The passive monitor does not generate nor derive any clocks. The testbench provides clocking to the passive monitor. Refer to the examples directory for how clocks are handled in the desired interface type.
- ❖ Configuration Space Tracking is provided for endpoint monitoring only (that is, when the monitored device is a PCIe endpoint device).



## 18.7 Types of Monitor Configuration Objects

Configuration data objects convey the system level and port level testbench configuration. The configuration of agents is done in the build() phase of environment or the testcase. If the configuration needs to be changed later, it can be done through reconfigure() method/

The configuration object properties can be of two types:

- ❖ Static configuration properties. Static configuration parameters specify configuration which cannot be changed when the system is running.
- ❖ Dynamic configuration properties. Dynamic configuration parameters specify configuration which can be changed at any time, irrespective of whether the system is running or not. Example of dynamic configuration parameter is timeout values.

## 18.8 Functional Coverage

### 18.8.1 LTSSM State Coverage

Coverage of LTSSM state transitions as tracked by the monitor is provided in the svt\_pcie\_ltssm\_state\_cov class. See the provided HTML documentation for further details.

### 18.8.2 Error Check Coverage

Each error check has PASS/FAIL coverage available. For example, for the check TXN\_02\_00\_02 coverage is available via the svt\_err\_check\_stats\_cov\_txn\_02\_00\_02 class. See the provided HTML documentation for further details.

### 18.8.3 Transaction Coverage

Transaction coverage is not yet provided.

### 18.8.4 Coverage Callback Classes

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

The coverage data callback class is extended from respective callback class. Examples listed below.

- ❖ The class svt\_pcie\_ltssm\_def\_cov\_data\_callback is extended from svt\_pcie\_ltssm\_callback.
- ❖ The class svt\_pcie\_tl\_monitor\_def\_cov\_data\_callback is extended from svt\_pcie\_tl\_monitor\_callback.
- ❖ The class svt\_pcie\_dl\_monitor\_def\_cov\_data\_callback is extended from svt\_pcie\_dl\_monitor\_callback.
- ❖ The class svt\_pcie\_pl\_monitor\_def\_cov\_data\_callback is extended from svt\_pcie\_pl\_monitor\_callback.

Various methods are implemented for triggering coverage events.

### 18.8.5 Enabling Default Coverage

The default functional coverage can be enabled by setting the following attributes in the port configuration class `svt_PCIE_port_configuration` to '1'. To disable coverage, set the attributes to '0'. The attributes are:

- ❖ `toggle_coverage_enable`
- ❖ `state_coverage_enable`
- ❖ `transaction_coverage_enable`

Note: By default, the coverage is disabled.

### 18.8.6 Coverage Shaping and Control

You provide a handle to the port configuration class `svt_pcie_port_configuration` in the class `svt_pcie_port_monitor_def_cov_callback`, which implements the default cover groups. Based on the port configuration, the coverage bins are shaped. The unwanted bins are ignored.

In addition, you also have ability to disable coverage at a cover group level. Class `svt_pcie_port_configuration` provides members `svt_pcie_port_configuration::<cover_group_name>_enable`, to enable/disable cover groups. By default, the value to these members is 1.

## 18.9 Interfaces and modports

SystemVerilog models signal connections using interfaces and modports. Interfaces define the set of signals which make up a port connection. Modports define collection of signals for a given port, the direction of the signals, and the clock with respect to which these signals are driven and sampled.

List of all the monitor interfaces:

- ❖ `svt_pcie_mpipe_x32_8g_if.svi`
- ❖ `svt_pcie_mpipe_x8_8g_if.svi`
- ❖ `svt_pcie_mpipe_x16_8g_if.svi`
- ❖ `svt_pcie_mpipe_x4_if.svi`
- ❖ `svt_pcie_spipe_x16_8g_if.svi`
- ❖ `svt_pcie_spipe_x32_8g_if.svi`
- ❖ `svt_pcie_spipe_x4_if.svi`
- ❖ `svt_pcie_spipe_x8_8g_if.svi`
- ❖ `svt_pcie_serdes_x4_if.svi`
- ❖ `svt_pcie_serdes_x8_if.svi`
- ❖ `svt_pcie_serdes_x32_if.svi`
- ❖ `svt_pcie_serdes_x16_if.svi`

For link widths higher than x4 use the x32 interface and connect only the required lanes.

For PIPE instantiations, DUTs might implement certain PIPE defined signals as either per-lane or as a single wire, fanned out to all lanes. The testbench can be modified to accommodate differences between the DUT and the VIP pertaining to how these signals are implemented.

The following table shows how the PCIe VIP has implemented such signals:

**Table 18-1 Passive Monitor Signals**

Signal	PIPE Spec Version 4.0	Active Component	Monitor
CLK	Shared	Shared	
Max PCLK	Shared	Shared	Shared
EncodeDecodeBypass	Optional - shared or on per-lane basis	Not implemented.	Not implemented.
BlockAlignControl	Optional - shared or on per-lane basis	Shared	Shared
FS	Optional - shared or on per-lane basis	Shared	Per-lane basis
LF	Optional - shared or on per-lane basis	Shared	Per-lane basis
TxSwing	Optional - shared or on per-lane basis	Shared	Shared
TxMargin	Optional - shared or on per-lane basis	Shared	Shared
TxDetectRx/Loopback	Optional - shared or on per-lane basis	Shared	Shared
Rate	Optional - shared or on per-lane basis	Shared	Shared
Width	Optional - shared or on per-lane basis	Shared	Shared
PCLK rate	Optional - shared or on per-lane basis	Shared	Shared
Reset#	Optional - shared or on per-lane basis	Shared	Shared
TxDataValid	Optional - shared or on per-lane basis	Per-lane basis	Per-lane basis
PCLK	Optional - shared or on per-lane basis	Shared	Shared
Txdata, txdatak	Per-lane basis	Per-lane basis	Per-lane basis
Rxdata, rxdatak	Per-lane basis	Per-lane basis	Per-lane basis
TxStartBlock	Per-lane basis	Per-lane basis	Per-lane basis
TxElecIdle	Per-lane basis	Per-lane basis	Per-lane basis
TxCompliance	Per-lane basis	Per-lane basis	Per-lane basis

**Table 18-1 Passive Monitor Signals (Continued)**

Signal	PIPE Spec Version 4.0	Active Component	Monitor
RxPolarity	Per-lane basis	Per-lane basis	Per-lane basis
RxValid	Per-lane basis	Per-lane basis	Per-lane basis
RxElecIdle	Per-lane basis	Per-lane basis	Per-lane basis
RxStatus	Per-lane basis	Per-lane basis	Per-lane basis
RxDataValid	Per-lane basis	Per-lane basis	Per-lane basis
RxStartBlock	Per-lane basis	Per-lane basis	Per-lane basis
TxDemph	Per-lane basis	Per-lane basis	Per-lane basis
PowerDown	Per-lane basis	Shared	Shared
PhyStatus	Per-lane basis	Shared	Shared
RxPresetHint	Per-lane basis	Per-lane basis	Per-lane basis
RxEqEval	Per-lane basis	Per-lane basis	Per-lane basis
LinkEvaluationFeedbackFigureMerit	Per-lane basis	Per-lane basis	Per-lane basis
LinkEvaluationFeedbackDirectionChange	Per-lane basis	Per-lane basis	Per-lane basis
InvalidRequest	Per-lane basis	Per-lane basis	Per-lane basis
TxSyncHeader	Per-lane basis	Per-lane basis	Per-lane basis
RxSyncHeader	Per-lane basis	Per-lane basis	Per-lane basis
Local_preset_index	Per-lane basis	Per-lane basis	Per-lane basis
Get_local_preset_coefficients	Per-lane basis	Per-lane basis	Per-lane basis
RxStandby	Per-lane basis	Shared	Shared
RxStandbyStatus	Per-lane basis	Shared	Shared
Local_fs	Per-lane basis	Per-lane basis	Per-lane basis
Local_if	Per-lane basis	Per-lane basis	Per-lane basis
Local_tx_coefficients_valid	Per-lane basis	Per-lane basis	Per-lane basis
Local_tx_preset_coefficients	Per-lane basis	Per-lane basis	Per-lane basis

## 18.10 Programming the Passive Monitor

This section gives you information on the various classes and members for programming the Passive Monitor.

### 18.10.1 Synopsys Passive Monitor Example

Synopsys has provided an example. It is located at:

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

The example shows the following:

- ❖ A top-level testbench in SystemVerilog
- ❖ A dummy EP DUT in the testbench, -
- ❖ A UVM verification environment
- ❖ Directed and random transaction generation.
- ❖ Programming of the Monitor's config space via the backdoor.

The README provides information about the structure of the example.

Below are the steps for installing and running the example `tb_pcie_svt_uvm_monitor_basic_sys`.

