VC Verification IP UART OVM User Guide

Version O-2018.09, September 2018

Contents

Preface

About This Guide

This guide contains installation, setup, and usage material of the VC VIP for UART on the Open Verification Methodology (OVM). This guide is for design or verification engineers who want to verify UART operations using an OVM testbench written in SystemVerilog. Readers are assumed to be familiar with UART, Object-Oriented Programming (OOP), SystemVerilog, and OVM techniques.

Guide Organization

The chapters of this guide are organized as follows:

- ❖ Chapter 1, "Introduction", introduces UART VIP and its features.
- ❖ Chapter 2, "Installation and Setup", describes system requirements and provides instructions on how to install, configure, and begin using UART VIP.
- ❖ Chapter 3 , "General Concepts", introduces UART VIP within an OVM environment and describes the data objects and components that comprise the VIP.
- ❖ Chapter 5, "Verification Topologies", describes the topologies to verify DTE, DCE and interconnect DUT.
- ❖ Chapter 4, "Using UART Verification IP", shows how to install and run a getting started example.
- ❖ Chapter 6, "Usage Notes", covers the properties of DTE transactions and DCE transactions.
- ❖ Chapter 7, "Troubleshooting", provides some useful information that can help you to troubleshoot common issues that you may encounter while using UART VIP.
- ❖ Appendix A, "Reporting Problems", outlines the process for working through and reporting UART VIP issues.

Web Resources

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

Customer Support

For Customer Support, perform any of the following tasks:

- Enter a call through SolvNet:
  - Go to http://solvnet.synopsys.com/EnterACall
    and click **Open A Support Case** to enter a call.
  - Provide the requested information, including:
    - **Product**: Verification IP
    - **Sub Product 1**: UART SVT
    - **Product Version**: O-2018.09
    - Fill in the remaining fields according to your environment and your issue
    - If applicable, provide the information noted in Appendix A, "Reporting Problems" on page 83
- Send an e-mail message to support_center@synopsys.com
    - Include Product name, Sub-Product name, and Product Version as discussed previously.
    - If applicable, provide the information noted in Appendix A, "Reporting Problems" on page 83.
- 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

UART Verification IP supports the verification of SoC designs that include interfaces implementing UART specifications. This document describes the use of the VIP in testbenches that comply with the SystemVerilog OVM. This approach leverages advanced verification technologies and tools that provide the following features:

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

Refer the UART VIP class reference HTML documentation, which is installed at the following location:

```
$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/uart_svt_ovm_class_reference/html/index.ht
ml
```

This chapter consists of the following sections:

1.1 Product Overview

UART VIP is a suite of OVM-based verification components that are compatible for use with SystemVerilog-compliant testbenches. The UART VIP suite simulates UART transactions through active agents, as defined by UART specifications.

The VIP provides a VIP agent which can be configured as a Data Terminal Equipment (DTE) agent or as a Data Communication Equipment (DCE) agent. The VIP agent supports all functionality normally associated with active and passive OVM components, including the creation of transactions, checking and reporting the protocol correctness, transaction logging, and functional coverage. After instantiating the VIP agent in the verification environment, you can specify the active or passive mode based on your requirements.

1.2 Language and Simulator Support

The UART VIP suite supports the following languages and simulators:

- Languages
  - SystemVerilog
- Methodologies and simulators: For details about methodologies and simulators, refer to the UART VIP Release Notes from the following location:

  ```
  $DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/uart_svt_release_notes.pdf
  ```

1.3 Features Supported

This section consists of the following sub-sections:

- "Protocol Features" on page 10
- "Verification Features" on page 11
- "Methodology Features" on page 12

1.3.1 Protocol Features

UART VIP currently supports the following protocol functions:

- Capable to verify a DCE and DTE
- Configurable Baud Rate Divisor, that is, you can select the baud rate divisor according to the UART DUT requirement
- Configurable Fractional Baud Rate Divisor, that is, you can select the fractional baud rate divisor according to the UART DUT requirement
- Configurable data-width with 9-bits as well as 5-bits, 6-bits, 7-bits, and 8-bits
- Supports the full-duplex operation, that is, simultaneous transmission and reception
- Supports Line break generation and detection
- Provides Hardware or out-band flow control
- Provides Software or in-band flow control
- Supports programmable 1, 1.5, and 2 stop bits
- Programmable parity (Even, Odd, Stick and No-parity)

- Provides receiver FIFO (Configurable depth)
- Implements programmable baud generator, which divides any input clock by 1 to (2^{16}-1) and generates the 16x clock
- Supports programmable sample rate
- Supports RS485 mode checking

1.3.2 Verification Features

UART VIP currently supports the following verification functions:

- Configurability for the VIP agent
- Built-in protocol checks
- Built-in functional coverage (transaction, toggle, and pattern coverage)
- Verification planner
- Exceptions (error injections)
- Sequence collection
- Source code visibility
- Configuration creator support
- UART VIP provides the following transaction features:
    - Configurable delay between two consecutive packets
    - Configurable delay in the assertion of Request to Send (RTS) and Clear to Send (CTS) pins
    - Configurable delay to flush the receiver buffer (FIFO) when buffer gets full
    - Control of the packet generation by the VIP driver
    - Controllability of transmitting the XON and XOFF data pattern on the bus on high priority in case of the software flow control
    - Controllability of driving the Data Terminal Ready (DTR) pin of DTE and the Data Set Ready (DSR) pin of DCE in case of the hardware flow control
- Ease-of-use features include the following features:
    - The Protocol Analyzer support for debug
    - Basic and intermediate-level examples to illustrate the use of the VIP in SystemVerilog
    - The HTML documentation for VIP classes
    - Quickstart guides for basic and intermediate-level examples

1.3.3 Methodology Features

UART VIP currently supports the following methodology functions:

- The VIP is organized as the UART DTE agent and the UART DCE agent.
- Analysis ports for connecting DTE or DCE agents to subscribers, such as Scoreboard and coverage collectors.
- Callbacks support for the DTE and DCE agent

2 Installation and Setup

This chapter leads you through the installing and setting up UART VIP. After completing this checklist, the provided example testbench will be operational and UART VIP will be ready to use.

This chapter consists of the following major steps:

- "Verifying Hardware Requirements" on page 13
- "Verifying Software Requirements" on page 14
- "Preparing for Installation" on page 14
- "Downloading and Installing" on page 15
- "Setting Up a Testbench Design Directory" on page 17
- "Licensing Information" on page 18
- "Environment Variable and Path Settings" on page 19
- "Determining Your Model Version" on page 20
- "Integrating a UART Verification IP into Your Testbench" on page 20

2.1 Verifying Hardware Requirements

UART VIP requires the following configuration for Solaris or Linux workstation:

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

2.2 Verifying Software Requirements

UART VIP is qualified for use with the certain versions of platforms and simulators. This section lists the software required by UART VIP and consists of the following sub-sections:

- "Platform/OS and Simulator Software" on page 14
- "SCL Software" on page 14
- "Other Third-Party Software" on page 14

2.2.1 Platform/OS and Simulator Software

Platform/OS and VCS: You need the versions of your platform or OS and simulator that have been qualified for use.

For more details, refer UART Release Notes.

2.2.2 SCL Software

The Synopsys Common Licensing (SCL) software provides the licensing function for UART VIP. For details on acquiring the SCL software, see the installation instructions in "Licensing Information" on page 18.

2.2.3 Other Third-Party Software

Following is the list of other third-party software:

- **Adobe Acrobat**: UART VIP documents are available in Acrobat PDF files. Adobe Acrobat Reader is available for free from http://www.adobe.com.
- **HTML Browser:** UART VIP includes a class-reference documentation in HTML that supports the following browser/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 Preparing for Installation

Perform the following steps to prepare for installation:

a. Set `DESIGNWARE_HOME` to the absolute path where Synopsys UART VIP is to be installed:

```
setenv DESIGNWARE_HOME absolute_path_to_designware_home
```

b. Ensure that your environment and PATH variables are set correctly, including 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. Also, include the absolute path to the third-party executable in your PATH variable.

```
% setenv LM_LICENSE_FILE <my_license_file|port@host>
```

- `SNPSLMD_LICENSE_FILE` – The absolute path to a file that contains the license keys for the SCL software or the `port@host` reference to this file.

```
% setenv SNPSLMD_LICENSE_FILE $LM_LICENSE_FILE
<my_Synopsys_license_file|port@host>
```

- `DW_LICENSE_FILE` – The absolute path to a file that contains the license keys for VIP product software or the port@host reference to this file.

```
% setenv DW_LICENSE_FILE <my_VIP_license_file|port@host>
```

2.4 Downloading and Installing

You can download 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 additional information, refer to the following web page:

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

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.

This section consists of the following sub-sections:

- "Downloading From EST (Download Center)" on page 15
- "Downloading Using FTP With a Web Browser" on page 16

2.4.1 Downloading From EST (Download Center)

a. Point your web browser to http://solvnet.synopsys.com.

b. Enter your Synopsys SolvNet Username and Password.

c. Click the `Sign In` button.

d. Make the following selections on SolvNet to download the `.run` file of the VIP (See Figure 2-1).

   i. `Downloads` tab

   ii. VC VIP Library product releases

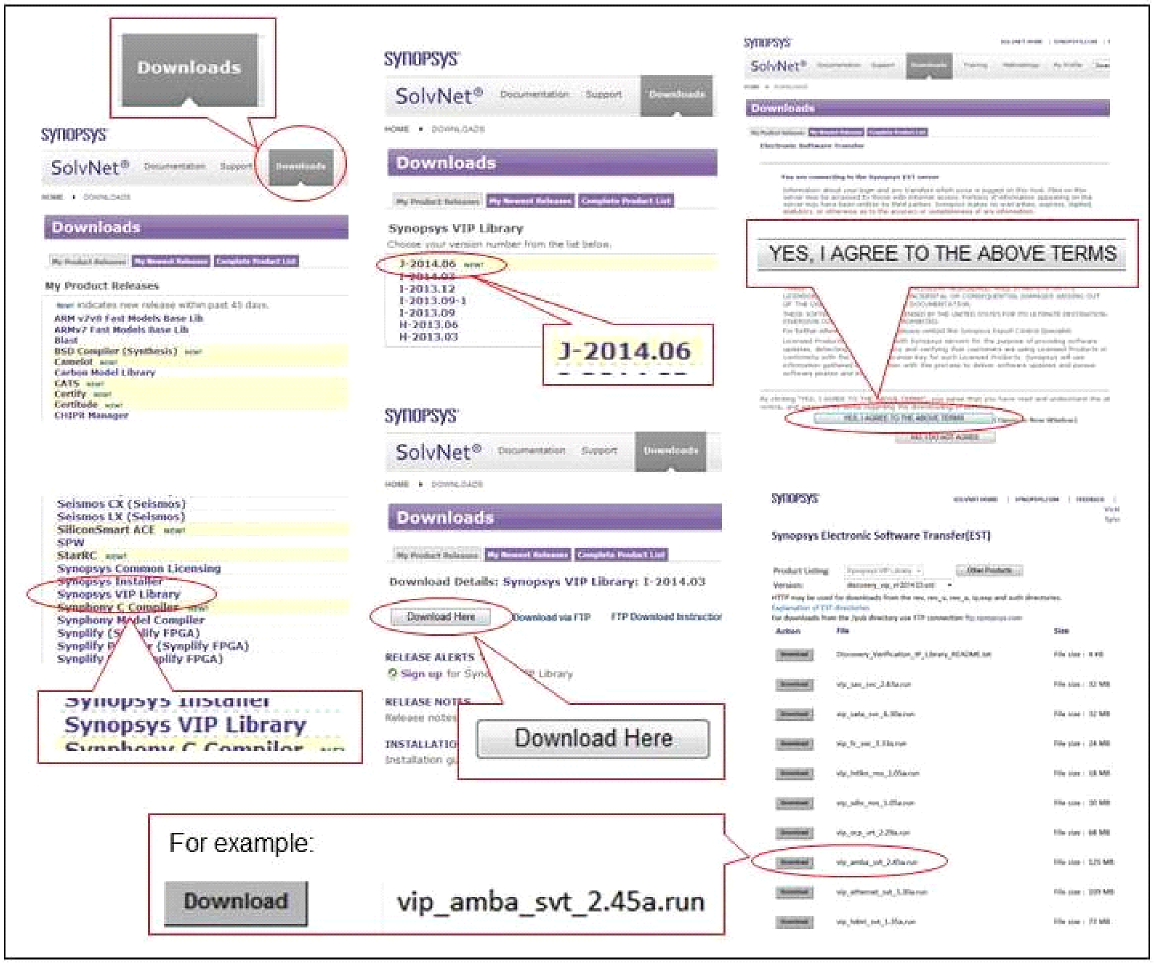
   iii. <release_version>

   iv. `Download Here` button

   v. `Yes, I Agree to the Above Terms` button

   vi. Download `.run` file for the VIP

Figure 2-1 SolvNet Selections for VIP Download



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

```
% setenv DESIGNWARE_HOME VIP_installation_path
```

f. 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.

2.4.2 Downloading Using FTP With a Web Browser

Follow Step a to Step e of Section 2.4.1 and then perform the following steps:

a. Click the **Download via FTP** link instead of the **Download Here** button.

b. Click the **Click Here To Download** button.

c. Select the file(s) that you want to download.

d. Follow browser prompts to select a destination location.

2.5 Setting Up a Testbench Design Directory

A design directory is where UART VIP is set up for use in a testbench. The `dw_vip_setup` utility is provided as a design directory is required for using the VIP.

The `dw_vip_setup` utility allows you to create the design directory (`design_dir`), which contains the VIP components, support files (include files), and examples (if any). Add a specific version of UART VIP from `DESIGNWARE_HOME` to the design directory.

For a complete description of `dw_vip_setup`, see "The dw_vip_setup Utility" on page 22.

The following models are provided with UART VIP:

- `uart_agent_svt`
- `uart_monitor_svt`
- `uart_txrx_svt`

Use the following command to create a design directory and add a model to be used in a testbench:

```
%$DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a(dd) <model1> -svtb


For example, %$DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a uart_agent_svt -svtb
Or
%$DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a(dd) <model1> <model2>
<model3> -svtb


For example, %$DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a uart_agent_svt
uart_monitor_svt uart_txrx_svt -svtb
Or
%$DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a(dd) -model_list
<input_file_containing_models_one_per_line> -svtb


For example, %$DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a -model_list
<filelist> -svtb


cat filelist:
uart_agent_svt
uart_monitor_svt
uart_txrx_svt
```

After running the above command, the model files are installed at the following location:

```
<design_dir>/include and <design_dir>/src
```

Note You need to specify the pointer to these installed directories on simulator analyze or compile-options.

2.6 Licensing Information

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

http://www.synopsys.com/keys

You can enable UART VIP by performing the license check in the order listed below. Once a required feature or a set of features are successfully checked-out, the VIP stops looking for other licenses. The type of license depends on product and portfolio type. You can use one of the following set of hierarchies:

- `VIP-UART-SVT`
- `VIP-PROTOCOL-SVT`
- `VIP-SOC-LIBRARY-SVT`
- `VIP-LIBRARY-SVT + DesignWare-Regression`

Only one license is consumed per simulation session, irrespective of how many VIP products are instantiated in the design.

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 Variable and Path Settings" on page 19.

This section consists of the following sub-sections:

- "Controlling License Usage" on page 18
- "License Polling" on page 19
- "Simulation License Suspension" on page 19

2.6.1 Controlling License Usage

You can control which license is used, using the `DW_LICENSE_OVERRIDE` environment variable, as follows:

- To use only `DESIGNWARE-REGRESSION` and `VIP-LIBRARY-SVT` licenses, set `DW_LICENSE_OVERRIDE` to `DESIGNWARE-REGRESSION VIP-LIBRARY-SVT`
- To use only the `VIP-UART-SVT` license, set `DW_LICENSE_OVERRIDE` to `VIP-UART-SVT`. If `DW_LICENSE_OVERRIDE` is set to a value and the corresponding feature is not available, a license error message is issued.

2.6.2 License Polling

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

- To enable the license polling, set the `DW_WAIT_LICENSE` environment variable to 1.
- To disable the license polling, unset the `DW_WAIT_LICENSE` environment variable. By default, the license polling is disabled.

2.6.3 Simulation License Suspension

All the Verification IP products support the license suspension. The simulator that support the license suspension allows the model to check-in its license token while the 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 the license suspension.

2.7 Environment Variable and Path Settings

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

1. Set `DESIGNWARE_HOME` to the following absolute path where the Synopsys UART VIP is recommended to be installed:
   ```
   setenv DESIGNWARE_HOME absolute_path_to_designware_home
   ```
2. Ensure that your environment and `PATH` variables are set correctly:

   - `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 file(s) 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 contain Synopsys VIP licenses. Also, include the absolute path to the third party executable in your PATH variable.

