Verification Continuum™

VC Verification IP
I2C
UVM User Guide

Version Q-2020.06, June 2020

SYNOPSYS®

www.synopsys.com

Contents

Preface

About This Guide

This guide contains installation, setup, and usage material for VC VIP for I2C on the Universal Verification Methodology (UVM). This guide is for design or verification engineers who want to verify I2C operations using a UVM testbench written in SystemVerilog. Readers are assumed to be familiar with I2C, Object-Oriented Programming (OOP), SystemVerilog, and UVM techniques.

Guide Organization

The chapters of this user guide are organized as follows:

- Chapter 1, "Introduction", introduces the I2C VIP and its features.
- Chapter 2, "Installation and Setup", describes system requirements and provides instructions on how to install, configure, and begin using the I2C VIP.
- Chapter 3, "General Concepts", introduces the I2C VIP within a UVM environment and describes the data objects and components that comprise the VIP.
- Chapter 4, "Verification Topologies", describes the topologies to verify Master, Slave and interconnect DUT.
- Chapter 5, "Using I2C Verification IP", shows how to install and run a getting started example.
- Chapter 6, "Usage Notes", covers the properties of Master transactions and Slave transactions.
- Chapter 7, "Frequently Asked Questions", covers the frequently asked questions on the I2C protocol.
- Chapter 7, "VIP Tools", describes the VIP tools supported by I2C.
- Chapter 8, "Troubleshooting", provides some useful information that can help you to troubleshoot common issues that you may encounter while using the I2C VIP.
- Appendix A, "Reporting Problems", outlines the process for working through and reporting I2C VIP issues.

Web Resources

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

Customer Support

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

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

   Enter the information according to your environment and your issue.

2. Send an e-mail message to support_center@synopsys.com

   ✦ Include the Product name, Sub Product name, and Product version for which you want to register the problem.

3. Telephone your local support center.

   ✦ North America:

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

   ✦ All other countries:

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

1 Introduction

The Synopsys I2C verification IP supports the verification of SoC designs that include interfaces implementing I2C specification. This document describes the use of the VIP in testbenches that comply with the SystemVerilog UVM. 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 Synopsys I2C VIP class reference HTML documentation, which is installed at the following location:

```
$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/index.html
```

This chapter consists of the following sections:

1.1 Product Overview

I2C UVM VIP is a suite of UVM-based verification components that are compatible for use with SystemVerilog-compliant testbenches. The I2C VIP suite simulates I2C transactions through active agents, as defined by I2C specifications (Version 2.1 and the fast-mode plus-speed feature of Version 3.0).

The VIP provides an I2C System environment that contains Master agents and Slave agents. The Master and Slave agents support all functionality normally associated with active and passive UVM components, including the creation of transactions, checking and reporting the protocol correctness, transaction logging, and functional coverage. After instantiating the system environment, you can configure agents in the active or passive mode to create an environment that verifies I2C features in the DUT.

Master agents and Slave agents can also be used in the stand-alone mode, that is, they can be instantiated in the testbench directly without the system environment.

1.2 Language and Simulator Support

The Synopsys I2C VIP suite supports the following languages and simulators:

- Languages
  - SystemVerilog
- Methodologies and Simulators

  For details about methodologies and simulators, refer to the I2C VIP Release Notes from the following location:

  `$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_release_notes.pdf`.

1.3 Features Supported

This section consists of the following sub-sections:

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

1.3.1 Protocol Features

The I2C VIP currently supports the following protocol functions:

- All the features of Version 2.1 and the fast-mode plus-speed feature of Version 3.0
- Standard-mode, Fast-mode, Fast-mode plus, and High speed
- Multi-Master and Multi-Slave configurations
- Address modes: 7-bit Slave address and 10-bit address
- General call address
- Start byte
- Arbitration
- Repeated start or stop

1.3.2 Verification Features

The I2C VIP currently supports the following verification functions:

- Configurability for the master agent and the slave agent
- Built-in protocol checks
- Exceptions (Error injection)
- Sequence collection
- Built-in functional coverage
- Verification planner
- Source code visibility
- Event-pool support to access events
- The I2C VIP slave configuration: As generic I2C Slave or as I2C EEPROM Slave, NACK response enabling
- The I2C VIP master transaction features are as follows:
    - Configurable delay between consecutive transactions
    - Configurable number of retries in case of not getting an ACK from the slave
    - Enable or disable the retry mechanism in case of NACK from the slave
    - Configurable timing values

- Ease-of-use Features includes the following features:
  - Protocol Analyzer support for debug
  - A basic-level example and an intermediate example to illustrate the use of the VIP in SystemVerilog
  - The HTML documentation for the VIP classes
  - A quickstart guide for a basic-level example

1.3.3 Methodology Features

The I2C VIP currently supports the following methodology functions:

- The VIP is organized as the I2C System environment, which includes the set of master agents and slave agents.
- Analysis ports for connecting master or slave agents to subscribers, such as Scoreboard and coverage collectors.
- Callbacks for the master agent and the slave agent

2
Installation and Setup

This chapter leads you through the installing and setting up the I2C VIP. After completing this checklist, the provided example testbench will be operational and the I2C VIP will be ready to use.

This chapter consists of the following major steps:

2.1 Verifying Hardware Requirements

The I2C 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

The I2C VIP is qualified for use with the certain versions of platforms and simulators. This section lists the software required by the I2C VIP and consists of the following sub-sections:

2.2.1 Platform/OS and Simulator Software

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

For more details, refer I2C Release Notes.

2.2.2 SCL Software

The Synopsys Common Licensing (SCL) software provides the licensing function for the I2C 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**: The I2C VIP documents are available in Acrobat PDF files. Adobe Acrobat Reader is available for free from http://www.adobe.com.
- **HTML Browser:** The I2C 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 the Synopsys I2C 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.

