Verification Continuum™

VC Verification IP SWD UVM User Guide

Version Q-2020.06, June 2020

SYNOPSYS®

Contents

Preface

About This Guide

This guide contains installation, setup, and usage material for VC VIP for Serial Wire Debug (SWD). Also, it is for the design or verification engineers who want to verify SWD VIP operation using an Universal Verification Methodology (UVM) testbench written in SystemVerilog. Readers are assumed to be familiar with SWD VIP, Object Oriented Programming (OOP), SystemVerilog, and UVM techniques.

Guide Organization

The chapters of this guide are organized as follows:

- Chapter 1, Introduction, introduces the SWD VIP and its features.
- Chapter 2, Download and Installation, describes system requirements and provides instructions on how to install, configure, and begin using the SWD VIP.
- Chapter 3, Design Directory Maintenance, leads you through setting up the VIP.
- Chapter 4, Overview of the SWD VIP, provides detail for each UVM-derived class.
- Chapter 5, VIP Tools, provides useful information that can help you troubleshoot common issues that you may encounter while using the SWD VIP.
- Appendix A, Reporting Problems, outlines the process for working through and reporting VC VIP for SWD issues.

Customer Support

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

- ❖ Accessing SolvNetPlus

  Access documentation through SolvNetPlus from the following location:

  https://solvnetplus.synopsys.com (Synopsys password required)

- ❖ Contacting the Synopsys Technical Support Center
  - ✦ Go to https://solvnetplus.synopsys.com and open a case.

    Enter the information according to your environment and your issue.

    If applicable, provide the information noted in Appendix A, Reporting Problems.
  - ✦ Send an e-mail message to support_center@synopsys.com
    - ✧ Include the Product name, Sub Product name, and Product version for which you want to register the problem.
    - ✧ If applicable, provide the information noted in Appendix A, Reporting Problems.
  - ✦ Call 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

VC VIP for SWD supports the verification of designs that include interfaces implementing SWD specifications. This document describes the use of SWD VIP in testbenches that comply with the SystemVerilog Universal Verification Methodology (UVM). This approach leverages advanced verification technologies and tools that provide the following features:

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

This document assumes that you are familiar with the SWD protocol, object oriented programming, SystemVerilog, and UVM.

This chapter discusses the following topics:

- Product Overview
- Simulator Support
- Methodology Features
- SWD Feature Support
- SWD Specification

1.1 Product Overview

The SWD VIP is a suite of UVM-based verification components that are compatible for use with SystemVerilog-Compliant testbenches. The SWD VIP suite simulates SWD transactions, through active agents, as defined by the SWD specification. The suite provides a SWD agent that contains a Master/Slave component and a Monitor component along with the handles to activity ports. After instantiating the agent, you can select and combine active and passive modes of the agent to create an environment that verifies SWD features in the DUT. The SWD agent supports all the functionalities normally associated with active and passive UVM components, including the creation of transactions, checking and reporting protocol correctness, and transaction logging.

1.2 Simulator Support

The SWD VIP suite supports the following languages and simulators:

- Languages
  - SystemVerilog
- Simulators
  - VCS
  - IUS
  - MTI

1.3 Methodology Features

The SWD VIP currently supports the following methodology features:

- IP organized as an agent, which includes a Master/Slave component.
- Analysis ports for connecting to the scoreboard or any other component.
- Callbacks

1.4 SWD Feature Support

The SWD VIP supports the following protocol-related features:

- SWD Master feature set
  - Supports programmable clock frequency of operation
  - Configurable turnaround period
  - Programmability to send/not-send the Data phase after `WAIT` or `FAULT` response
  - Supports the following command types:

    i.Debug Port Write
    ii.Debug Port Read
    iii.Access Port Write
    iv.Access Port Read
    v.Line Reset
  - Extensive Error Injection support
  - Built-in protocol checks
- SWD Slave feature set
  - Configurable turnaround period
  - Supports the following ACK types:

    i.OK
    ii.WAIT
    iii.FAULT

iv.INVALID

- ✧ Callback support after request phase so that you can set the required ACK Type.
- ✧ Callback support after ACK phase so that you can set the required data value for READ commands.

1.5 SWD Specification

The SWD VIP supports the following SWD standard specification:

IHI0031A_ARM_debug_interface_v5.pdf

2 Download and Installation

The Download and Installation chapter discusses the following topics:

- Hardware Requirements
- Software Requirements
- Downloading From SolvNet
- Licensing
- Environment Variables
- dw_vip_setup Administrative Tool

Note If you encounter any problem in installing the SWD VIP, contact customer support.

2.1 Hardware Requirements

The SWD VIP requires the following configuration for Solaris or Linux workstation:

- 400 MB available disk space for installation
- 16 GB Virtual memory (recommended)

2.2 Software Requirements

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

This section consists of the following subsections:

- Platform/OS and Simulator Software
- Software Common Licensing Software
- Third-Party Software

2.2.1 Platform/OS and Simulator Software

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

SWD VIP Release Notes

2.2.2 Software Common Licensing Software

The Software Common Licensing (SCL) software provides the licensing function for the SWD VIP. For more information, see Licensing.

2.2.3 Third-Party Software

You can access the SWD VIP documentation using the following third-party softwares:

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

2.3 Downloading From SolvNet

You can download the software from the Download Center using either HTTPS or FTP, or with a command-line FTP session. If you do not know your Synopsys SolvNet password or you do not remember it, go to http://solvnet.synopsys.com.