Simulator-Specific Settings

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

2.8 Determining Your Model Version

The following steps describes how to check your model version:

Note 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
  ```

2.9 Integrating a UART Verification IP into Your Testbench

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

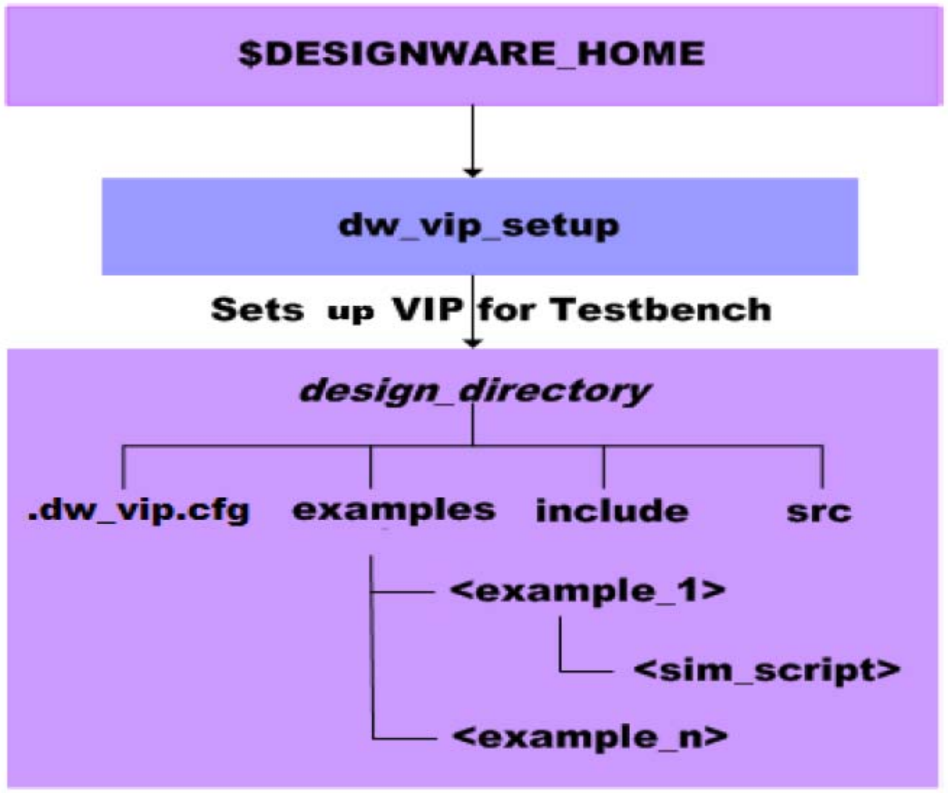
- ✦ "Creating a Testbench Design Directory" on page 20
- ✦ "The dw_vip_setup Utility" on page 22

2.9.1 Creating a Testbench Design Directory

A design directory contains a version of the VIP that is set up and ready to use in a testbench. The `dw_vip_setup` utility is used to create the design directories. For more information on `dw_vip_setup`, see "The dw_vip_setup Utility" on page 22.

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

Figure 2-2 Design Directory Created by dw_vip_setup



A design directory contains the following sub-directories:

examples	Each VIP includes example testbenches. The `dw_vip_setup` utility adds them in this directory, along with a script for simulation. If an example testbench is specified on the command line, this directory contains all the files required for model, suite, and system testbenches.
include	The language-specific include files that contain the critical information for VIP models. This directory is specified in simulator command lines.
src	The VIP-specific include files (not used by all VIP). This directory may be specified in simulator command lines.
.dw_vip.cfg	A database of all the VIP models used in the testbench. The `dw_vip_setup` utility 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.

2.9.2 The dw_vip_setup Utility

The `dw_vip_setup` utility provides the following features:

- Adds, removes, or updates VIP models in a design directory
- Adds example testbenches to a design directory, the 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
- Supports Protocol Analyzer (PA)
- Supports the FSDB wave format

This section consists of the following sub-sections:

- "Setting Environment Variables" on page 22
- "The dw_vip_setup Command" on page 22

2.9.2.1 Setting Environment Variables

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

2.9.2.2 The dw_vip_setup Command

From the command prompt, invoke `dw_vip_setup`. The `dw_vip_setup` command checks 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 the `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] (model [-v[ersion] version]) …	Adds the specified model or models to the specified design directory or current working directory. If you do not specify a version, the latest version is assumed. The model names are as follows: • uart_agent_svt • uart_monitor_svt • uart_txrx_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 program does not attempt to remove any include files used solely by the specified model or models. The model names are as follows: • uart_agent_svt • uart_monitor_svt • uart_txrx_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: • uart_agent_svt • uart_monitor_svt • uart_txrx_svt The -update switch causes dw_vip_setup to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-e[xample] {scenario | model/scenario} [-v[ersion] version]	The dw_vip_setup script configures a testbench example for a single model or a system testbench for a group of models. The program creates a simulator run program for all the supported simulators. If you specify a scenario (or system) example testbench, the models needed for the testbench are included automatically and do not need to be specified in the command. Note: Use the -info switch to list all the available system examples.
-ntb	Not supported.
-svtb	Use this switch to set up models and example testbenches for SystemVerilog OVM. The resulting design directory is streamlined and can only be used in SystemVerilog simulations.
-c[lean] {scenario | model/scenario}	Cleans the specified scenario/testbench in either the design directory (as specified by the -path switch) or the current working directory. This switch deletes all files in the specified directory, then restores all Synopsys-created files to their original contents.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-i/nfo design | home[:<product>[:<version>[:<methodology>]]]	Generate 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 modellist file for 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.
model	UART VIP models are as follows: • uart_agent_svt • uart_monitor_svt • uart_txrx_svt The model argument defines the model or models that dw_vip_setup acts upon. This argument is not needed with the -info or -help switches. All switches that require the model argument may also use a model list. You may specify a version for each listed model, using the -version option. If omitted, dw_vip_setup uses the latest version. The -update switch ignores model version information.
-m[odel_list] filename	The -model_list argument causes dw_vip_setup to use a user-specified file to define the list of models on which the program acts. The model_list, like the model argument, can contain model version information. Each line in the file contains the following syntax: model_name [-v version] –or– # Comments
-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`.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-waves	Enables 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.

Note The `dw_vip_setup` utility treats all lines beginning with "#" as comments.

3 General Concepts

This chapter introduces UART VIP within an OVM environment and describes the data objects and components that comprise the VIP. This chapter assumes that you are familiar with SystemVerilog.

This chapter consists of the following sections:

3.1 Introduction to OVM

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

For more information, see the following:

- For the IEEE SystemVerilog standard, refer the following:
  - IEEE Standard for SystemVerilog - Unified Hardware design, specification, and verification language
- For an essential reference guide describing OVM as it is represented in SystemVerilog along with a class reference, refer the following:
  - *Open Verification Methodology (OVM) 2.1.2 Class Reference*

3.2 VIP Agent

The VIP agent can be configured as either of the following mode:

- "DTE Agent"
- "DCE Agent"

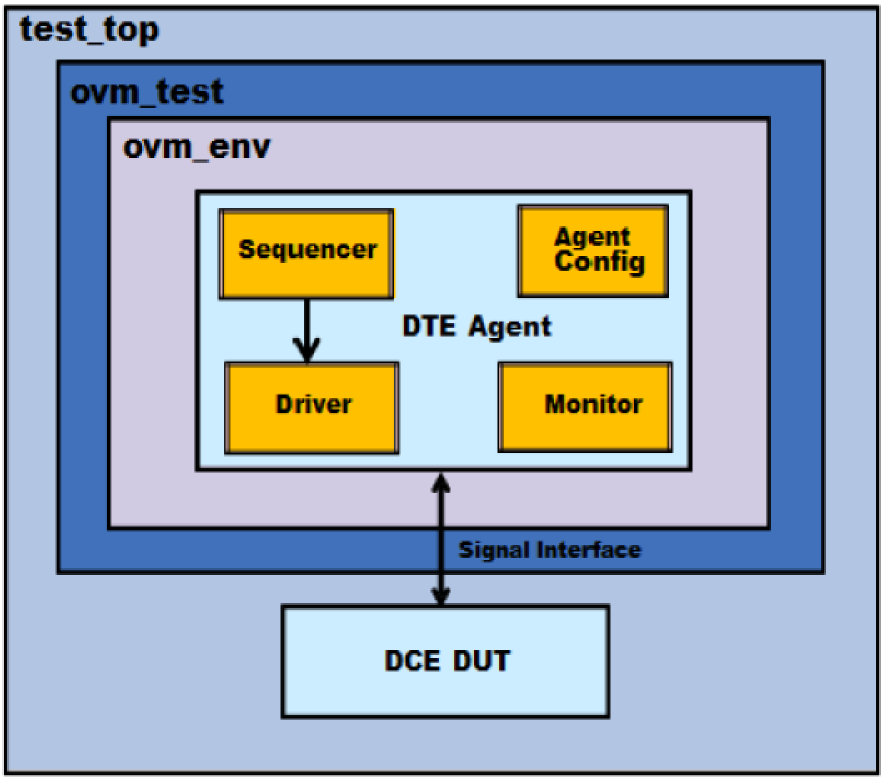
3.2.1 DTE Agent

When the VIP agent is configured in the DTE mode, it encapsulates a DTE sequencer, DTE driver, and DTE monitor. DTE driver gets transactions from DTE sequencer, DTE driver then drives transactions on a port. After a transaction is complete, the analysis port of DTE monitor provides the completed sequence item to testbench. You can run user-defined sequences on DTE sequencer.

You can configure the VIP agent in the DTE mode to operate either in an active mode or a passive mode. You can configure the VIP agent as the DTE mode using the *UART VIP agent configuration* property. You should provide the information of the UART VIP agent configuration in the build phase of a test.

Figures 3-1 shows the usage with standalone DTE agent.

Figure 3-1 Usage With Standalone DTE Agent



3.2.2 DCE Agent

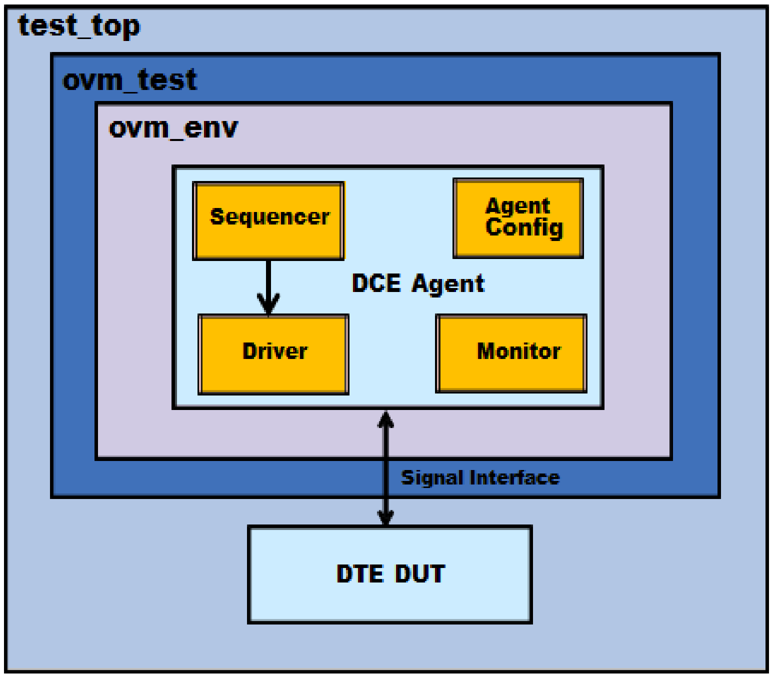
When the VIP agent is configured in the DCE mode, it encapsulates a DCE sequencer, DCE driver, and DCE monitor. The DCE driver gets transactions from the DCE sequencer, the DCE driver then drives transactions on a port. After a transaction is complete, the analysis port of the DCE monitor provides the completed sequence item to the testbench. You can run user-defined sequences on the DCE sequencer.

You can configure the VIP agent in the DCE mode to operate either in an active mode or a passive mode.

You can configure the VIP agent as the DCE mode using the *UART VIP agent configuration* property. You should provide the information of the UART VIP agent configuration in the build phase of a test.

Figures 3-2 shows the usage with standalone DCE agent.

Figure 3-2 Usage With Standalone DCE Agent



3.3 VIP User Interface

This section gives an overview of the user interface in UART VIP using the following sub-sections:

- "Configuration Objects" on page 30
- "Transaction Objects" on page 32
- "Analysis Ports" on page 34
- "Interfaces and Modports" on page 34

3.3.1 Configuration Objects

Configuration data objects convey the agent-level testbench configuration. The configuration of agents is done in the `build()` phase of the environment or the testcase. Configuration data objects contain built-in constraints, which come into effect when the configuration objects are randomized.

UART VIP defines the following configuration classes:

- "UART VIP Configuration" on page 30
- "UART VIP Agent Configuration" on page 31

3.3.1.1 UART VIP Configuration

The `svt_uart_configuration` class contains the UART protocol configuration information, which is applicable across the entire environment. The UART configuration mainly specifies the following attributes:

3.3.1.1.1 Common Attributes

- Data width
- Baud divisor
- Handshake type
- Enable or disable driving of the Baud Out pin
- Receiver buffer size
- Stop bit
- Parity type
- Enable or disable Fractional Baud divisor
- Fractional Baud divisor
- Fractional Divisor Period
- Fractional MULT Median
- Enable or disable calling of `put_response()` from driver to put response object back to sequencer

3.3.1.1.2 Attributes for Hardware Handshaking

- Enable or disable the RTS-CTS handshake
- Enable or disable the DTR-DSR handshake
- Enable or disable the Tx/Rx handshake

3.3.1.1.3 Attributes for Software Handshaking

- Data pattern for XON and XOFF
- Maximum delay between the XON packet and the XOFF packet
- Enable or disable wait for the XON packet before DTE sends the first transaction after power is up

3.3.1.2 UART VIP Agent Configuration

The agent configuration class, `svt_uart_agent_configuration`, is derived from the `svt_uart_configuration` class. This contains configuration information, which is applicable to individual UART DTE or DCE agent in the environment.

Common configuration attributes for DTE and DCE agents are as follows:

- Active or passive mode of the DTE or DCE agent
- Enable or disable protocol checks
- Enable or disable Functional coverage
- Enable or disable Toggle coverage
- Enable or disable Protocol Analyzer (PA)
- Enable or disable Monitor log
- Enable or disable protocol checks-related functional coverage

Refer to the UART VIP class reference HTML documentation for the details on individual members of configuration classes:

`$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/uart_svt_ovm_class_reference/html/class_svt_uart_agent_configuration.html`

3.3.2 Transaction Objects

Transaction objects, which extend from the `ovm_sequence_item` base class, define a unit of UART protocol information. The attributes of transaction objects are public and are accessed directly for setting and getting values. Most transaction attributes can be randomized.

UART transaction data objects store data content and the protocol execution information for UART transactions in terms of timing details of transactions.

UART transaction data objects are used to do the following:

- Generate Random and Directed stimulus
- Report observed transactions
- Collect functional coverage statistics
- Support error injection

The class properties are public and accessed directly to set and read values except those properties, which are not meant to be set by users. For example: `direction`, `received_packet`, and `received_parity`. The transaction data objects support randomization and provide built-in constraints.

UART VIP provides the following two constraints:

- The `valid_ranges` constraint: It limits the generated values to those acceptable to the driver. These constraints ensure basic VIP operations and should never be disabled.
- The `reasonable_*` constraint: It can be disabled individually or as a block. It limits the simulation by doing the following:
  - Enforcing the protocol. These constraints are typically enabled unless errors are injected in a simulation.
  - Setting simulation boundaries. Disabling these constraints may slow a 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.

UART VIP defines the following transaction classes:

- `svt_uart_transaction`: This class represents the data transaction for UART VIP and used by the `svt_uart_agent` class. It contains the attributes of a transaction, such as packet count, break condition, payload, and so on. It also provides the timing information of a transaction, that is, delays to assert RTS or CTS signals and inter-cycle delay between two packets.

The monitor uses the `svt_uart_transaction` class directly to construct a response object. The class has the following properties:

- ✧ It contains the following attributes which are relevant for the software handshaking (In-band):
  - `send_xoff_data_pattern`
  - `send_xon_data_pattern`
- ✧ It contains the following attributes which are relevant for the hardware handshaking (Out-band):
  - `delay_cts`
  - `delay_rts`
  - `drive_dsr_signal`
  - `drive_dtr_signal`
  - `dsr_signal_value`
  - `dtr_signal_value`
- ✧ It contains some other public attributes, which are as follows:
  - `enable_packet_generation`
  - `packet_count`
  - `inter_cycle_delay`
  - `buffer_flush_delay`
  - `break_cond`
  - `payload[]`
- ✧ It contains the following attributes to construct a response object:
  - `received_packet[]`
  - `received_parity[]`
  - `direction`

The transaction object contains a handle to the configuration object, which provides the configuration of the port on which the transaction is applied. The configuration is used during randomizing the transaction. The configuration is available in the sequencer of UART VIP agent. The user sequence should initialize the configuration handle in the transaction using the configuration available in the sequencer of the VIP agent. If the configuration handle in the transaction is null at the time of randomization, the transaction issues a fatal message.

For details on the individual members of transaction classes, refer to the following UART VIP Class reference HTML documentation:

`$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/uart_svt_ovm_class_reference/html/class_svt_uart_transaction.html`

3.3.3 Analysis Ports

In active as well as passive mode of operation of the DTE or DCE agent, you can use the analysis port for connecting to the scoreboard, or for any other purpose, where a transaction object, that is, `svt_uart_transaction` for the completed transaction is required. The monitor in the agent provides an analysis port. At the end of a transaction, the monitor within the agent provides the completed object, `svt_uart_transaction` from its analysis port.

The analysis ports for transmit and receive ports are as follows:

- `svt_uart_monitor::tx_xact_observed_port`
- `svt_uart_monitor::rx_xact_observed_port`

3.3.4 Interfaces and Modports

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

UART VIP provides the SystemVerilog interface, which can be used to connect the VIP to the DUT. Synopsys defines the top-level interface, `svt_uart_if`.

This SystemVerilog 'interface' definition declares all of the signals (scoped within an instance of this interface) that are specified by the UART protocol. The interface declares 'modports' that are to be used as logical port connections for components.

The port interface, `svt_uart_if`, contains the following modports, which you should use to connect the VIP to the DUT:

  - `svt_uart_monitor_modport`: This modport is used by the monitor component inside the DTE and DCE agents.

3.4 Functional Coverage

This section consists of the following sub-sections:

- "Built-In Coverage" on page 35
- "Coverage Callback Classes" on page 35
- "Enabling Built-In Coverage" on page 37

3.4.1 Built-In Coverage

The UART SVT VIP provides the following types of built-in coverage:

- "Transaction-Based Coverage" on page 35
- "Toggle-Based Coverage" on page 35
- "Pattern-Based Coverage" on page 35

For more details on covergroups, refer to the UART SVT VIP class reference HTML document:

```
$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/uart_svt_ovm_class_reference/html/level1_co
vergroups.html
```

3.4.1.1 Transaction-Based Coverage

The transaction-based coverage is a coverage collection based on the properties of transactions generated by the UART DCE and the UART DTE.

3.4.1.2 Toggle-Based Coverage

The toggle-based coverage is a signal-level coverage. The toggle coverage provides the baseline information that a system is connected properly and that a higher-level coverage or compliance failures are not simply the result of connectivity issues. The toggle coverage answers the question whether a bit changes from a value of 0 to 1 and back from 1 to 0. This type of coverage does not show every value of a multi-bit vector but it measures that all the individual bits of a multi-bit vector toggle.

3.4.1.3 Pattern-Based Coverage

It covers the pattern-based specific scenarios to recognize the sequence of a payload in transactions, etc.

3.4.2 Coverage Callback Classes

This section consists of the following sub-sections:

- "Coverage Data Callback" on page 36
- "Coverage Callback" on page 36
- "Toggle Coverage Data Callback" on page 36
- "Toggle Coverage Callback" on page 37

3.4.2.1 Coverage Data Callback

This callback class defines the default data and event information that are used to implement coverage groups. This class also includes the implementation of coverage methods that respond to coverage requests by setting the coverage data and triggering coverage events. This implementation does not include coverage groups.

The naming convention uses `def_cov_data` in class names for easy identification of these classes. The `def_cov_data` callback classes extend from the monitor callback class, that is `svt_uart_monitor_callback`. For example, the coverage data callback class name of the master monitor is `svt_uart_monitor_def_cov_data_callbacks`.

The following are the callback methods, which are implemented for sampling coverage:

- `pre_xact_observed_put`
- `xact_observed_cov`
- `transaction_started`
- `transaction_ended`

3.4.2.2 Coverage Callback

This callback class includes built-in covergroups based on data and events defined in the data class.

The naming convention uses `def_cov` in class names for easy identification of these classes. The `def_cov` callback classes extend from the coverage data callback class, that is `svt_uart_monitor_def_cov_data_callbacks`. The coverage callback class that implements built-in covergroups is `svt_uart_monitor_def_cov_callback`.

3.4.2.3 Toggle Coverage Data Callback

This callback class defines toggle data and event information that are used to implement coverage groups. This class also includes implementations of the coverage methods that respond to the coverage requests by setting the coverage data and triggering the coverage events. This implementation does not include coverage groups.

The naming convention uses `def_toggle_cov_data` in class names for easy identification of these classes. The `def_toggle_cov_data` callback classes extend from the monitor callback class, that is `svt_uart_monitor_callback`. The toggle coverage data callback class name is `svt_uart_monitor_def_toggle_cov_data_callback`.

The following methods are implemented for sampling the toggle coverage:

- `recognize_dtr_dsr_samples`
- `recognize_rts_cts_samples`
- `sample_rts_cts_toggle_bit_cov`
- `sample_dtr_dsr_toggle_bit_cov`

3.4.2.4 Toggle Coverage Callback

This callback class extends from the toggle coverage data callback class. This callback class includes built-in covergroups based on data and events defined in the data class.

The naming convention uses `toggle_def_cov` in class names for easy identification of these classes. The toggle coverage callback class implementing built-in covergroups is `svt_uart_monitor_toggle_def_cov_callback`.

3.4.3 Enabling Built-In Coverage

You can enable the built-in functional coverage by setting the `coverage_enable` attribute and the `toggle_coverage_enable` attribute in the configuration class, `svt_uart_agent_configuration`, to 1. To disable the coverage, set the attribute to 0.

By default, the coverage is disabled.

3.5 Exceptions

Exceptions are errors that are introduced into a transaction for the purpose of testing the DUT and to check its response in error scenarios. This can be done for agent transactions. The error injections are as follows:

- `PARITY_ERROR`
- `FRAMING_ERROR`
- `NO_OP_ERROR`

For more details on exceptions, refer to the following UART VIP class reference HTML document:

`$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/uart_svt_ovm_class_reference/html/class_svt_uart_transaction_exception_list.html`

3.6 Callbacks

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

- Callbacks are virtual methods with no initial code, so they do not provide any functionality unless they are extended. The exception to this rule is that some of the callback methods for functional coverage already contain a default implementation of a coverage model.

- The callback class is accessible to you, so the class can be extended. Including the testbench-specific extensions of default callback methods, the testbench-specific variables and methods control the behavior of supported callbacks in the testbench.

- Callbacks are called within the sequential flow at places where an external access is useful. In addition, the arguments to methods include the references to relevant data objects. For example, just before a monitor puts a transaction object into an analysis port is a good point to sample for functional coverage since the object reflects the activity that just happened on pins. A callback at this point with an argument referencing the transaction object allows this exact scenario.
- There is no need to invoke callback methods for the callbacks that are not extended. To avoid a loss of performance, callbacks are not executed by default.

UART VIP uses callbacks in the following three main applications:

- Access for functional coverage
- Access for scoreboarding
- Insertion of the user-defined code

This section consists of the following sub-section:

- "Agent Callbacks" on page 38

3.6.1 Agent Callbacks

In the agent, the callback methods are called by a driver and monitor components. The following callback classes, which contain the callback methods are invoked by the agent:

- `svt_uart_txrx_callback`
- `svt_uart_monitor_callback`

For details of these classes, refer to the following UART VIP class reference HTML documentation:

`$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/uart_svt_ovm_class_reference/html/class_svt_uart_txrx_callback.html`
and
`$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/uart_svt_ovm_class_reference/html/class_svt_uart_monitor_callback.html`

3.7 Protocol Checks

You can enable protocol checks by setting the configuration attribute, `check_enable`, in the `svt_uart_agent_configuration` class to `1`. To disable checks, set the attribute to `0`. By default, the protocol checks are enabled.

For a comprehensive list of all the protocol checks, refer to the UART VIP Class reference HTML documentation:

`$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/uart_svt_ovm_class_reference/html/protocolChecks.html`

3.8 Verification Features

This section consists of the following sub-sections:

- "Sequence Collection" on page 39
- "Verification Planner" on page 39
- "Source Code Visibility" on page 39
- "Protocol Analyzer Support" on page 39

3.8.1 Sequence Collection

UART VIP provides a collection of UART DTE and DCE sequences. These sequences can be registered with agent sequencers within DTE and DCE agents respectively to generate different types of UART scenarios. All UART DTE and DCE sequences extend from the base sequence, namely, `svt_uart_dte_base_sequence` and `svt_uart_dce_base_sequence` respectively.

For a list of all DTE and DCE sequences, refer to the UART VIP class reference HTML documentation:

```
$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/uart_svt_ovm_class_reference/html/sequencep
ages.html
```

Note Refer the `ts.sequence_collection_test.sv` and `ts.dce_dte_test.sv` tests in the intermediate example of the VIP.

3.8.2 Verification Planner

UART VIP provides verification plans, which track the verification progress of the UART protocol. UART VIP also provides a set of top-level plans and sub-plans. The verification plans are available at the following location:

```
$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/VerificationPlans
```

For more information, refer to the README file, which is available at the following location:
`$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/VerificationPlans/README`

3.8.3 Source Code Visibility

The source code visibility feature is available with Verdi to view some of the protected code of the VIP classes.

3.8.4 Protocol Analyzer Support

UART VIP supports Synopsys® Protocol Analyzer. Protocol Analyzer is an interactive graphical application, which provides protocol-oriented analysis and debugging capabilities.

For the UART SVT VIP, protocol file generation is enabled or disabled through the `enable_xml_gen` variable, which is defined in the `svt_uart_agent_configuration` class. The default value of this variable is `0`, which means that protocol file generation is disabled by default.

To enable protocol file generation, set the value of the `enable_xml_gen` variable to `1` in the agent configuration of each DTE or DCE for which protocol file generation is desired. The next time the environment is simulated, protocol files are generated according to the agent configuration. The protocol files are in the `.xml` format. Import these files into Protocol Analyzer to view protocol transactions.

To invoke Protocol Analyzer, use the following:

```
% $DESIGNWARE_HOME/bin/pa
```

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

```
$DESIGNWARE_HOME/vip/tools/pa/latest/doc/protocol_analyzer_getting_started.pdf
```

3.9 Using Native Protocol Analyzer for Debugging

3.9.1 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.

3.9.2 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_uart_agent_configuration >.enable_xml_gen = 1; // Default is 0
< svt_uart_agent_configuration>.pa_format_type =svt_xml_writer::
format_type_enum'(2); // 0 is XML, 1 FSDB and 2 both XML and FSDB, default it will
be zero
```

Runtime Option

```
+svt_enable_pa=fsdb
```

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

3.9.3 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.

3.9.4 Documentation

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

$VERDI_HOME/doc/Verdi_Transaction_and_Protocol_Debug.pdf

3.9.5 Limitations

Interactive support is available only for VCS.

4
Using UART Verification IP

This section describes SystemVerilog OVM example testbenches that show the general usage for various applications. Tables 4-1 lists the summary of the examples.

Table 4-1 SystemVerilog Example Summary

Example Name	Level	Description
tb_uart_svt_ovm_basic_sys	Basic	The example consists of the following: • A top-level testbench in SystemVerilog • A dummy DUT in the testbench, which has two UART interfaces • An OVM verification environment • UART VIP components in the OVM verification environment • Three tests illustrating base, directed, and random transaction generation. The quickstart for this example is available at the following location: `$DESIGNWARE_HOME/vip/svt/uart_svt/latest/examples/sverilog/tb_uart_svt_ovm_basic_sys/doc/tb_uart_svt_ovm_basic_sys/index_basic.html`
tb_uart_svt_ovm_intermediate_sys	Intermediate	The example consists of the following: • A top-level testbench in SystemVerilog • A dummy DUT in the testbench, which has two UART interfaces • An OVM verification environment • OVM VIP components in the OVM verification environment • Tests illustrating the usage of verification features, such as sequence collection, PA, exceptions, callbacks, coverage, subscriber and scoreboard

Table 4-1 SystemVerilog Example Summary

Example Name	Level	Description
		The quickstart for this example is available at the following location: `$DESIGNWARE_HOME/vip/svt/uart_svt/latest/examples/sverilog/tb_uart_svt_ovm_intermediate_sys/doc/tb_uart_svt_ovm_intermediate_sys/index_intermediate.html`
tb_uart_svt_ovm_disable_txrx_handshake_intermediate_sys	Intermediate	The example consists of the following: • A top-level testbench in SystemVerilog • A dummy DUT in the testbench, which has two UART interfaces • An OVM verification environment • OVM VIP components in the OVM verification environment • Tests illustrating the usage of full-hardware handshaking by disabling TX/RX handshake configuration

You can perform the following steps to install and run the `tb_uart_svt_ovm_basic_sys` example:.

a. Install the example using the following commands:

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

The example gets installed in the following location:
`<design_dir>/examples/sverilog/uart_svt/tb_uart_svt_ovm_basic_sys`

b. Run the testbench using the sim script. Tests are provided in the `tests` directory.

For example, to run the `ts.directed_test.sv` test, use the following command:

```
./run_uart_svt_ovm_basic_sys -w directed_test vcsvlog-svlog
```

Invoke `./run_uart_svt_ovm_basic_sys -help` to show more options.

For more details regarding installing and running the example, refer to the README file from the following location:
`$DESIGNWARE_HOME/vip/svt/uart_svt/latest/examples/sverilog/tb_uart_svt_ovm_basic_sys/README`

or

`<design_dir>/examples/sverilog/uart_svt/tb_uart_svt_ovm_basic_sys/README`

Note You can use the above steps to install and run the `tb_uart_svt_ovm_intermediate_sys` example also.

4.1 Getting Help on Example Run/make Scripts

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

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

```
run_uart_svt_ovm_basic_sys
run_uart_svt_ovm_basic_sys [-32] [-verbose] [-debug_opts] [-waves] [-clean] [-nobuild]
[-norun] [-pa] <scenario> <simulator>
where <scenario> is one of: all base_test directed_test random_test
<simulator> is one of: vcsvlog vcsmxvlog mtivlog vcsmxpcvlog vcsmxpipvlog ncvlog
vcspcvlog
      -32 forces 32-bit mode on 64-bit machines
      -verbose enable verbose mode during compilation
      -debug enable debug mode for SVT simulations
      -waves [fsdb|verdi|dump] enables waves dump and optionally opens viewer (VCS
      only)
      -clean clean simulator generated files
      -nobuild skip simulator compilation
      -norun exit after simulator compilation
      -pa invoke PA after execution
```

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

```
gmake help
Usage: gmake USE_SIMULATOR=<simulator> [VERBOSE=1] [DEBUG=1] [FORCE_32BIT=1]
[WAVES=fsdb|verdi|dump] [NOBUILD=1] [PA=1] [<scenario> ...]
Valid simulators are: vcsvlog vcsmxvlog mtivlog vcsmxpcvlog vcsmxpipvlog ncvlog
vcspcvlog
Valid scenarios are: all base_test directed_test random_test
```

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

5 Verification Topologies

The chapter consists of the following sections that show you how UART VIP can be used from a high-level to test the DTE or DCE DUT:

5.1 DTE DUT and DCE VIP

Scenario: The VIP is required to verify the UART DTE DUT.

Testbench Setup: Configure the UART agent configuration to have one DCE agent in an active mode. The active DCE agent responds to the transactions generated by the DTE DUT. The DCE agent also performs passive functions, such as protocol checking, coverage generation, and transaction logging.

Implementation of this topology requires the setting of the following properties:

(Assuming the instance name of the agent configuration is `dce_cfg`.)

Agent Configuration Settings:

```
dce_cfg.is_active = 1;
```

Also, the implementation of this topology requires the following setting in the top level:

```
Top-level setting:
/**
* Instantiate the SV Interface for DCE and connect the system clock to the clock signal
* in the interface */
  svt_uart_if uart_dce_if(SystemClock);
```

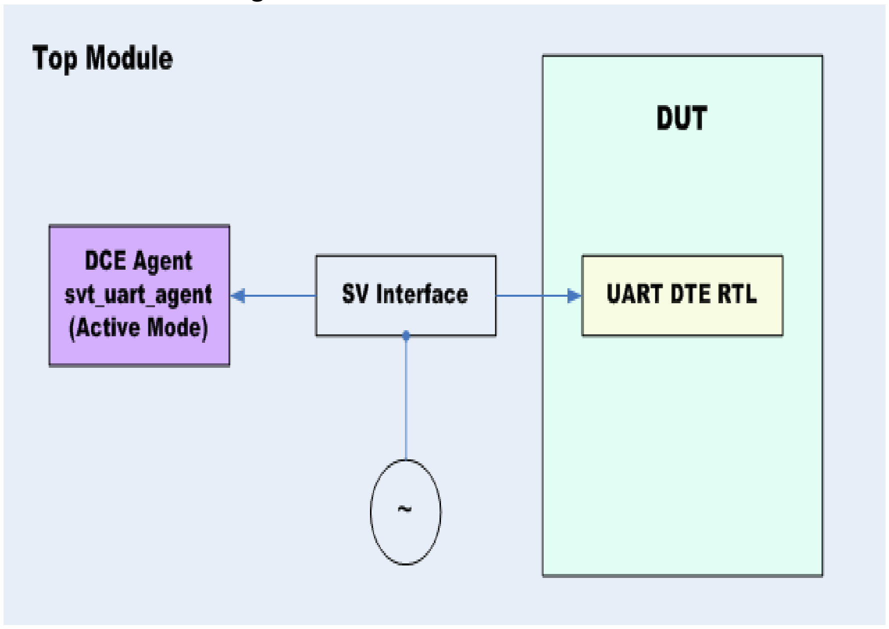
```
/** Instantiate the UART BFM wrapper acting as DCE*/
  svt_uart_bfm_wrapper  #(.UV_DEVICE_TYPE(`UV_DCE)) Bfm1 (uart_dce_if);
```

Note

Here, the `UV_DEVICE_TYPE` parameter, of the `svt_uart_bfm_wrapper` module, specifies the device type, that is, DTE or DCE and the ``UV_DCE` macro shows the agent as DCE.