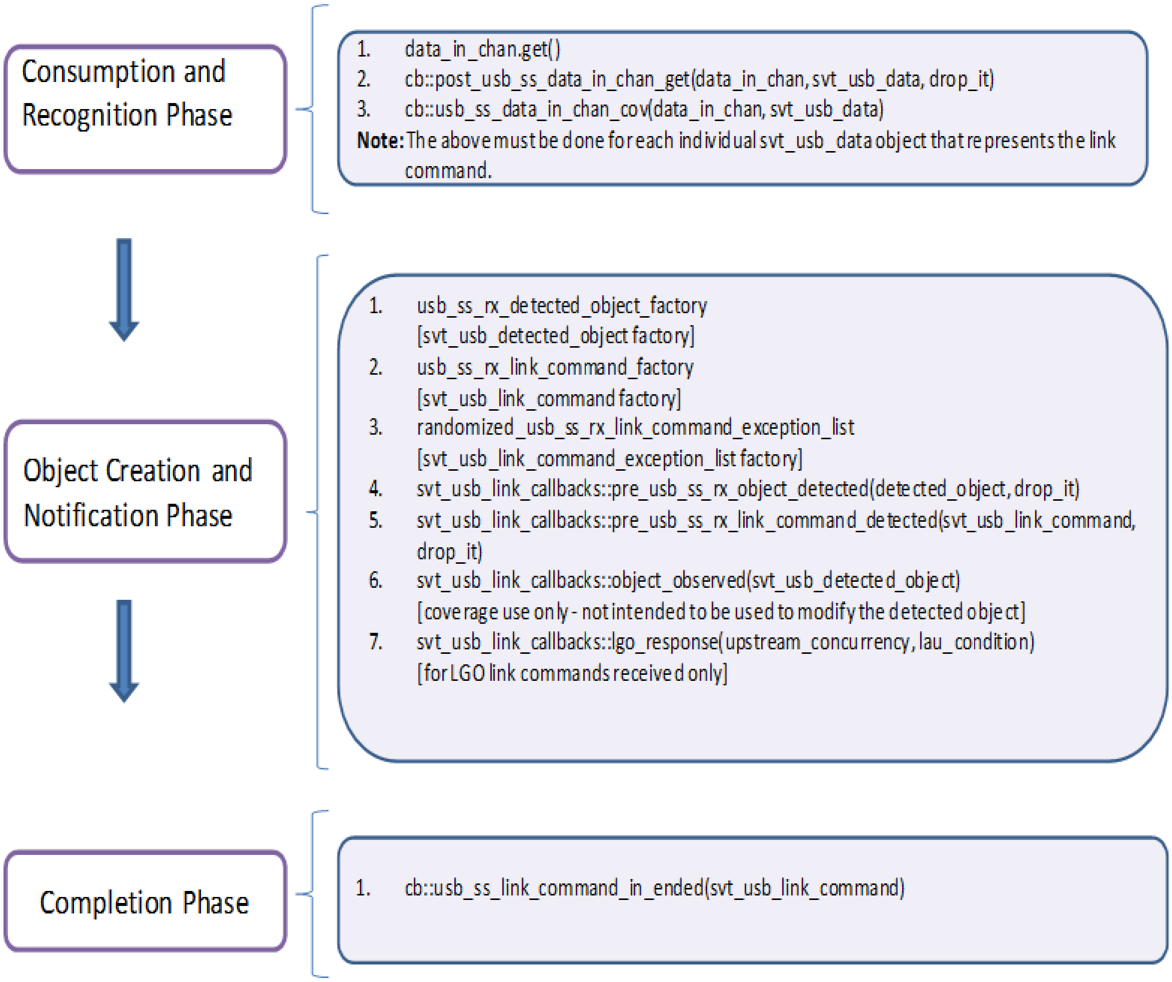
b. Enter your Synopsys SolvNet Username and Password.

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 the I2C VIP is set up for use in a testbench. The `dw_vip_setup` utility is provided as design directory is required for using 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 the I2C VIP from `DESIGNWARE_HOME` to a design directory.

For a complete description of `dw_vip_setup`, see "The dw_vip_setup Utility" on page 24.

The following models are provided with the I2C VIP:

- `i2c_master_agent_svt`
- `i2c_slave_agent_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, %$DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a i2c_master_agent_svt
-svtb
Or
%$DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a(dd) <model1> <model2>
<model3> -svtb
```

```
For example, %$DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a i2c_master_agent_svt
i2c_slave_agent_svt -svtb
Or
%$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:
i2c_master_agent_svt
i2c_slave_agent_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

The I2C 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

The I2C VIP uses a licensing mechanism that is enabled by the following license features:

- VIP-I3C-SVT

  or

- VIP-I2C-SVT

  or

- VIP-PROTOCOL-SVT

  or

- VIP-SOC-LIBRARY-SVT

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.

The Synopsys I2C VIP uses a licensing mechanism that is enabled by the following license feature:

This section consists of the following sub-sections:

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 the `VIP-I2C-SVT` license, set `DW_LICENSE_OVERRIDE` to `VIP-I2C-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, License Polling allows your request to exist until a license is available instead of exiting immediately. To control License Polling, use the DW_WAIT_LICENSE environment variable in the following way:

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

2.6.3 Simulation License Suspension

All the Verification IP products support License Suspension. The simulators that support 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 License Suspension.

2.7 Environment Variable and Path Settings

The following environment variables and path settings are required by the I2C UVM VIP verification models:

- DESIGNWARE_HOME: The absolute path to where the VIP is installed.
- DW_LICENSE_FILE - The absolute path to file that contains the license keys for the VIP product software or the port@host reference to this file.
- SNPSLMD_LICENSE_FILE: The absolute path to 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 please 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
The Synopsys VIP License can be set in either of the 3 license variables mentioned above with the order of precedence for checking the variables being:

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

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

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 an I2C Verification IP into Your Testbench

