Verification Continuum™

VC Verification IP
JEDEC UFS
UVM User Guide

Version Q-2020.06, June 2020

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

Preface

About This Guide

This guide contains installation, setup, and usage material for SystemVerilog Universal Verification Methodology (UVM) users of the VC VIP for Universal Flash Storage (UFS), and is for design or verification engineers who want to verify UFS operation using a UVM testbench written in SystemVerilog. Readers are assumed to be familiar with UFS protocol specification, Object Oriented Programming (OOP), SystemVerilog, and UVM techniques.

Guide Organization

The chapters of this data book are organized as follows:

- Chapter 1, "Introduction", introduces the UFS VIP and its features.
- Chapter 2, "Installation and Setup", describes the system requirements and provides instructions on how to install, configure, and use the UFS VIP.
- Chapter 3, "General Concepts", introduces the UFS VIP within an UVM environment and describes the data objects and components that comprise the VIP.
- Chapter 4, "Using the UFS Verification IP", describes the UVM concepts and techniques achieving a basic constrained random testbench that incorporates the UFS VIP.
- Chapter 5, "Usage Notes", provides the useful information about the UFS VIP.
- Chapter 6, "VIP Tools", provides the useful information about the UFS VIP.
- Chapter 7, "Troubleshooting", provides useful information to troubleshoot common problems that is encountered while using the UFS VIP.
- Chapter 8, "Reporting Problems", describes the process for reporting the SVT transactor issues to Synopsys.

Web Resources

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

Customer Support

To register a problem, perform any of the following tasks:

1. Go to https://solvnetplus.synopsys.com and open a case.

   Enter the information according to your environment and your issue.

2. Send an e-mail message to support_center@synopsys.com
   - Include the Product name, Sub Product name, and Product version for which you want to register the problem.
3. Telephone your local support center.
   - North America:

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

     http://www.synopsys.com/Support/GlobalSupportCenters

1 Introduction

1.1 Introduction

The UFS VIP supports verification of SoC designs that include interfaces implementing the UFS 3.0 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
- Proven testbench architecture that provides maximum reuse, scalability, and modularity
- Proven verification approach and methodology
- Transaction-level, self-checking tests
- Object oriented interface that allows OOP techniques

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

Refer:

- *JEDEC Standard for Universal Flash Storage (UFS) Version 2.0 JESD220B (Revision of JESD220A, June 2012) - September 2013, Version 2.1 JESD220C (Revision of JESD220B, September 2013) - March 2016, Version 3.0 JESD220D (Revision of JESD220C, March 2016) - January 2018*
- *JEDEC STANDARD Universal Flash Storage (UFS) Unified Memory Extention Version 1.0 JESD220-1 SEPTEMBER 2013, Version 1.1 JESD220-1A (Revision of JESD220-1, September 2013) March 2016*
- *JEDEC STANDARD Universal Flash Storage (UFS) Host Controller Interface, Version 2.1 JESD223C (Revision of JESD223B, September 2013) - March 2016, Version 3.0 JESD223D (Revision of JESD223C, September2016) - January 2018*
- *JEDEC STANDARD Universal Flash Storage (UFS) Host Controller Interface (USHCI), Unified Memory Extension Version 1.0 JESD223-1 September 2013, Version 1.1 JESD223-1A (Revision of JESD223-1, September 2013) March 2016*
- For UniPro information, see the *MIPI UniPro UVM User Guide*
- For M-PHY information, see the *MIPI M-PHY UVM User Guide*

- For UFS class reference, see:

  ```
  $DESIGNWARE_HOME/vip/svt/jedec_ufs_svt/latest/doc/jedec_ufs_svt_uvm_class_reference/html/index.html
  ```

- For UVM, the user examples are available at:

  ```
  $DESIGNWARE_HOME/vip/svt/jedec_ufs_svt/latest/examples/sverilog
  ```

  The README file, which resides in the example location provides the information about the examples.

1.2 Product Overview

The UFS VIP is a suite of UVM-based verification components that are compatible for use with SystemVerilog compliant testbenches. The UFS VIP suite simulates UFS transactions through active agents orenv, as defined by the UFS specification.

The UFS VIP can act as both Host Agent and Device Agent. After instantiating the agentsor env, you can select and combine active and passive agents to create an environment that verifies UFS features in the DUT. The Host Agent and Device Agent support all the functionalities normally associated with active and passive UVM components, including the creation of transactions, checking and reporting the protocol correctness, transaction logging, and functional coverage. The Host Agent and Device Agent can also be used in standalone mode.

1.3 Language and Methodology Support

The UFS VIP supports the following languages and methodology:

- Languages
  - SystemVerilog
- Methodology
  - UVM 1.1 and 1.2
    For more information on UVM 1.2, see http://www.accellera.org/.

1.4 UFS VIP Supported Features

The following sections discuss the supported features in UFS VIP.

1.4.1 Protocol Features

UFS VIP supports the following protocol features:

- UPIU supports NOPOUT, NOPIN, COMMAND, RESPONSE, REJECT UPIU, TASK_MANAGEMENT_REQUEST, TASK_MANAGEMENT_RESPONSE, DATAIN, DATAOUT, QUERY REQUEST, and QUERY RESPONSE
- Multiple NOPOUTs (NOP IN for each and every NOPOUT sent)
- Multiple outstanding COMMAND UPIU
- Task_management_request functions, such as, ABORT, ABORT Task Set, Clear Task Set, Query, Query Task Set
- Multiple Logical Units (upto a maximum number of 8)
- Controllable Logical Unit command depth

- RPMB Logical Unit
- Command Priority
- Multi-Initiator
- Secure Write Protection
- Device Life Span Mode
- Device Health Descriptor
- Production State Awareness
- SCSI commands: WRITE-6, READ-6, WRITE-10, READ-10, WRITE-16, READ-16, FORMAT UNIT, PRE-FETCH(10), PRE-FETCH(16), READ BUFFER, READ CAPACITY(10), READ CAPACITY(16), REPORT LUNS, REQUEST SENSE, SEND DIAGNOSTIC, START STOP UNIT, SYNCHRONIZE CACHE(10), SYNCHRONIZE CACHE(16), TEST UNIT READY, UNMAP, VERIFY(10)AND WRITE BUFFER, MODE SENSE, NODE SELECT, INQUIRY, SECURITY PROTOCOL IN, SECURITY PROTOCOL OUT
  - Zero transfer length of all the above commands
- Support to ignore PACP_EXT_CAP1 frame when the UniPro version is 1.4
- Hardware reset
- Supports UniPro v1.4, 1.6 and 1.8
- Universal Memory Extension (UME) support
- LCC Support

1.5 Verification Features

UFS VIP supports the following verification features:

- Protocol checks
- Sequence collection
- Verification Planner
- UFS, UniPro, and MPHY layer error injection for device VIP
- UFS Device VIP Response Management
- Configuring specification versions 1.1, 2.0, 2.1 and 3.0
- Clock Jitter in Serial and RMMI transmit
- Accessing Status Object from Testbench
- Timescale Setting and Scaling Down Timer Values
- Lane-to-Lane Skew Injection
- Symbol clock generation by VIP

1.6 Methodology Features

UFS VIP supports the following methodology features:

- UFS VIP is organized to act as a UFS Host Agent or UFS Device Agent
- Analysis ports for connecting Host/Device agent to scoreboard, or any other component

- Callbacks for Host/Device Driver and Monitor
- Notification to indicate start and end of info units

2 Installation and Setup

2.1 Introduction

This section provides information on how to install and set up the UFS VIP. After you complete performing the installation steps, the testbench will be operational and the UFS VIP will be ready to use.

The quick start lists the following steps:

- Verifying the Hardware Requirements
- Verifying the Software Requirements
- Prerequisites
- Download and Installation
- What's Next?
- Setting Up a New VIP
- Installing and Setting Up More than One VIP Protocol Suite
- Updating an Existing Model
- Include and Import Model Files into Your Testbench
- Compile-Time and Runtime Options

Note If you encounter any problems during installing the UFS VIP, see Customer Support.

2.2 Verifying the Hardware Requirements

UFS VIP requires the following configuration for a Solaris or Linux workstation:

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

2.3 Verifying the Software Requirements

UFS VIP is compatible with certain versions of platforms and simulators. This section lists the required softwares for UFS VIP.

2.3.1 Platform or Operating System and Simulator Software

Platform or OperatingSystem and VCS: Only compatible platform or OS versions are used for UFS VIP, see the support matrix for "SVT-based" VIP in the following document:

Support Matrix for JEDEC VIP: *JEDEC VIP Release Notes*

2.3.2 Synopsys Common Licensing Software

The SCL software provides the licensing function for the UFS VIP. For more details of installing SCL software see the installation instructions in Licensing Information.

2.3.3 Other Third Party Software

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

2.4 Prerequisites

Perform the following steps before installing UFS VIP:

1. Set DESIGNWARE_HOME to the absolute path where UFS VIP will be installed:

   ```
   setenv DESIGNWARE_HOME absolute_path_to_designware_home
   ```

2. Ensure that your environment and PATH variables are set correctly, include the following:
   - 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. In addition, 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
     ```

2.5 Download and Installation

Follow the following instructions to download 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.

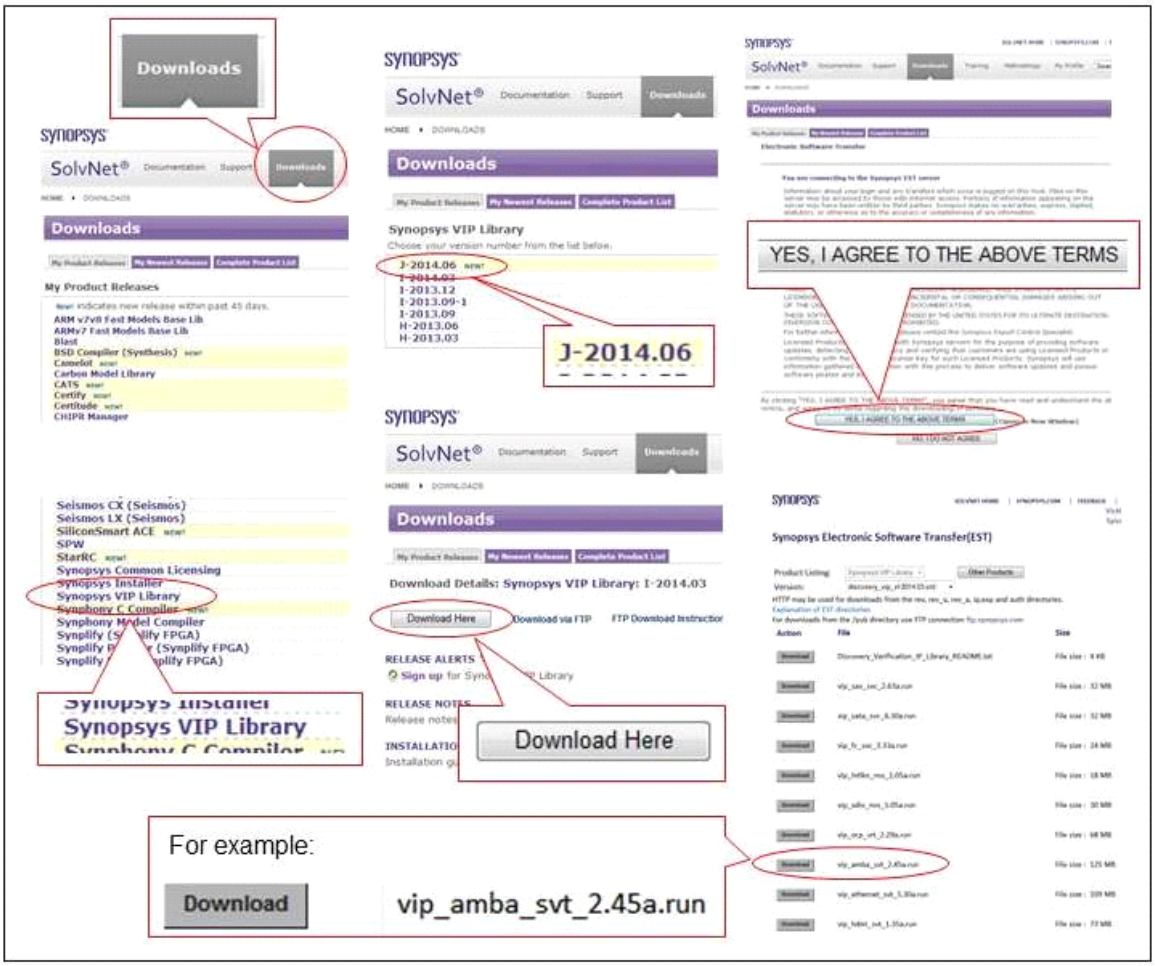
A passive mode FTP is required. The passive command toggles between passive and active mode. If your FTP utility does not support passive mode, use http. For additional information, see the following web page:

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

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

1. Point your web browser to http://solvnet.synopsys.com.
2. Enter your Synopsys SolvNet Username and Password.
3. Click the `Sign In` button.
4. Make the following selections on SolvNet to download the `.run` file of the VIP (See Figure 2-1).
   a. `Downloads` tab
   b. VC VIP Library product releases
   c. <release_version>
   d. `Download Here` button
   e. `Yes, I Agree to the Above Terms` button
   f. Download `.run` file for the VIP

Figure 2-1 SolvNet Selections for VIP Download



g. Set the `DESIGNWARE_HOME` environment variable to a path where you want to install the VIP.

```
% setenv DESIGNWARE_HOME VIP_installation_path
```

h. Execute the `.run` file by invoking its filename. The VIP is unpacked and all files and directories are installed under the path specified by the `DESIGNWARE_HOME` environment variable. The `.run` file can be executed from any directory. The important step is to set the `DESIGNWARE_HOME` environment variable before executing the `.run` file.

2.5.2 Downloading Using FTP with a Web Browser

a. Follow the above instructions through the product version selection step.

b. Click the "Download via FTP" link instead of the "Download Here" button.

c. Click the "Click Here To Download" button.

d. Select the files that you want to download.

e. Follow browser prompts to select a destination location.

2.6 What's Next?

The following section describes in detail the different steps that needs to be performed during installation and setup:

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

2.6.1 Licensing Information

The UFS VIP uses the SCL software to control its usage.

You can find general SCL information at:

http://www.synopsys.com/keys

The UFS VIP product is enabled by features defined below and in the order listed. Once a required feature or a set of features are successfully checked out, the VIP stops looking for other licenses.

- VIP-UFS-SVT
- VIP-LIBRARY-SVT + DesignWare-Regression

Only one license is consumed per simulation session, irrespective of the multiple Synopsys VIP models instantiated in the design.

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

2.6.1.1 License Polling

If you request for a license and it is unavailable, license polling queues up your request until a license is available instead of exiting immediately. To control license polling, use the following DW_WAIT_LICENSE environment variable:

- To activate license polling, set the DW_WAIT_LICENSE environment variable to 1.
- To deactivate license polling, unset the DW_WAIT_LICENSE environment variable. By default, license polling is disabled.

2.6.1.2 Simulation License Suspension

All VIP products support license suspension. The simulators that support license suspension allows a model to check-in its license token while the simulator is suspended, then check-out the license token when the simulation is resumed.

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

2.6.2 Environment Variable and Path Settings

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

- `DESIGNWARE_HOME`: The absolute path to where the VIP is installed.
- `DW_LICENSE_FILE` - The absolute path to file that contains the license keys for the VIP product software or the port@host reference to this file.
- `SNPSLMD_LICENSE_FILE`: The absolute path to files that contains the license keys for Synopsys software (VIP and/or other Synopsys Software tools) or the port@host reference to this file.

For faster license checkout of Synopsys VIP software, ensure to place the required license files at the front of the list of arguments to `SNPSLMD_LICENSE_FILE`.

- `LM_LICENSE_FILE`: The absolute path to a file that contains the license keys for both Synopsys software and/or your third-party tools.

The Synopsys VIP License can be set in either of the 3 license variables mentioned above with the order of precedence for checking the variables being:

- `DW_LICENSE_FILE` -> `SNPSLMD_LICENSE_FILE` -> `LM_LICENSE_FILE`, but also note If DW_LICENSE_FILE environment variable is enabled, VIP will ignore SNPSLMD_LICENSE_FILE and `LM_LICENSE_FILE` settings.

  Hence to get the most efficient Synopsys VIP license checkout performance, set the `DW_LICENSE_FILE` with only the License servers which contain Synopsys VIP licenses. Also, include the absolute path to the third-party executable in your PATH variable.

2.6.2.1 Simulator-Specific Settings

Your simulation environment and PATH variables must be set as per the requirement to support your simulator.

2.6.3 Determining Your Model Version

The version of the UFS VIP at the time of publication is O-2018.09. The following steps describes how to verify the version of the models you are using:

Verification IP products are released and versioned by the suite and not by the 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 following setup utility:

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

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

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

2.6.4 Integrating Synopsys IP into Your Testbench

After installing VIP, use the following procedures to set up VIP for use in testbenches:

- Creating a Testbench Design Directory
- The dw_vip_setup Utility

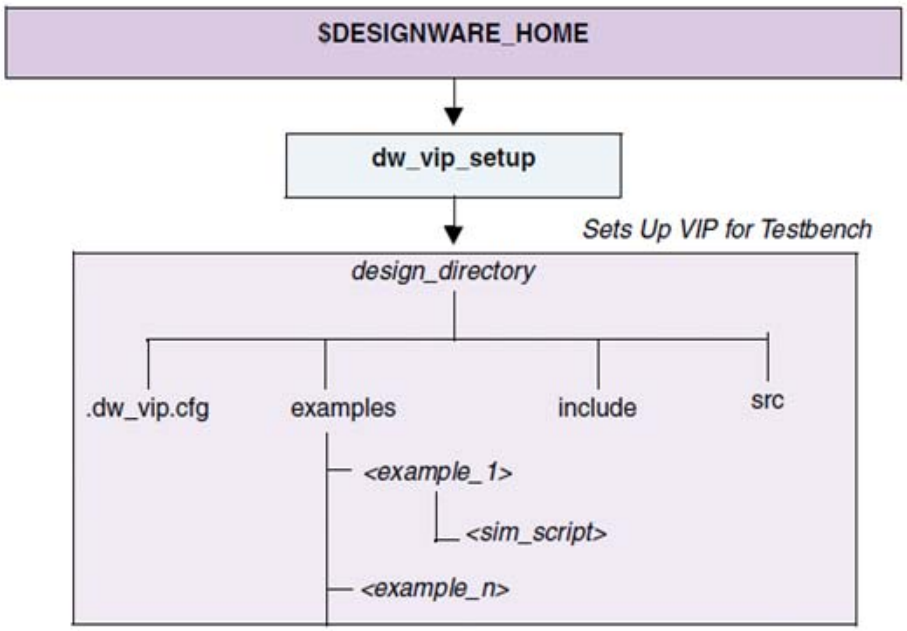
2.6.4.1 Creating a Testbench Design Directory

A design directory contains a version of VIP that is set up and ready to use in a testbench. The dw_vip_setup utility is used to create design directories. For more information on dw_vip_setup, see The dw_vip_setup Utility.

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

A design directory gives you the control over the version of VIP in your testbench as it is isolated from the DESIGNWARE_HOME installation. You can use dw_vip_setup to update the Synopsys VIP in your design directory. Figure 2-2 displays the process and the contents of a design directory.

Figure 2-2 Design Directory Created by dw_vip_setup



A design directory comprises of the following components:,

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 the files required for model, suite, and system testbenches.
include	Language-specific include files that contain critical information for VIP models. This directory is specified in simulator command lines.
src	VIP-specific include files (not used by all VIP). This directory might be specified in simulator command lines.
.dw_vip.cfg	A database of all the VIP models being used in the testbench. The dw_vip_setup program reads this file to rebuild or recreate a design setup.

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

2.7 Setting Up a New VIP

Once you have installed the VIP, you must set up the VIP for project and test bench use. All VIP suites contain various components such as transceivers, masters, slaves, and monitors depending on the protocol. The setup process gathers 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 MIPI UFS VIP contains the following components:

- svt_jedec_ufs_host_agent: Represents the UFS host agent component
- svt_jedec_ufs_host_protocol_transport_monitor: Represents the UFS host monitor
- svt_jedec_ufs_host_protocol_transport_active_common : Represents the UFS host driver component
- svt_jedec_ufs_device_agent: Represents the UFS device agent component
- svt_jedec_ufs_device_protocol_transport_monitor: Represents the UFS device component
- svt_jedec_ufs_device_protocol_transport_active_common: Represents the UFS device driver component

Note Some components are top level and they set up the entire suite. You have the choice to set up the entire suite, or just one component such as a monitor.

Attention There must be only one `design_dir` installation per simulation, regardless of the number of Synopsys Verification and Implementation IPs you have in your project. It is recommended not to create this directory in `$DESIGNWARE_HOME`.

2.8 Installing and Setting Up More than One VIP Protocol Suite

All VIPs for a 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 as `design_dir`, but you can use any name. In this example, assume you have the AXI-suite setup in the `design_dir` directory. In addition to the AXI VIP, you require the Ethernet and UFS VIP suites.