You require the passive mode of FTP. The passive command toggles between the passive and active mode. If your FTP utility does not support the passive mode, use HTTP. For more information, see the following Web page:

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

This section consists of the following subsections:

- Downloading From Electronic Software Transfer System
- Downloading Using FTP With a Web Browser

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

2.3.1 Downloading From Electronic Software Transfer System

To download from the Electronic Software Transfer (EST) System Download Center, perform the following steps:

1. Point your web browser to http://solvnet.synopsys.com.
2. Enter your Synopsys SolvNet Username and Password.
3. Click **Sign in**.
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

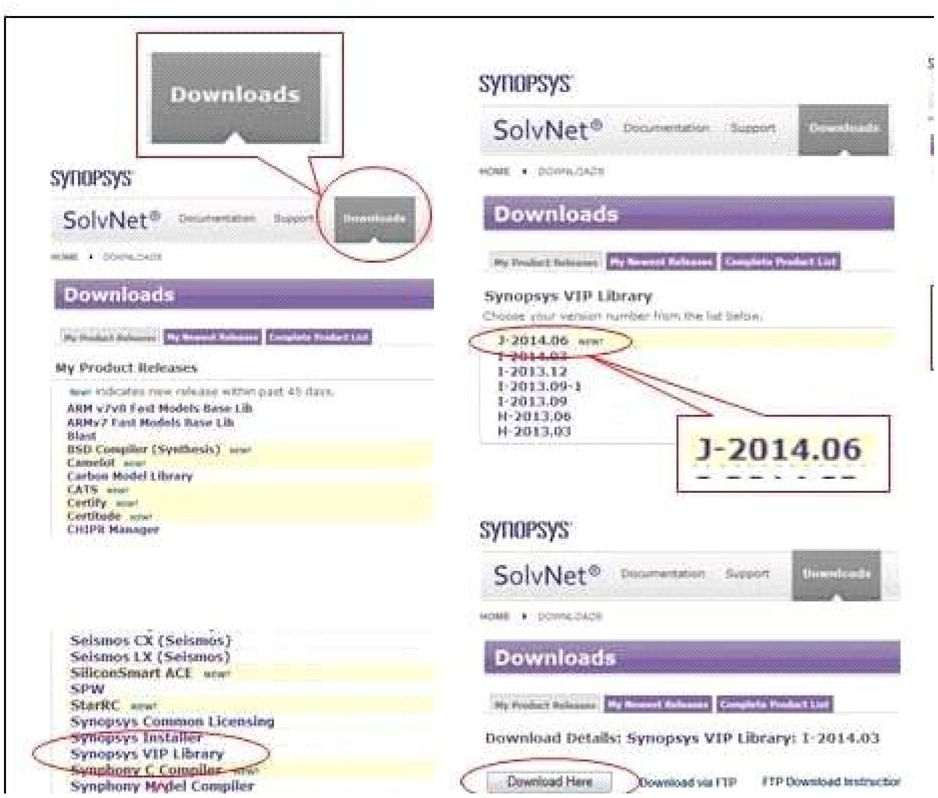
c. <release_version>

d. **Download Here** button

e. **Yes, I Agree to the Above Terms** button

f. **Download** *.run* file of the VIP

Figure 2-1 SolvNet Selections for VIP Download



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

```
% setenv DESIGNWARE_HOME VIP_installation_path
```

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.

Note
You may download multiple files simultaneously.

2.3.2 Downloading Using FTP With a Web Browser

Follow Step 1to Step 5 of Section 2.3.1 and then, perform the following steps:

1. Click the Download via FTP link instead of the Download Here button.
2. Click the Click Here To Download button.

3. Select the files that you want to download.
4. Follow browser prompts to select a destination location.

Note
If you are unable to download the Verification IP using these instructions, refer to "Customer Support" section to obtain support for download and installation.

2.4 Licensing

The SWD VIP uses the Synopsys Common Licensing (SCL) software to control its usage. You can find the general SCL information from the following link:

http://www.synopsys.com/keys

Perform the VIP License check order and feature names as per the following steps:

✦ `VIP-SWD-SVT`

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

The licensing key must reside in the files that are indicated by specific environment variables. For information about setting these licensing environment variables, see Environment Variables.

2.4.1 License Keys

This section consists of the following subsections:

✦ Simulation Modes
✦ Simulation License Suspension
✦ Simulation Modes

2.4.2 License Polling

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

✦ To enable license polling, set `DW_WAIT_LICENSE` to 1.
✦ To disable license polling, unset `DW_WAIT_LICENSE`. By default, license polling is disabled.

2.4.3 Simulation License Suspension

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

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

2.4.4 Simulation Modes

The SWD VIP supports the following two defines:

- Mandatory defines: The following defines are mandatory to make the VIP work in the UVM mode:
  - `SVT_UVM_TECHNOLOGY`: This define makes the SVT base classes to incorporate the UVM methodology base classes.
  - `SVT_SWD`: This define makes the methodology agnostic VIP to incorporate the SVT UVM base classes.
- Optional defines: The UVM define, namely `UVM_DISABLE_AUTO_ITEM_RECORDING`, is an optional define for the VIP. By default, it is defined in the VIP defines file. Note that if you execute `start_item/finish_item` (or `uvm_do* macro) executed from uvm_sequence#(REQ, RSP), it automatically triggers `begin_event` and `end_events` via calls to `begin_tr` and `end_tr`. While convenient, it is generally the responsibility of drivers to mark a transaction's progress during execution. To allow the driver to control sequence item timestamps, events, and recording, you must add `+define+UVM_DISABLE_AUTO_ITEM_RECORDING` when compiling the UVM package. Alternatively, you may use a transaction's event pool and events to define custom events for the driver to trigger and the sequences to wait on. Any in-between events such as marking the beginning of the address and data phases of the transaction execution could be implemented via the events pool.