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

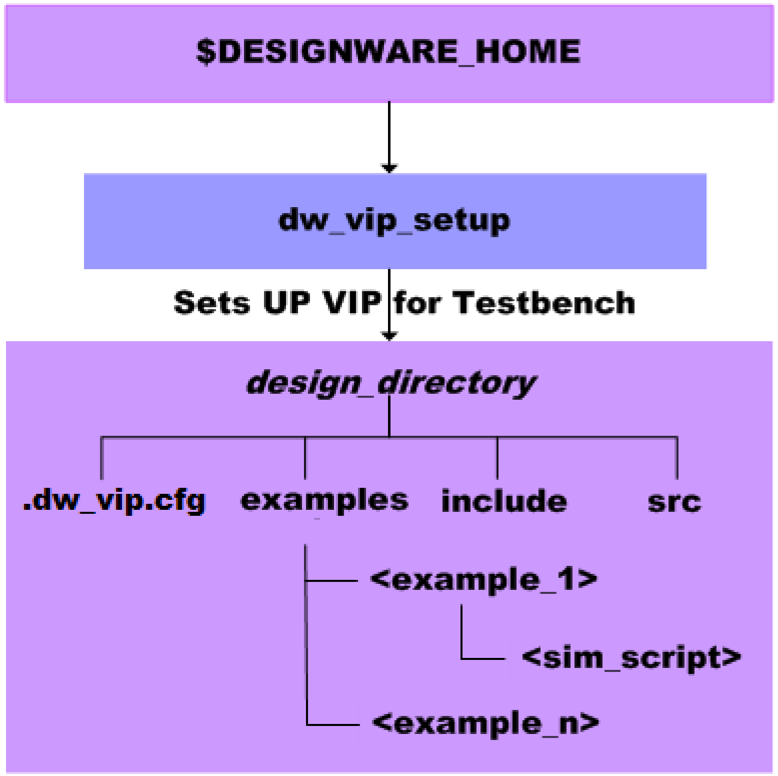
- "Creating a Testbench Design Directory" on page 20
- "The dw_vip_setup Utility" on page 24

2.9.1 Creating a Testbench Design Directory

A design directory contains a version of VIP that is set up and ready to use in a testbench. The `dw_vip_setup` utility is used to create the design directories. For more information on `dw_vip_setup`, see "The dw_vip_setup Utility" on page 24.

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

Figure 2-2Design 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 Running the Example with +incdir+

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

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

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

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

```
cd <testbench_dir>/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/
// To run the example using the generated run script with +incdir+
./run_i2c_svt_uvm_basic_sys -verbose -incdir <testname> vcsvlog
```

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

```
vcs -l ./logs/compile.log -q -Mdir=./output/csrc
+define+DESIGNWARE_INCDIR=<DESIGNWARE_HOME> \
+define+SVT_LOADER_UTIL_ENABLE_DWHOME_INCDIRS
+incdir+<DESIGNWARE_HOME>/vip/svt/i2c_svt/<vip_version>/sverilog/include \
-ntb_opts uvm -full64 -sverilog +define+UVM_DISABLE_AUTO_ITEM_RECORDING
+define+UVM_PACKER_MAX_BYTES=1500000 \
+define+UVM_NO_DEPRECATED -timescale=100ps/100ps \
+lint=none +define+SVT_UVM_TECHNOLOGY +define+SYNOPSYS_SV
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_i2c_svt_uvm_basic_sys/. \
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_i2c_svt_uvm_basic_sys/../
../env \
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_i2c_svt_uvm_basic_sys/../
env \
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_i2c_svt_uvm_basic_sys/
env \
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_i2c_svt_uvm_basic_sys/
dut \
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_i2c_svt_uvm_basic_sys/
hdl_interconnect \
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_i2c_svt_uvm_basic_sys/
lib \
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_i2c_svt_uvm_basic_sys/
tests \
-o ./output/simvcssvlog -f top_files -f hdl_files
```

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

2.9.3 Getting Help on Example Run/make Scripts

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

```
usage: run_i2c_svt_uvm_async_2X2_sys
run_i2c_svt_uvm_basic_program_sys
run_i2c_svt_uvm_basic_sys
run_i2c_svt_uvm_cci_sys
run_i2c_svt_uvm_cfg_validator
run_i2c_svt_uvm_disable_vip
run_i2c_svt_uvm_intermediate_sys
run_i2c_svt_uvm_multi_master_multi_slave
run_i2c_svt_uvm_multiplication_factor_sys
run_smbus_svt_uvm_basic_sys
[-32] [-incdir] [-verbose] [-debug_opts] [-waves] [-clean] [-nobuild] [-
buildonly] [-norun] [-pa] <test name> <simulator>
where   <test name> is one of: the tests included in tests folder of the test
bench
<simulator> is one of: vcsvlog mtivlog ncvlog
-32 forces 32-bit mode on 64-bit machines
-incdir use DESIGNWARE_HOME include files instead of design directory
-verbose enable verbose mode during compilation
-debug_opts  enable debug mode for VIP technologies that support this option
-waves [fsdb|verdi|dump] enables waves dump and optionally opens viewer (VCS only)
-seed run simulation with specified seed value
-clean clean simulator generated files
-nobuild skip simulator compilation
-buildonly exit after simulator build
-norun only echo commands (do not execute)
-pa invoke Verdi after execution
```

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

```
gmake help
Usage: gmake USE_SIMULATOR=<simulator> [VERBOSE=1] [DEBUG=1] [FORCE_32BIT=1]
[WAVES=fsdb|verdi|dump] [NOBUILD=1] [PA=1] [<test name> ...]
Valid simulators are: vcsvlog mtivlog ncvlog
Valid scenarios are: tests included in tests folder of the test bench
```

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

2.9.4 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 24
- "The dw_vip_setup Command" on page 24

2.9.4.1 Setting Environment Variables

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

2.9.4.2 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 2-1Setup 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: • i2c_master_agent_svt • i2c_slave_agent_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: • i2c_master_agent_svt • i2c_slave_agent_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: • i2c_master_agent_svt • i2c_slave_agent_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 UVM. 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 2-1Setup 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	The I2C VIP models are as follows: • i2c_master_agent_svt • i2c_slave_agent_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*	Specifies a file name, which contains a list of suite names to be added, updated, or removed from the design directory. This switch is valid during the following switch operations; for example, -add, -update, or -remove. The -m/odel_list switch displays one model name per line and each model includes a version selector. The default version is the latest. This switch is optional, but the filename argument is required whenever mentioned. Lines in the file starting with the pound symbol (#) are ignored. *model_name* [**-v** *version*] –or– **# Comments**
-s/uite_list <filename>	Specifies a file name, which contains a list of suite names to be added, updated, or removed from the design directory. This switch is valid during the following switch operations; for example, -add, -update, or -remove. The -s/uite_list switch displays one suite name per line and each suite includes a version selector. The default version is the latest. This switch is optional, but the filename argument is required whenever mentioned. Lines in the file starting with the pound symbol (#) are ignored.
-b/ridge	Updates the specified design directory to reference the current DESIGNWARE_HOME installation. All product versions contained in the design directory must also exist in the current DESIGNWARE_HOME installation.

Table 2-1Setup Program Switch Descriptions (Continued)

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

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

3
General Concepts

This chapter introduces the I2C VIP within a UVM 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 UVM

UVM is an object-oriented approach. It provides a blueprint for building testbenches using the constrained random verification. In addition, the resulting structure supports Directed testing. This chapter describes the data objects that support higher structures which comprise the I2C VIP.