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

Once installed, you must set up and locate the Ethernet and UFS suites in the same `design_dir` location as AMBA. Use the following commands:

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

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

//Add UFS to the same directory as AMBA and Ethernet.
%unix $DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/design_dir -add
jedec_ufs_host_agent_svt -svlog
```

To specify other model names, consult the VIP documentation.

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

2.9 Updating an Existing Model

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

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

   ```
   %unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path  /tmp/design_dir
   -add  <model_name>_svt -svlog
   ```
3. You can also update your design_dir by specifying the version number of the model.

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

The following sections contain three examples to show common usage scenarios:

- Adding or Updating VIP Models In a Design Directory
- Removing Synopsys VIP Models from a Design Directory
- Reporting Information About DESIGNWARE_HOME or a Design Directory

2.9.1 Adding or Updating VIP Models In a Design Directory

UFS VIP models include the following:

- svt_jedec_ufs_host_agent
- svt_jedec_ufs_device_agent

The following example adds a UFS VIP model to a design directory:

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

The following example updates a UFS VIP model in a design directory:

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

In these examples, the dw_vip_setup utility

1. Creates an include directory under the current directory and copies

- All files in the `svt_jedec_ufs_host_agent` model include directory
- All include files in the VIP suite
- The latest SVT library include files into the include directory

2. Creates the UFS VIP suite libraries and SVT libraries

2.9.2 Removing Synopsys VIP Models from a Design Directory

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

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

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

2.9.3 Reporting Information About DESIGNWARE_HOME or a Design Directory

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

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

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

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

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

2.9.4 Running the Example with +incdir+

In the current setup, you install the VIP under `DESIGNWARE_HOME` followed by creation of a design directory which contains the versioned VIP files. With every newer version of the already installed VIP requires the design directory to be updated. This results in:

- Consumption of additional disk space
- Increased complexity to apply patches

The new alternative approach of directly pulling in all the files from `DESIGNWARE_HOME` eliminates the need for design directory creation. VIP version control is now in the command line invocation.

The following code snippet shows how to run the basic example from a script:

```
cd <testbench_dir>/examples/sverilog/jedec_ufs_svt/tb_jedec_ufs_svt_uvm_basic_sys/
// To run the example using the generated run script with +incdir+
./run_jedec_ufs_svt_uvm_basic_sys -verbose -incdir nop_out_cmd vcsvlog
```

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

```
vcs -l ./logs/compile.log -q -Mdir=./output/csrc
+define+DESIGNWARE_INCDIR=<DESIGNWARE_HOME> \
+define+SVT_LOADER_UTIL_ENABLE_DWHOME_INCDIRS \
+incdir+<DESIGNWARE_INCDIR>/vip/svt/jedec_ufs_svt/<vip_version>/sverilog/include \
+incdir+<DESIGNWARE_INCDIR>/vip/svt/mipi_unipro_svt/<vip_version>/sverilog/include \
```

```
+incdir+<DESIGNWARE_INCDIR>/vip/svt/mphy_svt/<vip_version>/sverilog/include \
-ntb_opts uvm -full64 -sverilog +define+SVT_MPHY_SERIAL_INTERNAL_CLK=1
+define+UVM_DISABLE_AUTO_ITEM_RECORDING \
+define+UVM_PACKER_MAX_BYTES=1500000 -timescale=10ps/1ps +warn=noFCICIO
+define+SVT_UVM_TECHNOLOGY +define+SYNOPSYS_SV \
+incdir+<testbench_dir>/examples/sverilog/jedec_ufs_svt/tb_jedec_ufs_svt_uvm_basic_sys/
. \
+incdir+<testbench_dir>/examples/sverilog/jedec_ufs_svt/tb_jedec_ufs_svt_uvm_basic_sys/
../../env \
+incdir+<testbench_dir>/examples/sverilog/jedec_ufs_svt/tb_jedec_ufs_svt_uvm_basic_sys/
../env \
+incdir+<testbench_dir>/examples/sverilog/jedec_ufs_svt/tb_jedec_ufs_svt_uvm_basic_sys/
env \
+incdir+<testbench_dir>/examples/sverilog/jedec_ufs_svt/tb_jedec_ufs_svt_uvm_basic_sys/
dut \
+incdir+<testbench_dir>/examples/sverilog/jedec_ufs_svt/tb_jedec_ufs_svt_uvm_basic_sys/
hdl_interconnect \
+incdir+<testbench_dir>/examples/sverilog/jedec_ufs_svt/tb_jedec_ufs_svt_uvm_basic_sys/
lib \
+incdir+<testbench_dir>/examples/sverilog/jedec_ufs_svt/tb_jedec_ufs_svt_uvm_basic_sys/
tests \
-o ./output/simvcssvlog -f top_files -f hdl_files
```

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

2.9.5 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_jedec_ufs_svt_uvm_basic_sys
usage:  run_jedec_ufs_svt_uvm_basic_sys [-32][-incdir][-verbose] [-debug_opts] [-
waves] [-clean] [-nobuild] [-buildonly] [-norun] [-pa] <scenario> <simulator>
where   <scenario> is one of:   all backdoor_mem_write_read_serial base_test
dme_peer_communication_failure_serial host_init_link_start_up_rmmi
link_start_up_rmmi link_start_up_serial nop_out_cmd_hs_serial nop_out_cmd_rmmi
nop_out_cmd_serial nop_wr6_rd6_hs_rmmi reset_link_up_rmmi
rtt_delay_control_through_callback_serial
test_mode_config_change_peer_set_get_serial test_mode_config_change_rmmi
test_mode_linkup_phase_serial test_mode_seq_serial ufs_um_data_integrity_rmmi
ufs_um_data_integrity_serial wr10_rd10_rmmi wr6_rd6_serial
<simulator> is one of:  vcsmxvlog mtivlog vcsvlog vcszsimvlog vcsscvlog ncvlog
vcszebuvlog vcsmxpcvlog vcsvhdl vcsmxpipvlog ncmvlog vcspcvlog
-32          forces 32-bit mode on 64-bit machines
-incdir      use DESIGNWARE_HOME include files instead of design directory
-verbose     enable verbose mode during compilation
-debug_opts  enable debug mode for VIP technologies that support this option
```

```
-waves       [fsdb|verdi|dve|dump] enables waves dump and optionally opens viewer
(VCS only)
-seed        run simulation with specified seed value
-clean       clean simulator generated files
-nobuild     skip simulator compilation
-buildonly   exit after simulator build
-norun       only echo commands (do not execute)
-pa          invoke Verdi after execution
```

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

```
gmake help
Usage: gmake USE_SIMULATOR=<simulator> [VERBOSE=1] [DEBUG=1] [FORCE_32BIT=1]
[WAVES=fsdb|verdi|dump] [NOBUILD=1] [PA=1] [<scenario> ...]
Valid simulators are: vcsvlog vcsmxvlog mtivlog vcsmxpcvlog vcsmxpipvlog ncvlog
vcspcvlog vcsscvlog vcsvhdl ncmvlog
Valid scenarios are: all backdoor_mem_write_read_serial base_test
dme_peer_communication_failure_serial host_init_link_start_up_rmmi
link_start_up_rmmi link_start_up_serial nop_out_cmd_hs_serial nop_out_cmd_rmmi
nop_out_cmd_serial nop_wr6_rd6_hs_rmmi reset_link_up_rmmi
rtt_delay_control_through_callback_serial
test_mode_config_change_peer_set_get_serial test_mode_config_change_rmmi
test_mode_linkup_phase_serial test_mode_seq_serial ufs_um_data_integrity_rmmi
ufs_um_data_integrity_serial wr10_rd10_rmmi wr6_rd6_serial
```

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

2.9.6 The dw_vip_setup Utility

The dw_vip_setup utility are as follows:

- Adds, removes, or updates VIP models in a design directory
- Adds example testbenches to a design directory in the VIP models used (if necessary), and creates a script for simulating the testbench using anyone of the supported simulators
- Restores (cleans) example testbench files to their original state
- Reports information about your installation or design directory, including version information
- Supports Protocol Analyzer (PA)
- Supports the FSDB wave format

2.9.6.1 Setting Environment Variables

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

- DESIGNWARE_HOME – Points to the location where the VIP is installed

2.9.6.2 The dw_vip_setup Command

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

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

or

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

where

[**-p**[ath] *directory*] The optional -path argument specifies the path to your design directory. When omitted, dw_vip_setup uses the current working directory.

switch The *switch* argument defines dw_vip_setup operation. Table 2-1 lists the switches and their applicable sub-switches.

Table 2-1 Setup Program Switch Descriptions

Switch	Description
-a[dd] (m*odel* [**-v**[ersion] *version*]) …	Adds the specified model or models to the specified design directory or current working directory. If you do not specify a version, the latest version is assumed. The model names are as follows: • svt_jedec_ufs_host_agent • svt_jedec_ufs_device_agent The -add switch causes dw_vip_setup to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-r[emove] *model*	Removes ***all versions*** of the specified model or models from the design. The dw_vip_setup program does not attempt to remove any include files used solely by the specified model or models. The model names are as follows: • svt_jedec_ufs_host_agent • svt_jedec_ufs_device_agent
-u[pdate] (*model* [**-v**[ersion] *version*]) …	Updates to the specified model version for the specified model or models. The dw_vip_setup script updates to the latest models when you do not specify a version. The model names are as follows: • svt_jedec_ufs_host_agent • svt_jedec_ufs_device_agent The -update switch causes dw_vip_setup to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-e[xample] {*scenario* | *model*/*scenario*} [**-v**[ersion] *version*]	The dw_vip_setup script configures a testbench.For example single model or a system testbench for a group of models. The program creates a simulator run program for all the supported simulators. If you specify a *scenario* (or system) example testbench, the models required for the testbench are included automatically and is not specified in the command. **Note:** Use the -info switch to list all the available system examples.
-ntb	Not supported.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-svtb	Use this switch to set up models and example testbenches for SystemVerilog UVM. The resulting design directory is streamlined and is used in SystemVerilog simulations.
-c[lean] {*scenario* | *model*/*scenario*}	Cleans the specified scenario or testbench in either the design directory (as specified by the *-path* switch) or the current working directory. This switch deletes *all files in the specified directory*, then restores all Synopsys created files to their original contents.
-i[nfo] *design* | *home* [:<product>[:<version>[:<methodology>]]]	Generates an informational report on a design directory or VIP installation. design: If the -info design switch is specified, the tool displays product and version content within the specified design directory to standard output. This output can be captured and used as a model list file, as an input to this tool to create another design directory with the same content. home: If the -info home switch is specified, the tool displays product, version and example content within the VIP installation to standard output. Optional filter fields can also be specified such as <product>, <version> and <methodology> delimited by colons (:). An error will be reported if a nonexistent or invalid filter field is specified. Valid methodology names include: OVM, RVM, UVM, VMM and VLOG.
-h[elp]	Returns a list of valid dw_vip_setup switches and the correct syntax for each.
model	UFS VIP models are: • svt_jedec_ufs_host_agent • svt_jedec_ufs_device_agent The *model* argument defines the model or models that dw_vip_setup is performed. This argument is not required with the -info or -help switches. All switches that require the *model* argument might use a listed model. You might specify a version for each listed *model*, using the -version option. If omitted, dw_vip_setup uses the latest version. The -update switch ignores *model* version information.
-m[odel_list] *filename*	Specifies a file name, which contains a list of suite names to be added, updated, or removed from the design directory. This switch is valid during the following switch operations; for example, -add, -update, or -remove. The -m/odel_list switch displays one model name per line and each model includes a version selector. The default version is the latest. This switch is optional, but the filename argument is required whenever mentioned. The lines in the file starting with the pound symbol (#) are ignored.
-s/uite_list <filename>	Specifies a file name, which contains a list of suite names to be added, updated, or removed from the design directory. This switch is valid during the following switch operations; for example, -add, -update, or -remove. The -s/uite_list switch displays one suite name per line and each suite includes a version selector. The default version is the latest. This switch is optional, but the filename argument is required whenever mentioned. The lines in the file starting with the pound symbol (#) are ignored.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-pa	Enables the run scripts and Makefiles generated by dw_vip_setup to support PA. If this switch is enabled, and the testbench example produces XML files, PA will be launched and the XML files will be read at the end of the example execution. For run scripts, specify `-pa`. For Makefiles, specify `-pa = 1`.
-waves	Enables the run scripts and Makefiles generated by dw_vip_setup to support the `fsdb` waves option . To support this capability, the testbench example must generate an FSDB file when compiled with the WAVES Verilog macro set to `fsdb`, that is, `+define+WAVES=\"fsdb\"`. If a .fsdb file is generated by the example, the Verdi nWave viewer will be launched. For run scripts, specify `-waves fsdb`. For Makefiles, specify `WAVES=fsdb`.
-doc	Creates a doc directory in the specified design directory which is populated with symbolic links to the `DESIGNWARE_HOME` installation for documents related to the given model or example being added or updated.
-methodology <name>	When specified with -doc, only documents associated with the specified methodology name are added to the design directory. Valid methodology names include: OVM, RVM, UVM, VMM and VLOG.
-copy	When specified with -doc, documents are copied into the design directory, not linked.
-simulator <vendor>	When used with the `-example` switch, only simulator flows associated with the specified vendor are supported with the generated run script and Makefile. **Note**: Currently the vendors VCS, MTI, and NCV are supported.

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

2.10 Include and Import Model Files into Your Testbench

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

```
/* include uvm package before VIP includes. */
`include "svt_mipi_unipro.uvm.pkg"
/** Import UVM package */
import uvm_pkg::*;
`include "uvm_macros.svh"
/** Import the SVT UVM package */
import svt_uvm_pkg::*;
import svt_mphy_component_uvm_pkg::*;
```

```
import svt_mipi_unipro_uvm_pkg::*;
`include "svt_jedec_ufs_host_agent_source.uvm.svi"
`include "svt_jedec_ufs_device_agent_source.uvm.svi"
```

You must also include various VIP directories on the simulator's 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 design_dir directory would be /tmp/design_dir.

2.11 Compile-Time and Runtime Options

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

```
$DESIGNWARE_HOME/vip/svt/jedec_ufs_svt/test/sverilog/tb_jedec_ufs_svt_uvm_basic_sys/
```

The following files contain the options:

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

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

vcs_build_options

```
Required:   +define+UVM_PACKER_MAX_BYTES=66000
Required:   +define+UVM_DISABLE_AUTO_ITEM_RECORDING
Optional:   -timescale=1ns/1ps
Required:   +define+SVT_<model>_INCLUDE_USER_DEFINES
Required:   +define+SYNOPSYS_SV
```

Note

UVM_PACKER_MAX_BYTES define needs to be set to maximum value as required by each VIP title in your testbench. For example, if VIP title 1 needs UVM_PACKER_MAX_BYTES to be set to 8192, and VIP title 2 needs UVM_PACKER_MAX_BYTES to be set to 500000, you need to set UVM_PACKER_MAX_BYTES to 500000.

vcs_run_options

Required: +UVM_TESTNAME=$scenario

Note

scenario is the uvm test name you pass to VCS.

3 General Concepts

3.1 Introduction to UVM

The UFS VIP suite is a set of SVT -based verification components, intended to be used in System Verilog UVM compliant testbenches. The VIP suite infrastructure builds upon System Verilog and adds to UVM, and is used to create a common look and feel among VIP suites. This common infrastructure is referred as the Synopsys SVT toolkit. The UFS VIP suite is built using this toolkit. This document explains the components and usage information for the components in the UFS VIP suite.

For more information,

- For the IEEE SystemVerilog standard,see the following:
  - IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language
- For an essential reference guide describing UVM as it is represented in SystemVerilog, along with a class reference, see the following:
  - *Universal Verification Methodology (UVM) User's Manual* at: http://www.accellera.org/
- See the Class Reference HTML for a description of attributes and properties of the objects mentioned in this chapter.

This document describes the following elements of the UFS VIP suite:

- UFS VIP Components
- UFS Data Objects
  - Configuration Objects
  - Status Objects
  - Transaction Objects
- Interfaces and Modports
- Component Protocol Checkers
- Sequence Collection
- Callbacks

3.2 UFS VIP in an UVM Environment

UVM system-level verification environment is constructed with an UVM agent which contains the following three UVM components:

- Sequence generator
- Driver class
- Monitor class

The UFS VIP has Host Agent and Device Agent:

- UFS Host Agent: `svt_jedec_ufs_host_agent`
- UFS Device Agent: `svt_jedec_ufs_device_agent`

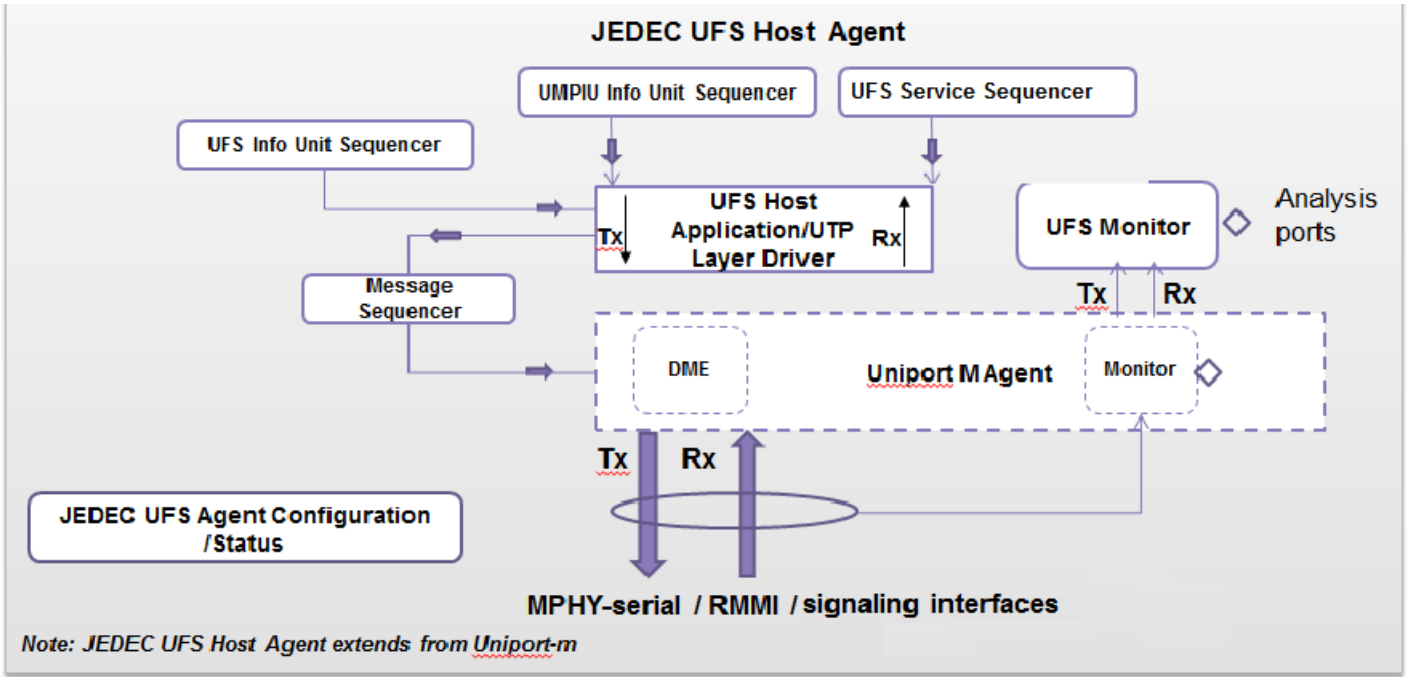
3.3 UFS Host Agent

The UFS Host Agent encapsulates UFS sequencers, UniPro-M agent, Drivers, and forward or reverse link monitors.

The agent can be configured to operate in active or passive mode. You can provide UFS Info Unit sequences to the UFS Info Unit sequencers and UMPIU sequences to the UMPIU Info Unit Sequencer.

Within UFS Agent, the UFS Transport Protocol gets sequences from UFS/UMPIU Info unit sequencers and converts to UniPro message transactions. The UFS Transport Protocol drives UFS Host Agent transactions on to the message sequencer connected to UniPro-M agent.

Figure 3-1 UFS Host Agent



3.4 UFS Host Agent Modes

The UFS Host Agent can be configured in the following modes by setting `is_active` and `enable_monitor` member of the UFS Agent configuration:

- Active mode
  - With Monitor

    This mode is set when `is_active` and `enable_monitor` variables are set to value of "1". In this mode, driver, monitor, and sequencers are created and sequencers are connected to relevant driver.
  - Without Monitor

    This mode is set when `is_active` is set to value of "1" and `enable_monitor` is set to value of "0". In this mode, driver, and sequencers are created and sequencers are connected to the corresponding driver.
- Passive mode

  This mode is set when `is_active` variable is set to value "0". In this mode, only monitor is available as part of the UFS Agent.

3.5 UFS Host Driver

The UFS Host driver defines the UFS Transport Protocol layer driver, which is used to process traffic in the TX and RX directions. For the TX direction, UFS Host driver receives UFS transactions from input TLM port and creates messages which is then delivered to the UniPro message sequencer. For the RX direction, UFS Host driver receives messages from the input TLM ports and creates a packet which is then delivered to the upstream layer (testbench or Host Controller Interface) through an output TLM port.

3.6 UFS Host Agent Monitor

The monitor in the UFS Agent is an UVM component object in an UVM-compliant environment that performs the following functions:

- Checks the UPIU/UMPIU transactions transmitted or received by the Host or Device Agents
- Implements UPIU transaction error checks

Note
Error checks are not supported for UMPIU transactions.