2.5 Environment Variables

The SWD VIP verification models require the following environment variables and path settings:

- `DESIGNWARE_HOME`: The absolute path to the directory where the VIP is installed.
- `DW_LICENSE_FILE`: The absolute path to a file that contains 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.

Note
For faster license checkout of Synopsys VIP software, ensure to place the desired 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.

Note
You can set the Synopsys VIP License using any of the three license variables in the following order:

```
DW_LICENSE_FILE -> SNPSLMD_LICENSE_FILE -> LM_LICENSE_FILE
```

If DW_LICENSE_FILE environment variable is enabled, the VIP will ignore SNPSLMD_LICENSE_FILE and LM_LICENSE_FILE settings. Therefore, to get the most efficient Synopsys VIP license checkout performance, set the DW_LICENSE_FILE with only the License servers, which contains Synopsys VIP licenses. Also, include the absolute path to the third-party executable in your PATH variable.

2.6 dw_vip_setup Administrative Tool

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

The `dw_vip_setup` utility provides the following features:

- Creates the design directory (*design_dir*), which contains transactors, support files (include files), and examples (if any).
- Add a specific version of SWD VIP from *DESIGNWARE_HOME* to a *design directory*.
- The `dw_vip_setup` utility is as follows:
  - Adds, removes, or updates SWD VIP models in a *design directory*.
  - Adds example testbenches to a *design directory*, the SWD VIP models they use (if necessary), and creates a script for simulating the testbench using any of the supported simulators.
  - Restores (cleans) example testbench files to their original state.
  - Reports information about your installation or *design directory*, including version information.
  - To create a *design directory* and add a model to use it in a testbench, use the following command:

  ```
  % $DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a
  swd_<master/slave>_agent_svt -svtb
  ```
  - Supports Verdi Protocol Analyzer
  - Supports the FSDB wave format

The SWD VIP provides the following models:

- `swd_master_agent_svt`
- `swd_slave_agent_svt`
- `swd_master_svt`
- `swd_slave_svt`

This section consists of the following subsections:

- Setting Environment Variables
- dw_vip_setup Command

2.6.1 Setting Environment Variables

Before running `dw_vip_setup`, the `DESIGNWARE_HOME` environment must point to the location where the VIP is installed.

2.6.2 dw_vip_setup Command

Invoke `dw_vip_setup` from the command prompt. The `dw_vip_setup` command checks the syntax of command-line arguments and makes sure that the requested input files exist. The syntax 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 switches and their applicable sub-switches.

Table 2-1 Setup Program Switch Descriptions

Switch	Description
-a`[dd]` (*model* [**-v**[ersion] *version*]) …	Adds the specified model or models to the specified design directory or the current working directory. If you do not specify a version, the latest version is assumed. The model names are as follows: • `swd_master_agent_svt` • `swd_slave_agent_svt` • `swd_master_svt` • `swd_slave_svt` The -add switch makes `dw_vip_setup` to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-r`[emove]` *model*	Removes all versions of the specified model or models from the design. The `dw_vip_setup` command does not attempt to remove any include files used solely by the specified model or models. The model names are as follows: • `swd_master_agent_svt` • `swd_slave_agent_svt` • `swd_master_svt` • `swd_slave_svt`
-u`[pdate]` (*model* [**-v**[ersion] *version*]) …	Updates to the specified model version for the specified model or models. The `dw_vip_setup` script updates to the latest models when you do not specify a version. The model names are as follows: • `swd_master_agent_svt` • `swd_slave_agent_svt` • `swd_master_svt` • `swd_slave_svt` The -`update` switch makes `dw_vip_setup` to build suite libraries from the same suite as the specified models, and to copy other necessary files from *$DESIGNWARE_HOME*.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-e`[xample]` {*scenario* | *model/scenario*} [**-v**[ersion] *version*]	The `dw_vip_setup` script configures a testbench example for a single model or a system testbench for a group of models. The script creates a simulator-run program for all supported simulators. If you specify a scenario (or system) for example, testbench the models needed for the testbench are included automatically and do not need to be specified in the command. **Note** Use the `-info` switch to list all available system examples.
`-ntb`	Not supported.
`-svtb`	Use this switch to set up models and example testbenches for SystemVerilog. The resulting design directory is streamlined and you can use it 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*	When you specify the `-info design` switch, `dw_vip_setup` prints a list of all models and libraries, installed in the specified design directory or the current working directory, and their respective versions. You can use the output from `-info` design to create a `model_list` file. When you specify the `-info home` switch, `dw_vip_setup` prints a list of all models, libraries, and examples, available in the currently-defined `$DESIGNWARE_HOME` installation, and their respective versions. The command prints reports to STDOUT.
-h`[elp]`	Returns a list of valid `dw_vip_setup` switches and their correct syntax.
model	The SWD VIP models are as follows: • `swd_master_agent_svt` • `swd_slave_agent_svt` • `swd_master_svt` • `swd_slave_svt` The *model* argument defines the model or models that `dw_vip_setup` acts upon. This argument is not needed with the `-info` or `-help` switches. All switches that require the model argument may also use a model list. You may specify a version for each listed model using the `-version` option. If omitted, `dw_vip_setup` uses the latest version. The -`update` switch ignores the model-version information.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-m[odel_list] *filename*	Specifies a file name which contains a list of model names to be added, updated or removed in the design directory. This switch is only valid when following an operation switch such as -add, -update or -remove. Only one model name per line and each model may include a version selector. The default version is 'latest'. This switch is optional, but if given the filename argument is required. Lines in the file starting with the pound symbol (#) will be ignored.
-s[uite_list] *filename*	Specifies a file name which contains a list of suite names to be added, updated or removed in the design directory. This switch is only valid when following an operation switch such as -add, -update or -remove. Only one suite name per line and each suite may include a version selector. The default version is 'latest'. This switch is optional, but if given the filename argument is required. Lines in the file starting with the pound symbol (#) will be ignored.
`-b/ridge`	Updates the specified design directory to reference the current `DESIGNWARE_HOME` installation. All product versions contained in the design directory must also exist in the current `DESIGNWARE_HOME` installation.
-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`.
`-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.