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_uvm_monitor_basic_sys -svtb
```

This installs the example under:

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

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

- a. Use the Makefile:

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

```
gmake USE_SIMULATOR=vcsvlog base_pipe_test WAVES=1
```

To see more options and tests you can run, invoke "gmake help".

- b. Use the sim script:

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

```
./run_pcie_svt_uvm_monitor_basic_sys -w base_pipe_test vcsvlog
```

To see more options and tests you can run, invoke " ./run\_pcie\_svt\_uvm\_monitor\_basic\_sys - help".

### 18.10.2 Configuring the Monitor in the Example Testbench

The configuration data objects contain built-in constraints, which come into effect when the configuration objects are randomized.

The PCIe VIP example testbench defines following configuration class:

- ❖ Shared Cfg. The `pcie_shared_cfg` class contains configuration information that is used by both the active and passive sides. Since the Passive part can only be instantiated as a standalone agent the test must provide the monitor with a unique instance of this class. In the provided examples this instance is called `cfg_passive`:

```
pcie_shared_cfg cfg_passive;
```

The monitor makes special use of the Cfg Space Tracking sub-class of the `pcie_shared_cfg` class. The monitor maintains this as a mirror of the Configuration Space of a monitored endpoint device. This space can be programmed via an enumeration sequence on the link or by the use an Analysis Port provided on the monitor.

### 18.10.3 Callbacks

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

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

#### 18.10.3.1 Callbacks in the PCIe Passive Agent

Please refer to class reference HTML documentation for details of these classes. Refer to:

[\\$DESIGNWARE\\_HOME/vip/svt/pcie\\_svt/latest/doc/pcie\\_svt\\_uvm\\_public/html/index.html](#)

## 18.11 Configuration Space Tracking

The Configuration Space Tracking feature of the Passive Monitor purpose is to maintain a copy of the (Type 0) Configuration Space of the monitored device when the monitored device is a PCIe Endpoint. When the monitored device is a Root Complex or Switch downstream port, then the Passive Monitor maintains a Type 0 Configuration Space for programming purposes.

For the purpose of programming protocol checks, the Passive Monitor relies on values in its copy of the Configuration Space whenever possible. Note that there are many checks in the Passive Monitor which have parameters that are not covered by the PCIe Configuration Space; in these cases the Passive Monitor uses the configuration object instead.

When the monitored device is a PCIe Endpoint, the Configuration Space maintained by the PCIe Passive Monitor will be a copy of the Configuration Space in the DUT itself. The Passive Monitor's copy of the Configuration Space can be maintained completely by 'back door' accesses from the test bench or by a combination of back door accesses and by updates that the Passive Monitor performs by observing Configuration Read and Write transactions on the link, including those that occur as part of a discovery and enumeration process. For test benches that bypass enumeration it is typical to simply update the Passive Monitor's Configuration Space by back door accesses only.

When the monitored device is an RC or switch downstream port, then the monitor still maintains a Type 0 Configuration Space in order to program certain checks. In this instance, the Configuration Space is programmed through back door accesses only.

The Passive monitor currently implements a partial list of capabilities listed in PCI Express Specification. 'Implementing a capability set' means the passive monitor recognizes the capability ID, register list for the capability and fields for all the capability registers.



### Attention

The Passive Monitor does not support all the capability structures that it can recognize. For example, the PCI Express Root Complex Link Declaration Capability can be established and loaded through enumeration or through backdoor accesses, but the passive monitor does not have any checks or specific support for that capability.

### 18.11.1 Configuration Space Features

The following lists PCI Configuration Space Capabilities:

- ❖ Power management capability set
- ❖ MSI capability set
- ❖ MSI-X capability set
- ❖ PCI Express Capability set
- ❖ PCI Express Extended Capabilities
- ❖ Advanced Error Reporting Capability
- ❖ Virtual Channel Capability
- ❖ Multi-Function Virtual Channel Capability
- ❖ Device Serial Number Capability
- ❖ Power Budgeting Capability
- ❖ Vendor-Specific Capability

The following lists the ACS Extended Capability set for the Passive Monitor:

- ❖ Advanced Error Reporting Capability
- ❖ Virtual Channel Capability
- ❖ Multi-Function Virtual Channel Capability
- ❖ Device Serial Number Capability
- ❖ Power Budgeting Capability
- ❖ Vendor-Specific Capability
- ❖ ACS Extended Capability
- ❖ ARI Capability
- ❖ ATS (Address Translation Services) Capability
- ❖ Page Request Capability
- ❖ SR-IOV Capability
- ❖ Latency Tolerance Reporting (LTR) Capability
- ❖ PASID Extended Capability
- ❖ L1 PM Substates Extended Capability
- ❖ PCI Express Root Complex Link Declaration Capability
- ❖ PCI Express Root Complex Internal Link Control Capability
- ❖ RCRB Header Capability

### 18.11.2 Programming Passive Monitor's Configuration Space

Synopsys strongly recommends that the Passive Monitor Configuration Space be programmed to correspond to the monitored device (DUT). In the case of an RC DUT, the Passive Monitor Configuration Space is still utilized for certain checks.

#### 18.11.2.1 Understanding the Configuration Space Capabilities List (Linked-List) Structure

While the Passive Monitor can track link activity for the purpose of programming and maintaining its copy of an End Point DUT's configuration space (see section 2.2 Configuration Space Modes), you are still required to program the Configuration Space with at least the linked list structure of Configuration Space capabilities for all functions to be used during the simulation. This is required so that the Passive Monitor can perform checks on configuration space programming during the simulation.

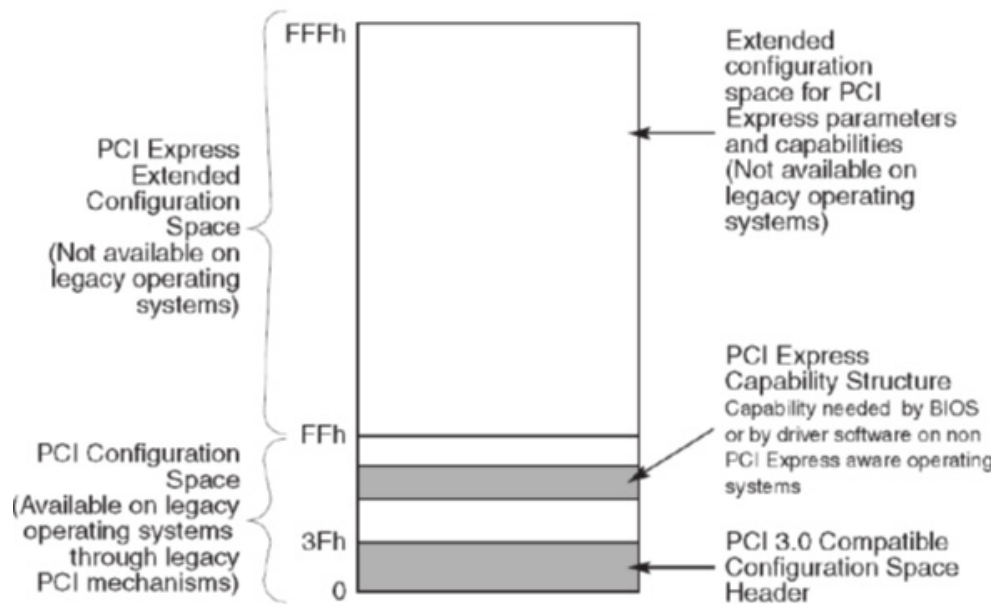
This section briefly describes configuration space capabilities lists. See the PCIe base specification for more information. Start with the following:

- ❖ A capability is set of registers define in PCI, PCI express or PCI Express ECN specification. Every capability has a unique ID to identify itself in enumeration process.
- ❖ Certain parts of the PCI Express configuration space apply to the entire device and certain parts are repeated on a per-function basis.
- ❖ The Configuration Space for each function is comprised of the following parts:
  - ◆ PCI Compatible Configuration Space header
  - ◆ Legacy and PCI Express Capability
  - ◆ PCI Express Extended Configuration Space



Figure 18-2 shows the layout of Configuration Space for one function.

**Figure 18-2 PCI Express Configuration Space Layout**

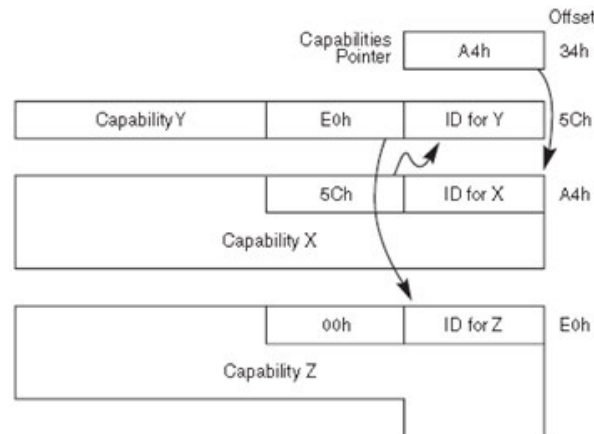


All capabilities will have an ID and pointer to start of next capability in the list (Next Capability pointer/offset field). The last capability in the list will have all zeros in the Next Capability pointer/offset field, indicating there are no more capabilities to be discovered.

In PCI Express configuration space, there are two linked-lists for every function:

- ❖ PCI Configuration space linked list starting from offset address 16'h0000 to 16'h00FF. For Function 0 PCI Configuration space linked list start from 16'h0000 to 16'h00FF, for Function 1 Configuration space linked list start from 16'h1000 to 16'h10FF, similarly for other functions.
- ❖ PCI Express Extended Configuration Space linked list starting from offset 16'h0100 to 16'h0FFF for Function 0.

Refer to Figure 18-3 for an example of capability list organization in PCI Configuration Space region and for PCI Express Extended Configuration Space region.

**Figure 18-3 Example PCI Capabilities list**

### 18.11.3 Passive Monitor Configuration Space Modes

You can program passive monitor's configuration space in one of the three modes:

- ❖ CFG\_SPACE\_DISABLED
- ❖ CFG\_SPACE\_BACKDOOR\_UPDATE
- ❖ CFG\_SPACE\_ENUMERATION\_UPDATE

#### 18.11.3.1 CFG\_SPACE\_DISABLED mode

For passive monitor CFG\_SPACE\_DISABLED is the default mode. In this mode configuration space programming is not needed, which also implies that monitor does not have a copy of the DUT's configuration space registers. This mode will affect the checks that rely on configuration space register values. Some checks will substitute a value from the `svt_pcie_configuration` class, some checks will use only the default value from specification and a few checks will be disabled altogether. While this is not the preferred mode of operation, it might be useful during initial bring-up of the Passive Monitor.

#### 18.11.3.2 2.2.2 CFG\_SPACE\_BACKDOOR\_UPDATE

In CFG\_SPACE\_BACKDOOR\_UPDATE mode, the test bench programs the complete capability list structure of the DUT's configuration space and all the registers and fields which differ from their default value (as defined in the PCIe Base Specification). This programming occurs through backdoor methods before link activity begins. In this mode, configuration write or read transactions occurring on the link will not cause updates to the Passive Monitor's copy of the Configuration Space.

#### 18.11.3.3 CFG\_SPACE\_ENUMERATION\_UPDATE

In CFG\_SPACE\_ENUMERATION\_UPDATE mode the Passive Monitor updates its copy of the Configuration Space based on configuration write & read transactions that occur on the link (ie, the discovery and enumeration process). As mentioned above the test bench must program the complete capability list structure the DUT's configuration space and all the registers and fields which differ from default value (as defined in the PCIe Base Specification) through backdoor methods before link activity begins. Once the link is up, the Passive Monitor will update its copy of the Configuration Space based upon the observation of successful configuration write or read transactions that occur on the link.

#### 18.11.4 Test Bench access to PCIe Passive Monitor Configuration Space.

Access to the Passive Monitor's copy of the Configuration Space is through a transaction layer service request. Specifically the `svt_pcie_tl_service` object is passed to the TL level analysis port.

**Table 18-2 List of Configuration Space Accessing Service Requests**

Service Request	Description
MON_CONFIG_SPACE_WRITE_ADDR	Used to set a particular value at the specified address except for the capability set mappings. Once established, capability set mappings are not allowed to change. There is no other address validity checking.
MON_CONFIG_SPACE_READ_ADDR	Used to retrieve a particular value from the specified address.
MON_CONFIG_SPACE_SET_BAR_RO_MAP	Used to set the size of BAR through backdoor.
MON_CONFIG_SPACE_GET_BAR_RO_MAP	Used to retrieve size of BAR.
MON_CONFIG_SPACE_PRINT	Used to print the contents of register space in transcript when called.
MON_CONFIG_SPACE_DUMP	Used to dump the contents of register space to the file specified. The file is created if the write mode is used, else the contents are appended to the existing file when append mode is used.
MON_CONFIG_SPACE_SET_FIELD	Used to set a particular value to a register field, it provides the backdoor access to the register fields.
MON_CONFIG_SPACE_GET_FIELD	Used to retrieve the value of a particular register field of a function number from the monitor's image of the configuration space.

Note : The `svt_pcie_tl_service` object is used by both the PCIe active and passive agents, not all the service type and fields are used by the passive monitor. Refer to the `svt_pcie_tl_service` class documentation (html docs) for full details.

NOTE: When the Passive Monitor receives a `MON_CONFIG_SPACE_SET_FIELD` or `MON_CONFIG_SPACE_GET_FIELD` service request, the Passive Monitor will check whether the complete configuration space capability list structure is programed. If the complete configuration space capability list is not set or incorrectly set the monitor will issue an error such as one of the following:

```
UVM_ERROR /<path>/vip/pcie_svt/src/svt_pcie_config_space_status.sv(1930) @ 205000.00 ps:
reporter [set_capability_base_addr] Configuration space detected unsupported PCIE
Capability set 00 while mapping the capability sets for function 0. This may result in
unexpected behavior.
```

```
UVM_ERROR /<path>/vip/pcie_svt/src/svt_pcie_config_space_status.sv(2467) @ 16071734.80
ps: reporter [manipulate_addr] Configuration space is trying to access unmapped
capability set SVT_PCIE_CONFIG_SPACE_PCI_EXPRESS_CAP for function 0. Hence the field
ltr_mechanism_en is unsupported. This may result in unexpected behavior.
```

```
UVM_ERROR /<path>/vip/pcie_svt/src/svt_pcie_config_space_status.sv(1675) @ 16071734.80
ps: reporter [get_reg_field] Trying to read unmapped register field ltr_mechanism_en of
function 0 configuration space.
```

NOTE: The first service request call with `MON_CONFIG_SPACE_SET_FIELD`, `MON_CONFIG_SPACE_GET_FIELD`, `MON_CONFIG_SPACE_PRINT` or `MON_CONFIG_SPACE_DUMP` will initialize configuration space. That is, the Passive Monitor will check the validity of the linked list structure that has been programmed, check that the capability ID's are from the list of recognized capabilities and create internal objects for all the capability registers & their corresponding fields. At this time the Passive Monitor will establish initial values for all the registers defined by the link list that has been established. There are two possible initial values for each register. If the testbench has written to the register address for a particular register (using the `MON_CONFIG_SPACE_WRITE_ADDR` service request), the monitor will use the provided value as the initial value for the register. Else the register will be initialized with the default value as per the PCI Express specification.

Once the Passive Monitor's copy of the Configuration Space is initialized, Read-Only fields cannot be changed. This means that no more capabilities can be added. The test bench should not call `MON_CONFIG_SPACE_SET_FIELD`, `MON_CONFIG_SPACE_GET_FIELD`, `MON_CONFIG_SPACE_PRINT` and `MON_CONFIG_SPACE_DUMP` service request before the full capability list is set.

The below table lists `svt_pcie_tl_service` fields relevant to Passive Monitor Configuration Space service requests.

**Table 18-3 svt\_pcie\_tl\_service Fields relevant to Configuration Space Access Service Requests**

Service Request	Relevant field	Description
MON_CONFIG_SPACE_WRITE_ADDR	mon_cfg_space_ecam_addr	PCIe ECAM register address
	mon_cfg_space_bit_mask	mask for write Dword
	mon_cfg_space_dword_data	Dword to update the register with
MON_CONFIG_SPACE_READ_ADDR	mon_cfg_space_ecam_addr	PCIe ECAM register address
	mon_cfg_space_bit_mask	mask for write Dword
	mon_cfg_space_dword_data	Dword read from register anded with mask
MON_CONFIG_SPACE_SET_BAR_RO_MAP	mon_cfg_space_bdf_num	Function number. For setting size of BAR only mon_cfg_space_bdf_num [7:0] is used as function number only.
	mon_cfg_space_bar_num	BAR number
	mon_cfg_space_dword_data	Size of BAR. Write 1'b1 for every read-only bit for BAR, i.e. 32'h0000_00FF for 256 byte BAR size.

**Table 18-3 svt\_pcie\_tl\_service Fields relevant to Configuration Space Access Service Requests (Continued)**

Service Request	Relevant field	Description
MON_CONFIG_SPACE_GET_BAR_RO_MAP	mon_cfg_space_bdf_num	Function number. For setting size of BAR only mon_cfg_space_bdf_num[7:0] is used as function number only.
	mon_cfg_space_bar_num	BAR number
	mon_cfg_space_dword_data	Return the size of the BAR.
MON_CONFIG_SPACE_PRINT		Prints the contents of configuration space in transcript with called.
MON_CONFIG_SPACE_DUMP	mon_cfg_space_dump_filename	File name for dump configuration space content
	mon_cfg_space_dump_file_mode	Indicates the mode in which to open the file for dumping configuration space. The argument can either be "w" for write or "a" for append mode.
MON_CONFIG_SPACE_SET_FIELD	mon_cfg_space_bdf_num	Function number for which the field has to be set.
	mon_cfg_space_fld_id	Field ID
	mon_cfg_space_dword_data	value that needs to be set into the field
MON_CONFIG_SPACE_GET_FIELD	mon_cfg_space_bdf_num	Function number from which the field has to be retrieved.
	mon_cfg_space_fld_id	Field ID
	mon_cfg_space_dword_data	Value of retrieved field value

For the MON\_CONFIG\_SPACE\_SET\_FIELD and MON\_CONFIG\_SPACE\_GET\_FIELD service requests, the argument mon\_cfg\_space\_fld\_id value uniquely identifies the field in the configuration space (for the function number identified by the value in the mon\_cfg\_space\_bdf\_num parameter). The mon\_cfg\_space\_fld\_id value in conjunction with function number is used internally by the monitor to calculate the register address of the register and bit position of the fields.

### 18.11.5 Example Programming Monitor Configuration Space

For both the CFG\_SPACE\_BACKDOOR\_UPDATE and CFG\_SPACE\_ENUMERATION\_UPDATE modes, the test bench programs the complete capability list structure of the DUT's configuration space and all the register & fields that differ from default value, through backdoor methods before link activity begins.

For example, consider the case where the passive monitor is to be programmed corresponding to capability structures shown in [Figure 18-4](#).

NOTE: To the below display show in Figure 4 was created through MON\_CONFIG\_SPACE\_DUMP service request.

Figure 18-4 Example Capability Structures

REGISTER NAME	ADDRESS	VALUE
Function 0 configuration space :		
*****		
PCI 3.0 Compatible Registers :		
vendor_device	0x00000000	0x00000000
command_status	0x00000004	0x00100007
rev_id_class_code	0x00000008	0x00000000
cache_lat_hdr_bist	0x0000000c	0x00000000
base_addr_reg_0	0x00000010	0x10000008
base_addr_reg_1	0x00000014	0x20000008
base_addr_reg_2	0x00000018	0x30000003
base_addr_reg_3	0x0000001c	0x00000003
base_addr_reg_4	0x00000020	0x40000000
base_addr_reg_5	0x00000024	0x50000001
cardbus_cis_ptr_reg	0x00000028	0x00000000
subsystem_id_reg	0x0000002c	0x00000000
exp_rom_base_addr_reg	0x00000030	0x00000000
capability_ptr_reg	0x00000034	0x00000040
intr_line_reg	0x0000003c	0x00000000
PCI Power Management Capability Set Registers :		
pme_cap_reg	0x00000040	0x00025001
pme_cntrl_stat_reg	0x00000044	0x00000000
Message Signaled Interrupt Capability Set Registers :		
msi_cap_reg	0x00000050	0x00006005
msi_msg_addr_reg_0	0x00000054	0x00000000
msi_msg_data_reg	0x00000058	0x00000000
PCI Express Capability Set Registers :		
pcie_cap_reg	0x00000080	0x00020010
dev_cap_reg	0x00000084	0x00000000
dev_cntrl_stat_reg	0x00000088	0x000058b0
link_cap_reg	0x0000008c	0x00000011
link_cntrl_stat_reg	0x00000090	0x00010000
dev_cap_2_reg	0x000000a4	0x00000000
dev_cntrl_stat_2_reg	0x000000a8	0x00000000
link_cap_2_reg	0x000000ac	0x00000002
link_cntrl_stat_2_reg	0x000000b0	0x00000001
SVT_PCIE_CONFIG_SPACE_MSIX Capability Set Registers :		
msix_control_reg	0x00000060	0x00018011
msix_table_bir_reg	0x00000064	0x00000000
msix_pba_bir_reg	0x00000068	0x00000000
Virtual Channel Extended Capability Set Registers :		
virtual_channel_cap_reg	0x00000100	0x00010002
port_vc_cap_reg_1	0x00000104	0x00000000
port_vc_cap_reg_2	0x00000108	0x1c000000
port_vc_cntrl_stat_reg	0x0000010c	0x00000000
vc_resource_ctrl_reg_0	0x00000114	0x800200ff
vc_resource_stat_reg_0	0x00000118	0x00000000

To create the capability list show in Figure 18-4, the test bench needs only to write into the six registers highlighted in red through a MON\_CONFIG\_SPACE\_WRITE\_ADDR service request. In addition, after programming the capability list the test bench will need to set Bus Master Enable, Memory Space Enable and IO Space Enable through either MON\_CONFIG\_SPACE\_WRITE\_ADDR or MON\_CONFIG\_SPACE\_SET\_FIELD service request.

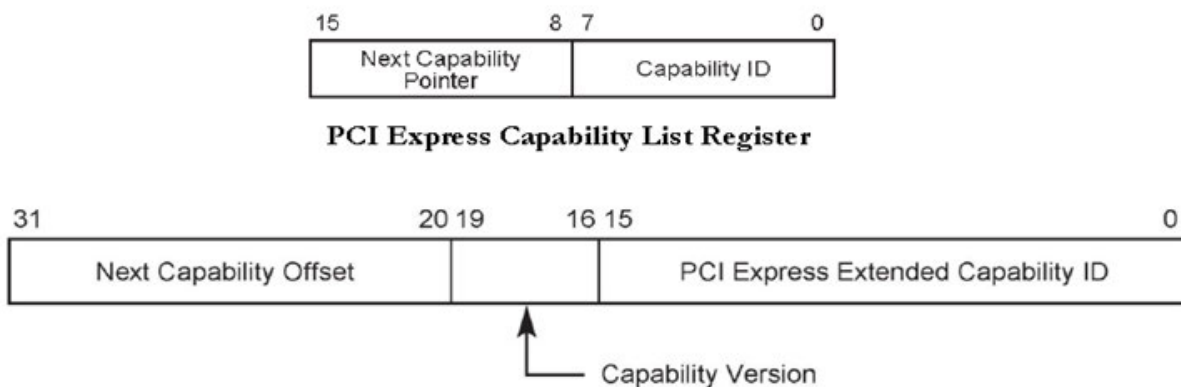
NOTE: Using MON\_CONFIG\_SPACE\_WRITE\_ADDR rather than MON\_CONFIG\_SPACE\_SET\_FIELD is generally easier when the testbench has a full, address based, copy of the DUT configuration space to work with. It still might be necessary however, to translate the DUT's absolute addresses to those used by the Passive Monitor. The Passive Monitor's Type 0 Header begins at address 0.

Programming capability list linked-list structure (shown in ) means the following fields and registers would be programmed as shown below for all functions:

- ❖ Capabilities Pointer in Common Configuration Space Header
  - ◆ Capability Pointer Register – offset address 0x34
- ❖ Capability ID for PCI and PCI Express capabilities & Next capability pointer.
  - ◆ PCI Power Management Capability ID – 0x01, Next capability Pointer – 0x50
  - ◆ MSI Capability ID – 0x05, Next Capability Pointer – 0x60
  - ◆ MSI-X Capability ID – 0x11, Next Capability Pointer – 0x80
  - ◆ PCI Express Capability ID – 0x10, Next Capability Pointer – 0x00 indicating end of PCIe Capability's.
- ❖ PCI Express Extended Capability ID and Next Capability Offset for all extended capabilities.
  - ◆ Virtual Channel Capability ID – 0x0002, Next Capability Offset – 0x000 indicating end of PCIe extended capabilities.

NOTE: The test bench should take care that the Next Capability pointer is 8-bit field [15:8] and Capability ID is also 8-bit field [7:0] for PCI & PCIe Capabilities. Next Capability Offset is 12 bit field [31:20] and PCIe Extended Capability ID is 16 bit field [15:0]. See the following illustration.

**Figure 18-5 Capability ID and Next Capability Pointer/Offset**



After programming the capability list, the test bench can alter the setting of specific fields as needed. In the example capability list in the previous, we have set Bus Master Enable; Memory Space Enable and IO Space enable to 1'b1 in Command Register (offset address - 0x4).

#### 18.11.5.1 In CFG\_SPACE\_BACKDOOR\_UPDATE mode

The example test case `monitor_configuration_space_backdoor_load_test` shows the backdoor configuration space corresponding to the previous figure, and is part of the `tb_pcie_svt_uvm_monitor_basic_sys` example. The path to the test case is:

```
$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/examples/sverilog/
tb_pcie_svt_uvm_monitor_basic_sys/tests/ts.monitor_configuration_space_backdoor_load_test.sv
```

#### 18.11.5.2 Steps For Initializing Monitor Config Space Through Backdoor Access

This section explains the steps involved in programming the passive monitor's configuration space, as done in the `tb_pcie_svt_uvm_monitor_basic_sys` example. There are multiple files in this example, refer to



README for example details. Following is the directory structure and file list for `tb_pcie_svt_uvm_monitor_basic_sys` example.

```
<tb_pcie_svt_uvm_monitor_basic_sys>
| -> top.sv
| -> top.pcie_device_pipe_x4.sv
| -> env
|     -> pcie_device_basic_env.sv
|     -> pcie_device_base_test.sv
|     -> pcie_device_random_traffic_sequence.sv
|     -> pcie_driver_transaction_directed_sequence.sv
|     -> pcie_shared_cfg.sv
| -> hdl_interconnect
|     -> svt_pcie_device_pipe_x4.sv
| -> tests
|     -> ts.base_pipe_test.sv
|     -> ts.monitor_configuration_space_backdoor_load_test.sv
|     -> ts.monitor_configuration_space Enumeration_test.sv
| -> vcs_build_options
| -> sim_build_options
| -> sim_run_options
| -> methodology
| -> modellist
| -> Makefile
| -> README
```

Following are the steps for configuration space updates through the backdoor:

1. [“Add Analysis Port of svt\\_pcie\\_tl\\_service type In Environment.”](#) on page 346
2. [“Add Tasks to Create and Send Service Request to Passive Monitor”](#) on page 346
3. [“Set Up Configuration Space Capability List”](#) on page 348
4. [“Program BARs”](#) on page 348
5. [“Set the Configuration Register Fields”](#) on page 348

#### 18.11.5.2.1 Add Analysis Port of svt\_pcie\_tl\_service type In Environment.

Refer to the testbench file `tb_pcie_svt_uvm_monitor_basic_sys/env/pcie_device_basic_env.sv`

**Declare the analysis port:**