When the DUT has a single UART DTE agent to be verified, the testbench can use the DCE agent. Figures 5-1 shows the DTE DUT and DCE VIP.

Figure 5-1 DTE DUT and DCE VIP - Usage



5.2 DCE DUT and DTE VIP

Scenario: The VIP is required to verify the UART DCE DUT.

Testbench Setup: Configure the UART agent configuration to have one DTE agent in an active mode. The active DTE agent generates UART transactions for the DCE DUT. The DTE agent also performs passive functions, such as protocol checking, coverage generation, and transaction logging.

When the DUT has a single UART DCE agent to be verified, the testbench can use the DTE agent.

Implementation of this topology requires the setting of the following properties:

(Assuming the instance name of the agent configuration is `dte_cfg`.)

Agent Configuration Settings:

```
dte_cfg.is_active = 1;
```

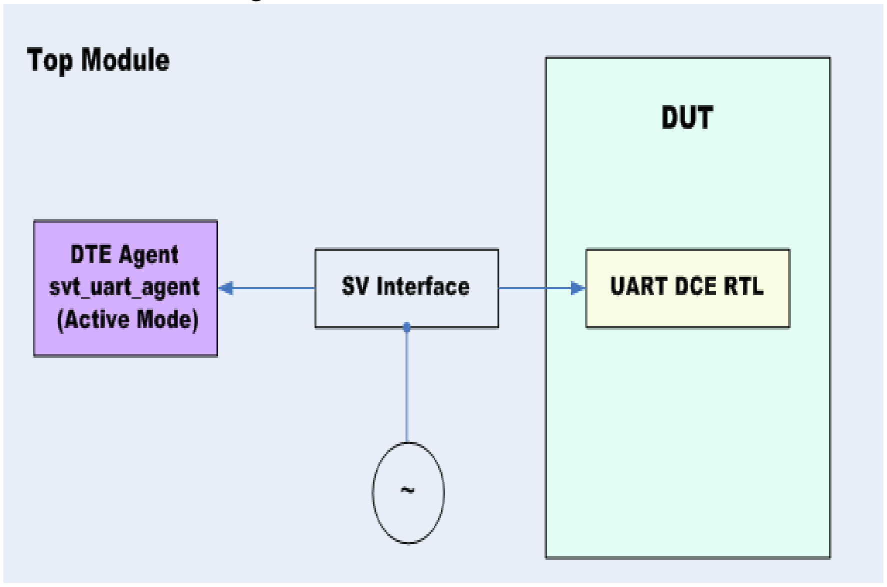
Also, the implementation of this topology requires the following setting in the top level:

```
Top-level setting:
/**
* Instantiate the SV interface for DTE and connect the system clock to the clock signal
in the interface */
svt_uart_if uart_dte_if(SystemClock);

/** Instantiate the UART BFM wrapper acting as DTE*/
svt_uart_bfm_wrapper  #(.UV_DEVICE_TYPE(`UV_DTE)) Bfm0 (uart_dte_if);
```

Note Here, the `UV_DEVICE_TYPE` parameter, of the `svt_uart_bfm_wrapper` module, specifies the device type, that is, DTE or DCE and the `` `UV_DTE `` macro shows the agent as DTE.

Figures 5-2 shows the DCE DUT and DTE VIP.

Figure 5-2 DCE DUT and DTE VIP - Usage



5.3 DTE DUT With Passive DTE VIP and DCE VIP

Scenario: The VIP is required to verify the UART DTE DUT.

Testbench Setup: Configure the UART agent configuration to have one DCE agent in an active mode and one DTE agent in a passive mode. The active DCE agent responds to the transactions generated by the DTE DUT. The DCE and DTE agents also perform passive functions, such as protocol checking, coverage generation, and transaction logging.

Implementation of this topology requires the setting of the following properties:

(Assuming the instance name of the agent configuration is `dce_cfg` and `dte_cfg`.)

Agent Configuration Settings:

```
dce_cfg.is_active = 1;
dte_cfg.is_active = 0;
```

Also, the implementation of this topology requires the following setting in the top level:

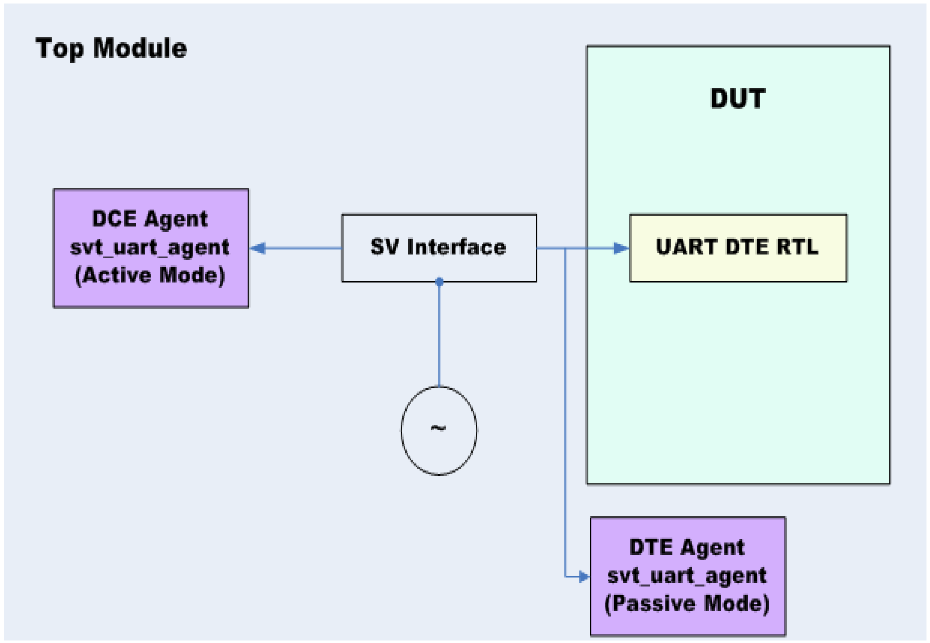
```
Top-level setting:
```

```
/**
* Instantiate the SV interface for DTE and connect the system clock to the clock signal
* in the interface */
svt_uart_if uart_dte_if(SystemClock);

/**
* Instantiate the SV interface for DCE and connect the system clock to the clock signal
* in the interface */
svt_uart_if uart_dce_if(SystemClock);

/** Instantiate the UART BFM Wrapper acting as DTE*/
svt_uart_bfm_wrapper  #(.UV_DEVICE_TYPE(`UV_DTE)) Bfm0 (uart_dte_if);

/** Instantiate the UART BFM Wrapper acting as DCE*/
svt_uart_bfm_wrapper  #(.UV_DEVICE_TYPE(`UV_DCE)) Bfm1 (uart_dce_if);
```

Note Here, the `UV_DEVICE_TYPE` parameter, of the `svt_uart_bfm_wrapper` module, specifies the device type, that is, DTE or DCE. `` `UV_DTE `` and `` `UV_DCE `` macros show the agent as DTE and DCE respectively.

When the DUT has a single UART DTE agent to be verified, the testbench can use the DCE agent. Figures 5-3 shows the DTE DUT with the passive DTE VIP and DCE VIP.

Figure 5-3 DTE DUT With Passive DTE VIP and DCE VIP - Usage



5.4 DTE VIP and DCE DUT With Passive DCE VIP

Scenario: The VIP is required to verify the UART DCE DUT.

Testbench Setup: Configure the UART agent configuration to have one DTE agent in an active mode and one DCE agent in a passive mode. The active DTE agent responds to the transactions generated by the DCE DUT. The DTE and DCE agents also perform passive functions, such as protocol checking, coverage generation, and transaction logging.

Implementation of this topology requires the setting of the following properties:

(Assuming the instance name of the agent configuration is `dce_cfg` and `dte_cfg`.)

Agent Configuration Settings:

```
dte_cfg.is_active = 1;
dce_cfg.is_active = 0;
```

Also, the implementation of this topology requires the following setting in the top level:

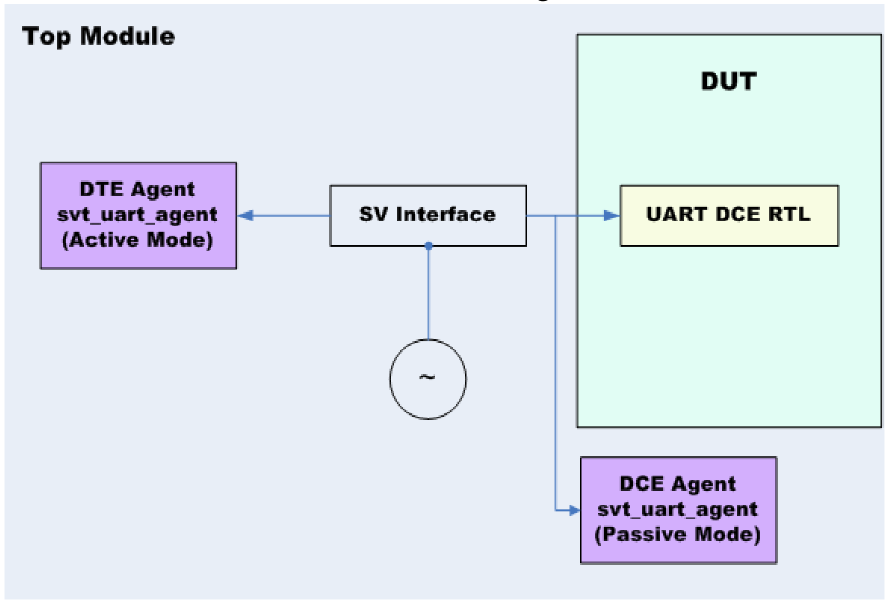
```
Top-level setting:
/**
* Instantiate the SV interface for DTE and connect the system clock to the clock signal
* in the interface */
svt_uart_if uart_dte_if(SystemClock);

/**
* Instantiate the SV interface for DCE and connect the system clock to the clock signal
* in the interface */
svt_uart_if uart_dce_if(SystemClock);

/** Instantiate the UART BFM wrapper acting as DTE*/
svt_uart_bfm_wrapper  #(.UV_DEVICE_TYPE(`UV_DTE)) Bfm0 (uart_dte_if);

/** Instantiate the UART BFM Wrapper acting as DCE*/
```

```
svt_uart_bfm_wrapper  #(.UV_DEVICE_TYPE(`UV_DCE)) Bfm1 (uart_dce_if);
```

Note Here, the `UV_DEVICE_TYPE` parameter, of the `svt_uart_bfm_wrapper` module, specifies the device type, that is, DTE or DCE. `` `UV_DTE `` and `` `UV_DCE `` macros show the agent as DTE and DCE respectively.

When the DUT has a single UART DCE agent to be verified, the testbench can use the DTE agent. Figures 5-4 shows the DCE DUT with the DTE VIP and passive DCE VIP.

Figure 5-4 DCE DUT With DTE VIP and Passive DCE VIP - Usage



5.5 DTE DUT With Passive DTE VIP and DCE DUT With Passive DCE VIP

Scenario: The VIP is required to monitor the UART DCE and DTE DUT.

Testbench Setup: Configure the UART agent configuration to have one DTE agent in a passive mode and one DCE agent also in a passive mode. The DTE and DCE agents also perform passive functions, such as protocol checking, coverage generation, and transaction logging.

Implementation of this topology requires the setting of the following properties:

(Assuming the instance name of the agent configuration is `dce_cfg` and `dte_cfg`.)

Agent Configuration Settings:

```
dte_cfg.is_active = 0;

dce_cfg.is_active = 0;
```

Also, the implementation of this topology requires the following setting in the top level:

```
Top-level setting:
/**
* Instantiate the SV interface for DTE and connect the system clock to the clock signal
* in the interface */
svt_uart_if uart_dte_if(SystemClock);
```

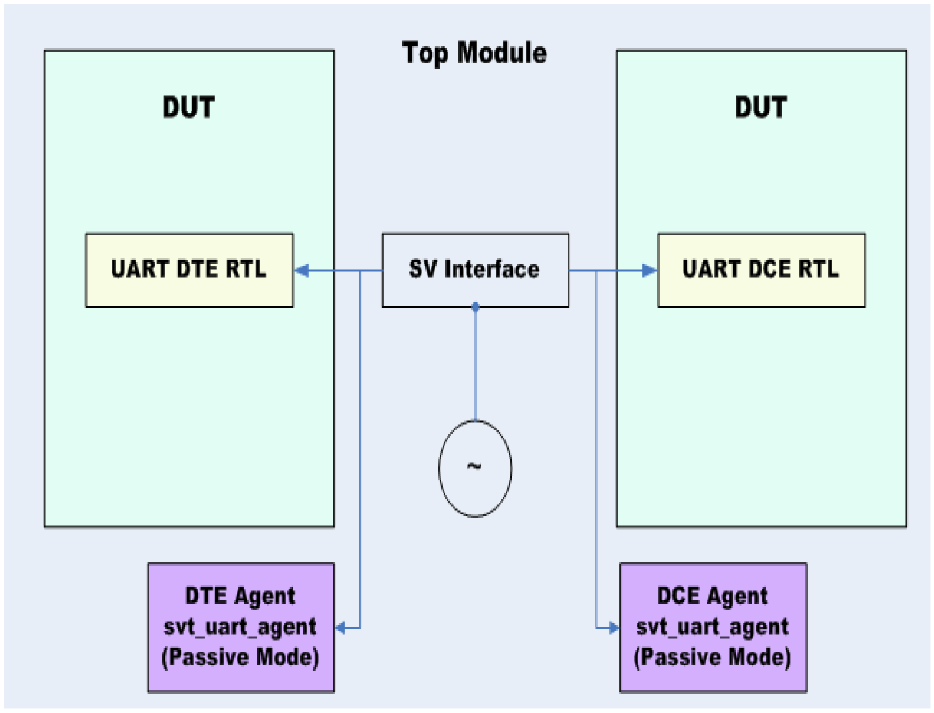
```
/**
* Instantiate the SV interface for DCE and connect the system clock to the clock signal
* in the interface */
svt_uart_if uart_dce_if(SystemClock);

/** Instantiate the UART BFM wrapper acting as DTE*/
svt_uart_bfm_wrapper  #(.UV_DEVICE_TYPE(`UV_DTE)) Bfm0 (uart_dte_if);

/** Instantiate the UART BFM wrapper acting as DCE*/
svt_uart_bfm_wrapper  #(.UV_DEVICE_TYPE(`UV_DCE)) Bfm1 (uart_dce_if);
```

Note

Here, the `UV_DEVICE_TYPE` parameter, of the `svt_uart_bfm_wrapper` module, specifies the device type, that is, DTE or DCE. `` `UV_DTE `` and `` `UV_DCE `` macros show the agent as DTE and DCE respectively.

Figures 5-5 shows the DCE DUT with the passive DTE VIP and DCE VIP with passive DCE VIP.

Figure 5-5 DCE DUT With Passive DTE VIP and DCE VIP With Passive DCE VIP- Usage



6 Usage Notes

To model different UART protocol scenarios, the properties of UART transactions can be used to create different stimulus from the DTE or DCE agent. This chapter covers the code snippets of DTE and DCE sequences to cover the common UART verification scenarios.

This chapter consists of the following sections:

6.1 UART Transaction Properties

The UART transaction class has different properties, which are used to create the different stimulus from the DTE agent or the DCE agent. For details on the properties of the transaction class, refer to the UART VIP class reference HTML documentation:

```
$DESIGNWARE_HOME/vip/svt/uart_svt/latest/doc/uart_svt_ovm_class_reference/html/class_svt
_uart_transaction.html
```

6.2 Flow Control

Flow control is the process of managing the rate of data transmission between two nodes to prevent a fast sender from over running a slow receiver. For example, as the DTE to DCE speed is a few times faster than the DCE to DCE speed, the PC can send data to the modem at a higher rate. In this connection sooner or later, the data may get lost because of the buffer overflow, so there is a need to monitor the control of data flow.

Flow control mechanisms are classified based on whether a receiving node sends feedback to a sending node. UART VIP supports the following two types of the flow control mechanism:

- "Hardware Flow Control" on page 52
- "Software Flow Control" on page 56

6.2.1 Hardware Flow Control

Hardware flow control is also known as the Request to Send (RTS) or Clear to Send (CTS) flow control. To implement this control, use two additional wires (RTS and CTS) in the sequential cable. This results in increasing the data transmission rate.

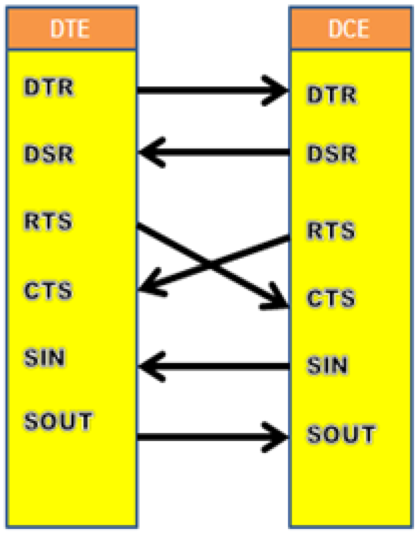
Here, the transmission activates the RTS line when the computer is ready to send data. If the modem has a free buffer for this data, it activates the CTS line in response and the computer starts sending the data. If the modem lacks free memory, it does not activate the CTS line. The hardware handshaking is also known as out-of-band flow control because the signals are generated and observed outside the flow of the data. The hardware -flow control is proactive. You do not start transmitting until you receive the Go signal. If you never get it, you can never start.