Note The `dw_vip_setup` command treats all lines beginning with “#” as comments.

3 Design Directory Maintenance

The Design Directory Maintenance chapter discusses the following topics:

3.1 Adding a Single VIP

After installing the VIP, you must set up the VIP for use. All VIP suites contain various components such as transceivers, masters, slaves, and monitors depending on a 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 components. For example, the SWD VIP contains the following components:

- `swd_master_agent_svt`, `swd_slave_agent_svt`: This is the VIP agent model, which encapsulates a sequencer, a driver, and a monitor.
- `swd_master_svt`, `swd_slave_svt`: This is the VIP agent driver or receiver model.

You can set up either an individual component, or the entire set of components within one protocol suite. Use the `dw_vip_setup` Synopsys tool available in *`$DESIGNWARE_HOME/bin`*.

To get help on `dw_vip_setup`, use the following command:

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

To set up a model component—`swd_<master/slave>_agent_svt` to the *`design_dir`* directory, use the following command:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/design_dir -add
swd_<master/slave>_agent_svt -svlog
```

To set up an entire set of components to the *design_dir* directory, use the following command:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/design_dir -add
swd_<master/slave>_agent_svt -add  swd_<master/slave>_svt -svlog
```

or

```
%$DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/design_dir -add -model_list
<input_file_containing_models_one_per_line> -svlog
For example, %$DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/design_dir -add -model_list
<file_name> -svlog

cat <file_name>:
swd_<master/slave>_agent_svt
swd_<master/slave>_svt
```

This command sets up all required files in */tmp/design_dir*. The dw_vip_setup utility creates three directories in design_dir, which contains all necessary model files. The following three directories include files for every VIP:

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

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.

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

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 Verification and Implementation IPs you have in your project. It is recommended not to create this directory in $DESIGNWARE_HOME.

3.2 Adding Multiple VIPs

All VIPs for a project must be set up in a single common directory after you execute the `*.run` file. You might 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 SWD and SWD VIP suites.

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

After installing, you must set up and locate the SWD and SWD 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 SWD to the same design_dir directory as AXI.
%unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/design_dir
-add swd_<master/slave>_agent_svt swd_<master/slave>_svt -svlog
// Add USB to the same design_dir directory as AMBA and SWD
%unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/design_dir
-add usb_system_env_svt -svlog
```

To specify other model names, see the VIP documentation.

By default, all VIPs use the latest installed version of SystemVerilog technology (SVT). Synopsys maintains backward compatibility with the previous versions of SVT. As a result, you may select different models using the previous versions of SVT.

If you participate in an Early Adopter (EA) program, you might get a VIP which brings along a newer SVT version other than what is currently installed. In this case, note the following after you set up the EA version of the model:

- All installed VIP models use the EA SVT after the EA VIP is installed.
- Synopsys attempts to maintain backward compatibility of new EA SVT releases with the latest LCA VIPs.
- In the case, where the EA SVT changes are not backward incompatible, you can use the `-svt` option of `dw_vip_setup` to use a specific version of SVT.
- Synopsys does not recommend you to use the `-svt` option as you must remember to remove this when all VIPs move to a compatible version of SVT. Use `-svt` as a workaround only.

Following is an example of using the `-svt` switch:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/vip_model_design_dir -add
amba_system_env_svt svlog -svt K-2015.09
```

3.3 Updating and Deleting VIP Components

To update an existing model, perform the following steps:

1. Install the model at 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 swd_<master/slave>_agent_svt -add  swd_<master/slave>_svt -svlog
```

You can also update your *`design_dir`* by specifying the version number of the model. Use the following command to update your *`design_dir`*:

```
%unix> dw_vip_setup -path design_dir -add swd_<master/slave>_agent_svt -add
swd_<master/slave>_svt -v Q-2020.06
```

3.4 Include and Import VIP Files into Your Test Environment

After setting up the models, you must include and import various files into your top testbench files to use the VIP. The code snippet of the includes and imports for the SWD VIP is as follows:

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

/* include AXI, AHB, and APB VIP interface */
`include "svt_ahb_if.svi"
`include "svt_axi_if.svi"
`include "svt_apb_if.svi"

/** Include SWD interface*/
`include "svt_swd_if.svi"

/** Include the AMBA SVT UVM package */
`include "svt_amba.uvm.pkg"

/** Include SWD SV UVM package */
`include "svt_swd.uvm.pkg"

/** Import UVM Package */
import uvm_pkg::*;

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

/** Import the SWD SVT UVM Package*/
import svt_swd_uvm_pkg::*;
```

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

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

Supported vendors include 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*.