```
/** Analysis port to pass TL service request to monitor */
`SVT_XVM(analysis_port) #(svt_pcie_tl_service) endpoint_passive_tl_service_port;
```

**Define the analysis ports in build\_phase:**

```
endpoint_passive_tl_service_port=new("endpoint_passive_tl_service_port",this);
```

**Connect TL service analysis port to endpoint monitor in connect\_phase:**

```
endpoint_passive_tl_service_port.connect
(endpoint_passive.pcie_agent.tl_mon.tl_service_in_port);
```

#### 18.11.5.2.2 Add Tasks to Create and Send Service Request to Passive Monitor

Add tasks to create and send service requests to the Passive Monitor, preferably in the base test so that it will be accessible to all the test cases extended from the base test. In this example, the tasks for creating and sending configuration space service requests are part of backdoor test, instead of base test in order to keep the example simple.



List of tasks for creating and sending service request related to configuration space backdoor to endpoint monitor in `ts.monitor_configuration_space_backdoor_load_test.sv` are:

```
send_cfg_space_wr_addr_tl_service_to_ep_mon
send_cfg_space_rd_addr_tl_service_to_ep_mon
send_cfg_space_set_fld_tl_service_to_ep_mon
send_cfg_space_get_fld_tl_service_to_ep_mon
send_cfg_space_set_bar_ro_map_service_to_ep_mon
send_cfg_space_get_bar_ro_map_service_to_ep_mon
send_cfg_space_print_service_to_ep_mon
send_cfg_space_dump_service_to_ep_mon
```

Below implementation of task for sending `MON_CONFIG_SPACE_WRITE_ADDR` service request as an sample. For other task implementation, refer to the file `ts.monitor_configuration_space_backdoor_load_test.sv` in the example directory.

```
//-----
// Method for sending svt_pcie_tl_service::MON_CONFIG_SPACE_WRITE_ADDR to endpoint
// monitor for writing to monitor's configuration space register
//-----
virtual task send_cfg_space_wr_addr_tl_service_to_ep_mon(bit [27:0] cfg_space_ecam_addr,
                                                         bit [31:0] cfg_space_bit_mask,
                                                         bit [31:0] cfg_space_dword_data);

    svt_pcie_tl_service  tl_service      = new();

    tl_service.service_type      = svt_pcie_tl_service::MON_CONFIG_SPACE_WRITE_ADDR;
    tl_service.mon_cfg_space_ecam_addr = cfg_space_ecam_addr;
    tl_service.mon_cfg_space_bit_mask  = cfg_space_bit_mask;
    tl_service.mon_cfg_space_dword_data = cfg_space_dword_data;
    env.endpoint_passive_tl_service_port.write(tl_service);
    tl_service.end_event.wait_on();
endtask

// BAR 0 - 1MB prefetchable 32 bit Memory Bar
send_cfg_space_wr_addr_tl_service_to_ep_mon('SVT_PCIE_CONFIG_SPACE_FUNC_0_BASE_ADDR + 'SVT_PCIE_CONFIG_SPACE_BAR_0_OFFSET, 32'hffff_ffff, 32'h1000_0008);
send_cfg_space_set_bar_ro_map_service_to_ep_mon(0,0,32'h000F_FFFF);
```

### 18.11.5.2.3 Set Up Configuration Space Capability List

Set up configuration space capability list corresponding to the following code.

```
// Initialize the capability set header registers and enabling the capability list
// in the Status Register.
send_cfg_space_wr_addr_tl_service_to_ep_mon('SVT_PCIE_CONFIG_SPACE_FUNC_0_CAP_STAT_ADDR, 32'hffff_ffff, 32'h0010_0000);

// Set the capability pointer to 40h.
send_cfg_space_wr_addr_tl_service_to_ep_mon('SVT_PCIE_CONFIG_SPACE_FUNC_0_CAP_PTR_ADDR, 32'hffff_ffff, 32'h0000_0040);

//-----
// PCI Capability Sets
//-----
// Set PME capability register to address 40h with next capability set pointer pointing to address 'h50
send_cfg_space_wr_addr_tl_service_to_ep_mon('SVT_PCIE_CONFIG_SPACE_FUNC_0_BASE_ADDR + 28'h000_0040, 32'hffff_ffff, 32'h0002_5001);

// Set MSI capability register to address 'h50 with next capability set pointer pointing to address 60h
send_cfg_space_wr_addr_tl_service_to_ep_mon('SVT_PCIE_CONFIG_SPACE_FUNC_0_BASE_ADDR + 28'h000_0050, 32'hffff_ffff, 32'h0000_6005);

// Set MSI-X capability register to 60h with next capability set pointer pointing to address 80h
send_cfg_space_wr_addr_tl_service_to_ep_mon('SVT_PCIE_CONFIG_SPACE_FUNC_0_BASE_ADDR + 28'h000_0060, 32'hffff_ffff, 32'h0001_8011);

// Set PCIE capability register to 80h, next pointer pointing to null
send_cfg_space_wr_addr_tl_service_to_ep_mon('SVT_PCIE_CONFIG_SPACE_FUNC_0_BASE_ADDR + 28'h000_0080, 32'hffff_ffff, 32'h0002_0010);

//-----
// PCI Extended Capability Sets
//-----
// Set Virtual Channel capability register to 'h100, next pointer pointing to null ('h000)
send_cfg_space_wr_addr_tl_service_to_ep_mon('SVT_PCIE_CONFIG_SPACE_FUNC_0_EXT_CAP_PTR_ADDR, 32'hffff_ffff, 32'h0001_0002);
// Add programming of extended VC count .
send_cfg_space_wr_addr_tl_service_to_ep_mon('SVT_PCIE_CONFIG_SPACE_FUNC_0_EXT_CAP_PTR_ADDR + 28'h000_0004, 32'hffff_ffff, 32'h0000_0007);
// Set VC ARB TABLE OFFSET to null.
send_cfg_space_wr_addr_tl_service_to_ep_mon('SVT_PCIE_CONFIG_SPACE_FUNC_0_EXT_CAP_PTR_ADDR + 28'h000_0008, 32'hffff_ffff, 32'h0000_0000);
```

### 18.11.5.2.4 Program BARs

For programming BARs through backdoor, the test bench programs the BAR type, starting address of the BAR and BAR size. The BAR starting address and BAR type can be programmed together through MON\_CONFIG\_SPACE\_WRITE\_ADDR service request. The size of the BAR is programmed with a MON\_CONFIG\_SPACE\_SET\_BAR\_RO\_MAP service request.

For example, let us consider for Function 0, BAR0 is 1 MB Prefetchable 32-bit Memory BAR with starting address of 32'h1000\_0000.

The BAR type is the last nibble of BAR register, and for Prefetchable 32-bit Memory BAR it will be 4'h8. Combining BAR type with the starting address will give us the DWord value for MON\_CONFIG\_SPACE\_WRITE\_ADDR service request, i.e. 32'h1000\_0008

For addressing 1 MB you need 20 bit's, i.e. 32'h000F\_FFF.

```
// BAR 0 - 1MB prefetchable 32 bit Memory Bar
send_cfg_space_wr_addr_tl_service_to_ep_mon('SVT_PCIE_CONFIG_SPACE_FUNC_0_BASE_ADDR + 'SVT_PCIE_CONFIG_SPACE_BAR_0_OFFSET, 32'hffff_ffff, 32'h1000_0008);
send_cfg_space_set_bar_ro_map_service_to_ep_mon(0,0,32'h000F_FFFF);
```

### 18.11.5.2.5 Set the Configuration Register Fields

After setting up the capability list and BAR, the testbench can alter some fields as needed if the test requires that these fields have values which are different from the default value in the PCIe Base Specification. In

this example, we are setting set Bus Master Enable; Memory Space Enable and IO Space enable to 1'b1 in Command Register (offset address - 0x4).

To set or retrieve fields from monitor's configuration space MON\_CONFIG\_SPACE\_SET\_FIELD and MON\_CONFIG\_SPACE\_GET\_FIELD service request are used.