UART VIP provides the following features of the hardware-flow control:

- Configuration to enable or disable the RTS-CTS handshake.
- Configuration to enable or disable the DTR-DSR handshake.
- Configuration to enable or disable the Tx/Rx handshake
- Configurable receiver-buffer size.
- Support for inserting delay to assert the RTS pin.
- Support for inserting delay to assert the CTS pin.
- Support for control of the DTR pin from DTE.
- Support for control of the DSR pin from DCE.
- Support for specifying delay to flush the receiver buffer (FIFO).
- Configuration to support the auto-flow mechanism

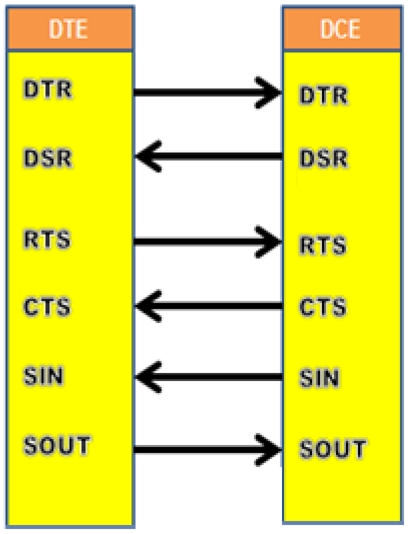
Figures 6-1 shows the handshaking in both data-transfer directions, that is DTE <--> DCE (The output of the transmitter RTS comes on the input CTS side of the receiver). In case of the DTE to DCE transfer to happen on SOUT, the handshaking first starts from asserting RTS from DTE which in turn is input to the CTS pin on the DCE side. In this mode, the RTS signal asserts only when the receiver reaches empty FIFO state. When the receiver reaches a full buffer, the RTS signal de-asserts, which in turn is input to the CTS pin on the DTE side, which signifies DTE to hold further transmission until the receiver buffer gets empty. Similarly, the DCE to DTE transfer also happens on the SIN handshaking.

Figure 6-1 Handshaking in Both Directions (DTE <--> DCE)



Figures 6-2 shows the handshaking in a single direction, that is DTE --> DCE.

Figure 6-2 Handshaking in a Single Direction (DTE --> DCE)



UART VIP supports the following possible verification environments:

- ✦ "Full-Hardware Handshaking" on page 54
- ✦ "Partial-Hardware Handshaking" on page 54
- ✦ "Without-Hardware Handshaking" on page 55

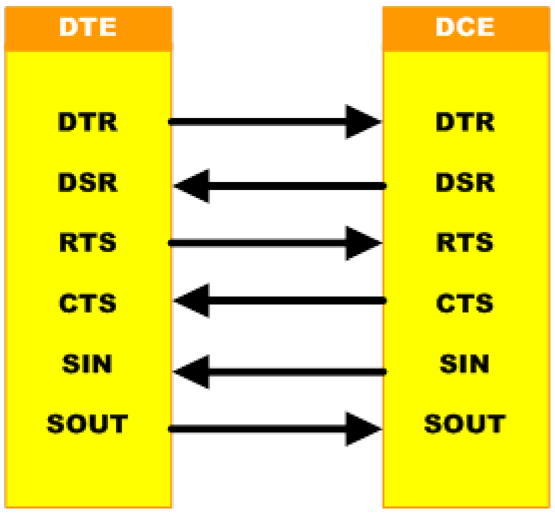
6.2.1.1 Full-Hardware Handshaking

In the full-hardware handshaking, DTE and DCE use RTS-CTS and DTR-DSR signals to exchange the flow-control information. Once all signals get asserted, then only data transfer on SIN-SOUT signals can happen.

The handshaking starts when DTE asserts its RTS signal to signify DCE that it is ready to exchange data. In reference to RTS, DCE asserts its CTS signal if its receiver buffer is free to accept data. The full handshaking completes when DTE asserts its DTR signal and in reference to it, DCE asserts its DSR signal. After completion of the handshaking, data transmission can happen on SIN and SOUT.

Figures 6-3 shows the full-hardware handshaking.

Figure 6-3 Full-Hardware Handshaking



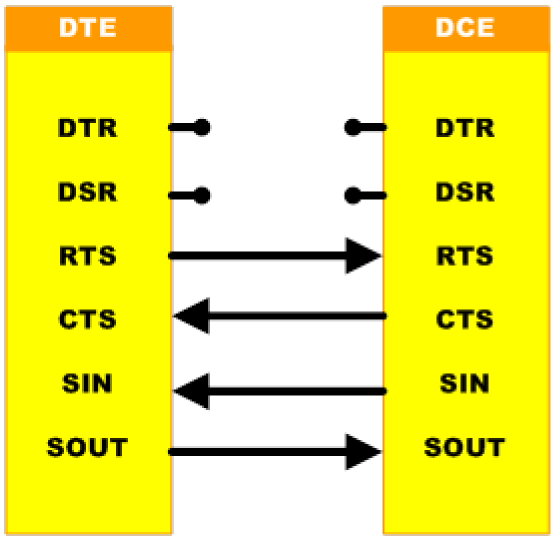
6.2.1.2 Partial-Hardware Handshaking

In the partial-hardware handshaking, DTE and DCE use RTS-CTS to exchange the flow-control information. In this handshaking, DTR-DSR signals of DTE and DCE are not connected to each other. Once RTS-CTS signals get asserted, then only data transfer on SIN-SOUT signals can happen.

The handshaking starts when DTE asserts its RTS signal to signify DCE that it is ready to exchange data. The handshaking completes when in reference to RTS, DCE asserts its CTS signal if its receiver buffer is free to accept data. After completion of the handshaking, data transmission can happen on SIN and SOUT.

Figures 6-4 shows the partial-hardware handshaking.

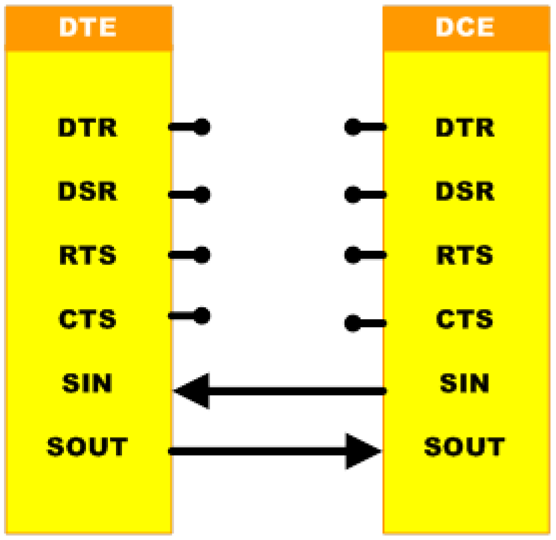
Figure 6-4 Partial-Hardware Handshaking



6.2.1.3 Without-Hardware Handshaking

In this handshaking, only data ports are connected to each other. All the other ports have no connection. Figures 6-5 is the connection that does not required RTS-CTS and DTR-DSR signals to exchange data with each other.

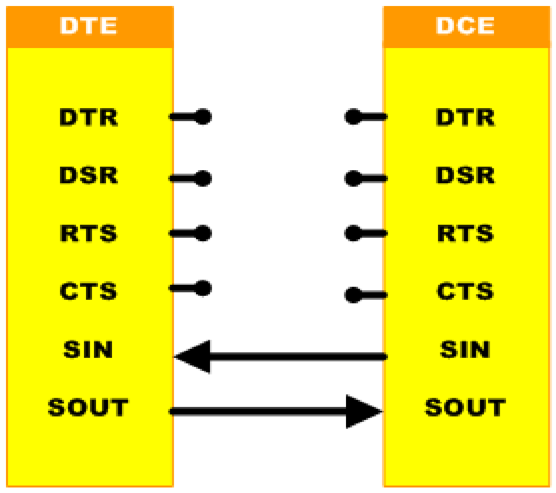
Figure 6-5 Without-Hardware Handshaking



6.2.2 Software Flow Control

In this handshaking, only data ports are connected to each other. It uses XON and XOFF data characters for flow control. Figures 6-6 shows the software handshaking.

Figure 6-6 Software Handshaking



6.3 Fractional Baud Rate Divisor

The baud rate defines the transfer rate of the number of symbols per second and it is derived as:

```
Baud rate=(input clock)/(sample rate * divisor)
```

Here, the sample rate of UART is 16 (commonly used value) and the input clock is the peripheral clock.

For example, if the input clock (peripheral clock) = 25 MHz, Baud rate = 9600 and sample rate = 16 then,

```
Divisor value=(25*10^6)/(16*9600)
```

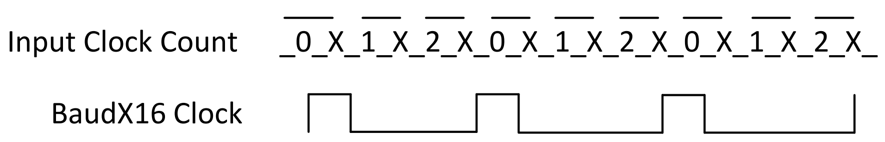
Therefore, `Divisor value = 162.76 (Integer = 162 and Fractional = 0.76)`

6.3.1 Implementation Details

6.3.1.1 Integer Divisor

In case of normal integer values, the baud clock is generated after the Nth cycle of the input clock. For example, the baud clock generated for a divisor of 3, that is N=3 is shown in Figure 6-7:

Figure 6-7 Integer Divisor of 3



6.3.1.2 Fractional Divisor

In case of a fractional baud divisor, the output of baud clock generation will depend on the value of MULT.

There are two ways in which a configurable fractional divisor can be implemented:

6.3.1.2.1 Achieving a Fractional Baud Divisor Value Over Single Bit Period

Take an example where:

```
Reference clock = 25MHz
Uart sampling rate (bit period) = 16
Divisor value = 1.76 (Integer = 1 and Fractional = 0.76) which derives, N (Integer) = 1
MULT = 0.76 * (bit period) = 0.76 * 16 = approx. 12 (exact value is 12.16)
Total duration = bit period = 16
```

Round off the value of MULT based on a configurable real variable, such as the median. If the median is configured to 0.4, then 12.4 or 12.39 will be rounded off to 12 and 12.41 and above will be rounded off to 13.

```
Baud clock=(input clock)/(N + MULT/(bit period))
```

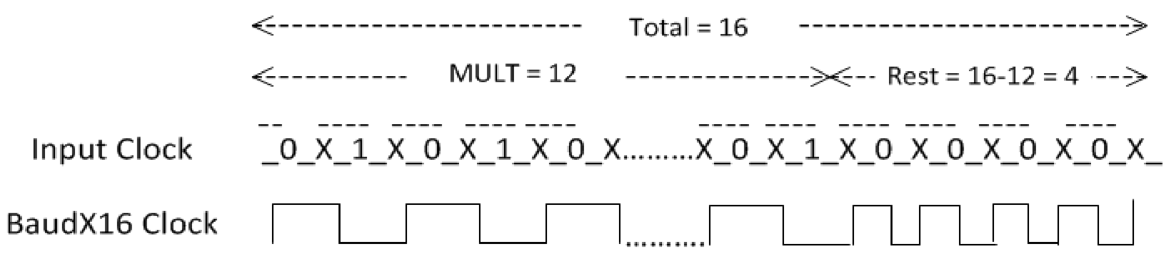
Therefore, `Baud clock = 25 MHz/ (1 + (12/16))`

For the duration of the MULT, each of the output baudX16 clock cycles are generated after 'N+1' input clocks.

For the remaining (Total duration – MULT) output baudX16 clock cycles, each baudX16 clock is generated after 'N' input clocks.

Divisor = 1.76, N=1 and MULT = 12

Figure 6-8 Fractional Baud Divisor Value Over Single Bit Period



As shown in Figure 6-8, 12 baudX16 cycles are generated using (N+1) reference clock cycle, that is two reference clock cycles and the remaining (4 out of 16) baudX16 cycles are generated using N reference clock cycle that is one reference clock cycle, thus giving an approximate average of 1.76 over a bit period.

6.3.1.2.2 Achieving a Fractional Baud Divisor Value Over a Period of (2^REG_WIDTH) Number of BaudX16 Cycles

`REG_WIDTH` is the width of the register that holds the fractional part of the divisor.

Take an example where:

```
Reference clock = 25 MHz
REG_WIDTH = 6
Divisor value = 1.76 (Integer = 1 and Fractional = 0.76)
N = 1;
MULT = 0 .76 * (2^6) = 0.76 * 64 = approx. 48 (exact value is 48.64)
Total duration = bit period = 16
```

Round off the value of MULT based on a configurable real variable, such as the median. If median is configured to 0.4, then 48.4 or 48.39 will be rounded off to 48 and 48.41 and above will be rounded off to 49.

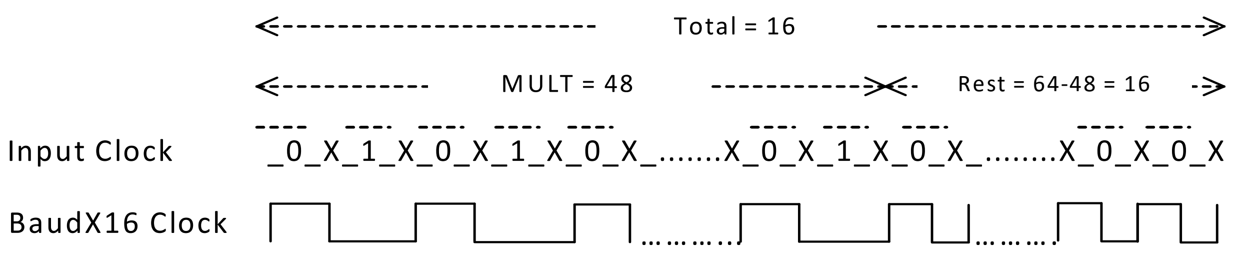
```
Baud clock=(input clock)/(N + MULT/(2^REG_WIDTH))
```

Therefore, `Baud clock = 25 MHz/ (1 + (48/64))`

For the duration of the MULT, each of the output baudX16 clock cycles is generated after 'N+1' input clocks. For the remaining (Total duration – MULT) output baudX16 clock cycles, each baudX16 clock is generated after 'N' input clocks.

Divisor = 1.76 = N=1 and MULT = 48

Figure 6-9 Fractional Baud Divisor Value Over a Period



As shown in Figure 6-9, 48 baudX16 cycles are generated using N+1, that is two reference clock cycles and the remaining (16 out of 64) baudX16 cycles are generated using one reference clock cycle, thus giving an approximate average of 1.76 over 64 baudX16 cycles.

6.4 UART VIP OVM Integration Reference

6.4.1 Introduction

This section provides UART VIP top-level modification steps which help you integrate UART VIP into your testbench environment. UART VIP can be used with the following verification topologies:

❖ DTE as DUT and DCE as VIP as shown in Figure 6-10

❖ DCE as DUT and DTE as VIP as shown in Figure 6-11

Figure 6-10 DTE as DUT and DCE as VIP

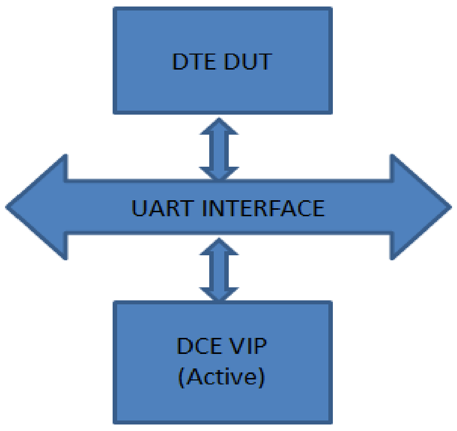
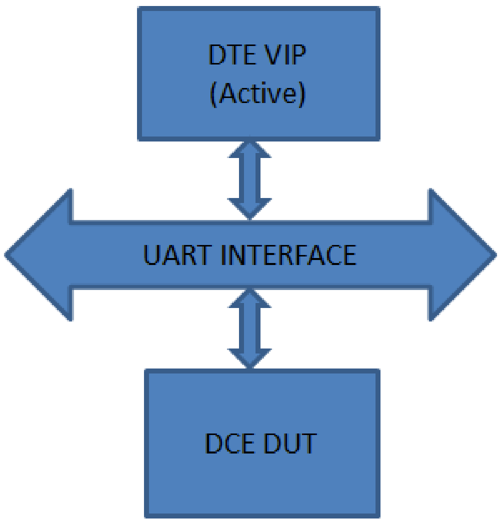


Figure 6-11 DCE as DUT and DTE as VIP



6.4.2 Inclusion of VIP files

To integrate UART VIP into your testbench environment, you need to modify its scripts to include the necessary directories, files and defines by referring the compilation log created after running any one of the tests available as a part of the VIP product.