- Provides the UPIU/UMPIU transaction packets to scoreboard for comparison

3.6.1 UFS Host Agent or Monitor Ports

3.6.1.1 Agent Ports (Host Agent)

The following port information is used if a host sequencer has to send the transaction to the host. The sequencer generates the transaction using the `svt_jedec_ufs_info_unit` transaction class and drives into the port.

UFS comprises of three different ports; the Task Management port, Command port, and Device Control. These ports are declared in the host and device driver logic:

- TX Upstream UDM Sequence Item Pull Port:

  SVT_XVM (seq_item_pull_port)#(svt_jedec_ufs_info_unit) udm_tx_iu_in_port;

- TX Upstream UTP CMD Sequence Item Pull Port:

```
`SVT_XVM (seq_item_pull_port)#(svt_jedec_ufs_info_unit) utp_cmd_tx_iu_in_port;
```

- TX Upstream UTP TM TLM Sequence Item Pull Port:

```
`SVT_XVM (seq_item_pull_port)#(svt_jedec_ufs_info_unit) utp_tm_tx_iu_in_port;
```

- TX Upstream UBM TLM sequence Item Pull Port:

```
SVT_XVM(seq_item_pull_port)#(svt_jedec_ufs_unified_mem_protocol_info_unit)
ubm_umpiu_response_port;
```

- RX Upstream UDM TLM Put Port:

```
svt_debug_opts_blocking_put_port#(svt_jedec_ufs_info_unit) udm_rx_iu_out_port;
```

- RX Upstream UTP CMD TLM Put Port:

```
svt_debug_opts_blocking_put_port#(svt_jedec_ufs_info_unit)
utp_cmd_rx_iu_out_port;
```

- RX Upstream UTP TM TLM Put Port:

```
svt_debug_opts_blocking_put_port#(svt_jedec_ufs_info_unit) utp_tm_rx_iu_out_port;
```

- RX Upstream UBM TLM Put Port

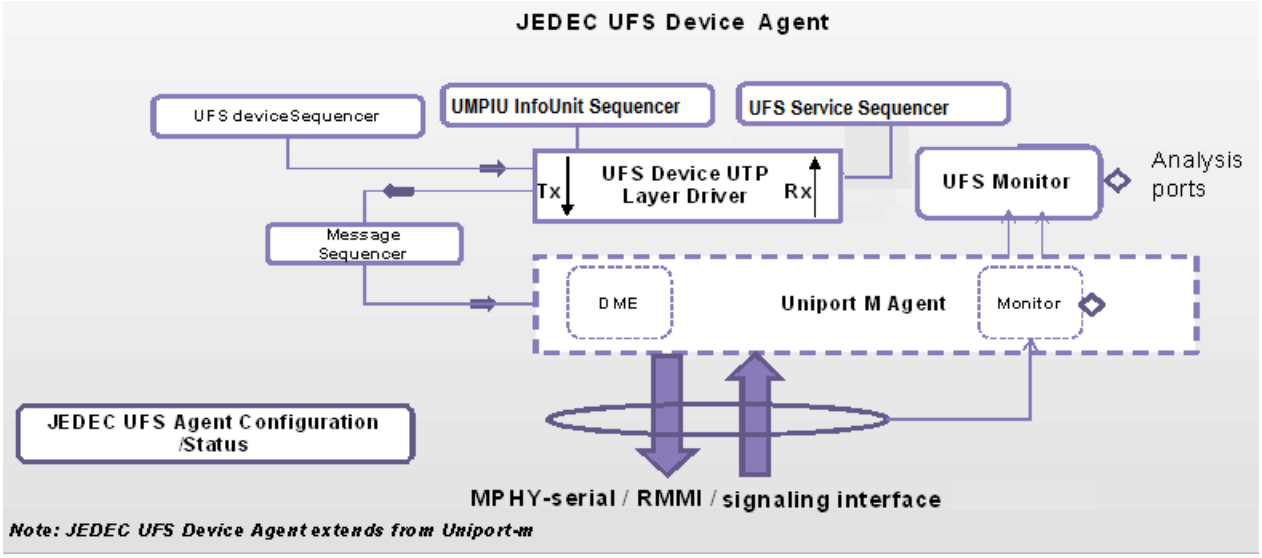
```
svt_debug_opts_blocking_put_port#(svt_jedec_ufs_unified_mem_protocol_info_unit)
ubm_umpiu_out_port;
```

3.7 UFS Device Agent

The UFS Device Agent encapsulates UFS Info Unit sequencers, UniPro-M agent, Drivers, and monitors.

The UFS Agent can be configured to operate in active or passive mode. Within UFS Agent, the UFS Transport Protocol receives the request for UPIU from the UFS Host. Based on the request UPIU, UFS device creates response UPIU and converts it to UniPro message transactions. Later, it drives these transactions on to the message sequencer connected to UniPro-M agent.

Figure 3-2 UFS Device Agent



3.8 UFS Device Agent Modes

The UFS Device Agent can be configured in the following modes by setting `is_active` and `enable_monitor` member of the UFS Agent configuration:

- Active mode:
  - With Monitor

    This mode is set when `is_active` and `enable_monitor` variables are set to value of "1". In this mode, driver, monitor, and sequencers are created and sequencers are connected to relevant driver.
  - Without Monitor

    This mode is set when `is_active` is set to value of "1" and `enable_monitor` is set to value of "0". In this mode, driver, and sequencers are created and sequencers are connected to relevant driver.
- Passive mode:

  This mode is set when `is_active` variable is set to value of "0". In this mode, only monitor is available as part of the UFS Agent.

3.9 UFS Device Driver

UFS Device Driver processes traffic in the TX and RX directions. For the RX direction, the UFS Device Driver receives UniPro messages from input TLM port and creates UPIU transactions. For the TX direction, based on the UPIU transactions received, it creates UPIU transaction and then transmits the corresponding UniPro messages to the message sequencer.

3.10 UFS Device Agent Monitor

The monitor in the UFS Agent is an UVM component object in an UVM-compliant environment which performs the following functions:

- Checks the UPIU/UMPIU transactions transmitted or received by the Host or Device Agents.

Note
Provides the UPIU/UMPIU transaction to scoreboard for comparison.

3.10.1 UFS Device Agent/Monitor Ports

The following ports are used to interact with the Unipro:

- RX Upstream TLM Put Port, provides a mechanism for sending Info Unit that is recognized by the Upper Layer.
  - `svt_debug_opts_blocking_put_port#(svt_jedec_ufs_info_unit) rx_iu_out_port;`
  - `svt_debug_opts_blocking_put_port#(svt_jedec_ufs_unified_mem_protocol_info_unit) ubm_umpiu_out_port;`
  - ```
    `SVT_DEBUG_OPTS_IMP_PORT(blocking_put, svt_mipi_unipro_message,
    svt_mipi_unipro_queue_downstream_imp#(svt_mipi_unipro_message))
    rx_msg_in_port[`SVT_JEDEC_UFS_CPORTS_USED];
    ```

3.11 Host UFS Data Objects

UFS VIP defines several classes designed for an UVM environment. This section introduces the major UFS VIP objects. The UFS VIP classes extend base classes to handle specific needs of the protocol and provide predefined constraints. The predefined constraints can be used "as is" to produce a wide range of stimulus, or extends to create specific test conditions. An object and its constraints are referred to as a factory object, or factory when used to control the production of, or randomization of a transaction data object. The sequencers create sequences, use factories to create streams of randomized objects. The sequencers are responsible for creating sequence items based on the factories.

The following sections explains the UFS VIP objects:

- Configuration Objects
- Status Objects
- Transaction Objects

3.11.1 Configuration Objects

The configuration data objects convey the component setup information. These data objects contain built-in constraints which come into effect when the configuration objects are randomized. If the configuration needs to be changed later, it can be done through the reconfigure method on the agent.

The following are the types of configuration:

- Static configuration properties: Static configuration parameters specify a configuration value that cannot be changed when the system is functional. For example, coverage enable.
- Dynamic configuration properties: Dynamic configuration parameters specify a configuration value that can be changed at any time, regardless of the system being functional or not.

The configuration data objects contain built-in constraintsthat are effective when the configuration objects are randomized.
    ```

The UFS VIP defines the following configuration class:

- **UFS Agent Configuration (svt_jedec_ufs_agent_configuration)**:
  UFS Agent configuration data and methods are used to configure the UFS Agent. The agent configuration class controls the internal components that are constructed.

  The UFS Agent Configuration configures the following parameters:

  - Interface type M-PHY Serial or RMMI
  - Active or Passive Component
  - Monitor enable or disable
  - Checker enable or disable
  - Coverage enable or disable
  - Exceptions enable or disable
  - Reporting enable or disable
  - Tracing enable or disable
  - XML generation enable or disable
- **UFS Device Configuration (svt_jedec_ufs_device_configuration)**:
  UFS Device configuration data and methods are used to configure the protocol features of the UFS Device components. This object contains randomizable member variables as well as valid and reasonable constraints on these member variables.
- **UFS Device LUN Configuration (svt_jedec_ufs_lun_configuration)**
  UFS Device LUN configuration data and methods are used to configure the protocol features of the UFS Logical Unit components. This object contains randomizable member variables as well as valid and reasonable constraints on these member variables.

Note
Configuration data objects are extended from the `svt_configuration` class, which is extended from the uvm_sequence_item base class. These objects implement all the methods specified for the uvm_sequence_item class.

3.11.2 Configuration Classes

3.11.2.1 svt_jedec_ufs_agent_configuration Class

The `svt_jedec_ufs_agent_configuration` configuration class defines the following:

- Enable attributes, for example, enable UFS layer tracing => enable_protocol_transport_tracing.
- Defines the attributes that are specified in the User Conf as No. For example, LUQueueDepth in Table 14-9 Unit Descriptor of the UFS specification 2.1, is a part of the `svt_jedec_ufs_lun_configuration` class. Such attributes are defined in three different classes:
  - `svt_jedec_ufs_protocol_transport_configuration` and contained classes
  - `svt_jedec_ufs_device_configuration`
  - `svt_jedec_ufs_lun_configuration`

3.11.2.2 svt_jedec_ufs_protocol_transport_configuration Class

The `svt_jedec_ufs_protocol_transport_configuration` class is contained as part of `svt_jedec_ufs_agent_configuration` class.

The agent configuration class also includes an instance of the shared status class. The configuration class reflects the initial setting of the attributes (before the start of the simulation), and subsequently may or may not reflect the current setting.

3.11.2.3 svt_jedec_ufs_status Class

The `svt_jedec_ufs_status` class defines the attributes that are specified in the User Conf as Yes.

For example, bLogicalBlockSize in Table 14-10 Unit Descriptor of UFS specification 2.1 is a part of the `svt_jedec_ufs_lun_status` class.

The `svt_jedec_ufs_status` class follows a similar structure as the configuration class and contains the `svt_jedec_ufs_protocol_transport_status` class. Such attributes are defined in three different classes:

- `svt_jedec_ufs_protocol_transport_status` and contained classes
- `svt_jedec_ufs_device_status`
- `svt_jedec_ufs_lun_status`

The status class attributes are READ ONLY and reflect the current setting. Only the VIP internally updates them based on the protocol communication or state.

3.11.3 Status Objects

Status objects provides the state information of the various components in the UFS VIP. This information is provided through data and event fields.

The UFS VIP supports the following status classes:

- `svt_jedec_ufs_status class`
- `svt_jedec_ufs_device_status`
- `svt_jedec_ufs_lun_status`
- `svt_jedec_ufs_protocol_transport_status`

3.11.4 Transaction Objects

Transaction objects, which are extended from `svt_sequence_item` base class, defines UFS objects which are designed to carry protocol defined transactions from, to, or within the UFS VIP. The attributes of transaction objects are public and are accessed directly to set and get values. Most of the transaction attributes can be randomized. The transaction object represents the desired activity to be simulated on the bus, or the actual activity that is monitored.

UFS transaction data objects are used to:

- Generate random stimulus
- Report observed transactions
- Generate random response to transaction requests
- Collect functional coverage statistics

The class properties are public and accessed directly to set and read values. The transaction data objects support randomization and provide built-in constraints.

- valid_ranges constraints limit generated values to those acceptable to the drivers. These constraints ensure basic VIP operation and should never be disabled.

- reasonable_* constraints, can be disabled individually or as a block, limits the simulation by the following methods:
  - Enforcing the protocol:These constraints are typically enabled unless errors are being injected into the simulation.
  - Setting simulation boundaries:Disabling these constraints might slow the simulation and introduce system memory issues.

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

Individual reasonable_* constraints map to independent fields, each of which can be disabled. The class provides the reasonable_constraint_mode() method to enable or disable blocks of reasonable_* constraints.

The transaction data classes also contain a list of exceptions to generate packets with certain error conditions. The exception list is null by default and can be populated manually or randomized and then associated with a transaction object.

UFS VIP defines the following transaction classes:

- **UFS Transaction Class (svt_jedec_ufs_info_unit)**

  This transaction class represents the CPC or CCI transactions and contains attributes such as Pdu Type, Transfer Size, Attribute Id and so on. This class is used by all the UFS Transmitter and Receiver VIP components such as drivers, monitors, functional coverage, and scoreboard.

- **UFS SCSI Transaction Class (svt_jedec_ufs_scsi_command)**

  This transaction class represents the UFS SCSI commands and contains attributes such as operation code, page_code, logical block address, logical length and so on. This class is embedded within `svt_jedec_ufs_info_unit` class.

- **UFS UMPIU Transaction Class (svt_jedec_ufs_unified_mem_protocol_info_unit)**

  This transaction class represents the UFS UMPIU commands and contains attributes such as UM ID, source UM area offset, Target UM area offset, `data_segment_length` among other attributes.

For more information on attributes of the transaction classes and their default values, see the Class Reference HTML documentation.

3.11.5 Interfaces and Modports

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

JEDEC Host and Device Agents provide the SystemVerilog interface required to connect the VIP to a DUT. The interface, `svt_jedec_ufs_if`, which contains M-Tx and M-Rx interfaces are used to connect to M-Rx and M-Tx of DUT interfaces respectively.

JEDEC Host and Device Agents additionally provide the SystemVerilog interface for the transport layer CPort signaling. The interface is `svt_jedec_ufs_cport_non_mux_if`that contains the non-multiplexed CPort signals and also a debug port for debugging messages. For more information on interfaces, see the Class Reference HTML documentation.

3.11.6 Component Protocol Checkers

The VIP component uses an object-based mechanism for defining and encapsulating checks dynamically performed by the components. Such an object oriented approach is useful for controlling checks and for

functional coverage of check execution and outcome. Thus, the classes explained in this section are the "container" classes (known to their associated components) within which these structured checks are defined and controlled.

- UFS host/device transport layer checks (svt_jedec_ufs_protocol_transport_err_check)

By default, the checkers are enabled. For the list of checks, see the Class Reference HTML documentation.

3.11.7 Callbacks

The callbacks are an access mechanism that enable the insertion of the user-defined code and allow access to objects for scoreboarding and functional coverage. Each UFS component has an associated callback class that contains a set of callback methods. UFS VIP supports the following callback classes.

- `svt_jedec_ufs_host_protocol_transport_callback`
- `svt_jedec_ufs_host_protocol_transport_monitor_callback`
- `svt_jedec_ufs_host_protocol_transport_monitor_def_cov_callback`
- `svt_jedec_ufs_host_protocol_transport_monitor_def_cov_data_callback`
- `svt_jedec_ufs_device_protocol_transport_callback`
- `svt_jedec_ufs_device_protocol_transport_monitor_callback`
- `svt_jedec_ufs_device_protocol_transport_monitor_def_cov_callback`
- `svt_jedec_ufs_device_protocol_transport_monitor_def_cov_data_callback`

For more information on Callbacks, see the Class Reference HTML documentation.

3.11.8 Sequence Collection

UFS VIP provides a collection of UFS UPIU/UMPIU sequences that are applicable only for the UFS host. These sequences can be registered with the sequencer within the host agent to generate different types of UFS UPIU/UMPIU sequences and the public attributes are overwritten by using the `uvm_config_db#()::set` function from the test case.

All the UFS sequences are extended from the base sequence `svt_jedec_ufs_info_unit_base_sequence`

All the UMPIU sequences are extended from the base sequence

`svt_jedec_ufs_unified_mem_protocol_info_unit_base_sequence`

The UMPIU sequence collection from the host is a response sequence collection, as the host responds to UMPIU requests from device.

For a list of all the host sequences and their public attributes, see the Class Reference HTML documentation.

3.12 Functional Coverage

UFS Verification IP coverage implementation has the following types of functional coverage:

- UFS coverage
- Unipro coverage
- MPHY coverage

3.12.1 Coverage Features

Transport Layer Coverage - UPIU: transaction code, flags, task_tag, data segment, CDB, response, Scsi status, sense data, task management function, task management response, query function, query response, andinput and output parameters.

3.12.2 Covergroups Organization

- Basic Header :
  - hd
  - dd
  - transaction_code
  - task_tag
  - wlun_id
  - lun_id
  - boot_lun
  - trans_code_lun
  - command_set_type
  - total_ehs_length
- Command UPIU
  - flags_r
  - flags_w
  - flags_attr
  - read_attr
  - write_attr
  - command_data_segment_length
  - expected_data_transfer_length
  - cdb_opcode
  - command_cdb_attr
  - Rdwr6_logical_block_addr
  - rdwr10_logical_block_addr
  - rdwr16_logical_block_addr
  - rdwr6_transfer_length
  - rdwr10_transfer_length
  - rdwr16_transfer_length
  - rdwr_dpo
  - rdwr_fua
  - rdwr_fua_nv
  - rdwr_group_number

- scsi_lun

- Response UPIU
  - flags_o
  - flags_u
  - flags_d
  - response_data_segment_length
  - response
  - residual_transfer_count
  - overflow_residual_transfer_count
  - underflow_residual_transfer_count
  - response_upiu
  - scsi_status
  - sense_data_length
  - sense_data_valid
  - sense_data_filemark
  - sense_data_eom
  - sense_data_ili
  - sense_data_information
  - sense_key
  - sense_data_additional_sense_length
  - sense_data_command_specific_information
  - sense_data_additional_asc
  - sense_data_additional_ascq
  - sense_data_additional_fruc
  - sense_data_sense_key_specific
  - sense_data_response_code
- DATA OUT/IN and RTT UPIU
  - data_buffer_offset
  - data_transfer_count
  - buffer_offset
  - transfer_count
  - data_segment
- Task Management Request or Response UPIU:
  - tm_data_segment_length
  - task_manag_function
  - input_parameter_1

- input_parameter_2
- task_mgmnt_req
- task_management_service_response

- Query Request or Response UPIU:
  - query_data_segment_length
  - query_function
  - opcode
  - query_response
  - opcode_query_function
  - opcode_query_response
  - descriptor_idn
  - attribute_idn
  - flag_idn
  - query_descriptor_req
  - query_attribute_req
  - query_flag_req
  - osf_index
  - osf_selector
  - osf_length
  - osf_value
  - osf_flag_value
- Reject :
  - reject_data_segment_length
  - reject_response
  - basic_header_status
  - e2e_status

3.12.3 Enable or Disable Functional Coverage

The default functional coverage is enabled by setting the attributes in the UFS agent configuration class.

```
svt_jedec_ufs_agent_configuration::enable_protocol_transport_cov=1 (TL coverage)
```

3.13 High Level Verification Plans

The high level verification plans are provided for UFS verification topologies.

The top level verification plans are found after installation at

```
$DESIGNWARE_HOME/vip/svt/jedec_ufs_svt/latest/doc/VerificationPlans/ ufs-host-verification-plan
```

You can back annotate the HVP with UVM planner using the following command:

```
hvp annotate -plan < Top level xml plan > -dir < combined vdb dir >
```

Note

Top level XML plan when Host is the DUT - svt_ufs_host_dut_top_level_plan.xml Device is the DUT - svt_ufs_device_dut_top_level_plan.xml.

Figure 3-3 provides the snapshot of top-level verification plan.

Figure 3-3 Verification Plan

hvp plan svt_ufs_host_dut_toplevel_fc_plan	include	value jedec_ufs_svt.Group	$reference	Feature	subfeature		subfeature	subfeature	measure jedec_ufs_svt.source	$comment
	svt_ufs_host_tx_toplev									Subplan for UFS
	svt_ufs_host_rx_sublink									Subplan for UFS
	unipro_device_A_dut_t		jesd220_ufs_specification_v1.1							
skip	NOTE: ufs_host_dut_toplevel_fc_plan embeds the below 1. UTP layer coverage : Consists of UFS coverage 2. MIPI Unipro layer coverage : Consists of MIPI-Unipro data coverage 3. MIPI MPHY layer cover Consists of MPHY layer coverage 4. Coverage categorized into Transmit direction coverage (Tx)& Receive direction coverage (Rx) 5. Similar strategy will be employed on the UFS device side, which will b covered in next doc release 6. Next Doc release will be having more intricate details of the coverage strategy									

Figure 3-4 provides the snapshot of sub-plan of the verification plan.

Figure 3-4 Sub-Plan of Verification Plan

skip				HOST UFS Coverage					
skip					Tx Sublink				
skip							UFS layer coverage		
		19.41%						subplan svt_ufs_host_tx_toplevel_sublink	
skip					Rx Sublink				
skip							UFS layer overage		
		13.36%						subplan svt_ufs_host_rx_sublink #(INST_GROUP_COV_attr="instance",I	
skip				Unipro Toplevel					
skip								subplan unipro_device_A_dut_toplevel_fc_plan	

4 Using the UFS Verification IP

4.1 Introduction

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

4.2 Creating a Test Environment

4.2.1 Base Class: uvm_env

The `uvm_env` base class provides a testbench template to organize both flow and the code associated with a test. You can create a test environment by defining a new class extended from `uvm_env`. The environment class contains (or affects) the entire test environment, as it provides an overall structure. The majority of the code in an UVM testbench is contained in the environmentthat can be reused by other tests or projects. Because of this, the `uvm_env` class has a large impact on the user coding, thus understanding the `uvm_env` class is central for writing the environment code effectively. Another result of the over arching nature of uvm_env helps in organizing the discussion of UVM and UFS VIP techniques and methods. Therefore the subjects presented here are organized and presented in the context of `uvm_env`.

The `uvm_env` base class has several methods that correspond to the major steps followed by all the tests.

Most of these methods are declared in the base class as virtual and should be extended by the user. To create a user environment, define a new class extended from `uvm_env` and extend the methods above to add test-specific code. To retain the core functionality of the base class methods, each extended method must call super method as the first line of code. The following code snippet shows the user extension of `uvm_env`, creating the class user_env. The `user_env` class instantiates all the components of the testing environment, which might include VIP products, user designs, generators, scoreboards, and so on. At the end of this extended class is a list of the methods that are extended.