```
send_cfg_space_set_fld_tl_service_to_ep_mon(0, 'SVT_PCIE_CMD_REG_IO_SPACE_EN_FLD, 1'b1);  
send_cfg_space_set_fld_tl_service_to_ep_mon(0, 'SVT_PCIE_CMD_REG_MEM_SPACE_EN_FLD, 1'b1);  
send_cfg_space_set_fld_tl_service_to_ep_mon(0, 'SVT_PCIE_CMD_REG_BUS_MASTER_EN_FLD, 1'b1);
```

For setting or getting a configuration space field value from test bench, the field ID value for the respective file should be known. Field ID is a value uniquely identifying the Fields per function. The complete list of macro's define for all the recognized configuration space register fields can be found in the following include file used by the Passive Monitor:

```
$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/sverilog/include/  
svt_pcie_config_space_fld_id_defines.svi
```

### 18.11.6 In CFG\_SPACE\_ENUMERATION\_UPDATE Mode

The example test case monitor\_configuration\_space\_enumeration\_test shows the programming of monitor's configuration space in enumeration mode corresponding to capability list in Figure 4 as part of the tb\_pcie\_svt\_uvm\_monitor\_basic\_sys example. The path to the test case is

```
$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/examples/sverilog/  
tb_pcie_svt_uvm_monitor_basic_sys/tests/ts.monitor_configuration_space_enumeration_test.sv
```

As stated above, in enumeration mode, the test bench must program the capability list before link-up. The test bench can either program BARs and registers (and register fields) through backdoor access or through CFG Read and Write transactions as part of the enumeration process. The Passive Monitor will update the configuration space register with the data captured from bus after each successful CFG transaction.

## 18.12 Equalization Support

PCIe SVT passive monitor supports equalization at 8G as well as at 16G speeds. The Equalization configuration attributes are values provided by the test bench to enable the PCIe SVT Passive Monitor to check the equalization progress and compliance to the specification. The Equalization Status Attributes can be accessed by the test bench for specific checking on various aspects of the equalization procedure.

### 18.12.1 Equalization Configuration Attributes

The following configuration attributes found in the svt\_pcie\_pl\_configuration class are used by the PCIe SVT Passive Monitor for tracking equalization states and equalization evaluation cycles and evaluating rules (protocol checks) specific to equalization at the 8G and 16G speeds. Please refer to HTML reference guide for detailed description of these properties.

**Table 18-4 Attributes Common for 8G and 16G Equalization**

No.	Attribute	Description
1	highest_enabled_equalization_phase	Enables VIP to go through respective equalization phases. Passive VIP's LTSSM does not transition to equalization state when this attribute is set to zero.
2	upstream_lanes_recovery_eq_phase0_timeout_ns	Represents LTSSM's timeout in ns for upstream lanes in Recovery.Equalization.0 state.
3	upstream_lanes_recovery_eq_phase1_timeout_ns	Represents LTSSM's timeout in ns for upstream lanes in Recovery.Equalization.1 state.
4	upstream_lanes_recovery_eq_phase2_timeout_ns	Represents LTSSM's timeout in ns for upstream lanes in Recovery.Equalization.2 state.
5	upstream_lanes_recovery_eq_phase3_timeout_ns	Represents LTSSM's timeout in ns for upstream lanes in Recovery.Equalization.3 state.
6	downstream_lanes_recovery_eq_phase1_timeout_ns	Represents LTSSM's timeout in ns for downstream lanes in Recovery.Equalization.1 state.
7	downstream_lanes_recovery_eq_phase2_timeout_ns	Represents LTSSM's timeout in ns for downstream lanes in Recovery.Equalization.2 state.
8	downstream_lanes_recovery_eq_phase3_timeout_ns	Represents LTSSM's timeout in ns for downstream lanes in Recovery.Equalization.3 state.
9	recovery_eq_to_speed_tolerance_timeout_ns	Represents tolerance timeout in ns for Recovery.Equalization to Recovery.Speed timeout transition of LTSSM.
10	get_local_preset_coefficients_timeout_ns	Represents time in ns at which PIPE MAC times out waiting for PIPE PHY to respond to preset to coefficients mapping request.
11	min_rx_eq_eval_delay	Represents time in ns required by PHY to respond to RxEqEval abort situation by asserting PhyStatus.
12	max_rx_eq_eval_delay	In MPIPE mode i.e. when monitored device is MAC PIPE it represents the maximum amount of time in ns MAC takes to assert RxEqEval for current evaluation cycle when accepted by link partner. In SPIPE mode i.e. when monitored device is PHY PIPE it represents the maximum amount of time in ns PHY takes to respond to RxEqEval by PhyStatus assertion.
13	enable_rxeqeval_default_settings_vector	Enables RxEqEval assertion checking for default settings in various equalization phases.
14	ltssm_recovery_equalization_rx_count	Represents number of Training Sets LTSSM must see on RX path prior to exiting current equalization phase.
15	ltssm_recovery_equalization_tx_count	Represents number of Training Sets LTSSM must see on TX path prior to exiting current equalization phase.



The attributes `min_rx_eq_eval_delay` and `max_rx_eq_eval_delay` are used differently by the active PCIe SVT VIP. The active VIP in SPIPE mode responds to `RxEqEval` assertion by asserting `PhyStatus` after delay of `N ns`, where `N` is a random number within `min_rx_eq_eval_delay` and `max_rx_eq_eval_delay` boundaries.

**Table 18-5 Attributes for 8G Equalization**

No.	Attribute	Description
1	<code>preset_to_coefficients_mapping_entry_valid</code>	Bit mapped enable for each entry in <code>preset_to_coefficients_mapping_table</code>
2	<code>preset_to_coefficients_mapping_table</code>	Represents monitored device's link partner's preset to coefficients mapping table for 8G equalization.
3	<code>expected_preset_to_coefficients_mapping_entry_enable</code>	Bit mapped enable for each entry in <code>expected_preset_to_coefficients_mapping_table</code>
4	<code>expected_preset_to_coefficients_mapping_table</code>	Represents monitored device's preset to coefficients mapping table for 8G equalization.