This chapter assumes that you are familiar with SystemVerilog and UVM. 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 essential reference guides describing UVM as it is represented in SystemVerilog, see http://www.accellera.org/.

3.2 I2C VIP Components

This section describes the following I2C VIP components:

- ✦ "Master Agent" on page 30
- ✦ "Slave Agent" on page 31
- ✦ "System Environment" on page 31

3.2.1 Master Agent

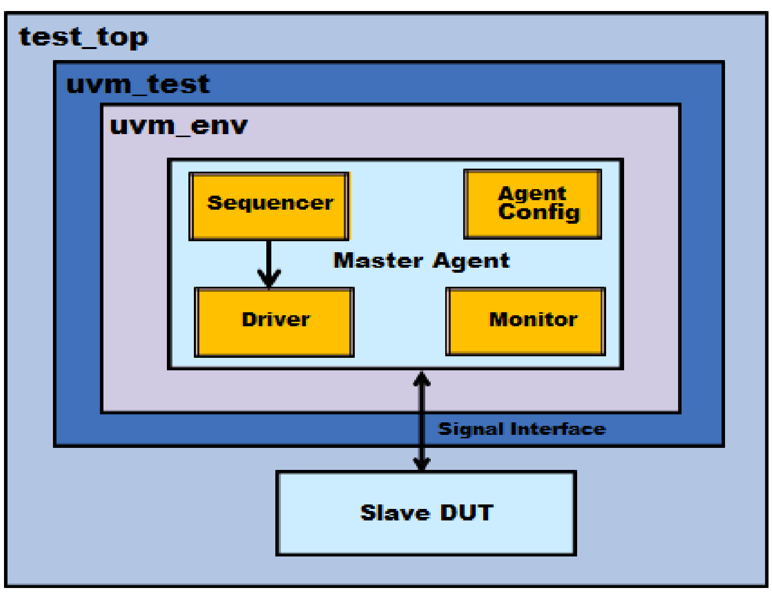
Master Agent encapsulates Master sequencer, Master driver, and Monitor. The agent can be configured to operate either in the active mode or in the passive mode. You can run user-defined sequences on Master sequencer.

Master agent is configured using the agent configuration, which is available in the System configuration.

Within Master agent, Master Driver gets transactions from Master Sequencer. Master Driver then drives the I2C transactions on the I2C port. After an I2C transaction on the bus is complete, the completed sequence item is provided to the analysis port of Monitor for use by the testbench.

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

Figure 3-1Usage With Standalone Master Agent



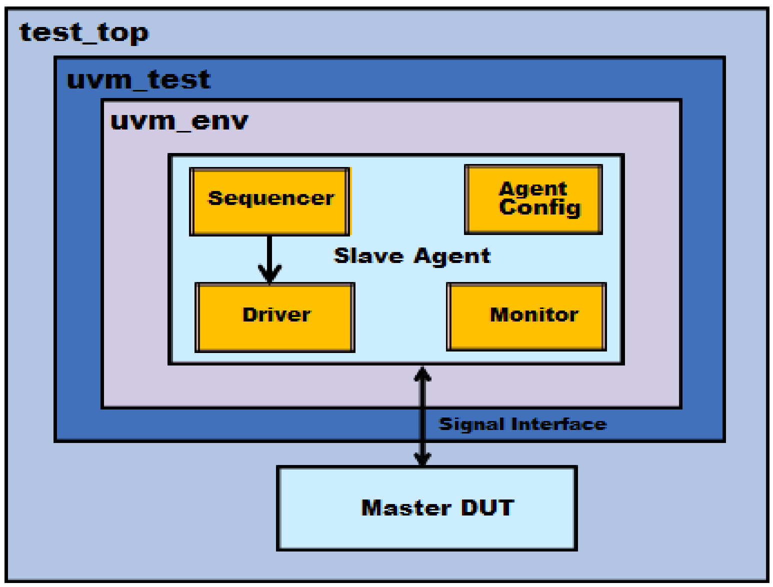
3.2.2 Slave Agent

Slave Agent encapsulates Slave Sequencer, Slave Driver, and Port Monitor. Slave Agent is configured using the agent configuration, which is available in the System configuration.

The response from Slave Agent can be configured by executing slave transactions on Slave Agent. For the details on the types of responses that can be configured by slave transactions, see the Transaction Objects section.

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

Figure 3-2Usage With Standalone Slave Agent



3.2.3 System Environment

The I2C system environment encapsulates master agents, slave agents, the system configuration, and the system sequencer. The system environment can be easily configured to have up to 16 slave agents and 16 master agents. This section discusses the following sub-sections:

- ✦ "System Configuration" on page 31
- ✦ "System Sequencer" on page 32

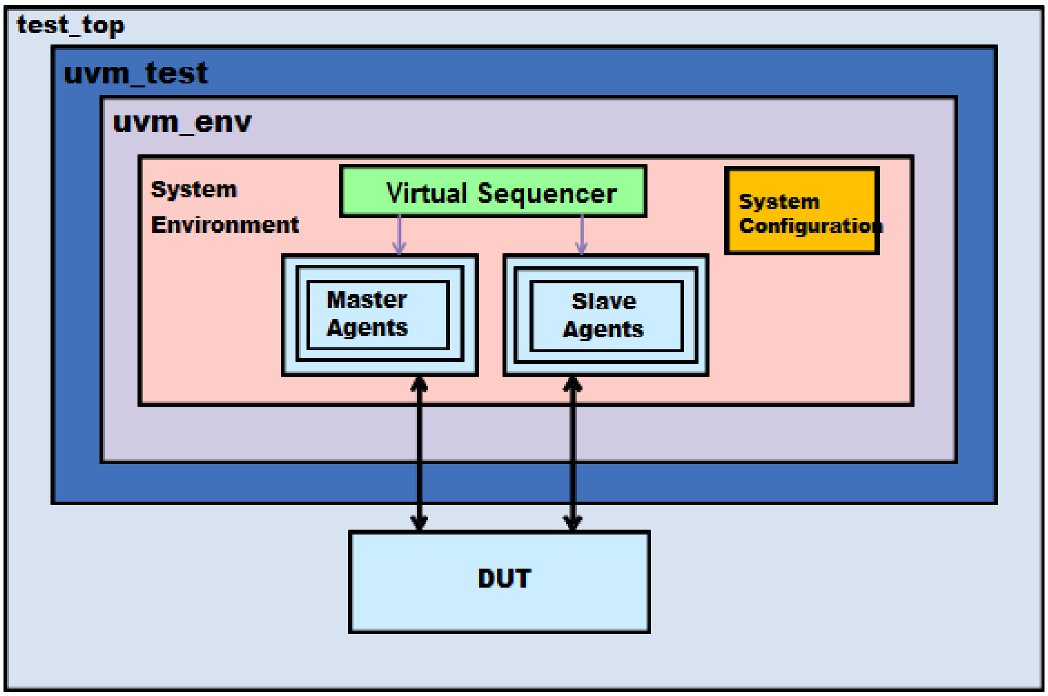
For more details about master agents and slave agents, see the Master Agent section and the Slave Agent section respectively.