```
////////////////////////////////////////////////////////////////////////
// Verification Environment
////////////////////////////////////////////////////////////////////////
class ufs_system_env extends uvm_env;
// Instantiate the objects that compose the environment
// JEDEC UFS SVT VIP models
  /**
```

```
   * An instance of VIP Host Agent
   */
  svt_jedec_ufs_host_agent host;
/**
   * An instance of VIP Device Agent
   */
  svt_jedec_ufs_device_agent dev;
/**
   * An instance of VIP Host Agent to act as UFS Host monitor
   */
  svt_jedec_ufs_host_agent pssv_host;
/**
* An Instance of VIP Device Agent to act as UFS Device monitor
  */
    svt_jedec_ufs_device_agent pssv_dev;
/**
* A parameter to enable/disable the passive agent on host and device
   */
  int enable_passive_agent = 0;
  /**
   * System configuration object reference.
   * This object is of a custom type, which contains sub-objects for the
   * configuration of the individual Jedec UFS stacks for both the hst
   * and the dev
   */
  ufs_system_configuration sys_cfg;
// List of methods in this class
extern function new (uvm_phase phase);
extern virtual function build_phase(uvm_phase phase);
extern virtual function connect_phase(uvm_phase phase);
extern virtual function void end_of_elaboration_phase(uvm_phase phase);  extern virtual
task main_phase(uvm_phase phase);
endclass: ufs_system_env
```

When the run_test method is invoked from the testbench top, the methods of the uvm_env and other uvm_components are invoked automatically in the following order.

```
build_phase -> connect_phase -> run_phase
```

Calling this one method (`run_test`) launches the entire test sequence inside a testbench top module:

```
initial
    begin
      run_test();
```

```
    $finish;
```

Since the testbench top has space constraint while using UVM , majority of the code in the environment is a reusable component. The test-specific code is minimized and is kept common so it is not replicated . This yields a less code base to maintain.

The user inherits structure and base functionality from the `uvm_env` classthat is customizable. In addition, the customization is controlled by the user. This is a common theme throughout UVM and the VIP products. The following sections explainthe individual steps in the test sequence..

4.3 Instantiating the VIP

The UFS SVT VIP models are instantiated in the testbench and appropriate configuration object is passed to the agent's object. These components can be configured as active or passive components.

```
//An instance of UFS SVT VIP Agent to act as a UFS Host
svt_jedec_ufs_host_agent host;

//An instance of UFS SVT VIP Agent to act as a UFS device
svt_jedec_ufs_device_agent dev;

//An instance of system configuration object. This will take care of
// the block unit size, number of Logical units, etc.,
ufs_system_configuration     sys_cfg;


//mphy interfaces
svt_mphy_rmmi_mrx_dut_controller_if  mphy_rmmi_mrx_dut_controller_if();
svt_mphy_rmmi_mtx_dut_controller_if  mphy_rmmi_mtx_dut_controller_if();
svt_mphy_rmmi_mrx_dut_phy_if         mphy_rmmi_mrx_dut_phy_if();
svt_mphy_rmmi_mtx_dut_phy_if         mphy_rmmi_mtx_dut_phy_if();

//creation of "host" and "device" agent object
host = svt_jedec_ufs_host_agent::type_id::create("host",this);
dev  = svt_jedec_ufs_device_agent::type_id::create("dev",this);

//Connect the virtual interface of the environment to mphy of host and device //agent.
Below connection is for RMMI_LOCAL

uvm_config_db#(virtual
svt_mphy_rmmi_mtx_dut_phy_if)::set(this,"host",$sformatf("mphy_tx_if",i),mphy_rmmi_tx_d
ut_phy_if);

uvm_config_db#(virtual
svt_mphy_rmmi_mrx_dut_phy_if)::set(this,"host",$sformatf("mphy_rx_if",i),mphy_rmmi_rx_d
ut_phy_if);

uvm_config_db#(virtual
svt_mphy_rmmi_mrx_dut_controller_if)::set(this,"dev",$sformatf("mphy_tx_if[%0d]",i),mph
y_rmmi_rx_dut_cntrlr_if[i]);

uvm_config_db#(virtual
svt_mphy_rmmi_mtx_dut_controller_if)::set(this,"dev",$sformatf("mphy_rx_if[%0d]",i),mph
y_rmmi_tx_dut_cntrlr_if[i]);
```

For more information, see theClass Reference HTML and the basic UVM example shipped with UFS VIP. model.

4.4 Configuring the VIP Models

The VIP models are configured using configuration objects that are ready for use. The configuration objects are similar to the other objects, they can be randomized and passed as an argument to a method. The objects come with constraints so that they adhere to protocol limits. They can be controlled or extended to create specific test conditions, or used to produce a wide range of stimulus.

UFS System configuration will have instances of both host and device configuration. Both host and device configuration objects are objects of agent configuration.

```
//passing configuration to "host" object
uvm_config_db#(svt_jedec_ufs_agent_configuration)::set(this,  "host", "cfg",
sys_cfg.host_cfg);
uvm_config_db#(svt_jedec_ufs_agent_configuration)::set(this,  "dev",  "cfg",
sys_cfg.device_cfg);
```

The configuration objects and the required attributes can be created as per your requirements. This is one approach to control the values of attributes. In most cases, the default values of the configuration class are manually assigned and default values are used for other attributes.

The test case creates the configuration object, assigns the values required for the particular test, and assign the configuration object to the environment's configuration.

The test case contains the env object which in turn contains the system configuration object. Using this system configuration object, the host/device configuration can be modified.

If the configuration object is not provided by the test case, the environment can create its own configuration object. The examples below show how to configure the attributes of the UFS configuration.

4.4.1 Configuring the VIP Component as Active or Passive

The UFS host Agent can be configured as active or passive components. When configured as passive components, the data generator and drivers are switched off and the component works in monitor mode only. This can be configured by setting the `is_active`"parameter.

For example, when UFS Host Agent is to be used only as a monitor, the following configuration parameters needs to be set in the env:

```
svt_jedec_ufs_host_agent pass_host;
pass_host = svt_jedec_ufs_host_agent::type_id::create("pass_host",this);
uvm_config_int_db#(int)::set(this, "pass_host.cfg", "is_active", 0);
```

4.4.2 Generating Constrained Random Stimulus

The constrained random generation uses the `built-in randomize()` method that is possessed by all SystemVerilog objects. The most common use of random generation is to produce a series of random transactions which follow a set of applied constraints. The TBD-UVM basic example shows a UVM sequencer that is used to drive directed and random sequences.

4.5 Controlling the Test

All components and sequences in UVM use a common mechanism of raising and dropping objections to control the execution of simulation. The simulation completes when all the raised objections are dropped.

The base sequence takes care of raising objection through its constructor as follows:

```
function ufs_base_system_virtual_sequence::new(string
name="ufs_base_system_virtual_sequence");
  super.new(name, "UFS:library:SVT_LIC_VERSION_PROXY");

  /*
   * setting manage_objection bit so the base svt class handles
   * the raising and dropping of objections.
   */
  manage_objection = 1'b1;
endfunction : new
```

If test case is using any other sequences not extending from the base sequence then test should take care of raising the objection.

4.6 Injecting Delay in the UFS Device VIP Transmission

Transaction class variable, `response_delay_helper` can be used to inject Delay before transmitting the UPIU from the device.

UPIUs, such as, Ready to Transfer, Data In, Response, Query Response, and Task Management Response can be delayed using `response_delay_helper`.

The default value of this variable is 0.

You must extend the `svt_jedec_ufs_info_unit` transaction class from the test bench and to do factory override of the transaction class.

```
set_inst_override_by_type("*.resp_iu_factory", svt_jedec_ufs_info_unit::get_type(),
dly_rsp_two_lun::get_type());
```

Need to override the `reasonable_response_delay_helper` constraint in the base transaction class in the new one as shown as follows and to assign the required value:

```
class dly_rsp_two_lun extends svt_jedec_ufs_info_unit;
  /** all delay are having ns units */
  constraint reasonable_response_delay_helper {
    response_delay_helper == 30;
  }

  `uvm_object_utils_begin(dly_rsp_two_lun)
  `uvm_object_utils_end

  /** Defines the constructor "new".Constructs the override class info unit.    */
  function new(string name = "svt_jedec_ufs_info_unit");
    super.new(name);
    relevant_comparer.check_type = 0;
```

```
  endfunction : new

endclass
```

Refer `ts.constant_delay_rsp_two_lun_serial.sv` and `ts.rand_delay_rsp_three_lun_serial.sv` test cases in Intermediate example directory.

4.7 Injecting Error in the VIP Transmission through Callback

Refer `ts.dev_field_override_cb_wr6_rd6_serial.sv` test case in Intermediate example directory.

- When UFS VIP is a Device:
  - To extend the `svt_jedec_ufs_device_protocol_transport_callback` class as shown as follows:
  - The `info_unit_started` subroutine has a transaction class handle `iu`. Any field (other than Transaction code) can be corrupted using this callback in any UPIU.
  - In the following example, the Data Transfer count in Ready To Transfer UPIU is forced to a value 0.

```
class dev_cb_override_rsp extends svt_jedec_ufs_device_protocol_transport_callback;
   function new(string name = "dev_cb_override_rsp");
       super.new(name);
   endfunction

   virtual function void info_unit_started(svt_jedec_ufs_device_protocol_transport
   device_protocol_transport, svt_jedec_ufs_info_unit iu);
      if ((iu != null) && (iu.transaction_code ==
      svt_jedec_ufs_info_unit::READY_TO_TRANSFER) && (iu.data_buffer_offset == 0)) begin
      `uvm_info("info_unit_started for data_out", $sformatf("iu.transaction_code =%s and
      offset=%d and
      iu.data_transfer_count=%d",iu.transaction_code.name(),iu.data_buffer_offset,iu.data
      _transfer_count), UVM_LOW);
      iu.data_transfer_count = 0;
      `uvm_info("info_unit_started for data_out", $sformatf("iu.transaction_code =%s and
      offset=%d and
      iu.data_transfer_count=%d",iu.transaction_code.name(),iu.data_buffer_offset,iu.data
      _transfer_count), UVM_LOW);
          end
   endfunction
endclass
```

From the test case, the new callback must be registered as follows:

```
   class dev_field_override_cb_wr6_rd6_serial extends ufs_system_base_test;
                    dev_cb_override_rsp cb_override_rsp;

     /** elaboration_phase: To register callback objects. */
     function void end_of_elaboration_phase(uvm_phase phase);
       string method_name = "end_of_elaboration_phase";
       super.end_of_elaboration_phase(phase);
       cb_override_rsp = new("cb_override_rsp");

   uvm_callbacks#(svt_jedec_ufs_device_protocol_transport,svt_jedec_ufs_device_pr
```

```
otocol_transport_callback)::add(ufs_sys_env.dev.device_protocol_transport,cb_o
verride_rsp);
  endfunction
endclass
```

- When UFS VIP is a Host

  The scenario is same as the above with the following changes:

1. The `svt_jedec_ufs_device_protocol_transport_callback` should be replaced with `svt_jedec_ufs_host_protocol_callback`
2. The `svt_jedec_ufs_device_protocol_transport` should be replaced with `svt_jedec_ufs_host_protocol_transport`
3. The `device_protocol_transport` should be replaced with `host_protocol_transport`
4. `uvm_callbacks#(svt_jedec_ufs_device_protocol_transport,svt_jedec_ufs_device_protocol_transport_callback)::add(ufs_sys_env.dev.device_protocol_transport,cb_override_rsp);` should be replaced with `uvm_callbacks#(svt_jedec_ufs_host_protocol_transport,svt_jedec_ufs_host_protocol_transport_callback)::add(ufs_sys_env.host.host_protocol_transport,cb_override_rsp);`

4.8 Injecting Error in the VIP Transmission through Exception

- For error injection, extend `svt_jedec_ufs_info_unit_exception_list` class as shown as follows:
- Refer HTML file for various types of Errors,such as FIELD_MASK_ERROR, FIELD_VALUE_ERROR, and WLUN_ID_ERROR.
- Injecting error_kind as FIELD_MASK_ERROR with weightage as 1.
- Exception is applied on INPUT_PARAMETER_3 field (exception[i].field == svt_jedec_ufs_info_unit::INPUT_PARAMETER_3)
- Exception is applied to TASK_MANAGEMENT_REQUEST UPIU (xact_exe.xact.transaction_code == svt_jedec_ufs_info_unit::TASK_MANAGEMENT_REQUEST)

```
class cust_task_mgmt_input_parameter3_error_exception_list extends
svt_jedec_ufs_info_unit_exception_list;


  /** Instance of the customized packet exception (extension) class for this
testcase. */
  svt_jedec_ufs_info_unit_exception xact_exc = new("xact_exc");



  /**Constructor: create a new instance.*/
  function new(string name =
"cust_task_mgmt_input_parameter3_error_exception_list",
svt_jedec_ufs_info_unit_exception xact_exc = null);


    super.new(name,xact_exc);
    xact_exc = this.xact_exc;
```

```
    xact_exc.set_constraint_weights(0);
    xact_exc.error_kind = svt_jedec_ufs_info_unit_exception::FIELD_MASK_ERROR;
    xact_exc.FIELD_MASK_ERROR_wt = 1;
    randomized_exception = xact_exc;
  endfunction

  /** Constraint the number of exceptions. */
  constraint number_of_exceptions {
    if (xact_exc.xact.transaction_code ==
svt_jedec_ufs_info_unit::TASK_MANAGEMENT_REQUEST)
      num_exceptions == 1;
    else
      num_exceptions == 0;
  }

  constraint reasonable_field {
    if (xact_exc.xact.transaction_code ==
svt_jedec_ufs_info_unit::TASK_MANAGEMENT_REQUEST) {
      foreach(exceptions[i]) {
        exceptions[i].field == svt_jedec_ufs_info_unit::INPUT_PARAMETER_3;
      }
    }
  }

endclass
```

In the test case, add the exception to either Host or Device VIP whichever applicable as follows:

```
class task_mgmt_input_parameter3_error extends ufs_system_base_test;
  svt_jedec_ufs_info_unit cust_info_unit;
  /**Instance of exception list.*/
  cust_task_mgmt_input_parameter3_error_exception_list excpn_lst;

  function void build_phase(uvm_phase phase);
    string method_name = "build_phase";
    super.build_phase(phase);
    /** Create an object of transaction_exception_list */
    excpn_lst = new("excpn_lst");

    //Applying the exception list.
    //When UFS VIP is Host
```

```
svt_config_object_db#(svt_jedec_ufs_info_unit_exception_list)::set(this,"ufs_sys_env.ho
st.host_protocol_transport","upiu_exception_list",excpn_lst);
    //When UFS VIP is Device

svt_config_object_db#(svt_jedec_ufs_info_unit_exception_list)::set(this,"ufs_sys_env.de
v.device_protocol_transport","upiu_exception_list",excpn_lst);


  endfunction
endclass
```

5 Usage Notes

5.1 Protocol Usage Notes

Protocol Usage Notes covers the following topics:

- MK2 Extension Use Model
- Hardware Reset and Linkstart Up
- Lane Change Procedure
- Enable and Disable Scrambling
- UME Use Model
- Passive Host/Device Use Model

5.1.1 BOOT Sequence

This section highlights using the features of VIP for boot sequence in UFS. The code snippets provided in the section is when VIP is configured as Host and Device agents. When VIP is replaced by DUT, DUT takes care of the process. The sequence used in the example is `ufs_boot_up_serial_test_sequence`.

When the VIP is configured as device it responds to the commands from the host.

Specification Reference: See Figure13.3 - Device Initialization and Boot Procedure Sequence Diagram in the UFS specification for more information.

VIP example / test reference: `tb_jedec_ufs_svt_uvm_additional_sys/tests/ts.boot_up_serial.sv`

Boot sequence is a complex sequence with various primitive and complex single type sub sequences. This complex boot sequence is further subdivided into the following steps:

- Step 1: Execution of `power_off` and `power_on` sequence
- Step 2: Execution of link start up sequence on both Host and Device DME
- Step 3: Read device descriptor and setting of boot parameter for required boot lun
- Step 4: Check boot lun ready status
- Step 5: Check `bDeviceInit` flag status to complete the boot process

5.1.1.1 Step 1

1. Perform the following steps to set the DME initial state to `svt_mipi_unipro_types::DME_OFF_STATE` :Initially both UFS Host VIP and UFS Device VIP DME are in power_off state.

2. Both UFS Host VIP and UFS Device VIP DME is switched on by running power_on sequences simultaneously on UFS Host VIP and UFS Device VIP DME sequencer.
3. After running `power_on` sequence, it checks whether the Unipro DME state of Host and Device VIP are in "Disabled" state.

The following code snippet shows the setting of DME to initial state:

```
device_cfg.initial_dme_state = svt_mipi_unipro_types::DME_LINK_UP_STATE
```

The following code snippet shows VIP configured as Host agent:

```
host_cfg.protocol_transport_cfg.is_host = 1'b1;
```

The following code snippet shows VIP configured as Device agent:

```
device_cfg.protocol_transport_cfg.is_host = 1'b0;
```

5.1.1.2 Step 2

In this phase, DME is in disable state after the sequences performed in Phase 1. To transition DME to link-up state (UNIPRO + MPHY) perform the following steps:

1. Enable and link start up sequences performed in Phase 1 to bring both Host and Device DME to linkup state.

```
Class reference for Link start up
....

  /** boot phase2 related sequences objects **/
  svt_mipi_unipro_dme_service_bootup_sequence bootup_seq;
```

2. NOP_OUT sequence from HOST UFS VIP is executed and checked for NOP_IN as response to NOP_OUT command.