3.5 VIP Compile-Time and Runtime Options

Every Synopsys provided example has ASCII files containing compile-time and runtime options. The examples for models are located at the following location:

`$DESIGNWARE_HOME/vip/svt/swd_svt/latest/examples/sverilog/<test_name>`
For example,
`$DESIGNWARE_HOME/vip/svt/swd_svt/latest/examples/sverilog/tb_swd_svt_uvm_basic_sys`

The following files contain the options:

- For compile-time options

  `sim_build_options (also, vcs_build_options)`

- For runtime options

  `sim_run_options (also, vcs_run_options)`

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

```
sim_build_options
sim_build_options
Required: +define+UVM_PACKER_MAX_BYTES=1500000
Required: +define+UVM_DISABLE_AUTO_ITEM_RECORDING
Required: +define+SVT_SWD
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 `VM_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_build_options(VCS-specific) :
Required: -timescale=100ps/100ps

sim_run_options
Required: +UVM_TESTNAME=$scenario
```

where, `scenario` is the UVM testname you pass to the simulator.

3.6 Verifying Installation

This section consists of the following subsections:

❖ Installing and Running Examples

3.6.1 Installing and Running Examples

This section consists of the following subsections:

- Installing the Basic Example
- Running the Basic Example

3.6.1.1 Installing the Basic Example

This step occurs after you invoke the `*.run` file to install the entire SWD VIP test-suite.

To install the example, you need to use the `dw_vip_setup` script. For more information, see "dw_vip_setup Administrative Tool" on page 32. Use the following command to invoke `dw_vip_setup`:

```
$DESIGNWARE_HOME/bin/dw_vip_setup -path <design_dir> -e
swd_svt/tb_swd_svt_uvm_basic_sys -svtb
```

where, `tb_swd_svt_uvm_basic_sys` is the name of the example and `-path <design_dir>` is the install location of the example.

3.6.1.2 Running the Basic Example

You can use the `dw_vip_setup` generated script to run the example.

Note You must install UVM and also set the `UVM_HOME` variable.
For example, `setenv UVM_HOME $VCS_HOME/etc/uvm`

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

```
cd <design_dir>/examples/sverilog/swd_svt/tb_swd_svt_uvm_basic_sys/

// To run the example using the generated run script with a sample command line
// of a specific interface in VCS

./run_swd_svt_uvm_basic_sys basic_test vcsvlog-svlog

// To run the example using the generated run script with a sample command line
of a specific interface in VCS with waveform dumping enabled

./run_swd_svt_uvm_basic_sys -w basic_test vcsvlog-svlog

// To see all options with the run_swd_svt_uvm_basic_sys example type

./run_swd_svt_uvm_basic_sys -h
```

3.6.1.3 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/swd_svt/tb_swd_svt_uvm_basic_sys/

// To run the example using the generated run script with +incdir+

./run_swd_svt_uvm_basic_sys -verbose -incdir directed_test vcsvlog
```

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

```
vcs -l ./logs/compile.log -Mdir=./output/csrc
+define+DESIGNWARE_INCDIR=<DESIGNWARE_HOME> \
+define+SVT_LOADER_UTIL_ENABLE_DWHOME_INCDIRS
+incdir+<DESIGNWARE_HOME>/vip/svt/swd_svt/Q-2020.06/sverilog/include \
-CFLAGS -DVCS +incdir+<UVM_HOME>/latest/src <UVM_HOME>/latest/src/dpi/uvm_dpi.cc \
-full64 -sverilog +define+SVT_SWD +define+UVM_PACKER_MAX_BYTES=1500000
+define+SVT_SWD_DEBUG_BUS_ENABLE \
-unit_timescale=100ps/100ps +define+SVT_UVM_TECHNOLOGY +define+SYNOPSYS_SV
+incdir+<testbench_dir>/examples/sverilog/swd_svt/tb_swd_svt_uvm_basic_sys/. \
+incdir+<testbench_dir>/examples/sverilog/swd_svt/tb_swd_svt_uvm_basic_sys/../../env \
+incdir+<testbench_dir>/examples/sverilog/swd_svt/tb_swd_svt_uvm_basic_sys/../env \
+incdir+<testbench_dir>/examples/sverilog/swd_svt/tb_swd_svt_uvm_basic_sys/env \
+incdir+<testbench_dir>/examples/sverilog/swd_svt/tb_swd_svt_uvm_basic_sys/dut \
+incdir+<testbench_dir>/examples/sverilog/swd_svt/tb_swd_svt_uvm_basic_sys/hdl_interconn
ect \
+incdir+<testbench_dir>/examples/sverilog/swd_svt/tb_swd_svt_uvm_basic_sys/lib \
+incdir+<testbench_dir>/examples/sverilog/swd_svt/tb_swd_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.

3.6.1.3.1 Supported Methodologies with Simulators

Table 3-1 lists the methodologies supported with simulators.