For an active PCIe SVT VIP the attribute '`preset_to_coefficients_mapping_table`' represents the VIP's own preset to coefficients mappings for 8G equalization and the attribute '`expected_preset_to_coefficients_mapping_table`' represents the DUT's preset to coefficients mappings for 8G equalization.

**Table 18-6 Attributes for 16G Equalization**

No	Attribute	Description
1	<code>preset_to_coefficients_mapping_entry_valid_16g</code>	Bit mapped enable for each entry in <code>preset_to_coefficients_mapping_table_16g</code>
2	<code>preset_to_coefficients_mapping_table_16g</code>	Represents monitored device's link partner's preset to coefficients mapping table for 16G equalization.
3	<code>expected_preset_to_coefficients_mapping_entry_enable_16g</code>	Bit mapped enable for each entry in <code>expected_preset_to_coefficients_mapping_table_16g</code>
4	<code>expected_preset_to_coefficients_mapping_table_16g</code>	Represents monitored device's preset to coefficients mapping table for 16G equalization.

For an active PCIe SVT VIP attribute '`preset_to_coefficients_mapping_table_16g`' represents VIP's own preset to coefficients mappings for 16G equalization and an attribute '`expected_preset_to_coefficients_mapping_table_16g`' represents DUT's preset to coefficients mappings for 16G equalization.

### 18.12.2 Equalization Status Attributes

The PCIe SVT Passive Monitor gathers various equalization attributes of evaluation cycles in `svt_pcie_pl_status` class. The `svt_pcie_pl_status` class in turn uses `svt_pcie_eq_status` to represent equalization specific attributes which in turn uses values in the `svt_pcie_eq_eval_cycle` class to gather

information regarding equalization evaluation cycle. The following table lists the status attributes made available. Refer to the HTML reference guide for a detailed description of these properties.

**Table 18-7 svt\_pcie\_pl\_status Attributes Common for 8G and 16G Equalization**

No	Attribute	Description
1	start_equalization_w_preset	Represents PCIe specifications variable start_equalization_w_reset

**Table 18-8 svt\_pcie\_pl\_status attributes for 8G Equalization**

No	Attribute	Description
1	equalization_done_8gt_data_rate	Represents PCIe specifications variable equalization_done_8gt_data_rate
2	equalization_complete	Represents PCIe specifications variable 'Equalization Complete'
3	equalization_phase_1_successful	Represents PCIe specifications variable 'Equalization Phase 1 Successful'
4	equalization_phase_2_successful	Represents PCIe specifications variable 'Equalization Phase 2 Successful'
5	equalization_phase_3_successful	Represents PCIe specifications variable 'Equalization Phase 3 Successful'
6	eq_status	Represents various equalization specific properties captured from TS1s in different phases of 8G equalization. Refer to HTML reference guide on details on 'svt_pcie_eq_status' and 'svt_pcie_eq_eval_cycle' class attributes.

**Table 18-9 svt\_pcie\_pl\_status attributes for 16G equalization**

No	Attribute	Description
1	equalization_done_16gt_data_rate	Represents PCIe specifications variable equalization_done_16gt_data_rate
2	equalization_16g_complete	Represents PCIe specifications variable 'Equalization 16G Complete'
3	equalization_16g_phase_1_successful	Represents PCIe specifications variable 'Equalization 16G Phase 1 Successful'
4	equalization_16g_phase_2_successful	Represents PCIe specifications variable 'Equalization 16G Phase 2 Successful'
5	equalization_16g_phase_3_successful	Represents PCIe specifications variable 'Equalization 16G Phase 3 Successful'
6	eq_16g_status	Represents various equalization specific properties captured from TS1s in different phases of 16G equalization. Refer to HTML reference guide on details on 'svt_pcie_eq_status' and 'svt_pcie_eq_eval_cycle' class attributes.

# 19

## 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.

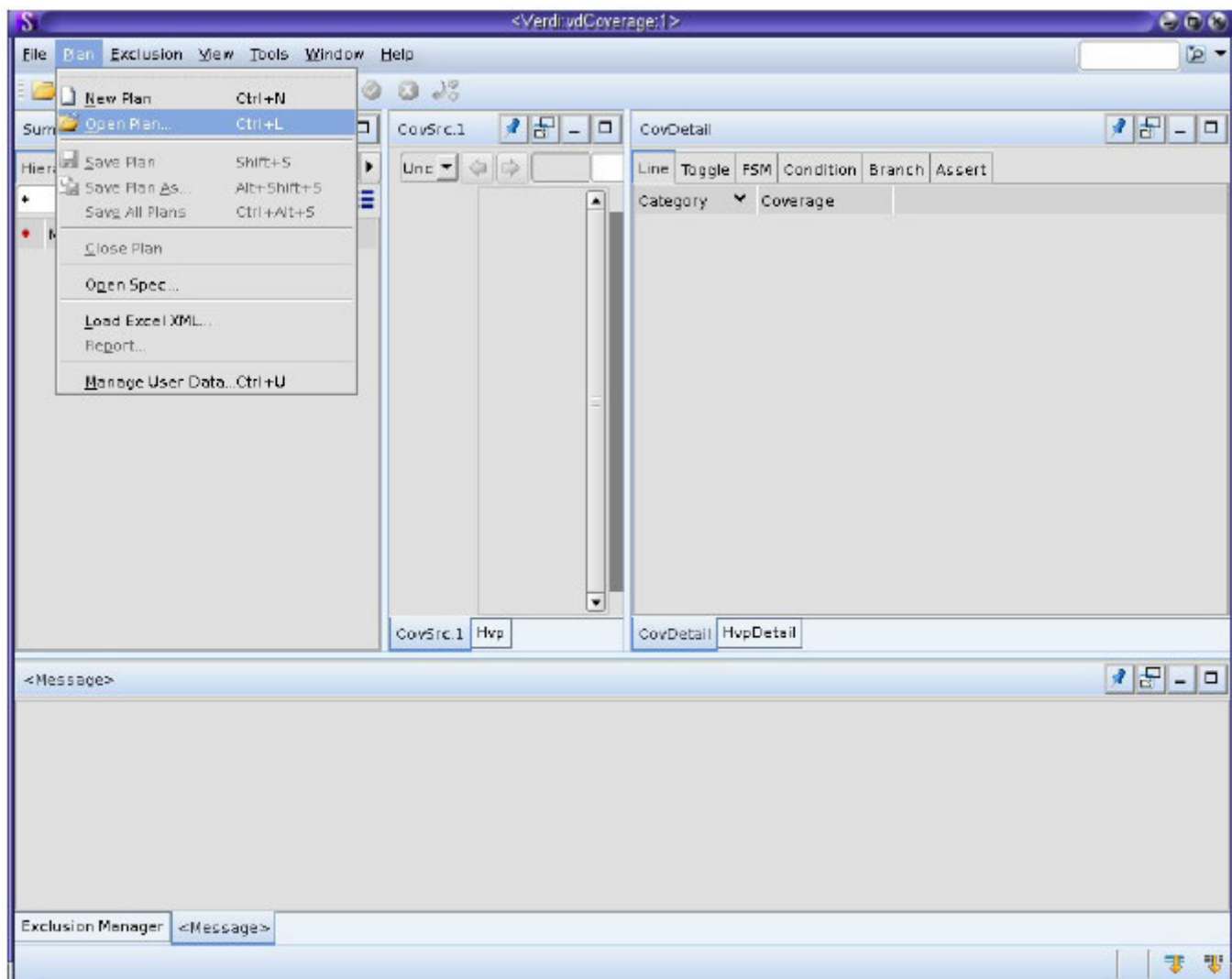
Coverage results are annotated to the plan that helps to map the verification completeness on a feature by feature basis at the aggregate level.

### 19.1 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.

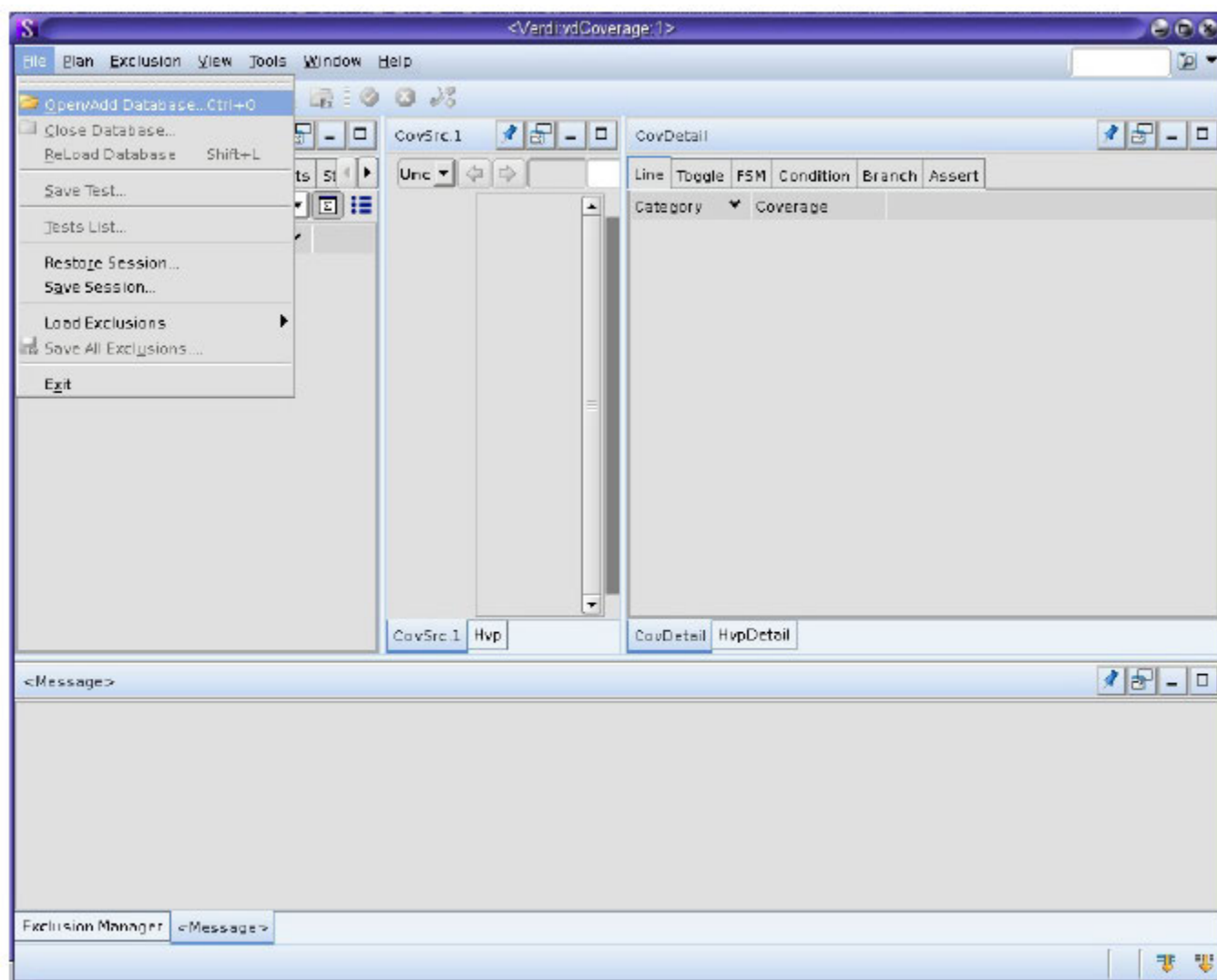
To leverage the verification plan, perform the following steps:

1. Navigate to the example directory using the following command:  
`cd <intermediate_example_dir>`
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 19-1 Open Plan**

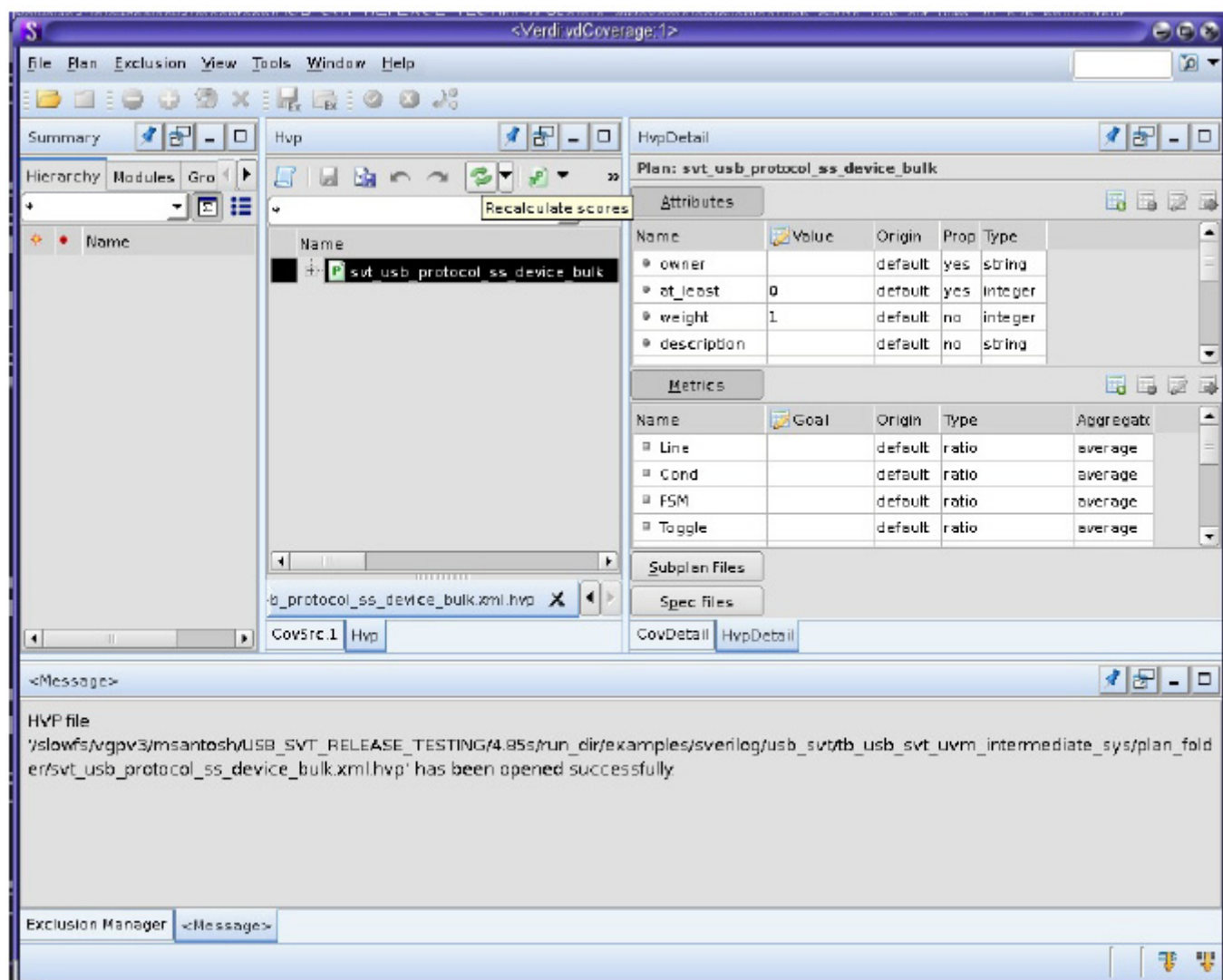
Load the coverage database (simvcssvlog.vdb) from the output folder in the current directory using the Verdi drop down menu, as shown in the following illustration.



**Figure 19-2 Verdi GUI**

Click Recalculate button to annotate the coverage results, as shown in the following illustration.

Figure 19-3 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\_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  
`$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/pcie_svt_uvm_class_reference/html/protocolChecks.html`
- MPHY SVT  
`$DESIGNWARE_HOME/vip/svt/pcie_svt/latest/doc/mpcie_svt_uvm_class_reference/html/protocolChecks.html`



### Note

The protocol checks for PHY layer are not supported in this release.



## 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	Spice Dir.	Lane 0 Mpipe / Spice Name	Mpipe / Spice 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 <code>is_pie8_mode_master = SVT_PCIE_IS_PIE8_MODE_MASTER_DEFAULT;</code> )
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's 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.

**Table B-6 MAC Internal PIE-8 "per Lane" Signals**

Signal Name	Description
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.
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	

**Table B-8     PIE-8 MSGCODES (Continued)**

Signal Name	Description
	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	
	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	