```
/** NOP OUT sequence from the Host */
svt_jedec_ufs_info_unit_nop_out_upiu_sequence nop_out_cmd_seq;

`uvm_do_on(nop_out_cmd_seq,p_sequencer.host_virt_seqr.utp_cmd_iu_seqr)
```

The following code snippet shows the transition of DME to link-up state in double-ended link start up process when VIP is configured as Host agent:

```
`uvm_do_on(bootup_seq,p_sequencer.host_virt_seqr.dme_svc_seqr)
```

The following code snippet shows the transition of DME to link-up state in double-ended link start up process when VIP is configured as Device agent:

```
`uvm_do_on(bootup_seq,p_sequencer.device_virt_seqr.dme_svc_seqr)

......
```

5.1.1.3 Step 3

In this phase, the sequence reads the UFS Device descriptor and parse device descriptor for various boot related parameters.

The boot related parameters are as follows:

- bBootEnable
- bDescrAccessEn
- bLunEnable
- bBootLunId

- bBootLunEn

Prior to sequencing read, from the UFS Device the following parameters are configured using write configuration descriptor to access BOOT LUN as per specification:

- bBootEnable = 1
- bDescrAccessEn = 1
- bLunEnable = 1
- bBootLunId = 01 (LUN A)
- bBootLunEn (attribute)= 01 (LUN A)

To enable the BOOTLUN, the following settings needs to be done:

```
device_cfg.initial_protocol_transport_status.lun_status[1].b_boot_lun_id =
`SVT_JEDEC_UFS_BOOT_B;
    device_cfg.initial_protocol_transport_status.lun_status[4].b_boot_lun_id =
`SVT_JEDEC_UFS_BOOT_A;
    device_cfg.initial_protocol_transport_status.b_boot_lun_en = `SVT_JEDEC_UFS_BOOT_A;
    device_cfg.initial_protocol_transport_status.device_status.b_boot_enable = 8'h01;
```

The following code snippet shows the sequence of the UFS device descriptor and parse device descriptor when VIP is configured as Host agent:

```
  /** starts Phase3 of boot sequence **/
    `uvm_do_on_with(rd_dd_descriptor,p_sequencer.host_virt_seqr.udm_iu_seqr,
                        {osf_descriptor_idn_l == `SVT_JEDEC_UFS_DEVICE;
                         osf_index_l          == 0;
                         osf_length_l         == `SVT_JEDEC_UFS_DD_B_LENGTH;
                        })
    test_b_boot_enable =
  rd_dd_descriptor.rsp.data_segment[`SVT_JEDEC_UFS_DD_B_BOOT_ENABLE];

    `uvm_do_on_with(rd_attribute_seq,p_sequencer.host_virt_seqr.udm_iu_seqr,
                        {osf_attribute_idn_l  == `SVT_JEDEC_UFS_DEV_B_BOOT_LUN_EN;
                         osf_index_l          == 0;
                        })
```

5.1.1.4 Step 4

In this phase, the sequencing is as follows:

- With all the above parameters and with WLUN id ‘h30, boot lun from address C130_0000_0000_0000 is accessible.
- Before accessing boot_lun verify the lun status as boot lun ready or not. If the boot lun status is set to 1 then send test unit ready UPIU to the boot lun with id 'h30.
- If test unit ready UPIU gets the response UPIU with response field as good, then boot lun is accessible for reading boot code from address C130_0000_0000_0000 by using rd6/rd10/rd16 SCSI command.

The following code snippet shows the boot sequence for phase 4 when VIP is configured as Host agent:

```
/** boot phase4 related sequences objects **/
  svt_jedec_ufs_info_unit_scsi_test_unit_sequence test_unit_rdy_seq;
  svt_jedec_ufs_info_unit_scsi_wlun_rd_cmd_sequence wlun_rd_seq;
```

```
/** The following steps can be executed only if bBootEnable field is set. **/
if (test_b_boot_enable == 1) begin
  `uvm_do_on_with(test_unit_rdy_seq,p_sequencer.host_virt_seqr.utp_cmd_iu_seqr,
                  {lun_id_l == `SVT_JEDEC_UFS_BOOT;
                   wlun_id_l== 1;})
```

5.1.1.5 Step 5

In this step, the sequencing is as follows:

- After accessing boot code, if `bDeviceInit` flag is enabled, then UFS Host VIP completes the boot seq
- Later on UFS Host VIP polls for bDeviceInit flag until the Device clears the flag bDeviceInit,which indicates the device initialization is completed.

Note
Boot up sequence is also involved in performing few of the events mentioned in the above phases.

The following code snippet shows the completion of the boot sequence:

```
/** starts Phase5 of boot sequence **/
    `uvm_do_on_with(dev_init_set_flag_seq,p_sequencer.host_virt_seqr.udm_iu_seqr,
                  {osf_flag_idn_l == `SVT_JEDEC_UFS_DEV_F_DEVICE_INIT;
                   osf_flag_value_l == 8'h01;
                  })
```

5.1.2 MK2 Extension Use Model

Specification Reference: MK2 extension is an enhanced feature provided as per the requirement in the UNIPRO specification version 1.6. The transmitter may close the burst by PA_PDU aligned M-PHY symbol pair of <MK2,MK2>.

When the MK2 extension feature is supported, it creates an additional guard time which results in an extended period of TX SYNC symbol transmission of the next consecutive frame. The additional guard time is defined by attribute `PA_MK2ExtensionGuardBand`. The support capability will be transmitted in the `PACP_CAP_EXT1_IND` frame in the last state of link startup before `PACP_CAP_IND` frame. The status[0] informs remote PA layer that local PA layer supports MK2 extension.

5.1.2.1 Host DUT is Connected to Device VIP

- The UNIPRO version on the device VIP needs to be changed to version 1.60 and the MK2 extension attribute enabled as shown in the following code snippet :

```
    device_cfg.unipro_protocol_version =
svt_mipi_unipro_types::UNIPRO_VERSION_1_60;
    device_cfg.initial_phy_adapter_status.pa_tx_mk2_extension = 1;
```

- The host has to set the attribute `PA_TxMk2Extension` through `DME_SET`.

5.1.2.2 Host VIP is Connected to Device DUT

- The UNIPRO version on the host VIP needs to be changed to version 1.60 as shown in the following code snippet:

```
    host_cfg.unipro_protocol_version =
svt_mipi_unipro_types::UNIPRO_VERSION_1_60;
```

```
host_cfg.initial_phy_adapter_status.pa_tx_mk2_extension = 1;
```

- Either the host sends PEER_SET to set MK2 extension attribute on device DUT as shown in the following code snippet:

```
`uvm_do_with(dme_peer_set_seq,{attr_set_type      == 0;
                               mib_attribute == 'h155A;
                               mib_value    == 1;
                               gen_selector_index == 0;})
```

The device DUT can also set it locally using DME_SET.

Figure 5-1 Pacp_cap_ext1_ind frame for MK2 extension support

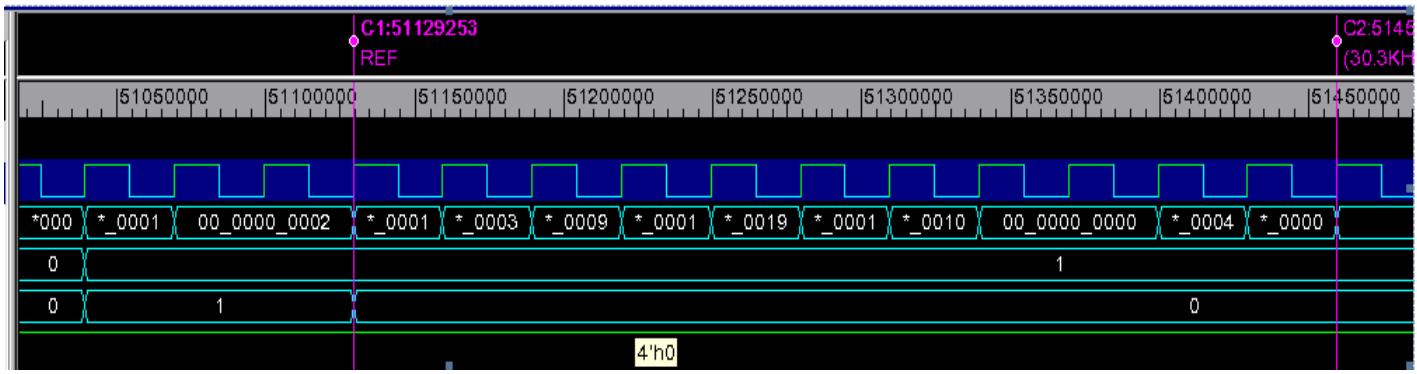
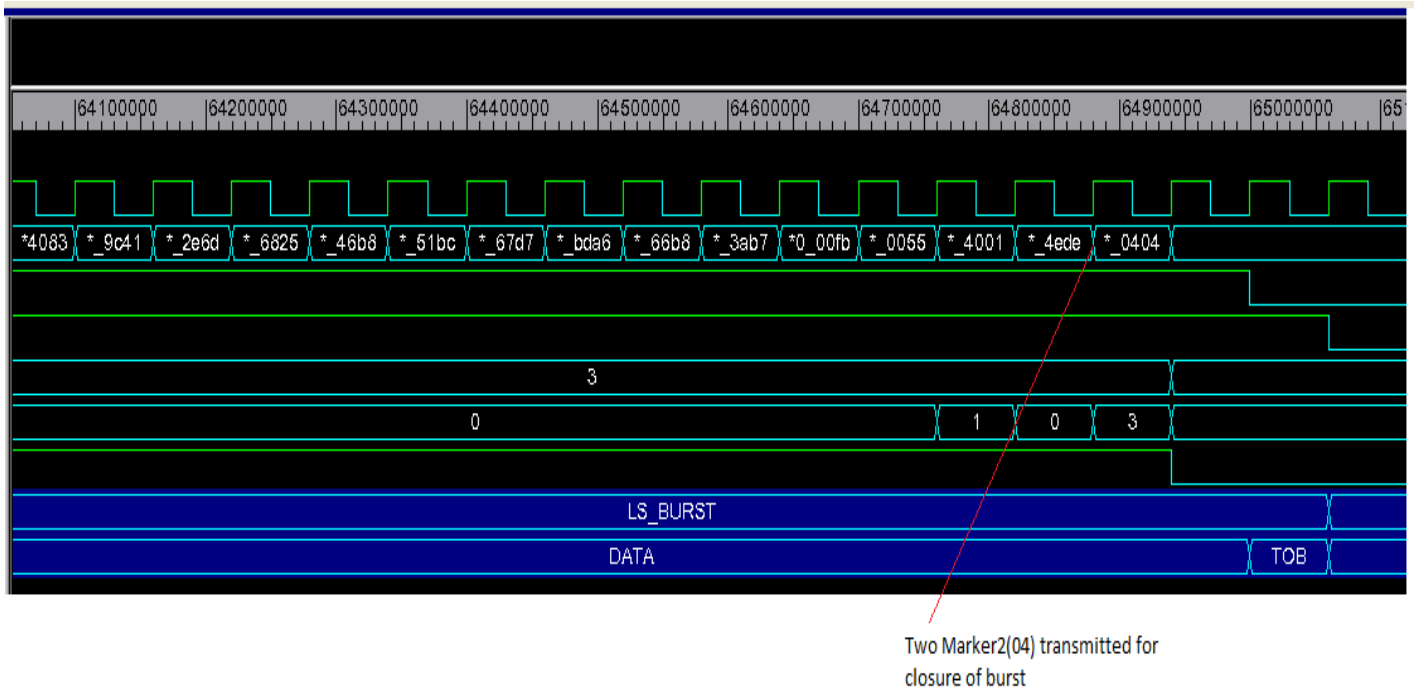


Figure 5-2 Trace File - Marker2 extension

Figure 5-3 Mk2 Extension During End of Burst



5.1.3 Hardware Reset and Linkstart Up

Hardware RESET is the output from the host VIP. It is an input to the device VIP. When the host VIP is used, you have to initiate the hardware reset from the sequence. When the device is VIP, it reacts to the Hardware reset. The timing for the hardware reset should be as per the Figure 7.2 - Hardware Reset in JESD220A. UFS VIP supports `hw_reset` at any time in a simulation.

The device will detect hardware reset only if Reset is asserted to LOW for minimum duration of 1us. The De-Assertion time between two hardware resets should be minimum of 1us, else it ignores the Hardware Reset.

The host will assert hardware reset for specific duration based on two parameters: `rst_n_pulse_width` in `svt_jedec_ufs_host_service_request` and replaceable define `` `SVT_JEDEC_UFS_HARDWARE_RESET_PULSE_WIDTH_UNIT `` (default value is 1us).

The product of `rst_n_pulse_width` and `` `SVT_JEDEC_UFS_HARDWARE_RESET_PULSE_WIDTH_UNIT `` gives the assertion time of hardware reset and de-assertion time is set to 1us.

Both Host and Device will detect Hardware Reset if it is asserted for 1us and will Reset itself and their respective UniPro layers. Once the Hardware Reset is De-Asserted, then both Host and Device will enable their layers followed by LinkStartUp.

Supporting Back-Back Resets, you need to explicitly wait for `DME_DISABLED_STATE` and then `DME_LINK_UP_STATE` from Host or Device VIP. The DME_DISABLED_STATE will occur after 1 us after de-asserting the hardware reset as per the UFS specification.

Specification Reference: See Figure 7.2 - Hardware Reset in section 7.1.2 Hardware Reset in JESD220A for more information.

VIP example / test reference: `tb_jedec_ufs_svt_uvm_additional_sys/tests/ts.hw_reset_serial.sv`

When the hardware reset is detected by the host or device VIP, `DME_RESET.req` is issued to their respective UniPro layers.

After `DME_RESET`, the link is in disabled state and the link is established using `DME_ENABLE` and link startup sequence.

The code snippet for generating reset is asfollows:

```
/** Instance of the virtual sequence */
  ufs_hw_reset_test_sequence test_virtual_seq;
```

Figure 5-4 Hardware Reset Log

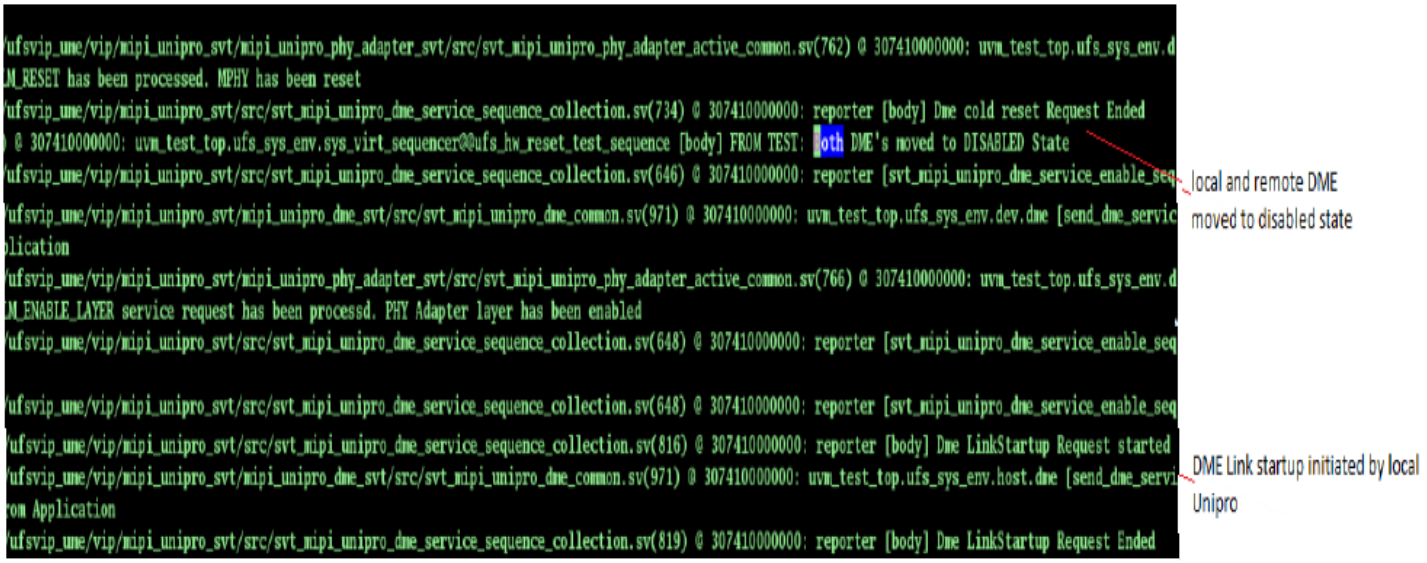
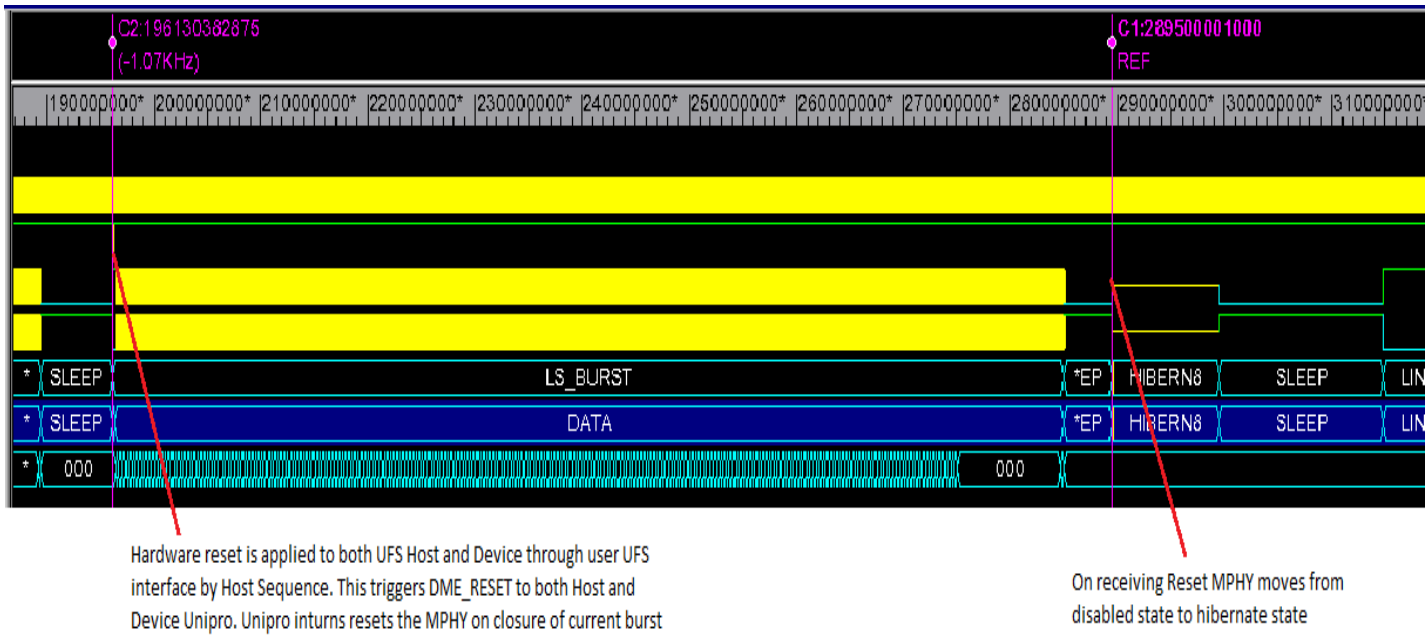


Figure 5-5 Hardware Reset Wave



5.1.4 Lane Change Procedure

This feature is implemented in VIP to increase the transmission and reception bandwidth when single lane is not sufficient as specified in Section 5.7.11 in the UniPro Specification. When multiple lanes are used, the PA symbols are transmitted or received in order starting from lane0 to (`PA_ActiveTxDataLanes - 1`) and (`PA_ActiveRxdataLanes - 1`) respectively.

After link initialization, by default, both remote and local UNIPRO power mode is PWMG1 and supports a single lane.

5.1.4.1 HOST VIP is Connected to Device DUT

To change the number of lanes, the host has to set the corresponding attribute register `PA_ActiveTxDataLanes` and `PA_ActiveRxDataLanes` through the DME SET primitive, and set `PA_PWRMode` using the `DME_SET` primitive to trigger the lane change as shown in the following code snippet :

```
//An instance to call the DME SET sequence
svt_mipi_unipro_dme_service_set_sequence dme_set_seq;


`uvm_do_on_with(dme_set_seq,p_sequencer.host_virt_seqr.dme_svc_seqr,{attr_set_type == 0;
            mib_attribute == svt_mipi_unipro_types::PA_ACTIVE_TX_DATA_LANES;
            mib_value == 1;
            gen_selector_index == 0;})

`uvm_do_on_with(dme_set_seq,p_sequencer.host_virt_seqr.dme_svc_seqr,{attr_set_type == 0;
            mib_attribute == svt_mipi_unipro_types::PA_ACTIVE_RX_DATA_LANES;
            mib_value == 1;
            gen_selector_index == 0;})
```

```
`uvm_do_on_with(dme_set_seq,p_sequencer.host_virt_seqr.dme_svc_seqr,{attr_set_type ==
0;
            mib_attribute == svt_mipi_unipro_types::PA_PWR_MODE;
            mib_value == 1;
            gen_selector_index == 0;})
```

L2 timers will be randomized with power mode change sequence.

5.1.4.2 Host DUT is Connected to Device VIP

The power mode change has to be initiated by the host. If the device initiates the power mode change, the host DUT has to reject the power mode request from the device.

The VIP provides this feature as an exception to initiate the lane change procedure from the device VIP to verify whether the DUT is ignoring the lane change procedure or not.

5.1.5 Enable and Disable Scrambling

Scrambling is done only when the VIP is operating in the FAST or FAST_AUTO mode. Scrambling support has to be enabled on both outbound and inbound links. However, it can be done only in one direction, as scrambling is done only in FAST or FAST_AUTO mode. It can be enabled or disabled at every power mode change request. The new scrambling setting becomes applicable at the first burst after the power mode change.

Data scrambling is applied across all lanes in one direction, but the scrambling is done independently on each lane. The `scrambling pa_scrambling` MIB attribute has to be used to enable or disable scrambling. Once set power mode change has to be initiated.

The following is the code snippet for enabling scrambling where a host VIP connected to a device VIP :

```
/** Instance of dme service sequence */
svt_mipi_unipro_dme_service_set_sequence dme_set_seq;