Table 3-1 Supported Methodologies with Simulators

Methodology	VCS	MTI	IUS
UVM	Supported	Supported	Not Supported
OVM	Not Supported	Not Supported	Not supported
VMM	Not Supported	Not Supported	Not supported
HDL	Not Supported	Not Supported	Not Supported

3.6.1.4 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_swd_svt_uvm_basic_sys [-32] [-incdir] [-verbose] [-debug_opts] [-waves] [-clean] [-
nobuild] [-buildonly] [-norun] [-pa] <scenario> <simulator>
  where   <scenario> is one of:   all base_test  directed_test
         <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
gmake USE_SIMULATOR=<simulator> [VERBOSE=1] [DEBUG_OPTS=1] [SEED=<value>]
[FORCE_32BIT=1] [WAVES=fsdb|verdi|dve|dump] [NOBUILD=1] [BUILDONLY=1] [PA=1]
[<scenario> ...]
       Valid simulators are: vcsmxvlog mtivlog vcsvlog vcszsimvlog vcsscvlog ncvlog
vcszebuvlog vcsmxpcvlog vcsvhdl vcsmxpipvlog ncmvlog vcspcvlog
       Valid scenarios are:  all base_test  directed_test
```

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

4 Overview of the SWD VIP

This chapter describes the various verification features available in the SWD Verification IP.

4.1 Introduction to UVM

UVM is an object-oriented approach. It provides a blueprint for building testbenches using the constrained random verification. In addition, the resulting structure supports directed testing. This chapter describes the data objects that support higher structures which comprise SWD VIP.

This chapter assumes that you are familiar with SystemVerilog and UVM. For more information, 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 http://www.accellera.org.

This chapter consists of the following sections:

- SWD VIP in a UVM Environment
- SWD VIP Programming Interface
- Interfaces and Modports
- More on Constraints

4.2 SWD VIP in a UVM Environment

The SWD agent encapsulates the following components:

- Sequencer – Data SWD sequencer
- Driver – Instance of a transceiver model

You can configure the above components in the agent using the agent configuration, which is available in the system configuration. You should provide the agent configuration to SWD Agent in the build phase of a test case.

You can provide SWD frames through sequences to Sequencer. Within the agent, Driver gets sequences from the Sequencer. Driver then drives the SWD transactions on the SWD Interface. After the SWD transaction on the bus is complete, the completed sequence item is provided to the analysis port for use by analysis components, such as Scoreboard.

In the layered approach that is typical for UVM, the VIP fits into the lower levels, which allow you to focus on the higher level of abstraction.

4.3 SWD VIP Programming Interface

This section gives an overview of the following programming interface in the SWD VIP:

- Configuration Objects
- Sequence Items
- Analysis Port
- Callbacks
- Tiered Messaging

4.3.1 Configuration Objects

Configuration objects convey the agent-level and protocol-level testbench configuration. These data objects contain built-in constraints, which come into effect when these objects are randomized. The configuration is of the following two types:

- **Static Configuration**: Static configuration parameters specify a configuration value which cannot be changed when the system is running. For example, `enable_<master/slavew>_cov` to enable the SWD coverage.
- **Dynamic Configuration**: Dynamic configuration parameters specify a configuration value which can be changed at any time, regardless of whether the system is running. An example of a dynamic configuration parameter is a timeout value. The configuration data objects contain built-in constraints, which come into effect when the configuration objects are randomized.

The SWD VIP describes the following configuration:

- **Agent Configuration**: The configuration has the `svt_swd_agent_configuration` base class for configuring agents, including Monitor, Transceiver, and Sequencer.

  For more information on this class, see the class-reference HTML documentation available in the following directory:

  $DESIGNWARE_HOME/vip//svt/swd_svt/latest/doc/swd_svt_uvm_class_reference/html/class_svt_swd_agent_configuration.html

  It is mandated by the SVT architecture that an instance of `svt_swd_agent_configuration` or extended configuration be passed to the agent, monitor, or driver assembly via `set` in `build_phase`. See the following code for the same:

```
uvm_config_db#(svt_swd_agent_configuration)::set(this, "*","cfg",cfg);
```

4.3.2 Sequence Items

The transaction objects, which extend from the `uvm_sequence_item` base class, define a unit of SWD protocol information that is passed across the bus. 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 objects represent the desired activity to be simulated on the bus, or the actual bus activity that is monitored. The transaction objects store data content and protocol execution information for SWD transactions in terms of the timing details of the transactions.

These data objects extend from the `uvm_sequence_item` base class and implement all the methods specified by UVM for that class. The transaction objects are used to perform the following tasks:

- Generate random stimulus
- Report observed transactions
- 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.

The SWD VIP also provides the following two constraints:

- **The valid_ranges constraint:** It limits the generated values to those acceptable to Driver. These constraints ensure basic VIP operations and should never be disabled. These correspond in most cases to limits set by the protocol.
- **The reasonable_* constraint:** It can be disabled individually or as a block. It limits the simulation by setting simulation boundaries. Disabling these constraints might slow the simulation and introduce system memory issues.

The VIP supports extending transaction data classes for customizing randomization constraints. This allows you to disable some `reasonable_*` constraints and replace them with constraints appropriate to your system. The individual `reasonable_*` constraint maps to independent fields, each of which can be disabled. The class provides the `reasonable_constraint_mode()` method to enable or disable blocks of the `reasonable_*` constraints. For more information, see More on Constraints.

The SWD VIP defines the following transaction classes:

- **Transaction Class (**`svt_swd_transaction`**):** It implements a base class for SWD transactions. SWD transaction generates a stimulus from the Tx side of the VIP. On the Rx side, the transaction gives the details on the frames transmitted and received across the VIP. For the detailed description about class members and methods, see the class-reference HTML documentation.

4.3.3 Analysis Port

At the end of a transaction, Monitor writes the transaction to an analysis port. You can use this analysis port to connect to Scoreboard, or any other purpose, where a transaction object for the completed transaction is required.

SWD VIP provides the following analysis ports:

- **Transmit Activity Analysis Port (**tx_xact_observed_port**):** It provides a mechanism for retrieving transaction results occurring from the SWD bus interface, which is transmitted by the VIP. These transactions are used by subscriber components such as Scoreboard for further checking on the Tx direction.
- **Rx Activity Analysis Port (**rx_xact_observed_port**):** It provides a mechanism for retrieving transaction results occurring from the SWD bus interface, which is received by the VIP. These transactions are used by the subscriber components such as scoreboard for further checking on the Rx direction.

4.3.4 Callbacks

Callbacks are an access mechanism that enables the insertion of user-defined code and allows access to objects, such as transaction objects and functional coverage status objects, and also can be used to deviate the normal transaction flow (for example, insert delay and so on.).

Both Driver and Monitor are associated with a callback class that contains a set of callback methods. These methods are called as part of the normal flow of the procedural code. Following are the differences between callback methods and other methods that set them apart.

- Callback methods are virtual methods with no code and they do not provide any functionality unless they are extended. The exception to this rule is that some of the callback methods for Functional coverage already contain the default implementation of a coverage model.
- You can access the callback class and extend the class including the testbench-specific extensions of the default callback methods, specific variables and/or methods that are used to control the behavior of the testbench using the callbacks support.
- Callbacks are called within the sequential flow at places where external access would be useful. In addition, the arguments to the methods include references to the relevant data objects. For example, just before Monitor puts a transaction object into an analysis port, which is an appropriate location to sample the Functional coverage since the object reflects the activity that took place on the SWD interface pins. A callback at this point with an argument referencing the transaction object allows this exact scenario.

The SWD VIP uses callbacks in the following three main applications:

- Access for Functional coverage
- Change the transaction fields
- Insertion of user-defined code

4.3.5 Tiered Messaging

The VIP includes the Tiered Messaging architecture to display debug messages, which helps you to find the status of the current simulation.

This section consists of the following two subsections:

- Verbosity Level
- Message Structure

4.3.5.1 Verbosity Level

Tiered messages are distributed over the following three levels of verbosity:

- `UVM_LOW`: This level includes the following important status messages about the VIP:
  - Transaction start and finish event message
  - Baud rate selection message
  - Slave and Master Id selected for a particular SWD transaction
- `UVM_MEDIUM`: This level includes the messages used for full transmission and reception of transactions—like sending a transaction byte-by-byte is printed in it.
- `UVM_HIGH`: This level includes the transitions of a State machine in the VIP BFM Rx side.

4.3.5.2 Message Structure

A tiered message has the following structure:

```
<message location>: [<message id>]: <message>
```

where,

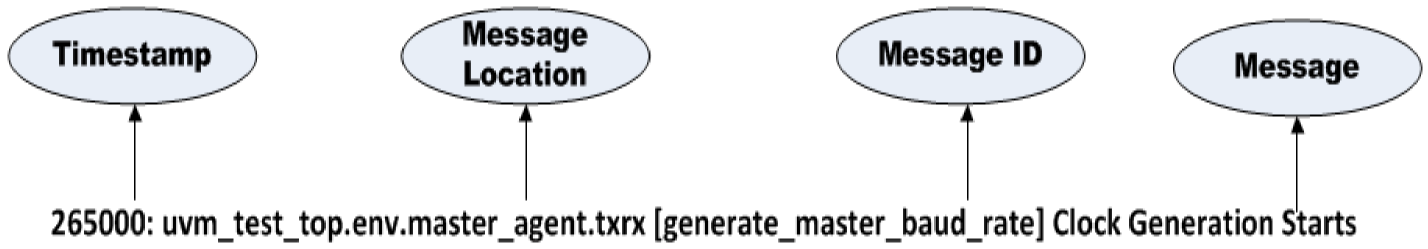
`<message location>`: This field indicates the agent instance from where messages are triggered. This field also states whether a message is from `*.driver` or `*.monitor`.

`<message id>`: Every message comes with a message id, which specifies the clause tag of the message. For example, generate_master_baud_rate means this message comes from the generate_master_baud_rate module.

`<message>`: The actual message to be printed.

Figures 4-1 shows the structure of a tiered message.

Figure 4-1 Message Structure


Timestamp
Message Location
Message ID
Message
265000: uvm_test_top.env.master_agent.txrx [generate_master_baud_rate] Clock Generation Starts

4.4 Interfaces and Modports

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

The top-level interface is `svt_swd_if`.

Each Agent gets an individual instance of interface. The connection between master and slave interface is done in top file.

Instantiate and create an agent in the UVM environment file:

```
svt_swd_agent agent_bfm0;
```

An agent gets an interface handle using any one of following methods:

- ✦ The agent is tied to interface in the top via set that happens in the *top.sv* file.

```
uvm_config_db#(virtual
svt_swd_if)::set(uvm_root::get(),"uvm_test_top.env*","vif", if_port);
```

- ✦ The interface can be passed on to an agent instance through svt_swd_agent_configuration

object.