For example, after you run the `random_test` from the `tb_uart_svt_ovm_basic_sys` example, the `compile.log` file is created in the logs directory. In the `compile.log` file, you can find the commands related to a specific simulator, the details of the directories which need to be included, the defines and other switches.

6.4.3 Integration of UART VIP in an OVM Based Setup

You can use the following file as a reference to integrate UART VIP in your testbench environment:

```
examples/sverilog/uart_svt/tb_uart_svt_ovm_<basic/intermediate/disable_txrx_handshake_i
ntermediate>_sys/top.sv
```

Perform the following steps to integrate UART VIP into your testbench.

1. Include the UART SVT OVM package

   ```
   `include "svt_uart.ovm.pkg"
   ```

2. UART Reset Requirement: Reset toggle (low--high--low) is always mandatory because UART VIP expects necessary reset toggle before any stimulus can be fired from it. You can use either of the following two ways to do this:

   a. Provide toggled reset to the VIP reset pin.

   b. Include the reset interface file in the top level testbench and reset.

      ```
      `include "uart_reset_if.svi"
      ```

Note The inclusion of uart_reset_if.svi is not required if you directly provide a toggled reset to the VIP

3. Declare the system clock if it is not already declared.

```
/** Signal to generate the clock */
bit SystemClock;
```

4. Import OVM and SVT and UART SVT based packages.

```
/** Import OVM Package */
import ovm_pkg::*;

/** Import the SVT OVM Package */
import svt_ovm_pkg::*;

/** Import UART SVT OVM Packages */
import svt_uart_ovm_pkg::*;
```

5. Includes all the test files.

```
`include "top_test.sv"
```

Note User can include own tests and sequences as well. Tests and sequences can be part of any other package or scope.

6. Instantiate the SV Interface for DTE/DCE (as per the DUT's requirement) and connect the system clock to the clock signal in the interface.

```
svt_uart_if uart_dte_if (SystemClock); // in case DTE VIP
```

or

```
svt_uart_if uart_dte_if (SystemClock); // in case DCE VIP
```

7. Instantiate UART BFM Wrapper acting as DTE/DCE.

```
svt_uart_bfm_wrapper # (.UV_DEVICE_TYPE (`UV_DTE)) Bfm0 (uart_dte_if); // in case
DTE VIP
```

Or

```
svt_uart_bfm_wrapper # (.UV_DEVICE_TYPE (`UV_DCE)) Bfm1 (uart_dce_if); // in case
DCE VIP
```

Note UV_DEVICE_TYPE is the parameter in the svt_uart_bfm_wrapper module. It can take the value (`UV_DTE or `UV_DCE) which tells the VIP to work in DTE mode or in DCE mode. Instance name Bfm0 & Bfm1 may differ based on your choice.

8. Instantiate reset interface and assign the system clock to reset the interface's clock pin and also assign the reset pin from the reset interface to the reset pins from the VIP DTE or DCE interface

```
/** Reset Interface instantiation */
uart_reset_if uart_reset_if ();
assign uart_reset_if.clk = SystemClock;
assign uart_dte_if.rst = uart_reset_if.reset; // in case DTE VIP
```

Or

```
assign uart_dce_if.rst = uart_reset_if.reset; // in case DCE VIP
```

Note This step is not mandatory if you are not using reset interface for reset purpose. You can directly provide the users reset to the VIP's reset, that is toggled reset.

For example,

```
assign uart_dte_if.rst = dut.reset; // dut is instance of DUT
```

Or

```
assign uart_dce_if.rst = dut.reset; // dut is instance of DUT
```

9. Generate the clock if required
10. Provide the UART SV interface to the UART ENV.

Note To achieve this in case of OVM set_config_object mechanism can be used (Refer the respective top.sv for reference).

11. The DUT's port connection with VIP in various verification environments and corresponding configuration settings in VIP are given below:

a. RTS Pin of One Device Connected to CTS Pin of Another Device

Figure 6-12 RTS pin of one device connected to CTS pin of another device

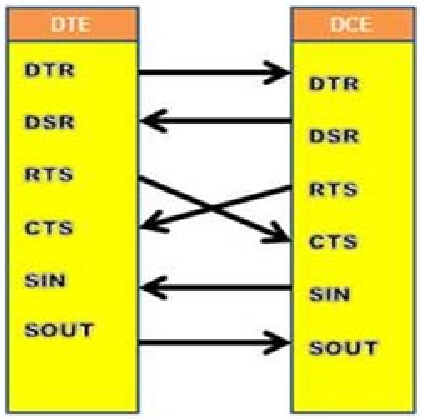


Figure 6-12 shows the cross connection between the RTS pin of one device with the respective CTS pin of another device. To meet this requirement, the VIP must configure the following attributes of the `svt_uart_configuration` class:

```
handshake_type = HARDWARE;
enable_tx_rx_handshake = 1; // must be set to 1 if cross connections between
RTS and CTS
enable_rts_cts_handshake = 1; // must be set to 1 if user needs RTS and CTS to
take part in Hardware handshaking otherwise 0
enable_dtr_dsr_handshake = 1; // must be set to 1 if user needs DTR and DSR to
take part in Hardware handshaking otherwise 0
```

There can be two topologies in this scenario:

i. DTE as DUT and DCE as VIP

In the following code snippet, the DTE is the DUT and the DCE is the VIP and an instance of the `svt_uart_if` is created with the name `uart_dce_if`. In this case direct assignment can be made between the DUT's instance and the VIP's interface and similarly other connections can be made.

```
assign uart_dce_if.dtr = dut.dtr;
assign dut.dsr = uart_dce_if.dsr;
assign uart_dce_if.cts = dut.rts;
assign dut.cts = uart_dce_if.rts;
assign uart_dce_if.sout = dut.sout;
assign dut.sin = uart_dce_if.sin;
```

ii. DCE as DUT and DTE as VIP

In the following code snippet, the DCE is the DUT and the DTE is the VIP and an instance of the `svt_uart_if` is created with the name `uart_dte_if`. In this case direct assignment can be made between the DUT's instance and the VIP's interface and similarly other connections can be made.

```
assign dut.dtr = uart_dte_if.dtr;
assign uart_dte_if.dsr = dut.dsr;
assign dut.cts = uart_dte_if.rts;
assign uart_dte_if.cts = dut.rts;
assign dut.sout = uart_dte_if.sout;
assign uart_dte_if.sin = dut.sin;
```

b. RTS and CTS Pin of One Device Connected to Corresponding RTS and CTS pin of Another Device Respectively

Figure 6-13 RTS and CTS Pin of one Device Connected to Corresponding RTS and CTS Pin of Another Device

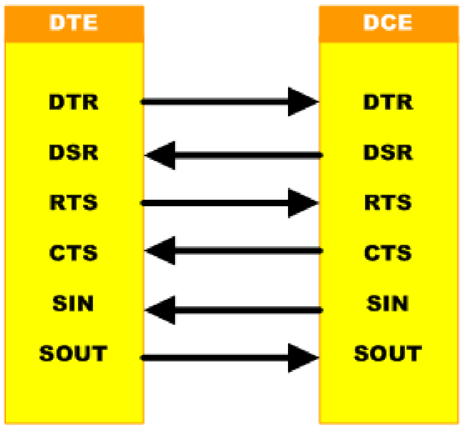


Figure 6-13 shows the RTS pin of the DTE connected to the RTS pin of the DCE and the CTS pin of the DCE connected to the CTS pin of the DTE. To meet this requirement, the VIP must configure the following attributes of the class `svt_uart_configuration`:

```
handshake_type = HARDWARE;
enable_tx_rx_handshake = 0; // must be set to 0
enable_rts_cts_handshake = 1; // must be set to 1 if user needs RTS and CTS to
take part in Hardware handshaking otherwise 0
enable_dtr_dsr_handshake = 1; // must be set to 1 if user needs DTR and DSR to
take part in Hardware handshaking otherwise 0
```

There can be two topologies in this scenario:

i. DTE as DUT and DCE as VIP

In the following code snippet, the DTE is the DUT and the DCE is the VIP and an instance of the `svt_uart_if` is created with the name `uart_dce_if`. In this case direct assignment can be made between the DUT's instance and the VIP's interface and similarly other connections can be made.

```
assign uart_dce_if.dtr = dut.dtr;
assign dut.dsr = uart_dce_if.dsr;
assign uart_dce_if.rts = dut.rts;
assign dut.cts = uart_dce_if.cts;
assign uart_dce_if.sout = dut.sout;
assign dut.sin = uart_dce_if.sin;
```

ii. DCE as DUT and DTE as VIP

In the following code snippet, the DCE is the DUT and the DTE is the VIP and an instance of the `svt_uart_if` is created with the name `uart_dte_if`. In this case direct assignment can be made between the DUT's instance and the VIP's interface and similarly other connections can be made.

```
assign dut.dtr = uart_dte_if.dtr;
assign uart_dte_if.dsr = dut.dsr;
assign dut.rts = uart_dte_if.rts;
assign uart_dte_if.cts = dut.cts;
assign dut.sout = uart_dte_if.sout;
assign uart_dte_if.sin = dut.sin;
```

c. In case of Software Handshaking

Figure 6-14 Software Handshaking

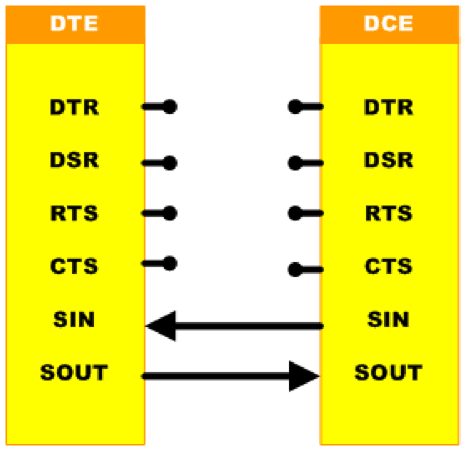


Figure 6-14 shows the connection between the SIN and SOUT pins. Only data pins are used to perform handshaking. To meet this requirement, the VIP must configure the following attributes of the class `svt_uart_configuration`:

```
handshake_type = SOFTWARE;

enable_tx_rx_handshake = 1; // optional to any value

enable_rts_cts_handshake = 0; // must be set to 0 in case of software
handshaking

enable_dtr_dsr_handshake = 0; // must be set to 0 in case of software
handshaking
```

There can be two topologies in this scenario:

i. DTE as DUT and DCE as VIP

In the following code snippet, DTE is the DUT and DCE is the VIP and an instance of `svt_uart_if` is created with the name `uart_dte_if`. In this case direct assignment can be made between the DUT's instance and the VIP's interface and similarly other connections can be made.

```
assign uart_dce_if.sout = dut.sout;
assign dut.sin = uart_dce_if.sin;
```

Note The code snippet takes Figure 6-14 as a reference. The name of the pin could differ for SIN and SOUT. So, while assigning these pins, you need to keep in mind that in case DCE is assigned as the VIP, SIN is the output from the DCE VIP and SOUT is input to the DCE VIP.

ii. DCE as DUT and DTE as VIP

In the following code snippet, DCE is the DUT and DTE is the VIP and an instance of `svt_uart_if` is created with the name `uart_dte_if`. In this case, direct assignment can be made between the DUT's instance and the VIP's interface:

```
assign dut.sout = uart_dte_if.sout;
assign uart_dte_if.sin = dut.sin;
```

Note The code snippet takes Figure 6-14 as a reference. The name of the pin could differ for SIN and SOUT. So, while assigning these pins, you need to keep in mind that in case DTE is assigned as the VIP, SOUT is the output from the DTE VIP and SIN is input to the DTE VIP.

6.5 UART OVM Scenario Reference Guide

Table 6-1 provides the scenario reference guide for UART OVM Configuration class attributes:

Table 6-1 OVM Scenario Reference Guide (Configuration Class)

Scenario	Configuration (svt_uart_ configuration) Class attributes	Reference Tests	Remarks
To configure different data widths	data_width	tb_uart_svt_ovm_intermediate_sys/tests/ts.data_parity_configuration_test.sv	Refer build phase of the test which showcases the data width configured as 5 bits.
To configure different parity types	parity_type	tb_uart_svt_ovm_intermediate_sys/tests/ts.data_parity_configuration_test.sv	Refer build phase of the test which showcases the type of parity configured as STICK_HIGH_PARITY.
To configure number of stop bits	stop_bit	tb_uart_svt_ovm_intermediate_sys/tests/ts.data_parity_configuration_test.sv	Refer build phase of the test which showcases the number of stop bits configured as 1 bit.

Scenario	Configuration (svt_uart_ configuration) Class attributes	Reference Tests	Remarks
To configure the handshake type	handshake_type	tb_uart_svt_ovm_intermediate_sys/tests/ts.partial_hardwr_hndshk_disable_rts_cts_test.sv	Refer build phase of the test which showcases type of handshake configured as HARDWARE.
		tb_uart_svt_ovm_intermediate_sys/tests/ts.partial_hardwr_hndshk_disable_dtr_dsr_test.sv	Refer build phase of the test which showcases type of handshake configured as HARDWARE.
		tb_uart_svt_ovm_intermediate_sys/tests/ts.softwr_hndshk_test.sv	Refer build phase of the test which showcases type of handshake configured as SOFTWARE.
		tb_uart_svt_ovm_intermediate_sys/tests/ts.fractional_baud_divisor_hardwr_hndshk_test.sv	Refer build phase of the test which showcases type of handshake configured as HARDWARE.
		tb_uart_svt_ovm_intermediate_sys/tests/ts.fractional_baud_divisor_softwr_hndshk_test.sv	Refer build phase of the test which showcases type of handshake configured as SOFTWARE.
To enable/disable RTS-CTS handshake	enable_rts_cts_handshake	tb_uart_svt_ovm_intermediate_sys/tests/ts.partial_hardwr_hndshk_disable_rts_cts_test.sv	Refer build phase of the test which showcases disabling of RTS-CTS handshake by setting the configuration enable_rts_cts_handshake to 0.
To enable/disable DTR-DSR handshake	enable_dtr_dsr_handshake	tb_uart_svt_ovm_intermediate_sys/tests/ts.partial_hardwr_hndshk_disable_dtr_dsr_test.sv	Refer build phase of the test which showcases disabling of DTR-DSR handshake by setting the configuration enable_dtr_dsr_handshake to 0.
To configure different XON and XOFF data pattern values in case of software handshaking	data_pattern_xon, data_pattern_xoff	tb_uart_svt_ovm_intermediate_sys/tests/ts.softwr_hndshk_test.sv	Refer build phase of the test which showcases XON and XOFF data pattern configured as 19 & 23 respectively.
To set integeral baud divisor value to generate different baud clocks	baud_divisor	tb_uart_svt_ovm_intermediate_sys/tests/ts.baud_divisor_test.sv	Refer build phase of the test which showcases baud divisor configured as 5.

Scenario	Configuration (svt_uart_ configuration) Class attributes	Reference Tests	Remarks
To set integeral as well as fractional baud divisor values to generate different baud clocks in case of hardware handshaking	handshake_type, baud_divisor, enable_fractional_brd, fractional_divisor, fractional_divisor_period, fractional_mult_median, enable_drive_baudout_pin	tb_uart_svt_ovm_intermediate_sys/tests/ts.fractional_baud_divisor_hardwr_hndshk_test.sv	Refer build phase of the test which showcases handshake type configured as HARDWARE, Integeral baud divisor configured as 3. Fractional baud divisor is enabled by setting enable_fractional_brd to 1, fractional baud divisor configured as 76, fractional divisor period configured as 16 and fractional MULT median configured as 70 along with enabling the baud out pin.
		tb_uart_svt_ovm_intermediate_sys/tests/ts.fractional_baud_divisor_softwr_hndshk_test.sv	Refer build phase of the test which showcases handshake type configured as SOFTWARE, configure XON data pattern as23, configure XOFF data pattern as27, configure integeral baud divisor as 1. Fractional baud divisor is enabled by setting enable_fractional_brd to 1, fractional baud divisor configured as 87, fractional divisor period configured as 128 and fractional MULT median configured as 50 along with enabling the baud out pin. Refer the sequence 'svt_uart_dce_soft_handshake_send_xoff_xon_sequence' which showcases sending of XON and XOFF data pattern on high priority by setting send_xoff_data_pattern and send_xon_data_pattern to 1.
To configure receiver buffer size	receiver_buffer_size	tb_uart_svt_ovm_intermediate_sys/tests/ts.softwr_hndshk_test.sv	Refer build phase of the test which showcases the configured receiver buffer size to 16.
		tb_uart_svt_ovm_intermediate_sys/tests/ts.fractional_baud_divisor_softwr_hndshk_test.sv	Refer build phase of the test which showcases the configured receiver buffer size to 20.