`uvm_do_on_with(dme_set_seq,p_sequencer.device_A_virt_seqr.dme_svc_seqr,{attr_set_type
== 0;
mib_attribute == svt_mipi_unipro_types::PA_SCRAMBLING;
mib_value == 1;
gen_selector_index == 0;})
```

5.1.6 UME Use Model

UME enables using the host memory as a device working memory. For example, as the L2P table cache, write buffer, andso on as specified in Figure 4.4 - Inter-Memory communication of UM Buffering in specification JESD220-1A. The UM area is physically located on the host side, but logically belongs to the device as if it would be physical RAM in the device. Thus the UM area is logically a property of the device, and it is a replacement of the device-integrated RAM.

Specification Reference: See Figure 4.4 - Inter-Memory communication of UM Buffering in specification JESD220-1A for more information.

Configuration and Status classes are enhanced to include UME (or UMA) specific attributes, such as,

dMinUMAreaSize/fUM/fSuspendUM/fUMSuspended/dUMAreaSize/bMaxUMPIURequests.

5.1.6.1 Device VIP Use Model

- By Default, UME is disabled.
- To enable UME operations, you must set the following UFS versions:
  - 2. 0 for UME 1.0
  - 2.1 for UME 1.1
    - Set `enable_ufs_version` in *svt_jedec_ufs_protocol_transport_configuration.sv* to `JEDEC_UFS_VERSION_2_0/JEDEC_UFS_VERSION_2_1`.
- `bDeviceSubClass` in Device Descriptor must be set to 4.
  - Set `b_device_sub_class` in *svt_jedec_ufs_device_configuration.sv* to 4.
- fUM Flags must be set to 1.
  - Set `enable_ume_wb_cache` in *svt_jedec_ufs_agent_configuration.sv* to 1.
- If L2P Cache operation is required, then set `enable_ume_l2p_cache` in *svt_jedec_ufs_agent_configuration.sv* to 1.
- SCSI Write Operation:
  - Logical unit x will perform UME operation only when
    - Expected Data transfer Length in command UPIU is less than or equal to the `ume_wb_cache_size`
    - `dUMAreaSize >= ume_l2p_cache_tag_size + ume_l2p_cache_size + ume_wb_cache_tag_size + (LUN ID + 1) * ume_wb_cache_size`
- SCSI Read Operation:
  - If the corresponding data is available in UM space, UME operations will be automatically executed.
- As shown in Figure 5-6, each LUN is pointed to a specific location is Host Controller Unified Memory space.
- When a write request is received for a LU, device will point to the specified location based on the LUN.
- Write Buffer Flush operation (copying from UMA area to Device area) will be performed when another write request is received to the same LUN (for UME v1.1).
- Write Buffer Flush operation will be performed at the end of the same SCSI Write command (for UME v1.0).

Apart from the fUM, various other configuration parameters have to be set in the `svt_jedec_ufs_agent_configuration` attribute in order to support cache operation.

Table 5-1 describes the enable signals to define the support of various cache as per UME and the size of each cache.

Table 5-1 Enable Signals

Cache	Enable Signals

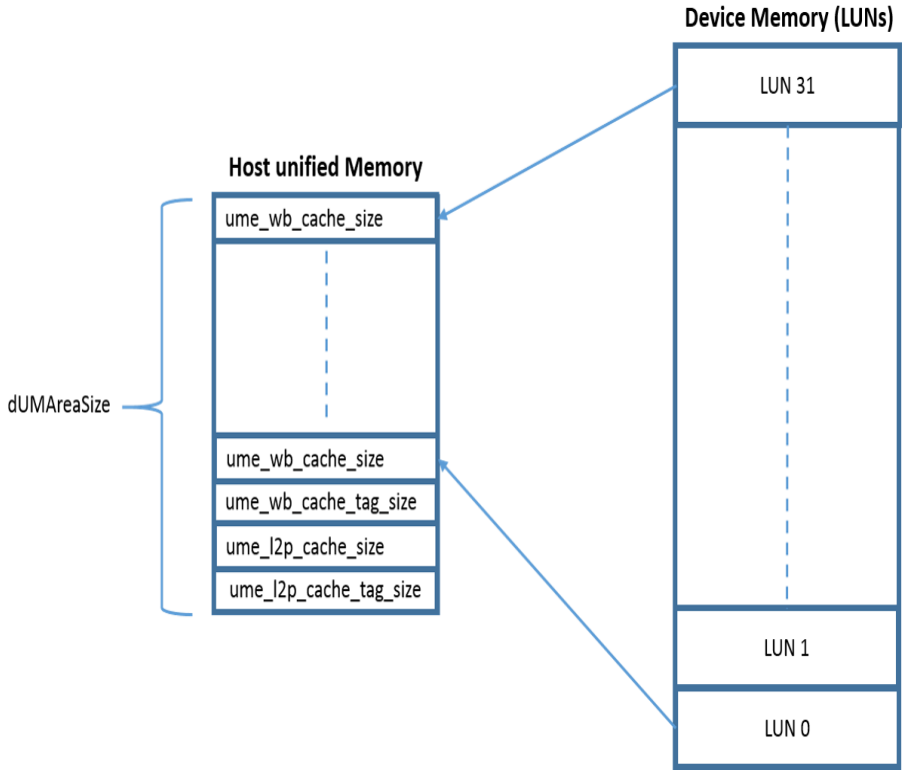
L2P Cache	`enable_ume_l2p_cache - (default value "0")`
	`ume_l2p_cache_tag_size - (default value "256")`
	`ume_l2p_cache_size - (default value "256")`

WB Cache	`enable_ume_wb_cache - (default value "0")`
	`ume_wb_cache_tag_size - (default value "256")`
	`ume_wb_cache_size - (default value "4096")`

For example, if you want to enable L2P and WB cache, the following parameters have to be set in the test:

- `enable_ume_wb_cache`
- `enable_ume_l2p_cache`

Figure 5-6 Unified Memory



5.1.6.2 UME Outstanding Request

The Device VIP supports maximum outstanding request based on the configuration parameter `ume_max_outstanding_transactions`. If `svt_jedec_ufs_ume_num_outstanding_req_supported` is greater than zero, then multiple outstanding process will be supported.

Host Driver will hold all the Outstanding UME requests. It sends the request to the Sequence, waits for its response, and sends another request through `ubm_umpiu_response_port`. This ensures that OutStanding UME requests are addressed in-order.

The following are the two examples tests placed in `/jedec_ufs_svt/test/sverilog/tb_jedec_ufs_svt_uvm_basic_sys/tests/`

- `ts.ufs_um_data_integrity_rmmi.sv`
- `ts.ufs_um_data_integrity_serial.sv`

Note
When all cache are enabled, their combined size should be equal to or less than dUMAreaSize attribute.

5.1.7 Passive Host/Device Use Model

The following is the use model for passive host/device:

- The passive monitor samples the transactions on the bus and responds as required
- The passive component detects the following transactions (ideally from the bus except Hardware Reset) and changes its state accordingly:
  - Hardware Reset
  - Endpoint reset
  - Link Lost condition
  - Link Startup Indication
  - PACP Power Mode change.
- The following transactions are issued to the passive VIP component from the testbench:
  - DME_POWERON.req from DME_OFF state
  - DME_ENABLE.req from DME_DISABLED_STATE
  - DME_SET or DME_GET

You must ensure the passive Host/Device is synchronized with the DME requests from the testbench.

The following code snippet must be executed for the passive component:

```
/** Instances needed to trigger transactions to Passive Host/Device  */
    ufs_system_env sys;
    uvm_component parent_comp;
    svt_mipi_unipro_dme_service dme_svc;

   /** Getting the handle for Host Agent */
    if(!$cast(host_agent, p_sequencer.find_host_agent(this))) begin
      `svt_xvm_fatal(get_full_name(), "Failed attempting to obtain handle to agent.");
    end

   /** Getting the handle for Device Agent */
    if(!$cast(device_agent, p_sequencer.find_device_agent(this))) begin
      `svt_xvm_fatal(get_full_name(), "Failed attempting to obtain handle to agent.");
```

```
    end

    /** Getting System Handle for the Environment to write into the passive Host/Device
Explicitly when required */
    parent_comp = host_agent.get_parent();
    if(!$cast(sys, host_agent.get_parent())) begin
     `svt_fatal("get_full_name()",$psprintf("Failed attempting to obtain handle to env.
type = %0s",parent_comp.get_type_name()));
    end
```

The following code snippet demonstrates issuing the DME_ENABLE request to the passive components:

```
  /**
     * Send Explicit DME_ENABLE.req to Passive component.
     * Since the write port id non-blocking, wait for Desired state
     */
    if(sys.enable_passive_agent == 1) begin
      dme_svc = new();
    dme_svc.direction = svt_mipi_unipro_types::TX;
      dme_svc.service_type = svt_mipi_unipro_dme_service::DME_ENABLE;
sys.pssv_host.dme_mon.mon_svc_in_port.write(dme_svc);
      wait(sys.pssv_host.shared_status.dme_state ==
svt_mipi_unipro_types::DME_LINK_DOWN_STATE);


      sys.pssv_dev.dme_mon.mon_svc_in_port.write(dme_svc);
      wait(sys.pssv_dev.shared_status.dme_state ==
svt_mipi_unipro_types::DME_LINK_DOWN_STATE);
```

For more information, see the `ufs_sequence_collection` file in the UFS Intermediate example directory which demonstrates issuing DME transactions to the passive component.

5.1.8 Secure In/Out Operation in RPMB Logical Unit

- The contents of the RPMB W-LUN can be read and written through authenticated read and write operations successfully.
- The Message Authentication Code (MAC) is calculated using HMAC SHA-256 Algorithm.
- The algorithm is developed based on "UFS Secure Hash Algorithm (SHA and HMAC-SHA)" as per JESD220B Specification.
- You can make of the use of the algorithm implemented in the VIP. The following is the use case that returns the 256 Bit MAC

```
svt_jedec_ufs_hmac_sha256 hmac_sha;
hmac_sha = new();
hmac_sha.sha256_encrypt(input bit [7:0]  Authentication_key[32], input bit
[7:0]  Data[])
```

- A configuration variable `ignore_mac_comparison` is provided. If you set this variable, it ignores the comparison of MAC requested in the Frame against the MAC calculated by the Device.
- The programmed data is written into RPMB Memory location despite of the MAC value in the Frame.
- You can set the `ignore_mac_comparison` variable as follows:

```
device_cfg.protocol_transport_cfg.ignore_mac_comparison = 1;
```

- You can program the required value in the Write counter as follows:

```
device_cfg.protocol_transport_status.rpmb_write_counter_value = 32'h12345678;
```

5.1.9 Maximum Number of LUNs Supported

In UFS 2.1, the maximum number of logical units supported in a UFS Device has increased from 8 to 32 when bMaxNumberLU is set to 1 in Geometric Descriptor. So the existing Define `SVT_JEDEC_UFS_MAXIMUM_LUNS` is replaced with method `get_max_num_of_luns()` which will return the Maximum number of LUNs.

You can access the method `get_max_num_of_luns()` in `svt_jedec_ufs_protocol_transport_configuration.sv` configuration class to get the LUN information.

Youhave to explicitly create these many number of LUN configuration/status instances from `ufs_system_configuration`. Refer ufs_system_configuration.sv file `tb_jedec_ufs_svt_uvm_basic_sys` example TB area.

```
for (int i = 0; i < device_cfg.protocol_transport_cfg.get_max_num_of_luns(); i++)
    begin
      device_cfg.initial_protocol_transport_status.lun_status.push_back(null);
      device_cfg.initial_protocol_transport_status.lun_status[i] = new();
      device_cfg.initial_protocol_transport_status.lun_status[i].b_lu_enable = 1'b1;
      device_cfg.initial_protocol_transport_status.lun_status[i].d_num_alloc_units =
32'h1000;
      device_cfg.protocol_transport_cfg.lun_cfg.push_back(null);
      device_cfg.protocol_transport_cfg.lun_cfg[i] = new();
      device_cfg.protocol_transport_cfg.lun_cfg[i].b_lu_queue_depth = 20;
    end
```

5.1.10 Purge Operation

As UFS specification is not clear with respect to data retrieval from the unmapped address space hence Device VIP does not store the data related to the unmapped address space.

Due to this implementation, Device VIP cannot emulate cases covering purge FSM, bpurgestatus, and sending Failure responses for commands issued during purge in progress. Also, Device VIP executes all the commands received during purge like in normal scenario.

5.1.11 Field Firmware Update Timeout

The `bFFUTimeout` timer is in terms of seconds and it can be reduced to smaller value by using the `ffu_timer_scaling_factor` scaling factor provided in the device configuration class.

5.1.12 Boot Memory Access

VIP uses different Boot Memory spaces based on the chosen configuration. This is to meet the requirement of different boot code programming for the BOOT LUNS A and B. The 'boot_memory_space_access'

configuration parameter in svt_jedec_ufs_protocol_transport_configuration decides single boot memory or two different memory spaces for Boot LUN A and Boot LUN B.

1. boot_memory_space_access = ACCESS_BOOT_LU_MEMORY_SPACE_FOR_BOOT_READ

   Both Boot LU A and LU B access from same memory location which is the physical address of BOOT LUN with address ranges from

   C1_30_00_00_00_00_00_00 to C1_30_FF_FF_FF_FF_FF_FF based on the Logical Block Address and the Transfer Length in SCSI Read command.

2. boot_memory_space_access = ACCESS_BOOT_LU_A_AND_LU_B_MEMORY_SPACE_FOR_BOOT_READ

   Boot LU A and LU B will read from their own LUN physical address locations with address range from 00_<LU_A_OR_B_ID>_00_00_00_00_00_00 or 00_<LU_A_OR_B_ID>_FF_FF_FF_FF_FF_FF based on the Logical Block Address and Transfer Length in SCSI Read command.

5.1.13 Lower Transfer Size Support

Based on the specification, the UFS Device must support data transfers with lengths multiple of `bLogicalBlockSize` and its minimum value is 4K. In SoC setups doing such big transfers is difficult. Therefore, we added lower data transfer support in VIP when `ignore_edtl_vs_tl_calculation` parameter is set in `svt_jedec_ufs_protocol_transport_configuration`.

It ignores all the calculations based on the specification and processing the command with data size as per Expected Data Transfer Length.

5.1.14 APIs on Device Side

The following APIs are supported on the Device side:

a. reset_ufs_device

   It issues a DME_RESET.req SAP to the UniPro layer

b. enable_ufs_device

   It issues a DME_ENABLE.req SAP to the UniPro layer

c. read_lun_data(input bit [6:0] unit_number_id, int logical_block_address, int offset, output bit [`SVT_JEDEC_UFS_MEM_MAX_DATA_WIDTH-1:0] data);

   It reads the data from the LUN Memory based on Logical Block Address, Offset.

d. write_lun_data(input bit [6:0] unit_number_id, int logical_block_address, int offset, bit [`SVT_JEDEC_UFS_MEM_MAX_DATA_WIDTH-1:0] data);

   It writes data to the LUN Memory based on Logical Block Address, Offset.

5.2 Verification Usage Notes

Verification Usage Notes covers the following topics:

- Configuring the UFS Host and Device
- Setting up Spec Version
- LCC support
- Accessing Status Object From Testbench

- Timescale Setting and Scaling Down Timer Values
- Using UFS exceptions
- Lane-to-Lane Skew Injection
- Test Mode
- Error Injection in UniPro Layers
- Using Internal VIP Clock in RMMI Mode
- Using Clock-Jitter in the Serial Mode

5.2.1 Configuring the UFS Host and Device

The UFS system configuration has instances of the host and device configuration (which are objects of `svt_jedec_ufs_agent_configuration`). With the host and device configurations, various parameters of the host and device (which signify UFS System as a whole) can be configured. Some of the configurations can be modified at run time.

For example, if the number of active LUNs are to be configured for the UFS system, the configuration is as follows:

```
// Enable LUN 0
device_cfg.initial_protocol_transport_status.lun_status[0].b_lu_enable = 1'b1;
device_cfg.initial_protocol_transport_status.lun_status[0].d_num_alloc_units =
32'h1000;
device_cfg.protocol_transport_cfg.lun_cfg[0].b_lu_queue_depth = 2;

// Enable LUN 4
device_cfg.initial_protocol_transport_status.lun_status[4].b_lu_enable = 1'b1;
device_cfg.initial_protocol_transport_status.lun_status[4].d_num_alloc_units = 32'h500;
device_cfg.protocol_transport_cfg.lun_cfg[4].b_lu_queue_depth = 2;
```

Note
The above example shows configuration of a particular LU's allocation units, its queue depth, enabling the LU.

5.2.2 Setting up Spec Version

The spec version is controlled by `enable_ufs_version` parameter in the `svt_jedec_ufs_protocol_transport_configuration.sv` file. By default, the spec version is defined as JEDEC_UFS_VERSION_1_1. You can configure it to spec version JEDEC_UFS_VERSION_2_0.

5.2.3 LCC support

5.2.3.1 Spec version attribute is set to v1.41

After a `PACP_CAP_IND` frame is received during capability exchange phase (phase 5) of the link startup, the VIP's PA layer sets the `TX_LCC_Enable` attribute of the MTX and order M-TX on logical lane 0, to perform `LCC-READ-CAPABILITY`, `LCC-READ-MFG-INFO`, and `LCC-READ-VEND-INFO` before closing the burst. During the link configuration in power mode change and during hibernate entry, the M-TXs on all connected lanes issue the `LCC-WRITE-ATTRIBUTE` command before closing the burst. No other additional settings are necessary.

Note
For UniPro version 1.4, LCC_DISABLE is provided for controlling LCC transmissions.

5.2.3.2 Spec Version Attribute is set to v1.6

After a `PACP_CAP_EXT1` frame has been received and if the `pa_local_tx_lcc_enable` and the `pa_peer_tx_lcc_enable` MIB attributes are set, `LCC-READ-CAPABILITY`, `LCC-READ-MFG-INFO`, and `LCC-READ-VEND-INFO` are performed before closing the burst. During the link configuration in power mode change and during hibernate entry, the M-TXs on all connected lanes issue the `LCC-WRITE-ATTRIBUTE` command before closing the burst. To enable the LCC functionality, the testbench has to set the VIP's `pa_local_tx_lcc_enable` to 1 using the initial configuration, and the DUT should also set this bit in its configuration registers.

Figure 5-7 PACP_CAP_IND Frame Transmission

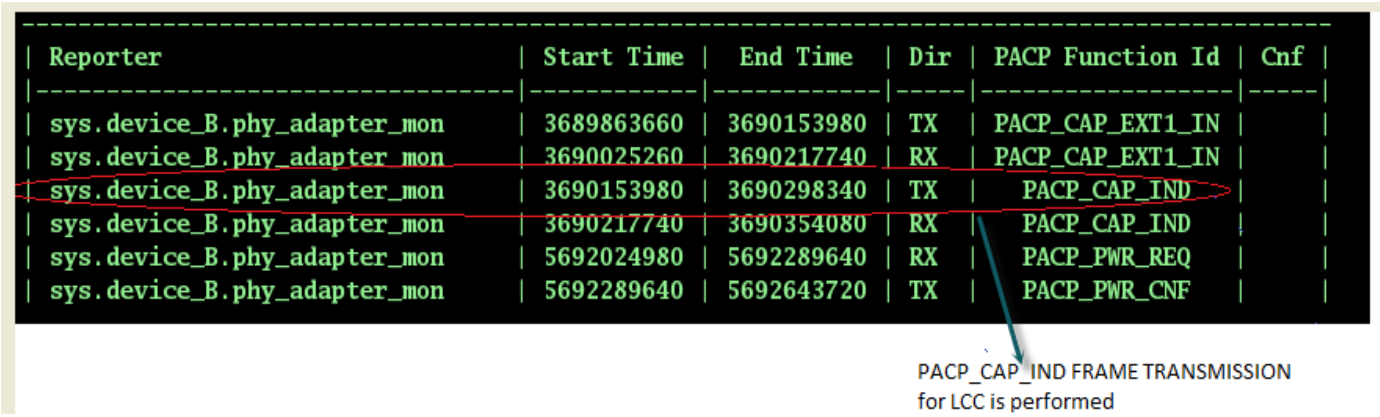
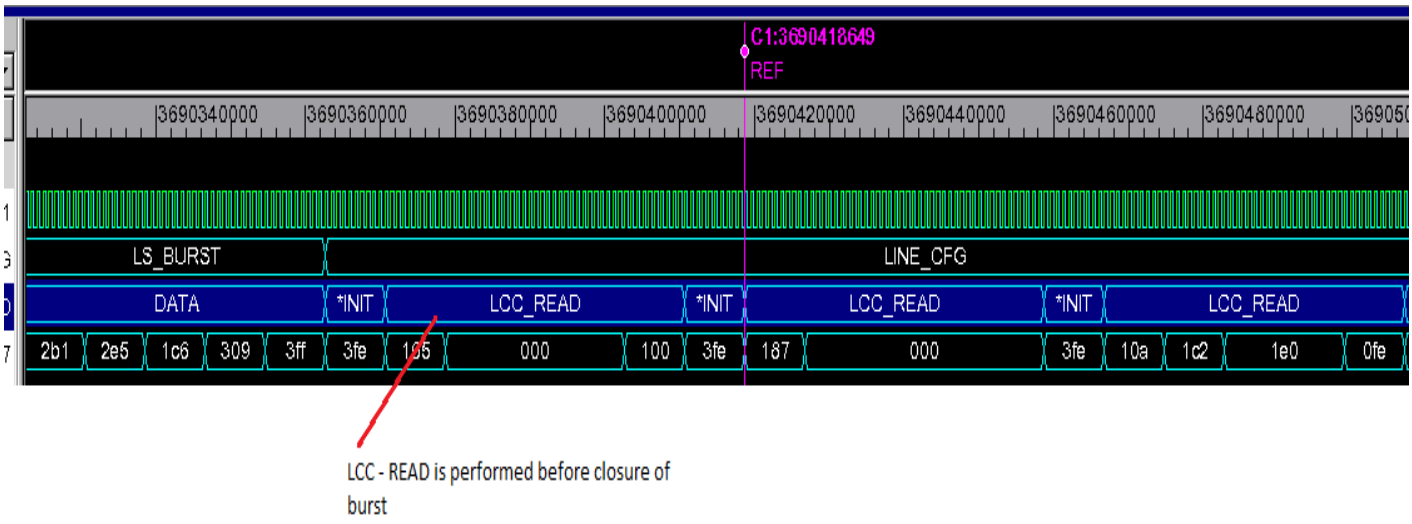


Figure 5-8 LCC Read



5.2.4 Accessing Status Object From Testbench

5.2.4.1 Configuration Class

The `svt_jedec_ufs_agent_configuration` class defines enable attributes. For example, the attribute to enable UFS layer tracing `is enable_protocol_transport_tracing`.

The configuration classes define the `User Configurable` protocol attributes specified in the UFS specification as "No".

Such attributes are defined in the following class:

- `svt_jedec_ufs_protocol_transport_configuration` and contained classes:
  - `svt_jedec_ufs_device_configuration`
  - `svt_jedec_ufs_lun_configuration`

The configuration class attributes can be accessed from the test bench.

5.2.4.2 Status Class

The status class attributes reflect the current settings. The attributes are READ ONLY from the testbench and should not be modified by the testbench in run time. Only VIP internally updates them based on the protocol communicationor state. The status class `svt_jedec_ufs_status` follows a similar structure as configuration class and contains the following:

- `svt_jedec_ufs_protocol_transport_status` class
- `svt_jedec_ufs_protocol_transport_status` and contained classes:
  - `svt_jedec_ufs_device_status`
  - `svt_jedec_ufs_lun_status`

The status class attributes class attributes can be accessed from the testbench. To access status class attributes, the UFS agents provide a method `get_jedec_ufs_shared_status()` to get the handle of the `svt_jedec_ufs_status` object. The following code snippet provides the usage information:

```
//First get the handle of the svt_jedec_ufs_status class.
svt_jedec_ufs_status current_ufs_status =
<env_inst_name>.<device_agent_inst_name>.get_jedec_ufs_shared_status();
for example: env_inst_name = sys; device_agent_inst_name = dev, then
svt_jedec_ufs_status current_ufs_status = sys.dev.get_jedec_ufs_shared_status();


// Access the attributes using the status object hande.
// For protocol specific status attributes which are related to the whole transport
// protocol use the following attributes
 current_ufs_status.protocol_transport_status.f_device_init


// For device specific status (device related configurable variables [attributes/flags]
// ). Additionally, you need to use device_status instance present in protocol transport
status class.
 current_ufs_status.protocol_transport_status.device_status.b_boot_enable


// For lun specific status (basically logical unit related configurable variables
[attributes/flags]). Additionally, you need to use lun_status instance present in
protocol transport status class.
 current_ufs_status.protocol_transport_status.lun_status[i].b_boot_lun_en
```

5.2.5 Timescale Setting and Scaling Down Timer Values

MPHY specification defines various timers which are used at certain places. You can scale down any of these timers to improve simulation time by setting configuration parameters.

The default value for `T_linereset` timer is 3.1ms as defined in the `svt_mphy_configuration` class. To scale down this value you can set any value to the `T_linereset`timer in the `shared_cfg` file. Linereset time should always be greater than prepare time.