3.2.3.1 System Configuration

The number of configured master and slave agents is based on the system configuration. In the build phase, the system environment builds master agents and slave agents. After master and slave agents are built, they are configured by the system environment using the agent configuration information available in the system configuration.

Figures 3-3 shows the usage with the system environment.

Figure 3-3Usage With System Environment



3.2.3.2 System Sequencer

I2C System sequencer is a virtual sequencer with references to each Master and Slave sequencers in the system. The sequencer is created in the build phase of the system environment. The System configuration is provided to System Sequencer. System Sequencer is used to synchronize between the sequencers in Master and Slave agents.

3.3 I2C VIP User Interface

The following sections give an overview of the user interface into the I2C VIP:

✦ "Configuration Objects" on page 33
✦ "Transaction Objects" on page 34
✦ "Analysis Ports" on page 35
✦ "Interfaces and Modports" on page 36
✦ "Events" on page 36

3.3.1 Configuration Objects

Configuration data objects convey the system-level and port-level testbench configuration. The configuration of agents is done in the `build()` phase of an environment or a testcase.

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

The I2C VIP defines the following configuration classes:

- System Configuration (`svt_i2c_system_configuration`)

  The System configuration class contains the configuration information, which is applicable across the entire system. The system-level configuration parameters can be specified through this class. The system configuration needs to be provided to the system environment from the environment or the testcase. The system configuration mainly specifies the following:

  - The number of master and slave agents in the system environment
  - Configurations for master and slave agents
  - Virtual top-level I2C interface
- Agent Configuration (`svt_i2c_agent_configuration`)

The agent configuration class contains the configuration information, which is applicable to individual I2C master or slave agents in the system environment. Some of the important information provided by the agent configuration class is as follows:

- Common configuration attributes for Master and Slave agents are as follows:
  - Active or Passive mode of the Master/Slave port agent
  - Enable or disable protocol checks
  - Enable or disable transaction or toggle coverage
  - The agent configuration contains the virtual interface for the port
  - Timing parameter values
  - Agent id
  - Speed mode
  - Enable or disable Protocol Analyzer (PA)
  - Enable or disable Monitor log
- Master agent-specific configuration attributes are as follows:
  - Master code
- Slave agent-specific configuration attributes are as follows:
  - Enable 7/10-bit addressing
  - Enable response to the general call
  - Slave address
  - Slave type: Generic Slave or EEPROM Slave

The port configuration objects within the system configuration object are created in the constructor of the system configuration.

For details on individual members of configuration classes, refer to the I2C VIP class reference HTML documentation:

```
$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/index.html
```

3.3.2 Transaction Objects

Transaction objects, which extends from the `uvm_sequence_item` base class, define a unit of the I2C protocol information that is passed across the bus. The attributes of transaction objects are public and are accessed directly for setting and getting values. Most transaction attributes can be randomized. Transaction objects represent the desired activity to be simulated on the bus, or the actual bus activity that was monitored.

I2C transaction data objects store data content and the protocol execution information for I2C transactions in terms of the timing details of transactions.

These data objects extend from the `uvm_sequence_item` base class and implement all methods specified by UVM for that class.

I2C transaction data objects are used to perform the following:

- Generate random and directed stimulus
- Report observed transactions
- Collect Functional coverage statistics

Class properties are public and accessed directly to set and read values. Transaction data objects support randomization and provide built-in constraints. It provides the following two set of constraints:

- The `valid_ranges` constraints limit generated values to those acceptable to drivers. These constraints ensure basic VIP operation and should never be disabled.
- The `reasonable_*` constraints, which can be disabled individually or as a block, limit the simulation by doing the following:
  - Enforcing the protocol. These constraints are typically enabled unless errors are injected in the simulation.
  - Setting simulation boundaries. Disabling these constraints may slow the simulation and introduce system memory issues.

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

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

The I2C VIP defines the following transaction classes:

- I2C Base transaction (`svt_i2c_transaction`)

This is the base transaction type, which contains all the physical attributes of a transaction, such as cmd_type(write/read/general call), slave address, and data, etc.

- I2C Master transaction (`svt_i2c_master_transaction`)

  The master transaction class extends from the I2C transaction base class, namely, `svt_i2c_transaction` and has properties to specify the following:

  - If Master has to dump the current transaction or retry if it loses arbitration.
  - Master to enable 10 -bit addressing.
  - Master has to arbitrate for the current transaction. Master waits till a START condition is generated by other master on the bus.
  - Time interval between the completion of a transaction to the start of next transaction.
  - Number of times Master will try to complete current transaction in case of not getting an not getting an ACK from slave or when it loses arbitration.
  - If Master has to abort the current transaction or retry in case of NACK from Slave.
  - Byte to be sent as a second byte after the General Call address.
  - If Master has to send a START byte at the beginning of a transaction.
  - If Master has to generate the repeated START (Sr) or a STOP (P)

- I2C Slave transaction (`svt_i2c_slave_transaction`)

  The slave transaction class extends from the I2C transaction base class, namely `svt_i2c_transaction`. Slave transactions only configure the response behavior of a Slave agent and these transactions are not seen on the I2C bus. Slave transaction has the following properties to configure the following responses from the slave agent:

  - If Slave has to respond with a NACK.
  - The number of times Slave has to respond with NACK.
  - Data byte number for which Slave should send NACK.
  - Duration to hold the SCL line low when Master transmits the slave address.
  - Duration to hold the SCL line low after every byte transmitted by Master.
  - Duration to hold the SCL line low when it receives or transmits the data byte.