Scenario	Configuration (svt_uart_ configuration) Class attributes	Reference Tests	Remarks
To enable/disable functional and toggel coverage along with enabling/ disabling of xml generation for Protocol Analyzer feature coverage_ enable	toggle_coverage_e nable enable_xml_gen	tb_uart_svt_ovm_intermediate_sys/te sts/ts.pa_and_coverage_test.sv	Refer build phase of the test which showcase enabling of functional and toggel coverage by setting the configuration coverage_enable and toggle_coverage_enable to 1 and enable_xml_gen to 1.
To showcase the use of a single sequence from the built-in sequence collection for DCE and DTE	handshake_type	tb_uart_svt_ovm_intermediate_sys/te sts/ts.sequence_collection_test.sv	Refer sequence collection files 'svt_uart_dce_transaction_sequenc e_collection.sv' and 'svt_uart_dte_transaction_sequenc e_collection.sv' for usage of different sequences.
To showcase the use of few complex level and primitive level sequences from the built-in sequence collection for DCE and DTE	handshake_type	tb_uart_svt_ovm_intermediate_sys/te sts/ts.dce_dte_test.sv	Refer sequence collection files 'svt_uart_dce_transaction_sequenc e_collection.sv' and 'svt_uart_dte_transaction_sequenc e_collection.sv' for usage of different sequences.
To enable/disable TX-RX Handshake	enable_tx_rx_hand shake	tb_uart_svt_ovm_disable_txrx_hands hake_intermediate_sys/tests/ts.full_ha rdwr_hndshk_disable_tx_rx_test.sv	Refer build phase of the test which showcase disabling of TX-RX handshake by setting the configuration enable_tx_rx_handshake to 0.
To generate invalid parity functionality using callback mechanism	N.A	tb_uart_svt_ovm_intermediate_sys/te sts/ts.parity_error_insertion_test.sv	Refer build phase of the test and refer file <intermediate_example/env>/cust_ svt_uart_txrx_parity_callback.sv

Table 6-2 provides the scenario reference guide for UART OVM Transaction class attributes:

Table 6-2 OVM Scenario Reference Guide (Transaction Class)

Scenario	Transaction (svt_uart_ transaction) Class attributes	Reference Tests	Remarks
To generate inter cycle delay between two consecutive packets	inter_cycle_delay	tb_uart_svt_ovm_basic_sys/tests/ts.directed_test.sv	Refer the body() task of uart_directed_sequence present inside file tb_uart_svt_ovm_basic_sys/env/uart_directed_sequence.sv. Inter_cycle_delay is set as 100. This delay will appear as multiples of 16. That is, if set to 1-16, the delay will be 0 similarly 17-32 --> 1, 33-48 -->2, ...so on 145-160 --> 9 etc.
To transmit user specified number of packets	packet_count	tb_uart_svt_ovm_basic_sys/tests/ts.directed_test.sv	Refer the body() task of uart_directed_sequence present inside file tb_uart_svt_ovm_basic_sys/env/uart_directed_sequence.sv. Here, packet_count is set as 5.
To enable/disable generation of packets through VIP	enable_packet_ generation	tb_uart_svt_ovm_intermediate_sys /tests/ts.softwr_hndshk_test.sv	Refer the body() task of sequence svt_uart_dce_soft_handshake_send_ xoff_xon_sequence which is part of svt_uart_dce_transaction_sequence_ collection.sv. Here enable_packet_generation is set to 0 to disable the packets generation.
To transamit user specified receiver buffer flush delay	buffer_flush_delay	tb_uart_svt_ovm_intermediate_sys /tests/ts.dce_dte_test.sv	Refer the body() task of sequence uart_dce_dte_virtual_sequence present inside file tb_uart_svt_ovm_basic_sys/env/uart_ dce_dte_virtual_sequence.sv. Here buffer_flush_delay is set to 200.
To Send the XON and XOFF data pattern on the bus on high-priority in case of the software handshaking	send_xon_data_ pattern send_xoff_data_ pattern	tb_uart_svt_ovm_intermediate_sys /tests/ts.softwr_hndshk_test.sv	Refer the body() task of sequence svt_uart_dce_soft_handshake_send_ xoff_xon_sequence which is part of svt_uart_dce_transaction_sequence_ collection.sv. Here send_xon_data_pattern and send_xoff_data_pattern are set to 1.

Scenario	Transaction (svt_uart_ transaction) Class attributes	Reference Tests	Remarks
To generate break condition from the VIP	break_cond	N.A	Refer the body() tasks of sequences svt_uart_dte_break_cond_sequence and svt_uart_dce_break_cond_sequence which is part of svt_uart_dte_transaction_sequence_collection.sv and svt_uart_dce_transaction_sequence_collection.sv respectively.
To read the received data back from the VIP through subscriber's API calls	packet_count payload	tb_uart_svt_ovm_intermediate_sys /tests/ts.subscriber_test.sv	Refer the body() task of uart_subscriber_sequence present inside file tb_uart_svt_ovm_intermdiate_sys/env /uart_subscriber_sequence.sv. Here, the packet_count is set as 10 and user defined data is sent. Refer the post_body() task of uart_subscriber_virtual_sequence present inside file tb_uart_svt_ovm_intermdiate_sys/env /uart_subscriber_virtual_sequence.sv. Here all subscriber's APIs like pop_fifo_char are called.

6.6 Example of Full Hardware Handshaking

This section covers the examples for the following:

- Configuring the handshaking type to the hardware flow control
- Providing the packet count to five packets
- Configuring the type of parity to the odd parity
- Configuring the data width to 5
- Configuring the stop bit to 1

Note To apply a constraint, you may have to override the reasonable constraints, such as `reasonable_delay_in_rts_assertion` and `reasonable_packet_count`, etc. in the `cust_svt_uart_transaction` class.

Figures 6-15 shows the hardware handshaking sequence with the packet count to 5.

Figure 6-15 Hardware Handshaking Sequence With the Packet Count to 5

```
class uart_full_handshaking_seq extends ovm_sequence #(svt_uart_transaction);

  svt_uart_transaction tx_xact;

  /** OVM object utility macro */
  `ovm_object_utils(uart_full_handshaking_seq)
  ............................
  ............................
  ............................
  ............................

  virtual task body();
    `ovm_info("body", "Entered ...", OVM_LOW)


       ............................
       ............................

    `ovm_create(tx_xact)
      start_item(tx_xact);
      if (tx_xact.randomize() with {
        // set number of packets to be sent
        tx_xact.packet_count              == 5   ;
      }); begin  end else begin  `ovm_error("body", "Randomization Failed") end

      finish_item(tx_xact);

  endtask : body

endclass : uart_full_handshaking_seq
```

Figures 6-16 shows the test configuring the hardware handshaking with the odd parity, data width to 5-bit, and 1-bit stop bit.

Figure 6-16 Hardware Handshaking With the Odd Parity, Data Width to 5-Bit, and 1-Bit Stop Bit

```
class uart_base_test extends ovm_test;

  /** Instance of the environment */
  uart_basic_env env;

  /** Configuration Instance for DTE agent */
  cust_svt_uart_agent_configuration dte_cfg;

  /** Configuration Instance for DCE agent */
  cust_svt_uart_agent_configuration dce_cfg;

  ....................................

  virtual function void build();
    `ovm_info("build", "Entered ...", OVM_LOW)
    ...................................

    /** Set the configuration values for DTE agent */
    dte_cfg.is_active                      = 1;

    /** Set the configuration values for DCE agent */
    dce_cfg.is_active                      = 1;

    /** Set the configuration values for Handshake Type of DTE agent */
    dte_cfg.handshake_type                 = svt_uart_configuation::HARDWARE;

    /** Set the configuration values for Handshake Type of DCE agent */
    dce_cfg.handshake_type                 = svt_uart_configuation::HARDWARE;

    /** Set the configuration values for Parity Type of DTE agent */
    dte_cfg.parity_type                    = svt_uart_configuation::ODD_PARITY;

    /** Set the configuration values for Parity Type of DCE agent */
    dce_cfg.parity_type                    = svt_uart_configuation::ODD_PARITY;

    /** Set the configuration values for Data Width of DTE agent */
    dte_cfg.data_width                     = svt_uart_configuation::FIVE_BIT;

    /** Set the configuration values for Data Width of DCE agent */
    dce_cfg.data_width                     = svt_uart_configuation::FIVE_BIT;

    /** Set the configuration values for Stop bits of DTE agent */
    dte_cfg.stop_bit                       = svt_uart_configuation::ONE_BIT;

    /** Set the configuration values for Stop bits of DCE agent */
    dce_cfg.stop_bit                       = svt_uart_configuation::ONE_BIT;

    .....................................

    `ovm_info("build", "Exited ...", OVM_LOW)
  endfunction : build

  ....................................

endclass : uart_base_test
```

6.7 Example of Full Hardware Handshaking With Delay

This section covers the examples for the following:

- ✦ Configuring the handshaking type to the hardware flow control
- ✦ Providing delay in assertion of RTS to 25
- ✦ Providing delay in assertion of CTS to 75
- ✦ Providing the inter-cycle delay to 100
- ✦ Providing the packet count to five packets
- ✦ Configuring the type of parity to the even parity
- ✦ Configuring data width to 8
- ✦ Configuring stop bits to 2

Figures 6-17 shows the hardware handshaking delay sequence with the packet count to 5, delay in RTS to 25, delay in CTS to 75, and Intercycle delay to 100.

Figure 6-17 Hardware Handshaking Delay Sequence

```
class uart_full_handshaking_seq_with_delay extends ovm_sequence #(svt_uart_transaction);

  svt_uart_transaction tx_xact;

  /** OVM object utility macro */
  `ovm_object_utils(uart_full_handshaking_seq_with_delay)
  ............................
  ............................
  ............................
  ............................

  virtual task body();
    `ovm_info("body", "Entered ...", OVM_LOW)


       ............................
       ............................

    `ovm_create(tx_xact)
      start_item(tx_xact);
      if (tx_xact.randomize() with {
        // set number of packets to be sent
        tx_xact.packet_count             == 5;
        tx_xact.delay_rts                == 25;
        tx_xact.delay_cts                == 75;
        tx_xact.inter_cycle_delay        == 100;
      }); begin  end else begin  `ovm_error("body", "Randomization Failed") end

      finish_item(tx_xact);

  endtask : body

endclass : uart_full_handshaking_seq_with_delay
```

Figures 6-18 shows the test configuring the hardware handshaking with the even parity, data width to 8-bit, and 2-bit stop bits.

Figure 6-18 Hardware Handshaking With the Even Parity

```
class uart_base_test extends ovm_test;

  /** Instance of the environment */
  uart_basic_env env;

  /** Configuration Instance for DTE agent */
  cust_svt_uart_agent_configuration dte_cfg;

  /** Configuration Instance for DCE agent */
  cust_svt_uart_agent_configuration dce_cfg;

  ....................................

  virtual function void build();
    `ovm_info("build", "Entered ...", OVM_LOW)
    ....................................

    /** Set the configuration values for DTE agent */
    dte_cfg.is_active                     = 1;

    /** Set the configuration values for DCE agent */
    dce_cfg.is_active                     = 1;

    /** Set the configuration values for Handshake Type of DTE agent */
    dte_cfg.handshake_type                = svt_uart_configuation::HARDWARE;

    /** Set the configuration values for Handshake Type of DCE agent */
    dce_cfg.handshake_type                = svt_uart_configuation::HARDWARE;

    /** Set the configuration values for Parity Type of DTE agent */
    dte_cfg.parity_type                   = svt_uart_configuation::EVEN_PARITY;

    /** Set the configuration values for Parity Type of DCE agent */
    dce_cfg.parity_type                   = svt_uart_configuation::EVEN_PARITY;

    /** Set the configuration values for Data Width of DTE agent */
    dte_cfg.data_width                    = svt_uart_configuation::EIGHT_BIT;

    /** Set the configuration values for Data Width of DCE agent */
    dce_cfg.data_width                    = svt_uart_configuation::EIGHT_BIT;

    /** Set the configuration values for Stop bits of DTE agent */
    dte_cfg.stop_bit                      = svt_uart_configuation::TWO_BIT;

    /** Set the configuration values for Stop bits of DCE agent */
    dce_cfg.stop_bit                      = svt_uart_configuation::TWO_BIT;

    .....................................

    `ovm_info("build", "Exited ...", OVM_LOW)
  endfunction : build

  ....................................

endclass : uart_base_test
```

6.8 Example of Software Handshaking

This section covers the examples for the following:

✦ Configuring the handshaking type to the software flow control

✦ Controllability of transmitting the XON and XOFF data pattern on the bus on a high priority

✦ Configuring the data pattern XON or XOFF value

✦ Providing the packet count to eight packets

Figures 6-19 shows the software handshaking sequence with the packet count to 8 and XON or XOFF data pattern control.

Figure 6-19 Software Handshaking Sequence

```
class uart_soft_handshaking_seq extends ovm_sequence #(svt_uart_transaction);

  svt_uart_transaction tx_xact;

  /** OVM object utility macro */
  `ovm_object_utils(uart_soft_handshaking_seq)
  ............................
  ............................
  ............................
  ............................

  virtual task body();
    `ovm_info("body", "Entered ...", OVM_LOW)


       ............................
       ............................

    `ovm_create(tx_xact)
      start_item(tx_xact);
      if (tx_xact.randomize() with {
        // set number of packets to be sent
        tx_xact.packet_count             == 8;
      }); begin  end else begin  `ovm_error("body", "Randomization Failed") end
      tx_xact.send_xoff_data_pattern     = 1;
      tx_xact.send_xon_data_pattern      = 1;

      finish_item(tx_xact);

  endtask : body

endclass : uart_soft_handshaking_seq
```

Figures 6-20 shows the test configuring the software handshaking with the data pattern for XON to 21 and the data pattern to XOFF to 23.

Figure 6-20 Software Handshaking With the Data Pattern for XON and XOFF

```
class uart_base_test extends ovm_test;

  /** Instance of the environment */
  uart_basic_env env;

  /** Configuration Instance for DTE agent */
  cust_svt_uart_agent_configuration dte_cfg;

  /** Configuration Instance for DCE agent */
  cust_svt_uart_agent_configuration dce_cfg;

  ....................................

  virtual function void build();
    `ovm_info("build", "Entered ...", OVM_LOW)
    ....................................

    /** Set the configuration values for DTE agent */
    dte_cfg.is_active                    = 1;

    /** Set the configuration values for DCE agent */
    dce_cfg.is_active                    = 1;

    /** Set the configuration values for Handshake Type of DTE agent */
    dte_cfg.handshake_type               = svt_uart_configuation::SOFTWARE;

    /** Set the configuration values for Handshake Type of DCE agent */
    dce_cfg.handshake_type               = svt_uart_configuation::SOFTWARE;

    /** Set the configuration values of data pattern for XON packet of DTE agent */
    dte_cfg.data_pattern_xon             = 9'd21;

    /** Set the configuration values of data pattern for XON packet of DCE agent */
    dce_cfg.data_pattern_xon             = 9'd21;

    /** Set the configuration values of data pattern for XOFF packet of DTE agent */
    dte_cfg.data_pattern_xoff            = 9'd23;

    /** Set the configuration values of data pattern for XOFF packet of DCE agent */
    dce_cfg.data_pattern_xoff            = 9'd23;

    .....................................

    `ovm_info("build", "Exited ...", OVM_LOW)
  endfunction : build

  ....................................

endclass : uart_base_test
```

6.9 Example of Software Handshaking With Delay

This section covers the examples for the following:

- ✦ Configuring the handshaking type to the software flow control
- ✦ Configurable delay to flush the receiver buffer (FIFO) when buffer gets full
- ✦ Controllability of transmitting XON and XOFF data pattern on the bus on high priority
- ✦ Configuring the data pattern XON or XOFF value
- ✦ Providing the packet count to 5 from a sequence

Figures 6-21 shows the software handshaking delay sequence with the packet count to 5, buffer flush delay to 200, and XON or XOFF data pattern control.

Figure 6-21 Software Handshaking With Delay

```
class uart_soft_handshaking_seq_with_delay extends ovm_sequence #(svt_uart_transaction);

  svt_uart_transaction tx_xact;

  /** OVM object utility macro */
  `ovm_object_utils(uart_soft_handshaking_seq_with_delay)
  ............................
  ............................
  ............................
  ............................

  virtual task body();
    `ovm_info("body", "Entered ...", OVM_LOW)


       ............................
       ............................


    `ovm_create(tx_xact)
      start_item(tx_xact);
      if (tx_xact.randomize() with {
        // set number of packets to be sent
        tx_xact.packet_count             == 5;
        tx_xact.buffer_flush_delay       == 200;
      }); begin  end else begin  `ovm_error("body", "Randomization Failed") end
      tx_xact.send_xoff_data_pattern     = 1;
      tx_xact.send_xon_data_pattern      = 1;

      finish_item(tx_xact);

  endtask : body

endclass : uart_soft_handshaking_seq_with_delay
```

Figures 6-22 shows the test configuring the software handshaking with the data pattern for XON and XOFF.