```
bfm_cfg.swd_if = vif;
swd_svt/test/sverilog/tb_swd_svt_uvm_basic_sys/env/swd_basic_env.sv:
uvm_config_db#(svt_swd_agent_configuration)::set(this, "agent_bfm0", "cfg",
bfm_cfg);
```

4.5 More on Constraints

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

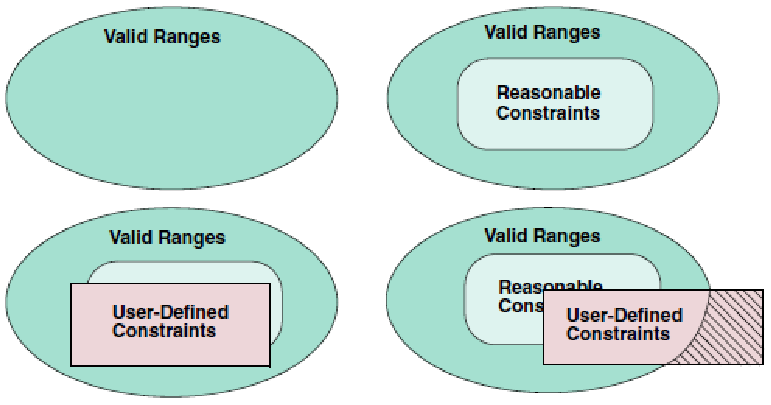
The classes that provide random attributes allow you to define the contents of the resulting object. When you call the `randomize()` method, all random attributes are randomized using all constraints that are enabled.

Constraint randomization is sometimes viewed as a process whereby the simulation engine takes away your control on class members. In fact, the opposite is true. Randomization is an additional way for you to assign class members and there are several ways to control the process. The following techniques apply when working with randomization:

- ✦ Randomization only occurs when the `randomize()` method of an object is called, and it is completely up to the test code when, or even if, this occurs.
- ✦ Constraints form a rule set to follow when randomization is performed. By controlling constraints, the testbench has influence over the outcome. A direct control can be exerted by constraining a member to a single value. The constraints can also be enabled or disabled.
- ✦ Each random member has a random mode that can be turned ON or OFF, giving you the control of what is randomized.
- ✦ You can assign a value to a member at any time. Randomization does not affect the other methods of assigning class members.

Figure 4-2 shows the scope of the constraints that are part of all VIP.

Figure 4-2Constraints: Valid Ranges, Reasonable, and User-Defined



The following list describes the constraints in detail:

- **Valid Range Constraints**
  - Provided with VIP
  - Keep values within a range that Driver can handle
  - Not tied to protocol limits
  - Turned on by default, and should not be turned off or modified
- **Reasonable Constraints**
  - Provided with VIP
  - Keep values within protocol limits (typically) to generate worthwhile traffic
  - In some cases, keep simulations to a reasonable length and size
  - Defined to be "reasonable" by Synopsys (you can override)
  - May result in conditions that are a subset of the protocol
  - Turned on by default and can be turned off or modified (you should review these constraints)
- **User-Defined Constraints**
  - Provide a way to define specific tests
  - Constraints that lie outside the valid ranges will result in the constraint failure.

All constraints that are enabled are included in the simulation.

5 VIP Tools

5.1 Using Native Protocol Analyzer for Debugging

5.1.1 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.

5.1.2 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 as shown below:

Configuration Variable Setting:

```
< svt_swd_agent_configuration >.enable_master_xml_gen = 1; // Default is 0
< svt_swd_agent_configuration >.enable_slave_xml_gen = 1; // Default is 0
< svt_swd_agent_configuration>.pa_format_type =svt_xml_writer::FSDB
// 0 is XML, 1 FSDB and 2 both XML and FSDB, default it will be zero
```

Runtime Option

```
+svt_enable_pa=<option>
```

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

5.1.3 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 issue the following command to invoke the GUI:

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

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

Interactive Mode

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

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

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

5.1.4 Documentation

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

$VERDI_HOME/doc/Verdi_Transaction_and_Protocol_Debug.pdf

A
Reporting Problems

A.1 Introduction

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

A.2 Debug Automation

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

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

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

A.3 Enabling and Specifying Debug Automation Features

Debug Automation is enabled through the use of a run-time plusarg named +*svt_debug_opts. T*his plusarg accepts an optional string-based specification to control various aspects Debug Automation. If this command control specification is not supplied, then the feature will default to being enabled on all VIP instances with the default options listed as follows:

Note the following about the plusarg:

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

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

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

The following table explains each control string:

Table A-1 Control Strings for Debug Automation plusarg

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

Examples:

Enable on all VIP instances with default options:

```
+svt_debug_opts
```

Enable on all instances:

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

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

Enable on all instances:

- starting at time 1000 until time 1500:

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

Enable debug feature on all instances using default options:

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

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

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

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

A.4 Debug Automation Outputs

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

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

The following are the output files generated:

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

A.5 FSDB File Generation

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

A.5.1 VCS

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

```
-debug_access
```

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

A.5.2 Questa

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

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

A.5.3 Incisive

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

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

A.6 Initial Customer Information

Follow these steps when you call the Synopsys Support Center:

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

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

A.7 Sending Debug Information to Synopsys

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

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

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

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

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

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

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

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

A.8 Limitations

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

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