**Table B-8     PIE-8 MSGCODES (Continued)**

Signal Name	Description
	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	
	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” on page 369
- “Driver Application Parameters” on page 370
- “Requester Parameters” on page 370
- “Completion Target Parameters” on page 371
- “Memory Target Parameters” on page 372
- “Transaction Layer Parameters” on page 372
- “Data Link Layer Parameters” on page 373
- “Physical Layer Parameters” on page 373
- “Physical Coding Sublayer (PCS) Parameters” on page 375
- “Serializer/Deserializer (SERDES) Parameters” on page 376

## 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.
CMB_TABLE_SIZE				



**Table C-3 Requester parameters (Continued)**

	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-7](#).

**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 <sup>16</sup>	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'b0011111010	Definition of positive COMMA character.
COMMA_N				
	Integer	Any 10-bit value	10'b1100000101	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 UVM class members, for users who are migrating from the SVC or SVT Verilog PCIe VIP to the UVM PCIe.

This appendix contains the following sections:

- [“Transaction Layer Verilog Tasks and Parameters to UVM Class Members Map”](#) on page 377
- [“Data Link Layer Verilog Tasks and Parameters to UVM Class Member Maps”](#) on page 379

## D.1 Transaction Layer Verilog Tasks and Parameters to UVM Class Members Map

This section contains the following tables:

- [“Transaction Layer Verilog Task to UVM Class Member Map”](#) on page 377
- [“Transaction Layer Verilog Parameters to UVM Class Members Map”](#) on page 378

### D.1.1 Transaction Layer Verilog Task to UVM 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 UVM class members**

Verilog Task	UVM 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 UVM class members (Continued)**

Verilog Task	UVM 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 UVM Class Members Map

Transaction Layer Verilog parameters are mapped to UVM class members in [Table D-2](#), listed alphabetically by Verilog parameter.

**Table D-2 Map of Transaction Layer class members to parameters**

Verilog Parameter	UVM 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	UVM 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 UVM Class Member Maps

- [“Data Link Layer Verilog Task to UVM Class Member Map”](#)
- [“Data Link Layer Verilog Parameter to UVM Class Member Map”](#)

### D.2.1 Data Link Layer Verilog Task to UVM Class Member Map

Data Link Layer Verilog tasks are mapped to UVM class members in [Table D-3](#), listed alphabetically by Verilog task.

**Table D-3 Map of Data Link Layer Verilog tasks to UVM class members**

Verilog Task	UVM 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 UVM class members (Continued)**

Verilog Task	UVM 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 UVM Class Member Map

Data Link Layer Verilog parameters are mapped to UVM 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	UVM 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	UVM 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	UVM 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

# SolvNet PCIe VIP Articles

---

The following table lists and links to all the SolvNet articles published on the PCIe VIP. The articles are organized by the following categories:

- [“Transaction Layer”](#) on page 385
- [“Data Link Layer”](#) on page 387
- [“PHY Layer”](#) on page 388
- [“Methodology, Testbench, and Debug”](#) on page 391

## 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 UVM 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 uninitialized 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 Completion 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 Gereneration 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 UVM 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 UVM 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 UVM 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 LOS 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\_pp Switch

<https://solvnet.synopsys.com/retrieve/2246895.html>

Title: VC VIP: Understanding Why the PCIe VIP Issues Null Object Access (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 UVM\_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 UVM\_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 UVM 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>
- Title: VC VIP: Getting a Fatal Error Stating 8G is Not Supported  
<https://solvnet.synopsys.com/retrieve/2272176.html>
- Title: VC VIP: Interfacing a PCIe UVM 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: Avoiding "Malformed OS detected " Errors Using the PCIe Passive Monitor  
<https://solvnet.synopsys.com/retrieve/2334608.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 UVM 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>



## A

# Reporting Problems

## A.1 Introduction

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

## A.2 Debug Automation

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

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

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

## A.3 Enabling and Specifying Debug Automation Features

Debug Automation is enabled through the use of a run-time plusarg named *+svt\_debug\_opts*. This plusarg accepts an optional string-based specification to control various aspects Debug Automation. If this

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

Note the following about the plusarg:

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

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

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

The following table explains each control string:

**Table A-1 Control Strings for Debug Automation plusarg**

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

#### Examples:

Enable on all VIP instances with default options:

```
+svt_debug_opts
```

Enable on all instances:

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

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

Enable on all instances:

- ❖ starting at time 1000 until time 1500:

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

Enable debug feature on all instances using default options:

- ❖ By setting the macro `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
```

### Note

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`.

## A.4 Debug Automation Outputs

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

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

The following are the output files generated:

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

## A.5 FSDB File Generation

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

### A.5.1 VCS

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

```
+define+SVT_FSDB_ENABLE -P $(VERDI_HOME)/share/PLI/VCS/{LINUX|LINUX64}/novas.tab
$(VERDI_HOME)/share/PLI/VCS/{LINUX|LINUX64}/pli.a
```

The simulation environment must be updated to add the following to LD\_LIBRARY\_PATH:

```
$(VERDI_HOME)/share/PLI/VCS/{LINUX|LINUX64}
```

### A.5.2 Questa

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

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

The simulation environment must be updated to add the following to LD\_LIBRARY\_PATH:

```
$(VERDI_HOME)/share/PLI/MODELSIM/{LINUX|LINUX64}
```

### A.5.3 Incisive

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

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

The simulation environment must be updated to add the following to LD\_LIBRARY\_PATH:

```
$(VERDI_HOME)/share/PLI/IUS/{LINUX|LINUX64}
```

## A.6 Initial Customer Information

Follow these steps when you call the Synopsys Support Center:

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

Enable the Debug Opts feature. For more information, see the [“Debug Automation”](#) on page 395.

## A.7 Sending Debug Information to Synopsys

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

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

- ◆ Testbench language (SystemVerilog or Verilog)
  - ◆ Simulator and version
  - ◆ DUT languages (Verilog)
3. Use the VIP case submittal tool to pack a file with the appropriate debug information. It has the following usage syntax:

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

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

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

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

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

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

**Note**

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.

## A.8 Limitations

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

- ❖ Only enable the VIP instance necessary for debug. By default, the +svt\_debug\_opts command enables Debug Opts on all instances, but the 'inst' argument can be used to select a specific instance.
- ❖ Use the start\_time and end\_time arguments to limit the verbosity changes to the specific time window that needs to be debugged.