Figure 6-22 Software Handshaking With Delay and With the Data Pattern for XON and XOFF

```
class uart_base_test extends ovm_test;

  /** Instance of the environment */
  uart_basic_env env;

  /** Configuration Instance for DTE agent */
  cust_svt_uart_agent_configuration dte_cfg;

  /** Configuration Instance for DCE agent */
  cust_svt_uart_agent_configuration dce_cfg;

  ....................................

  virtual function void build();
    `ovm_info("build", "Entered ...", OVM_LOW)
    ...................................

    /** Set the configuration values for DTE agent */
    dte_cfg.is_active                   = 1;

    /** Set the configuration values for DCE agent */
    dce_cfg.is_active                   = 1;

    /** Set the configuration values for Handshake Type of DTE agent */
    dte_cfg.handshake_type              = svt_uart_configuation::SOFTWARE;

    /** Set the configuration values for Handshake Type of DCE agent */
    dce_cfg.handshake_type              = svt_uart_configuation::SOFTWARE;

    /** Set the configuration values of data pattern for XON packet of DTE agent */
    dte_cfg.data_pattern_xon            = 9'd21;

    /** Set the configuration values of data pattern for XON packet of DCE agent */
    dce_cfg.data_pattern_xon            = 9'd21;

    /** Set the configuration values of data pattern for XOFF packet of DTE agent */
    dte_cfg.data_pattern_xoff           = 9'd23;

    /** Set the configuration values of data pattern for XOFF packet of DCE agent */
    dce_cfg.data_pattern_xoff           = 9'd23;

    .....................................

    `ovm_info("build", "Exited ...", OVM_LOW)
  endfunction : build

  ....................................

endclass : uart_base_test
```

6.10 Application Note for UART RS-485 Mode

❖ For using UART VIP in rs485 mode:
  ✦ Instantiate DTE Passive VIP agent.
  ✦ Define `SVT_UART_GPIO` macro.

  - ✦ Enable `enable_rs485` system configuration.
- ❖ Following are the set of VIP interface pins to which DUT (RS 485) pins should be connected:
  - ✦ `gpio[0]` – This pin is used for connecting `de` (driver enable) signal of DUT.
  - ✦ `gpio[1]` – This pin is used for connecting `re` (receiver enable) signal of DUT.
  - ✦ `gpio[2]` – This pin is used for connecting `rs485_en` signal of DUT.

Table 6-3 the set of `svt_uart_configuration` variables to configure VIP as per DUT RS 485 mode registers.

Table 6-3 Configuration Variables for RS 485 Mode

VIP Configurations		
Name	**Default Value**	**Description**
`enable_rs485`	0 (Disabled)	Determines whether to enable rs485 associated checker rules or not.
`de_polarity`	1 (active high)	Determines whether de signal is active high (1) or active low (0).
`re_polarity`	1 (active high)	Determines whether re signal is active high (1) or active low (0).
`de_assertion_delay`	5 (8'h5)	The assertion time is the time between the activation of the DE signal and the beginning of the START bit. The value represented is in terms of serial clock cycles.
`de_deassertion_delay`	5 (8'h5)	The de-assertion time is the time between the end of the last stop bit, in a transmitted character, and the de-activation of the DE signal. The value represented is in terms of serial clock cycles.
`re_to_de_TAT`	10 (16'ha)	If any transmit transfer is ongoing, then the signal waits until transmit has finished and after the turnaround time counter ('de to re') has elapsed.
`de_to_re_TAT`	10 (16'ha)	If any receive transfer is ongoing, then the signal waits until receive has finished, and after the turnaround time counter ('re to de') has elapsed.
`uart_rs485_transfer_mode`	RS485_FULL_DUPLEX	`RS485_FULL_DUPLEX`: The full duplex mode supports both transmit and receive transfers simultaneously. You can choose when to transmit or when to receive. Both 're' and 'de' can be simultaneously asserted or de-asserted at any time. No take care of turnaround timing. `RS485_HALF_DUPLEX`: The half duplex mode supports either transmit or receive transfers at a time but not both simultaneously. In this mode, it must be insured that a proper turnaround time is maintained while switching from 're' to 'de' or from 'de' to 're'.

Table 6-4 lists the set of checker rules..

Table 6-4 Checker Rules

Protocol Check Instance name	**Description**
`svt_err_rs485_re_to_de_turnaround`	RS485: In half duplex mode re to de tat time must be followed.

Table 6-4 Checker Rules

Protocol Check Instance name	Description
`svt_err_rs485_de_to_re_turnaround`	RS485: In half duplex mode de to re tat time must be followed.
`svt_err_rs485_de_deassertion`	RS485: If stop bit on sout is received then de must be de-asserted only after de-deassertion delay.
`svt_err_rs485_de_assertion`	RS485: If de is asserted start can come on sout only after de-assertion time delay.
`svt_err_rs485_no_derive_sin_if_re_off`	RS485: No derivation on sin if re is not asserted.
`svt_err_rs485_no_derive_sout_if_de_off`	RS485: No derivation on sout if de is not asserted.
`svt_err_rs485_no_derive_sout_sin_if_de_re_off`	RS485: No derivation on sout or sin if de /re is not asserted.
`svt_err_rs485_de_re_both_not_high_half_duplex_chk`	RS485: In rs485 half duplex mode de and re cannot be high at the same time.
`svt_err_rs485_no_re_de_if_rs485_pin_off`	RS485: No derivation on re de if rs485 pin is off.
`svt_err_rs485_de_re_rs485_pin_cannot_be_x_or_z`	RS485: Any of the three gpio pins for de, re, rs485 mode cannot be x 0r z.

Note

- RS485 mode feature must be used with DTE passive agent only.
- If enable_rs485 configuration is ON and gpio[2] (rs485_en pin of DUT) is LOW, then RS232 (UART) mode checker rules will be on.
- If enable_rs485 configuration is ON and gpio[2] (rs485_en pin of DUT) is LOW, then both RS485 mode and RS232 (UART) mode checker rules will be ON. This will be addressed in the upcoming release.

7 Troubleshooting

This chapter provides some useful information that can help you to troubleshoot common issues that you may encounter while using UART VIP.

The chapter consists of the following sections:

7.1 Enabling Traffic Logs

You can enable or disable traffic logs using the `enable_traffic_log` variable, which is defined in the `svt_uart_agent_configuration` class. The default value of the variable is 1, which enables traffic logs, by default.

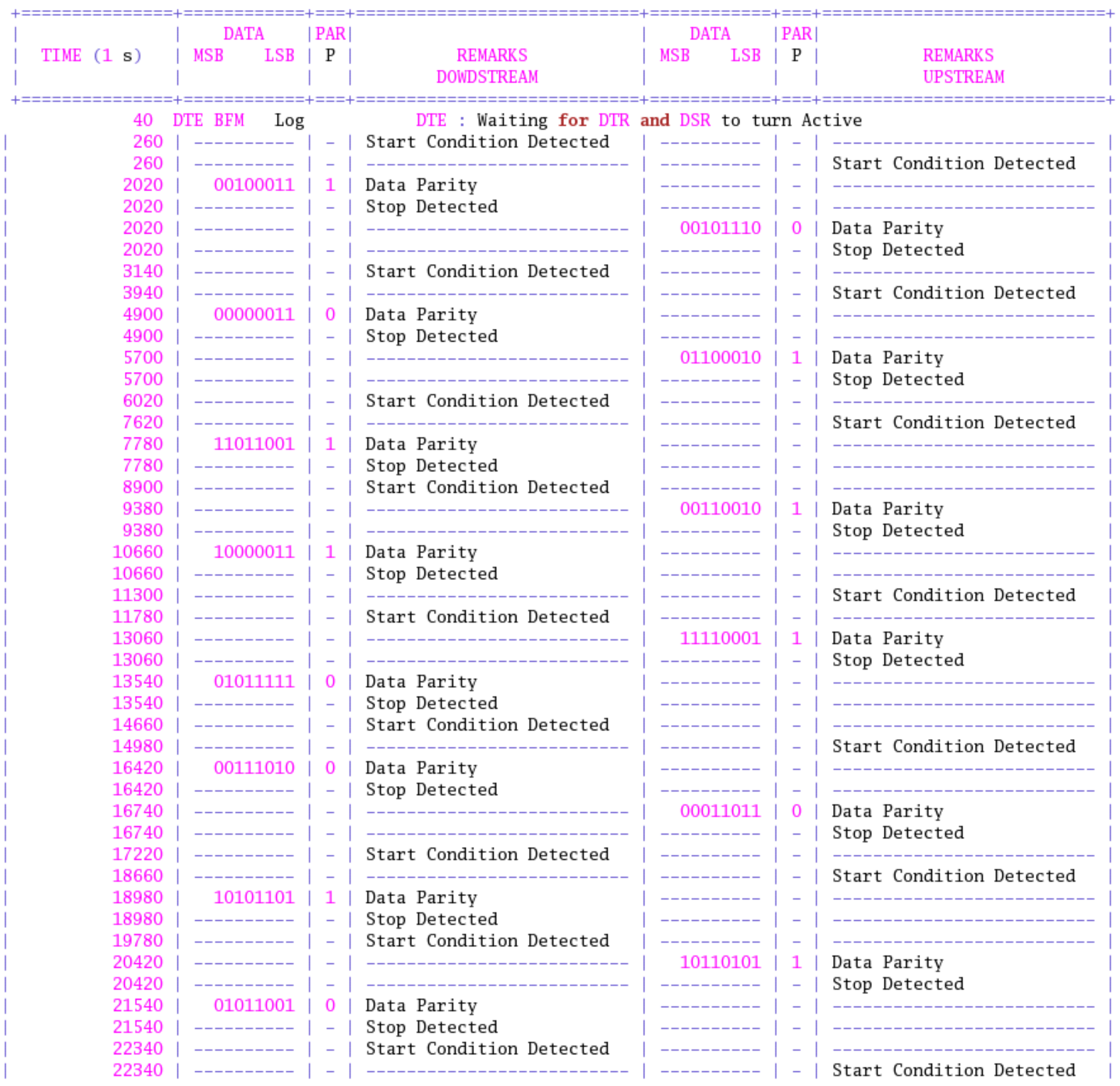
The following code setting illustrate how you can enable the traffic log assuming the `dte_cfg` handle is of the `cust_svt_uart_agent_configuration` class:

```
dte_cfg.enable_traffic_log = 1;
```

To disable traffic logs, set the value of the `enable_traffic_log` variable to 1 in the `cust_svt_uart_agent_configuration` class, which is used in your VIP agent. When the environment is simulated, traffic log files are generated according to the configuration.

Figures 7-1 shows the traffic log generated by the above code setting.

Figure 7-1 .Traffic Log



```
+==============+===========+===+==========================+===========+===+==========================+
|              |   DATA    |PAR|                          |   DATA    |PAR|                          |
|  TIME (1 s)  | MSB   LSB | P |         REMARKS          | MSB   LSB | P |         REMARKS          |
|              |           |   |        DOWDSTREAM        |           |   |         UPSTREAM         |
+==============+===========+===+==========================+===========+===+==========================+
          40  DTE BFM   Log         DTE : Waiting for DTR and DSR to turn Active
|          260 | ---------- | - | Start Condition Detected   | ---------- | - | -------------------------- |
|          260 | ---------- | - | -------------------------- | ---------- | - | Start Condition Detected   |
|         2020 |   00100011 | 1 | Data Parity                | ---------- | - | -------------------------- |
|         2020 | ---------- | - | Stop Detected              | ---------- | - | -------------------------- |
|         2020 | ---------- | - | -------------------------- |   00101110 | 0 | Data Parity                |
|         2020 | ---------- | - | -------------------------- | ---------- | - | Stop Detected              |
|         3140 | ---------- | - | Start Condition Detected   | ---------- | - | -------------------------- |
|         3940 | ---------- | - | -------------------------- | ---------- | - | Start Condition Detected   |
|         4900 |   00000011 | 0 | Data Parity                | ---------- | - | -------------------------- |
|         4900 | ---------- | - | Stop Detected              | ---------- | - | -------------------------- |
|         5700 | ---------- | - | -------------------------- |   01100010 | 1 | Data Parity                |
|         5700 | ---------- | - | -------------------------- | ---------- | - | Stop Detected              |
|         6020 | ---------- | - | Start Condition Detected   | ---------- | - | -------------------------- |
|         7620 | ---------- | - | -------------------------- | ---------- | - | Start Condition Detected   |
|         7780 |   11011001 | 1 | Data Parity                | ---------- | - | -------------------------- |
|         7780 | ---------- | - | Stop Detected              | ---------- | - | -------------------------- |
|         8900 | ---------- | - | Start Condition Detected   | ---------- | - | -------------------------- |
|         9380 | ---------- | - | -------------------------- |   00110010 | 1 | Data Parity                |
|         9380 | ---------- | - | -------------------------- | ---------- | - | Stop Detected              |
|        10660 |   10000011 | 1 | Data Parity                | ---------- | - | -------------------------- |
|        10660 | ---------- | - | Stop Detected              | ---------- | - | -------------------------- |
|        11300 | ---------- | - | -------------------------- | ---------- | - | Start Condition Detected   |
|        11780 | ---------- | - | Start Condition Detected   | ---------- | - | -------------------------- |
|        13060 | ---------- | - | -------------------------- |   11110001 | 1 | Data Parity                |
|        13060 | ---------- | - | -------------------------- | ---------- | - | Stop Detected              |
|        13540 |   01011111 | 0 | Data Parity                | ---------- | - | -------------------------- |
|        13540 | ---------- | - | Stop Detected              | ---------- | - | -------------------------- |
|        14660 | ---------- | - | Start Condition Detected   | ---------- | - | -------------------------- |
|        14980 | ---------- | - | -------------------------- | ---------- | - | Start Condition Detected   |
|        16420 |   00111010 | 0 | Data Parity                | ---------- | - | -------------------------- |
|        16420 | ---------- | - | Stop Detected              | ---------- | - | -------------------------- |
|        16740 | ---------- | - | -------------------------- |   00011011 | 0 | Data Parity                |
|        16740 | ---------- | - | -------------------------- | ---------- | - | Stop Detected              |
|        17220 | ---------- | - | Start Condition Detected   | ---------- | - | -------------------------- |
|        18660 | ---------- | - | -------------------------- | ---------- | - | Start Condition Detected   |
|        18980 |   10101101 | 1 | Data Parity                | ---------- | - | -------------------------- |
|        18980 | ---------- | - | Stop Detected              | ---------- | - | -------------------------- |
|        19780 | ---------- | - | Start Condition Detected   | ---------- | - | -------------------------- |
|        20420 | ---------- | - | -------------------------- |   10110101 | 1 | Data Parity                |
|        20420 | ---------- | - | -------------------------- | ---------- | - | Stop Detected              |
|        21540 |   01011001 | 0 | Data Parity                | ---------- | - | -------------------------- |
|        21540 | ---------- | - | Stop Detected              | ---------- | - | -------------------------- |
|        22340 | ---------- | - | Start Condition Detected   | ---------- | - | -------------------------- |
|        22340 | ---------- | - | -------------------------- | ---------- | - | Start Condition Detected   |
```

7.2 Setting Verbosity Levels

You can set the verbosity levels of the VIP debug either in the testbench or as an option during runtime. This section consists of the following sub-sections:

- ✦ "Setting Verbosity in the Testbench" on page 81
- ✦ "Setting Verbosity During Runtime" on page 81

7.2.1 Setting Verbosity in the Testbench

To set the verbosity level in the testbench, use the OVM-specified log levels in the code. The components extend from the `ovm_report_object` method. You can use the `ovm_report_object` method to change the verbosity for the agent.

Consider the following example:

```
vip_agent.set_report_verbosity_level(<level>);
```

Here, the following verbosity levels define all the possible levels:

- `OVM_NONE`: Prints the report always. The setting of the verbosity level cannot disable it.
- `OVM_LOW`: Prints the report if the configured verbosity is set to `OVM_LOW` or above.
- `OVM_MEDIUM`: Prints the report if the configured verbosity is set to `OVM_MEDIUM` or above.
- `OVM_HIGH`: Prints the report if the configured verbosity is set to `OVM_HIGH` or above.
- `OVM_FULL`: Prints the report if the configured verbosity is set to `OVM_FULL` or above.

7.2.2 Setting Verbosity During Runtime

You can change the verbosity by passing the above verbosity levels with `+OVM_VERBOSITY`. For example, to set the verbosity level as `OVM_MEDIUM`, use the following:

- `+OVM_VERBOSITY=OVM_MEDIUM`

7.3 Protocol Analyzer

You can use the Protocol Analyzer tool to understand protocol-related activities on the bus to figure out any violation as per protocol specifications. For more details, refer Section 3.8.4 “Protocol Analyzer Support”.

A
Reporting Problems

This chapter provides you an overview for working through and reporting SVT 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.

The chapter consists of the following section:

❖ "Initial Customer Information" on page 83

Initial Customer Information

Synopsys Support Center. To report a problem, perform the following steps:

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
   ✧ A description of your simulator and its version
   ✧ A description of operating system and the command-line of the simulator
2. Create a Value Change Dump (VCD) file
3. Generate the simulation log by running the simulation with the `OVM_HIGH` verbosity
4." Generate the traffic log file by setting the `enable_traffic_log` variable of the agent configuration class
5. Generate XML files for Protocol Analyzer. For this, you need to enable the following properties:

   `dte_cfg.enable_xml_gen = 1'b1; //`Enable the XML generation (PA) for DTE

   `dce_cfg.enable_xml_gen = 1'b1; //`Enable the XML generation (PA) for DCE

Providing this information ensures the accurate diagnosis of the problem.

If the requested information is not sufficient to determine the cause of your issue, the Synopsys Support Center may request a simple testbench demonstrating the issue. This is the most effective way to debug issues, but if an example cannot be provided, Synopsys may request additional information that could include regenerating the log file using different settings.

If your testbench initializes the random seed (see 'srandom()' in the VCS guide) in the host simulator, then the seed that can be used to demonstrate the problem must be included in the testbench or provided as information for executing the testbench.