```
// This will make timer value timescale independent
mphy_cfg.set_timer("t_linereset",100,"us");
```

Similarly, you can also scale the value of the timer up as shown in the following code snippet :

```
mphy_cfg.reset_timer = 10us;   // Its default value is 100ns
```

5.2.6 Using UFS exceptions

UFS exceptions are categorized into two groups as follows:

- **Functional Exceptions:** This group is based on functionality and introduce changes to the UPIU based upon the functional exception.
- **Field Based Exceptions:** This group of exceptions are field based and generic. They apply a mask or value to any filed in the UPIU, and also modify reserved fields of the UPIU. They can be used either in a randomized fashion where the mask or value also gets randomized or in a more directed way where you can mimic all the exceptions in the first group by directly setting the mask or value.

Host exceptions can be added to an `svt_jedec_ufs_info_unit` object through randomization of the `info_unit` in the sequence before sending it to the VIP, or through callbacks. Device side exceptions can be added to an `svt_jedec_ufs_info_unit` object, through a factory override or a callback. Exceptions are then carried along until the object is packed, at which point the exception is checked and applied to the attribute as a part of the packing algorithm.

5.2.6.1 Random Exception Injection Using Factory Override or Callback

- Set up a callback or a factory override that uses randomization of an exception list to inject exceptions at random times using weights.
- Register callback or factory override at the start of the simulation.
- Whenever the callback is called or an object is created using factory override, randomization is used to add an exception list with zero or more exceptions to the current object.
- Before transferring the object to the next layer, the exception if generated, will be applied to the object.

5.2.6.2 Directed Exception Injection Using Factory Override or Callback

- Set up a callback or a factory override that uses local static counter variables and information from the applicable object, to decide if an exception should be added and what kind of an exception needs to be added.
- Register callback or factory override at the start of the simulation.
- Whenever the callback is called or object is created using factory override, local variables are updated and exceptions generated based on these variables and the current object.
- Before transferring the object to the next layer, the exception if generated, will be applied to the object.

5.2.6.3 Directed Exception Injection Using Sequence Control or Callback

- Get the handle to components and register the callback that generate exceptions. Unregister the callback at end of sequence.
- Exceptions are generated whenever the callback is called.
- Before transferring the object to the next layer, the exception if generated, will be applied to the object.

5.2.6.4 PHY Adapter Layer

5.2.6.4.1 PHY Error on Lane 0 and No PHY Error on Lane 1

The following command selects the type of error that will be injected. The `error_kind` attribute is the top-selector of a hierarchical set of descriptors.

```
typedef enum svt_mphy_transaction_exception ::error_kind_enum
```

The `svt_mphy_transaction_exception` supports the following error types:

Table 5-2 Error Types

Error Kind Attribute	Description
CTRL_CODES_RSVD_ERROR	Reserved Control Code error
RD_ERROR	RD error
EN_3B4B_5B6B_ERROR	3b4b and 5b6b Coding error
PREPARE_LENGTH_ERROR	prepare_length error
SYNC_LENGTH_ERROR	sync_length error
NO_SYNC_ERROR	prepare_sync_allignment error
ILLEGAL_SYNC_SYMBOL_ERROR	sync_symbol error
NO_OP(7)	No error
INVALID_PWM_BURST_CLOSURE_CAPABILITY_ERROR(8)	Invalid PWM Burst Closure error

Since each lane has one instance of the `svt_mphy_tx` driver, you can create an `svt_mphy_transaction_exception_list` (which has a random `svt_mphy_transaction_exception` object, with some non-zero weights on one or more of these PHY level errors), and set it to the `exception_list` object of just one `svt_mphy_tx` driver instance but not others.

The `svt_mphy_transaction_exception_list` objects inside the `svt_mphy_tx` drivers are placed at:

- `<ufs_device_agent>.mphy_tx[0].tx_transaction_exception_list`
- `<ufs_device_agent>.mphy_tx[1].tx_transaction_exception_list`
- `<ufs_device_agent>.mphy_tx[2].tx_transaction_exception_list`
- `<ufs_device_agent>.mphy_tx[3].tx_transaction_exception_list`

5.2.6.5 Data Link Layer

5.2.6.5.1 AFCx_REQUEST_TIMER_EXPIRED

From a protocol perspective, this timer gets started in the DUT on reception of a data frame from a UFS device. The DUT transmits the AFC before the timer expires.

In the VIP device, it is possible not to use the timer for sending AFC packets for data from the host DUT. It would cause the REPLAY timer to expire in the Host DUT, and the packet will be sent again by the host.

5.2.6.5.2 NAC_FRAME_SYNTAX_ERROR

NAC_FRAME_SYNTAX_ERROR is supported by the `svt_mipi_unipro_frame_exception` class as one of the many frame-level exception types. All the frame-level exception types of the `svt_mipi_unipro_frame_exception` class are listed in Table 5-3.

Table 5-3 Exception Types

Exception Types
FRAME_NO_ERROR
ESC_TYPE_ERROR
CTRL_SYM_TYPE_ERROR
TC_ERROR
RSVD_FIELD_ERROR
DATA_FRAME_SEQ_NUM_ERROR
CTRL_FRAME_SEQ_NUM_ERROR
PADDING_BYTE_ERROR
CREQ_ERROR
RREQ_ERROR
DATA_FRAME_CRC_ERROR
DATA_FRAME_LENGTH_ERROR
AFC_FRAME_SYNTAX_ERROR
NAC_FRAME_SYNTAX_ERROR
EOF_FRAME_SYNTAX_ERROR
CTRL_FRAME_CRC_ERROR
FRAMING_SEQ_ERROR
COF_WITHOUT_SOF_ERROR
REPLACE_AFC_WITH_NAC_ERROR
DROP_CTRL_FRAME_ERROR

You can extend `svt_mipi_unipro_frame_exception_list` to create a `custom_unipro_frame_exception_list` with a random `svt_mipi_unipro_frame_exception` object, with non-zero weights on `FRAME_NO_ERROR` and `NAC_FRAME_SYNTAX_ERROR`.

You need an extra condition constraint, to inject this `NAC_FRAME_SYNTAX_ERROR` only on NAC type frames since, this error type is applicable only to NAC frames.

For example,

```
If (<svt_mipi_unipro_frame_exception object>.xact.frame_traffic_type ==
svt_mipi_unipro_frame::NAC_TT) { error_kind != FRAME_NO_ERROR; }
```

For this to work, you have to first cause a NAC frame. As an example, you can cause a NAC frame by corrupting the transmitted frame with a `DATA_FRAME_CRC_ERROR`.

You can then pass this custom exception list handle, to the `tx_frame_exception_list` object of the data link driver.

The svt_mipi_unipro_frame_exception_list object inside the svt_mipi_unipro_data_link driver is placed at <ufs_device_agent>.data_link.tx_frame_exception_list.

5.2.6.6 Network Layer

5.2.6.6.1 BAD_DEVICEID_ENC and LHDR_TRAP_PACKET_DROPPING

Both of these types are supported by the svt_mipi_unipro_packet_exception class. This provides the support for the following packet-level exceptions:

- PACKET_NO_ERROR
- L3S_ERROR
- N_DEVICE_ID_ERROR

You can extend svt_mipi_unipro_packet_exception_list to create a custom_unipro_packet_exception_list, with a random svt_mipi_unipro_packet_exception object, with non-zero weights on these error types.

There are other constraints that must be provided, aside from the weights on the error kind. For the BAD_DEVICEID_ENC error condition, the VIP checks whether a received N_PDU has a DestDviceID_Enc less than N_DeviceID, if N_DeviceID_valid is true. Therefore, it is important to set up the attribute n_deviceid_valid to be TRUE, and that the mask_or_value variable of the svt_mipi_unipro_packet_exception class, is constrained to be less than the n_device_id attribute.

For the LHDR_TRAP_PACKET_DROPPING error condition, just getting the error kind to L3S_ERROR to corrupt the L3s bit is sufficient.

You can then, pass this custom exception list handle to the tx_packet_exception_list object of the network driver.

The svt_mipi_unipro_packet_exception_list object inside the svt_mipi_unipro_network driver, is placed at <ufs_device_agent>.network.tx_packet_exception_list.

5.2.6.7 Transport Layer

5.2.6.7.1 CONTROLLED_SEGMENT_DROPPING

UFS does not require the UniPro E2E flow control mechanism. UFS will not use CSD and it will be disabled.

See section 9.6.4 UniPro Transport Layer in UFS JEDEC 220a specification for more information.

5.2.6.7.2 BAD_TC

UFS uses a single CPort0 and TC0 traffic class. BAD_TC happens, when a T_PDU is received with a traffic class, that is different from the T_TrafficClass attribute of the destination CPort.

As a device can only respond to commands from a host, it is not possible to generate BAD_TC error in a host for UPIU traffic situations.

See section 5.4.2 MIPI UniPro in UFS JEDEC 220a specification for more information.

5.2.6.7.3 E2E_CREDIT_OVERFLOW

UFS does not use the End-to-end Flow control feature of Unipro for data communication. See section 9.3 UniPro/UFS Transport Protocol Interface (Data Plane) in UFS JEDEC 220a specification for more information.

5.2.6.7.4 SAFETY_VALVE_DROPPING

UFS does not use Cport safety valve (CSV), therefore it is disabled. See section 9.6.4 UniPro Transport Layer in UFS JEDEC 220a specification for more information.

5.2.7 Lane-to-Lane Skew Injection

Skew injection is based on the `t_skew` MPHY configuration parameter which can have a value of 0 to 30 UI as per specification. VIP supports more than 30 UI on the Transmit path and up to 20 UI on the Receive path. This is according to Table 32 of the UniPro specification.

VIP supports fixed and variable skews:

- Fixed skew is applicable for both auto and non-auto modes:

  Tskew(i) = Tskew(i+1) for any “i”
- Variable skew is applicable for auto modes where burst ends after every transmission:

  Tskew(i) ≠ Tskew(i+1) for any “i”

To inject skew at the MPHY level, set the `t_skew` parameter for the lanes on which you want to inject skew in the serial mode of a UFS device transmit (assuming four lanes).

Refer the following code snippet that sets the `t_skew` parameter for four lanes:

```
ufs_device_instance_cfg>.mphy_tx_cfg[0].t_skew = $urandom_range(0,10);
<ufs_device_instance_cfg>..mphy_tx_cfg[1].t_skew = $urandom_range(0,10);
<ufs_device_instance_cfg>..mphy_tx_cfg[2].t_skew = $urandom_range(0,10);
<ufs_device_instance_cfg>..mphy_tx_cfg[3].t_skew = $urandom_range(0,10);
```

5.2.8 Test Mode

Test mode is a UniPro feature for testing the M-PHY by sending specific test patterns as described in the MIPI Alliance Specification for Unified Protocol (UniProSM) Version 1.6, Section 5.7.15.

Test mode can be initiated during LinkDown state and Phase 0 to Phase 5of the PHY Adapter LinkStartUp sequenceAs per the Table 9.1 - DME Service Primitives in the UFS JESD220B specification, the test mode request has to be initiated from the device to transmit test patterns from either the host or device.

5.2.8.1 Default Configuration to Support TEST_MODE

Thefollowing are the default configurations that support `TEST_MODE`:

- `UNIPRO state - LINK_DOWN for both UFS Host & Device VIP`
- `Gear - PWM_Gear_1`
- `Active-Lanes - 1 lane`
- `Mode - SLOW_AUTO_MODE`

Before sending the test patterns, the link can be set to the desired configuration, if it is different from the default configuration.

To set the attributes, a burst should be started on both outbound and inbound links by writing to the `ContBurst` bit of the `PA_PHY_TEST_CONTROL` MIB attribute, either through `DME_SET` or `DME_PEER_SET`.

To complete the link configuration process so that the new attributes come into effect, the burst on the inbound and outbound lanes should be turned off by resetting the `ContBurst` bit of the `PA_PHY_TEST_CONTROL` MIB attribute.

Note
After every DME_SET/PEER_SET, sufficient delay should be added to avoid internal races. Refer to the example for details.

5.2.8.2 Test Pattern Transmission and Test Mode Termination

1. **When host DUT is connected to device VIP**

   a. The device initiates the test pattern transmission by initiating PACP_TEST_MODE_req pacp frame followed by programming the PA_PHY_TEST_CONTROL attribute as shown in the following code snippet:

```
Initiating PACP_TEST_MODE_req pacp_frame class reference
`uvm_do_on(dme_test_mode_seq,p_sequencer.device_virt_seqr.dme_svc_seqr)

Programming PA_PHY_TEST_CONTROL attribute
`uvm_do_on_with(dme_set_seq,p_sequencer.device_virt_seqr.dme_svc_seqr,{attr_se
t_type == 0;
                                        mib_attribute ==
svt_mipi_unipro_types::PA_PHY_TEST_CONTROL;
                                        mib_value == 5'b11000;
                                        gen_selector_index == 0;})

To stop transmitting Test pattern
`uvm_do_on_with(dme_set_seq,p_sequencer.device_virt_seqr.dme_svc_seqr,{attr_se
t_type == 0;
                                        mib_attribute ==
svt_mipi_unipro_types::PA_PHY_TEST_CONTROL;
                                        mib_value == 5'b10000;
                                        gen_selector_index == 0;})
```

   b. If the host DUT wants to send test pattern, it has to program the local PA_PHY_TEST_CONTROL attribute through DME_SET, after it receives the TEST_MODE_req from the device VIP or the device can perform PEER_SET to program this attribute on the host.

2. **When the Host VIP is connected to Device DUT**

   a. PACP_TEST_MODE has to be initiated by device DUT by sending the PACP_TEST_MODE_req pacp_frame and programming the PA_PHY_TEST_CONTROL attribute through DME_SET.

   b. If the host VIP wants to send the test pattern, it has to program the local PA_PHY_TEST_CONTROL attribute through DME_SET or the device DUT can perform PEER_SET to program this attribute on the host as shown in the following code snippet:

```
`uvm_do_on_with(dme_set_seq,p_sequencer.host_virt_seqr.dme_svc_seqr,{attr_set_
type == 0;
mib_attribute  == svt_mipi_unipro_types:A_PHY_TEST_CONTROL;
mib_value   == 5'b11000;
gen_selector_index == 0;})
```

5.2.8.3 Termination

To exit from the test mode, warm or cold reset should be issued to UniPro.

5.2.8.4 Basic Test

Sequence starting with DME_RESET and followed by DME_ENABLE and Test mode.

Test Reference: tb_jedec_ufs_svt_uvm_basic_sys/tests/ts.test_mode_seq_serial.sv.

Figure 5-9 Pacp_test_mode_req frame

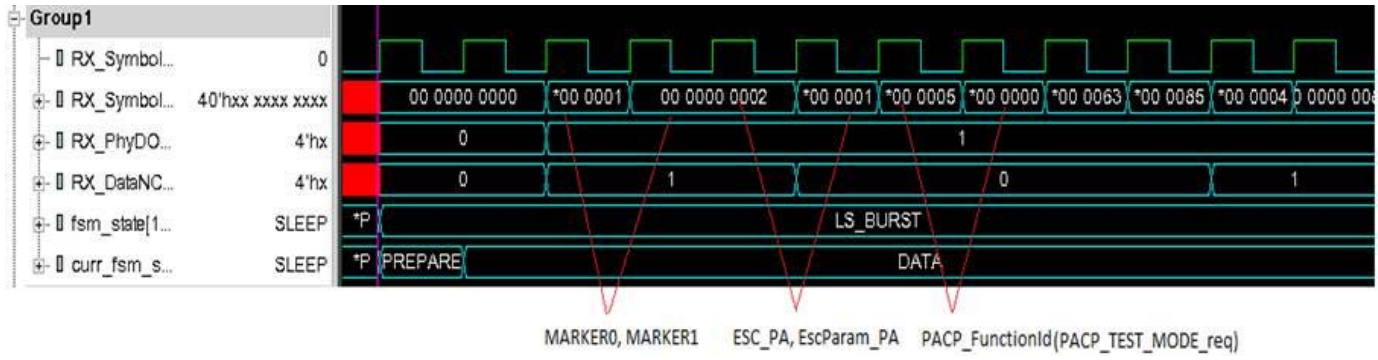


Figure 5-10 PACP_TEST_DATA_0 Frame with CRPAT Pattern

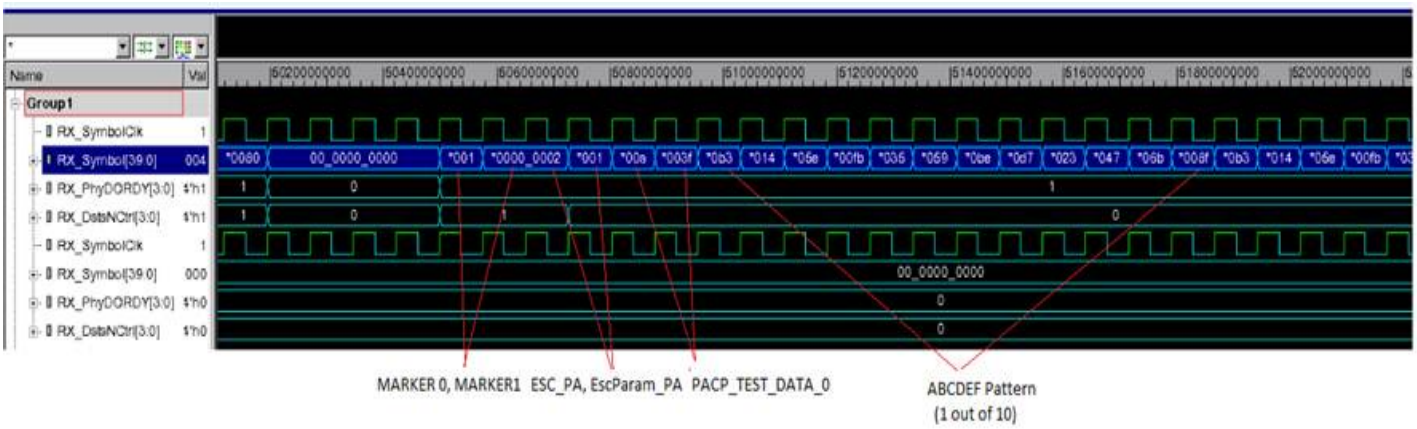
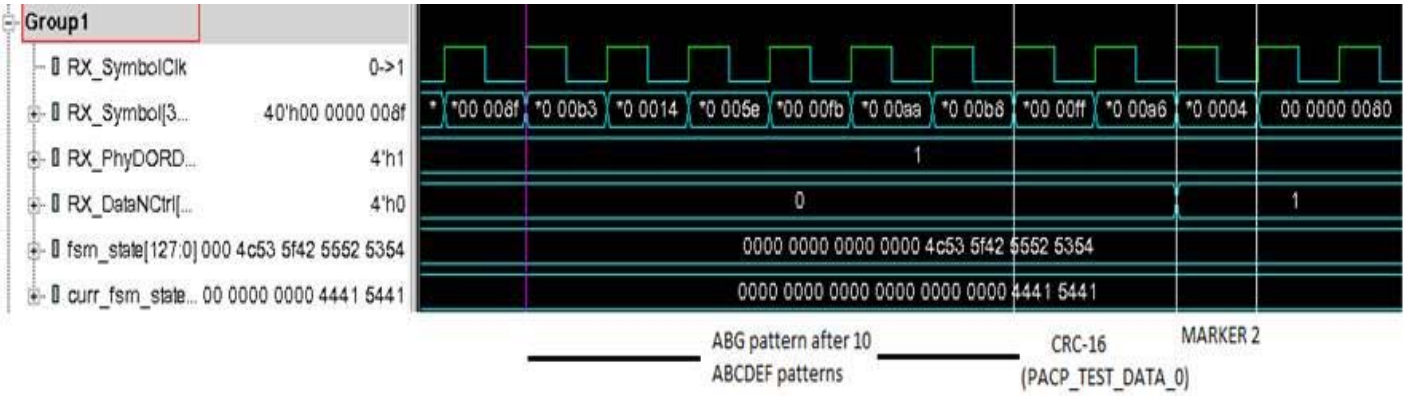


Figure 5-11 Last PACP_TEST_DATA_0 frame



5.2.9 Error Injection in UniPro Layers

Table 5-4 lists the error injections in UniPro for which this version of UFS VIP adds support.

Table 5-4 Error Injection in UniPro Layers

Commands	Transport Layer (TL) Error Code	Exception Messages
TL message	UNSUPPORTED_HEADER_TYPE	Message exception TYPE: L4S_ERROR
	UNKNOWN_CPORTID	Message exception TYPE: CPORT_ID_ERROR
	NO_CONNECTION_RX	Message exception TYPE: CPORT_ID_ERROR
Commands	**Networkt Layer (NL) Error Code**	**Exception Messages**
NL message	UNSUPPORTED_HEADER_TYPE	Not Applicable
	BAD_DEVICEID_ENC	Packet exception TYPE: N_DEVICE_ID_ERROR
	LHDR_TRAP_PACKET_DROPPING	Packet exception TYPE: L3S_ERROR
Commands	**Phy Layer Error Code**	**Exception Messages**
L1.5	PWR_ERROR_CAP. The request was rejected because the requested configuration exceeded the Link's capabilities.	PACP exception TYPE: CFG_RESULT_CODE_ERROR

The use model continues to be similar to other error injection features.

Note
Use the MPHY callback to achieve the indication of the Hibernate-Exit failure Hibernate-Exit-CTRL SAP

5.2.10 Using Internal VIP Clock in RMMI Mode

In RMMI mode, there are four clocks specified in the M-PHY specification. This section defines the model requirement for the clocks `RX_CfgClk` or `TX_CfgClk`. In RMMI mode, the `CfgClk` is an input to the model. The M-PHY specification does not define any specific frequency or frequency range for this clock. It also does not define any phase relation with the symbol clock. The M-PHY model works with any clock frequency provided by the you. All the control transactions are processed using this clock.

You have to ensure that there is sufficient `CfgClk` cycles provided for a successful RCT before starting the new burst. The M-PHY model has an inbuilt `EVENT_MPHY_RECONFIGURE_TRIGGER` notification for UVM that can be used in the testbench to ensure the RCT completion.

5.2.10.1 RX_SymbolClock or TX_SymbolClock

The `TX_SymbolClock` is generated by the TX controller and passed to TX PHY. `RX_SymbolClock` is generated by RX PHY and passed to the RX controller. When the model is used as TX RMMI Controller or RX RMMI PHY, it generates the `TX_SymbolClock` and `RX_SymbolClock` respectively. The clock frequency generated by the model is aligned to various gears and modes as defined in Table 8 and 9 in MIPI Alliance Specification for M-PHY Version 1.40.0.

If the model is used as TX RMMI PHY or RX RMMI controller, the symbol clock is expected as an input to the model. As per MIPI Alliance Specification for M-PHY Version 1.40.00 - 30 November 2011 and later, `TX_SymbolClk` is generated by MTX Controller and it is an input to MTX PHY. An enhancement,

TX_SymbolClk being generated as an output from MTX PHY is added in MPHY VIP. This feature can be enabled by setting the configuration parameter enable_phy_clock_gen in MPHY VIP cfg. When enable_phy_clock_gen is set to 1, TX_SymbolClk will be the output from MTX PHY VIP and will be the input to the MTX Controller DUT. Thus, the clock connections should be swapped in the top.sv file as shown below:

- **Remove the following connection**:

```
assign mtx_dut_controller_if[i].TX_SymbolClk = mtx_dut_phy_if[i].TX_SymbolClk;
```

- **Add the following connection**:

```
assign mtx_dut_phy_if[i].TX_SymbolClk = mtx_dut_controller_if[i].TX_SymbolClk;
```

- As per the MIPI Alliance Specification for M-PHY Version 1.40.00 - 30 November 2011 and later, RX_SymbolClk and TX_SymbolClk generation may get disabled when the FSM state is not in LINE-CFG, PWM-BURST, SYS-BURST, or HS-BURST states.

To enable this behavior in M-PHY VIP, you need to set the bit allow_symbol_clock_disable to 1 in the shared cfg file. In the DUT environment, if the DUT is not generating SymbolClk in save states, set allow_symbol_clock_disable bit to 1 in the shared cfg file and TX_SymbolClk is generated by MTX CONTROLLER. If MTX DUT PHY generates its own TX_SymbolClk, then it can be looped back into MPHY VIP CONTROLLER.

To enable this behavior, you need to set the bit external_symbol_clock to 1 in the shared cfg file. Also, the DUT TX_SymbolClk needs to be connected to the ext_TX_SymbolClk pin in HDL interconnect. When this behavior is enabled, DUT is responsible for the switching of TX_SymbolClk (on different gears) and both MTX DUT PHY and VIP DUT CONTROLLER work on the V RX_RefClock or TX_BitClk clock. These clocks are not used by the model.

V SYS clock runs on 19.2 Mhz, 26 Mhz, 38.4 Mhz, or 52 Mhz. This clock is used for SYS signaling on serial interface. Therefore, TX_SymbolClock or RX_SymbolClock runs at 1/10 times the SYS clock frequency, if symbol_length=10.

To set SYS clock at 26 Mhz and RX_SymbolClk at 2.6 Mhz at RMMI, use define in top level testbench file as shown as follows:

```
`define SVT_MPHY_SYS_CLK_PERIOD 38.46ns
```

5.2.11 Using Clock-Jitter in the Serial Mode

The M-PHY physical layer supports clock jitter in the serial mode. For serial mode, the jitter will be introduced for the internal transmit clock, generated by the VIP.

Table 5-5 shows the new attributes added for clock-jitter (svt_mphy_configuration class).

Table 5-5 Configuration Parameters

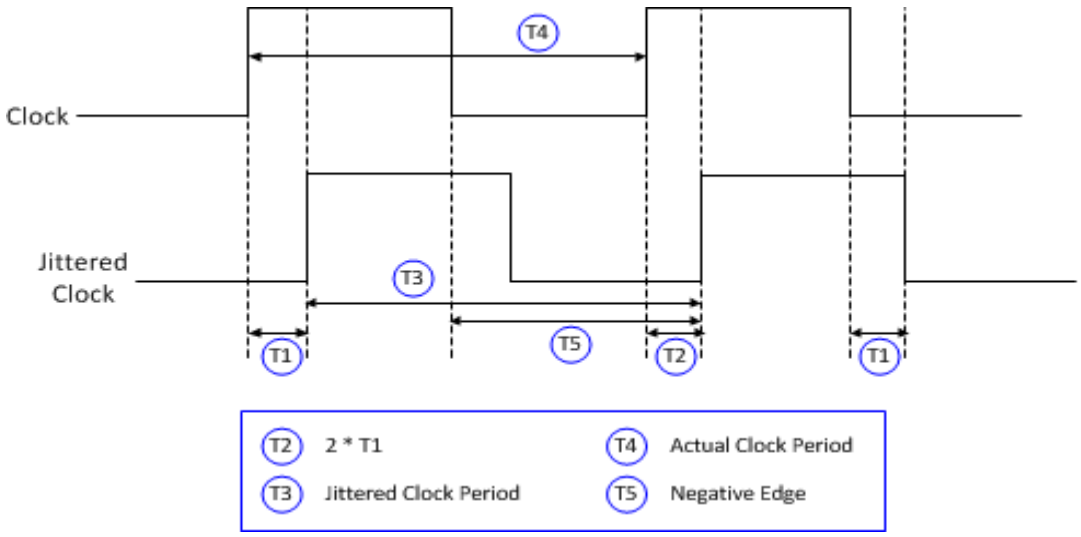
Configuration Parameter	Description
enable_clock_jitter Default = 0;	Enables the jitter insertion in the generated clock, resulting in serial-jittered data.
mphy_jitter_type Default = DETERMINISTIC_JITTER	Specifies the type of jitter to be introduced. 0 – RANDOM_JITTER 1 – DETERMINISTIC_JITTER 2 – CONSTANT_PERIODIC_JITTER

Table 5-5 Configuration Parameters

Configuration Parameter	Description
percentage_of_jitter_per_clk Default =17	Specifies the percentage of the jitter insertion per clock. As per the specification, the maximum allowable random jitter is 0.17 UI for the serial mode. This reduces the '1' width of the clock by half of the specified percentage jitter and increases the '0' width of the clock by specified percentage. This is applicable only in HS-MODE.
number_of_clks_with_jitter	Specifies the number of clocks with jitter insertion. The same number of clocks would be repeated without jitter in the clock. For example, if `number_of_clks_with_jitter` = 100, then there are 100 clocks with jitter followed by 100 actual frequency clocks as per the mode of the device. This pattern continues throughout. This is applicable only in HS-MODE and for CONSTANT_PERIODIC_JITTER type.
raise_of_jitter_per_clk Default = 1	Specifies the increase of jitter per clock till it reaches `percentage_of_jitter_per_clk`. Once it reaches this value, it starts again from 1% till `percentage_of_jitter_per_clk`. This is applicable only when the type of jitter is configured as DETERMINISTIC_JITTER. This is applicable only in HS-MODE.

Jitter Insertion parameters which co-relate to the configuration parameters in Table 5-5 is shown in Figure 5-12.

Figure 5-12 Jitter Insertion



- Jitter insertion is applicable only when the `enable_clock_jitter` is set to 1 and speed mode is HS-MODE.
- T1 specifies the drift of the positive edge as an effect of the jitter insertion.

- T2 is the percentage of the expected jitter insertion per UI specified by the `percentage_of_jitter_per_clk` configuration parameter.
- T3 - Clock position when jitter is introduced.
- T4 - Clock position when jitter is not introduced.
- T5 - Result of the drift of the negative edge as an effect of the jitter insertion.
- T4 - clock position When jitter was not introduced

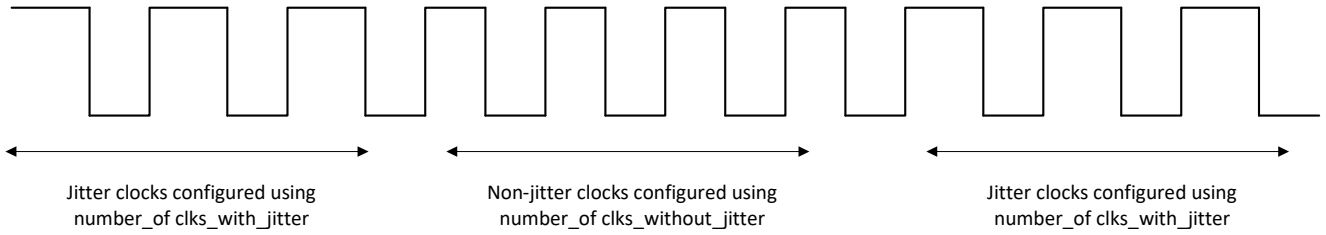
5.2.11.1 Usage Examples

5.2.11.1.1 Insertion of 17% of CONSTANT_PERIODIC_JITTER

- `interface_type` = SERIAL;
- `enable_clock_jitter` = 1;
- `number_of_clks_with_jitter` = 100;
- `percentage_of_jitter_per_clk` = 17;

The final generated clock is shown in Figure 5-13.

Figure 5-13 Clock Generated for Insertion of 17% CONSTANT_PERIODIC_JITTER

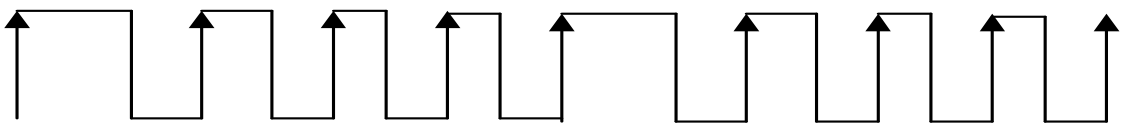


5.2.11.1.2 Insertion of 17% of RANDOM_JITTER

- `interface_type` = SERIAL;
- `enable_clock_jitter` = 1;
- `percentage_of_jitter_per_clk` = 17;

Every clock has a random jitter within the value of `percentage_of_jitter_clk` parameter. The final generated clock would be as shown in Figure 5-14.

Figure 5-14 Clock Generated for Insertion of 17% RANDOM_JITTER



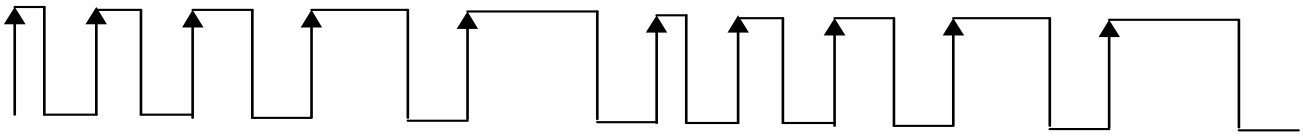
5.2.11.1.3 Insertion of 17% of DETERMINISTIC_JITTER

- `interface_type` = SERIAL;
- `enable_clock_jitter` = 1;

- `percentage_of_jitter_per_clk` = 17;
- `raise_of_jitter_per_clk` = 1;

Every clock has an incremental value of jitter as per the `raise_of_jitter_per_clk` parameter, and the pattern rotates throughout the simulation. The final generated clock would be as shown in Figures 5-15.

Figure 5-15 Clock Generated for Insertion of 17% DETERMINISTIC_JITTER



5.3 HCI Layer

The following are the HCI feature implementation in the UFS VIP:

- UFS host VIP is enhanced to function as HCI layer only without any lower layer stack present.
- This use model is applicable only in the test suite environment.
- The UFS-HOST-HCI agent would populate `UVM_REG_ITEM`s when for each of the command UPIUs instructed form the test environment respectively as per HCI protocol.
- This HCI feature is enabled only when `is_hci` agent configuration is set to 1 and user need not pass any interfaces (RMMI/Serial/CPORT) from the test environment.
- The HCI layer is applicable only when the VIP is used in a UFS test suite environment.

For more information, see the test suite document.

6 VIP Tools

6.1 Using Native Protocol Analyzer for Debugging

6.1.1 Introduction

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

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

6.1.2 Prerequisites

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

Compile Time Options

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

  For more information on how to set the FSDB dumping libraries, see Appendix B section in *Linking Novas Files with Simulators and Enabling FSDB Dumping* guide available at: `$VERDI_HOME/doc/linking_dumping.pdf`.

  You can dump the transaction database either by setting the `pa_format_type` configuration variable or by passing the runtime switch as shown below:

  Configuration Variable Setting:

  Set `enable_protocol_transport_xml_gen` parameter of `svt_jedec_ufs_agent_configuration` to enable the generation of .xml and/or FSDB files

  For example,

  ```
  device_cfg. enable_protocol_transport_xml_gen = 1;
  device_cfg.pa_format_type = svt_xml_writer::FSDB;
  ```

  Similarly for UFS receiver:

```
<jedec_ufs_rcvr_agent_configuration>.jedec_ufs_cfg.pa_xml_generation_enable = 1
```

Runtime Switch:

```
+svt_enable_pa=fsdb
```

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

6.1.3 Invoking Protocol Analyzer

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

Post-processing Mode

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

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

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

Interactive Mode

- Run 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 is updated during the simulation.

6.1.4 Documentation

The documentation for Protocol Analyzer is available at:

```
$VERDI_HOME/doc/Verdi_Transaction_and_Protocol_Debug.pdf.
```

6.1.5 Limitations

Interactive support is available only for VCS.

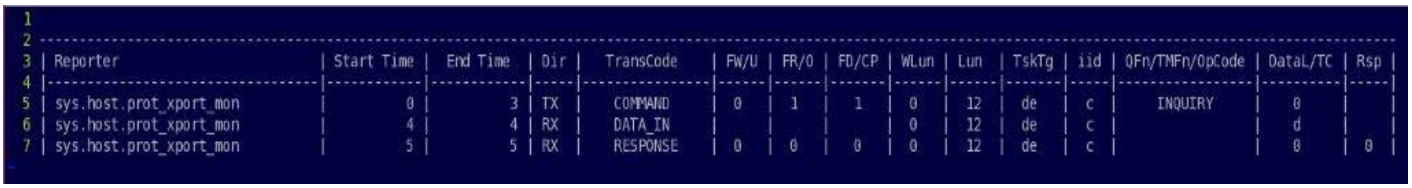
7 Troubleshooting

This chapter provides some useful information that regarding troubleshooting t common issues that you might encounter while using the UFS VIP. It explains the following topics:

- Info Unit Trace
- Enabling Tracing

7.1 Info Unit Trace

The Info Unit object from the UFS transport Layer are available in the following format:



Reporter	Start Time	End Time	Dir	TransCode	FW/U	FR/O	FD/CP	WLun	Lun	TskTg	iid	QFn/TMFn/OpCode	DataL/TC	Rsp
sys.host.prot_xport_mon	0	3	TX	COMMAND	0	1	1	0	12	de	c	INQUIRY	0	
sys.host.prot_xport_mon	4	4	RX	DATA_IN				0	12	de	c		d	
sys.host.prot_xport_mon	5	5	RX	RESPONSE	0	0	0	0	12	de	c		0	0

- Dir - Direction of the UPIU
- TransCode - Transaction Code of the UPIU
- FW/U - Flag_w and Flag_U overlapped on the same field (Flag_W is applicable for Command UPIU and Flag_U is applicable for Response UPIU)
- FR/O - Flag_r and Flag_o overlapped on the same field (Flag_R is applicable for Command UPIU and Flag_O is applicable for Response UPIU)
- FD - Flag_D applicable for Response UPIU
- WLun - wlun_id field indicates whether LUN is Wlun or not
- Lun - unit_number_id of the LU
- TskTg - Task Tag of the UPIU
- QFn/TMFn - Query Function and Task Management Function overlapped on the same field
- DataL/TC - Data Segment Length and Data Transfer count overlapped on the same field (in case of RTT, this field shows the data transfer count)
- Rsp - Response field of the UPIU

7.2 Enabling Tracing

```
<env_instance_name> <agent instance name>.<layer identifier_mon>.<traffic
class_sap identifier>.message_trace
```

where,

- `agent instance name`: Indicates the instance name of the corresponding VIP agent.
- `layer identifier_mon`: Indicates the monitor trace file of the UFS Transport layer.

The trace can be enabled or disabled by the following fields in the UFS Agent Config:

```
vip_cfg.enable_protocol_transport_tracing=1;
vip_cfg.enable_protocol_transport_reporting=1;
```

Table 7-1 Generated Trace Files

Trace File Type	Trace File Names
Info unit Trace	Trace sys.host.prot_xport_mon.info_unit_trace

8 Reporting Problems

8.1 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.

8.2 Debug Automation

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

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

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

8.3 Enabling and Specifying Debug Automation Features

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

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

Note the following about the plusarg:

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

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

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

The following table explains each control string:

Table 8-1 Control Strings for Debug Automation plusarg

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

Examples:

Enable on all VIP instances with default options:

```
+svt_debug_opts
```

Enable on all instances:

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

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

Enable on all instances:

- starting at time 1000 until time 1500:

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

Enable debug feature on all instances using default options:

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

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

Note

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

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

8.4 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.
- *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).

8.5 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.

8.5.1 VCS

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

```
-debug_access
```

For more information on how to set the FSDB dumping libraries, see Appendix B section in *Linking Novas Files with Simulators and Enabling FSDB Dumping* guide available at:
`$VERDI_HOME/doc/linking_dumping.pdf.`

8.5.2 Questa

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

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

8.5.3 Incisive

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

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

8.6 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.

8.7 Sending Debug Information to Synopsys

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

1. Create a description of the issue under investigation. Include the simulation time and bus cycle of the failure, as well as any error or warning messages that are part of the failure.
2. Create a description of your verification environment. Assemble information about your simulation environment, making sure to include:
   - OS type and version
   - Testbench language (SystemVerilog or Verilog)
   - Simulator and version
   - DUT languages (Verilog)
3. Use the VIP case submittal tool to pack a file with the appropriate debug information. It has the following usage syntax:

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

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

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

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

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

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

8.8 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.