For details on the individual members of transaction classes, refer to the I2C VIP class reference HTML documentation:

`$DESIGNWARE_HOME/vip/svt/i2c_svt/doc/i2c_svt_uvm_class_reference/html/index.html`

3.3.3 Analysis Ports

In active as well as passive mode of operation of the master or slave agent, you can use the analysis port for connecting to the scoreboard, or for any other purpose, where a transaction object, that is, `svt_i2c_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 `svt_i2c_transaction` object from its analysis port.

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

- `svt_i2c_master_monitor::xact_observed_port`

- `svt_i2c_master::pre_observed_port`
- `svt_i2c_slave_monitor::xact_observed_port`
- `svt_i2c_slave::pre_observed_port`

pre_observed_port:

`pre_observed_port` is defined as the pre-analysis port. The VIP Master/Slave pre-port is populated with the same transaction which is sent from the sequencer to Master/Slave VIP component.

Connections with scoreboard can be done as following:

```
i2c_system_env.master[0].driver.pre_observed_port.connect(sb.item_collected_ma
ster_pre)
```

xact_observed_port:

`xact_observed_port` is defined as the post analysis port. This port is populated with packets formed by sampling the bus. This packet is available at the end of particular transaction in both Master/Slave VIP component.

Connections with scoreboard can be done as following:

```
i2c_system_env.master[0].monitor.xact_observed_port.connect(sb.item_collected_
master)
```

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.

The I2C VIP provides the SystemVerilog interface, which can be used to connect the VIP to the DUT. Synopsys defines the top-level interface, `svt_i2c_if`.

The top-level interface is contained in the system configuration class. The top-level interface is specified to the system configuration class using the `svt_i2c_system_configuration::set_if` method. This interface is also passed to Master and Slave agents.

The port interface, `svt_i2c_if`, contains the following modports, which you should use to connect the VIP to the DUT:

- `svt_i2c_master_modport`: This modport is used by the driver inside the master agent.
- `svt_i2c_slave_modport`: This modport is used by the driver inside the slave agent.
- `svt_i2c_monitor_modport`: This modport is used by monitor component inside the master and slave agents.

3.3.5 Events

Events that are available from Master agent's driver are as follows:

- `event_mst_start_generated`: Signals the START condition from Master
- `event_mst_stop_generated`: Signals the STOP condition from Master

- `event_mst_ack_generated`: Signals generating ACK from Master
- `event_mst_nack_generated`: Signals generating NACK from Master
- `event_mst_ack_received`: Signals receiving ACK by Master
- `event_mst_nack_received`: Signals receiving NACK by Master
- `event_mst_repeated_start_generated`: Signals the REPEATED START condition
- `event_mst_start_byte_transmited`: Signals START BYTE from Master
- `event_mst_general_call_addr_sent`: Signals the GENERAL CALL command address sent from Master
- `event_mst_general_call_sec_byte_sent`: Signals the GENERAL CALL command second byte sent from Master
- `event_mst_arbitration_loss_detected`: Signals losing arbitration by Master

Events that are available from Slave agent's driver are as follows:

- `event_slv_start_detected`: Signals the START detection by Slave
- `event_slv_stop_detected`: Signals the STOP detection by Slave
- `event_slv_ack_received`: Signals receiving ACK by Slave
- `event_slv_nack_received`: Signals receiving NACK by Slave
- `event_slv_ack_generated`: Signals generating ACK from Slave
- `event_slv_nack_generated`: Signals generating NACK from Slave

Events that are available from Monitor inside Master and Slave agents are as follows

- `event_mon_start_condition`: Signals the START condition from Master
- `event_mon_stop_condition`: Signals the STOP condition from Master
- `event_mon_ack_received`: Signals receiving ACK on the bus
- `event_mon_nack_received`: Signals receiving NACK on the bus
- `event_mon_start_byte`: Signals START BYTE from Master
- `event_mon_gen_call`: Signals the GENERAL CALL command from Master
- `event_mon_slave_add_7_b`: Signals 7-bit slave address from Master
- `event_mon_slave_add_10_b`: Signals 10-bit slave address from Master
- `event_mon_get_packet`: Signals receiving a data byte on the bus
- `event_mon_repeated_start_condition`: Signals the REPEATED START condition

You can access events by cross-module reference. For example:

- To access the `event_mst_start_generated` event in the master agent’s driver, use the following code snippet:

  ```
  test_top.Master.Master.if_master.event_mst_start_generated
  ```

  where,

  - `test_top` is the top-level module.
  - `test_top.Master` is an instance of the `svt_i2c_master_wrapper` module.
- To access the `event_slv_start_detected` event in the slave agent's driver, use the following code snippet:

  ```
  test_top.Slave.Slave.if_slave.event_slv_start_detected
  ```

  where,
  - `test_top` is the top-level module.
  - `test_top.Slave` is an instance of the `svt_i2c_slave_wrapper` module.
- To access the `event_mon_start_condition` event in the master agent's monitor, use the following code snippet:

  ```
  test_top.Master.Master.monitor_master.if_mon.event_mon_start_condition
  ```

  where,
  - `test_top` is the top-level module.
  - `test_top.Master` is an instance of the `svt_i2c_master_wrapper` module.
- To access the `event_mon_start_condition` event in the slave agent's monitor, use the following code snippet:

  ```
  test_top.Slave.Slave.monitor_slave.if_mon.event_mon_start_condition
  ```

  where,
  - `test_top` is the top-level module.
  - `test_top.Slave` is an instance of the `svt_i2c_slave_wrapper` module.

3.4 Functional Coverage

This section consists of the following sub-sections:

- "Built-In Coverage" on page 38
- "Coverage Callback Classes" on page 39
- "Enabling the Built-In Coverage" on page 40

3.4.1 Built-In Coverage

The I2C SVT VIP provides the following types of built-in coverage:

- "Transaction-Based Coverage" on page 38
- "Toggle-Based Coverage" on page 39
- "Pattern-Based Coverage" on page 39

For more details on covergroups, refer to the I2C SVT VIP class reference HTML document.

3.4.1.1 Transaction-Based Coverage

The transaction-based coverage is a coverage collection based on the properties of transactions generated by the I2C master and the I2C slave.

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 for the type of command (Write followed by Read followed by General call) and addressing modes (7-bit addressing followed by 10-bit addressing), etc.

3.4.2 Coverage Callback Classes

This section consists of the following sub-sections:

✦ "Coverage Data Callback" on page 39
✦ "Coverage Callback" on page 39
✦ "Toggle Coverage Data Callback" on page 40
✦ "Toggle Coverage Callback" on page 40

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_i2c_master_monitor_callback`. For example, the coverage data callback class name of the master monitor is `svt_i2c_master_monitor_def_cov_data_callbacks`.

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

✦ `master_xact_observed_cov`

3.4.2.2 Coverage Callback

This callback class includes default 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_i2c_master_monitor_def_cov_data_callbacks`. The coverage callback class that implements default covergroups is `svt_i2c_master_monitor_def_cov_callbacks`.

3.4.2.3 Toggle Coverage Data Callback

This callback class defines the 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_i2c_monitor_callback`. The toggle coverage data callback class name is `svt_i2c_monitor_def_toggle_cov_data_callback`.

The following methods are implemented for sampling the toggle coverage:

- `recognize_scl_sda_samples`
- `sample_scl_sda_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_i2c_monitor_toggle_def_cov_callback`.

3.4.3 Enabling the Built-In Coverage

You can enable the default functional coverage by setting the `coverage_enable` attribute in the configuration class, `svt_i2c_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 master transactions as well as for slave transactions.

Exceptions for master transactions are as follows:

Table 3-1

Exception Macros	Des
DATA_SIZE_LESS_THAN_8_BIT_ERROR	/**< Data byte with less than 8 bits is transmitted*/
SETUP_TIME_VIOLATION_FOR_STOP_ERROR	/**< Setup time for stop violated*/
SETUP_TIME_VIOLATION_FOR_DATA_ERROR	/**< Setup time for data violated*/

Table 3-1

Exception Macros	Des
HOLD_TIME_VIOLATION_FOR_START_ERROR	/**< Hold time for start violated*/
HOLD_TIME_VIOLATION_FOR_DATA_ERROR	/**< Hold time for data violated*/
INVALID_TBUF_TIME_ERROR	/**< Tbuf time violated*/
INVALID_HIGH_CLK_DURATION_ERROR	/**< SCL High Time violated*/
INVALID_LOW_CLK_DURATION_ERROR	/**< SCL Low Time violated*/
MASTER_CODE_NOT_SENT_IN_HS_ERROR	/**< Master doesnot send master code in HS mode*/
NO_OP_ERROR	/**< This error kind selects no error*/
ACK_ON_LAST_READ_ERROR	/**< Master sends ACK on the last read byte instead of NACK*/
NACK_ON_RANDOM_READ_BYTE_ERROR	/**< Master, at random, responds with NACK on any read byte*/
START_STOP_ERROR	/**< Master generates START condition, immediately followed by STOP condition*/
ACK_FOR_MASTER_CODE_HS_ERROR	/**< Master generates ACK for MASTER CODE (instead of NACK)*/
MISSING_START_ERROR	/**< Master starts transaction without START condition*/
ABORT_TRANS_RANDOM_ERROR	/**< Master aborts transaction randomly*/
STOP_IN_BYTE	/**< Master generates STOP in the middle of command or write data byte*/
FALSE_STOP_BEFORE_START	/**< Master generates 'n' number of STOP conditions, where minimum value of n is 1*/
START_FOLLOWED_BY_START	/**< Master generates 'n' number of START conditions, where minimum value of n is 1*/
START_IN_BYTE	/**< Master generates START in the middle of command or write data byte*/
SETUP_TIME_VIOLATION_FOR_REP_START	/**< Master violates setup time for repeated START*/
NO_SR_OR_P_GEN_AFTER_NACK	/**< Master does not generate Rep Start or Stop after NACK is received on bus for write data*/
CMD_CORRUPT_FOR_10BIT_READ	?/**< Master corrupts the first frame sent as part of 10 bit read. It sends a READ instead of WRITE in first frame*/
SMBUS_ABORT_TRANS_RANDOM_ERROR	/**< Master aborts transaction at byte configured as <data_byte_pos>. Use of data_byte_pos is required to use this Exception */

Table 3-1

Exception Macros	Des
MST_CORRUPT_BYTE_COUNT	/**< To corrupt number of bytes to be sent by master."mst_corrupted_num_of_bytes" to be used to configure the number of bytes to be sent.*/
SMBUS_MISSING_REP_START	/**< Master does not generate repeated start in case of READ command or any process command */
I2C_INSERT_P_AFTER_8_BIT_RD_DATA	/**< Master insert Stop after 8th bit of Read data sent by slave. */
I2C_INSERT_P_BEFORE_1_BIT_RD_DATA	/**< Master insert Stop before 1st bit of Read data sent by slave. */
I2C_INSERT_P_AFTER_MASTER_CODE	/**< Master insert Stop after 8th bit of Master code in HS mode. */
I2C_MASTER_MISS_SR_AFTER_MASTER_CODE	/**< Master erroneously miss repeated start after master code in HS mode and continues th transaction afterwards. */
I2C_INSERT_REPEATED_START_AFTER_8_BIT_RD_DATA	/**< Master insert REPEATED START after 8th bit of Read data sent by slave. */
I2C_INSERT_REPEATED_START_BEFORE_1_BIT_RD_DATA	/**< Master insert REPEATED START before 1st bit of Read data sent by slave. */
I2C_MASTER_MISS_SR_AFTER_START_BYTE	/**< Master erroneously miss repeated start after Start Byte and continues th transaction afterwards. */

The following is the Exception for slave transactions and their description:

Table 3-2

Exception Slaves	Description
SETUP_TIME_VIOLATION_DATA_ERROR	/** Setup time for data violated*/
HOLD_TIME_VIOLATION_DATA_ERROR	/**Hold time for data violated*/
SEND_ACK_FOR_START_BYTE_ERROR	/** Ack sent for Start Byte*/
NO_OP_ERROR	/** This error kind selects no error*/
SEND_ACK_FOR_RESERVED_ADDR_ERROR	/**< Ack sent for RESERVED ADDRESS*/

The detailed use model of different exceptions are mentioned “Exceptions Use Model” on page 123. For more details on exceptions, refer to the I2C SVT VIP class reference HTML document.

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 Master and Slave 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. To execute callback methods, you must register the callback class with the component using the `` `uvm_register_cb `` macro.

The I2C SVT 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-sections:

- "Master Agent Callbacks" on page 44
- "Slave Agent Callbacks" on page 44

For more details on callbacks, refer to the I2C SVT VIP class reference HTML document.

3.6.1 Master Agent Callbacks

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

- `svt_i2c_master_callback`
- `svt_i2c_master_monitor_callback`

For details of these classes, refer to the following I2C SVT VIP class reference HTML documentation:

`$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/class_svt_i2c_master_callback.html`

and

`$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/class_svt_i2c_master_monitor_callback.html`

3.6.2 Slave Agent Callbacks

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

- `svt_i2c_slave_callback`
- `svt_i2c_slave_monitor_callback`

For details of these classes, refer to the following I2C SVT VIP class reference HTML documentation:

`$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/class_svt_i2c_slave_callback.html`

and

`$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/class_svt_i2c_slave_monitor_callback.html`

3.7 Sequence Collection

The I2C VIP provides a collection of I2C master and slave sequences. These sequences can be registered with the master and slave sequencers within master and slave agents respectively to generate different types of I2C scenarios. All I2C master sequences extend from the base sequence, namely, `svt_i2c_master_transaction_base_sequence`. All I2C slave sequences extend from the base sequence, namely, `svt_i2c_slave_transaction_base_sequence` respectively.

For a list of all master and slave sequences, refer to the following I2C VIP class reference HTML documentation:

`$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/sequencepages.html`

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

3.8 Protocol Checks

You can enable protocol checks by setting the configuration attribute, `checks_enable`, in the `svt_i2c_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 I2C VIP class reference HTML documentation:

`$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/protocolChecks.html`

3.9 Verification Planner

The I2C VIP provides verification plans, which track the verification progress of the I2C protocol. The I2C 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/i2c_svt/latest/doc/VerificationPlans`

For more information, refer to the README file, which is available at the following location:
`$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/VerificationPlans/README`

3.10 Source Code Visibility

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

For details on the classes, which are available with the source code visibility feature, refer to the following I2C VIP class reference HTML documentation:

`$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/sourcevisibility.html`

4
Verification Topologies

The chapter consists of the following chapters that show you from a high-level how the I2C VIP can be used to test Master, Slave, or Interconnect DUT:

- "Master DUT and Slave VIP" on page 45
- "Slave DUT and Master VIP" on page 46
- "System DUT and Passive VIP" on page 47
- "System DUT With a Mix of Active and Passive VIP" on page 49

4.1 Master DUT and Slave VIP

Scenario: The VIP is required to verify the I2C Master DUT.

Testbench Setup: Configure the I2C System configuration to have one slave agent in an active mode. The active slave agent responds to the transactions generated by Master DUT. The Slave 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 system configuration is `sys_cfg`.)

System Configuration Settings:

```
// Instantiate at least one master agent and configure it as a passive agent

sys_cfg.num_masters = 1;
sys_cfg.num_slaves = 1;
```

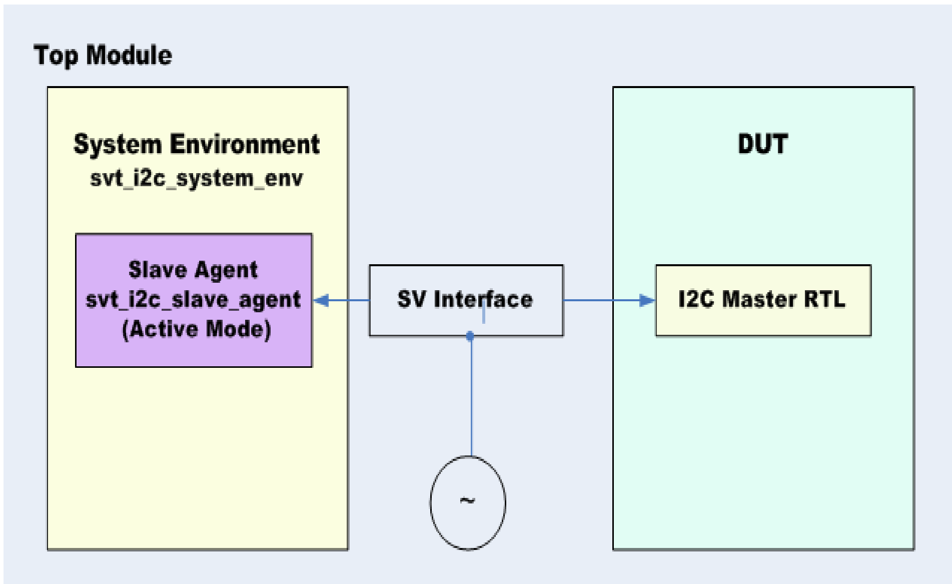
Agent Configuration Settings:

```
sys_cfg.slave_cfg[0].is_active = 1;
sys_cfg.master_cfg[0].is_active = 0;
```

When the DUT has a single I2C master port to be verified, the testbench can either use a slave agent in the stand-alone mode, or use a system environment configured for a single slave agent. In the stand-alone mode, the testbench becomes light weight as the system environment and related infrastructure is not required. However, the testbench does not remain scalable. If you need to increase the number of I2C master ports that needs to be verified, you need to replace the stand-alone slave agent with the system environment or you need to instantiate multiple slave agents.

Figures 4-1 shows Master DUT and Slave VIP: Usage with the system environment.

Figure 4-1Master DUT and Slave VIP - Usage With the System Environment



4.2 Slave DUT and Master VIP

Scenario: The VIP is required to verify the I2C Slave DUT.

Testbench Setup: Configure the I2C System configuration to have one master agent in an active mode. The active master agent generates I2C transactions for the Slave DUT. The Master agent also performs passive functions, such as Protocol checking, Coverage generation, and Transaction logging.

When the DUT has a single I2C slave port to be verified, the testbench can either use a master agent in the stand-alone mode, or use the system environment configured for a single master agent. In the stand-alone mode, the testbench becomes light weight as the system environment and related infrastructure is not required. However, the testbench does not remain scalable. If you need to increase the number of I2C slave ports that needs to be verified, you need to replace the stand-alone master agent with the system environment or you need to instantiate multiple master agents.

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

(Assuming the instance name of system configuration is `sys_cfg`.)

System Configuration Settings:

```
// Instantiate at least one slave agent and configure it as a passive agent
```

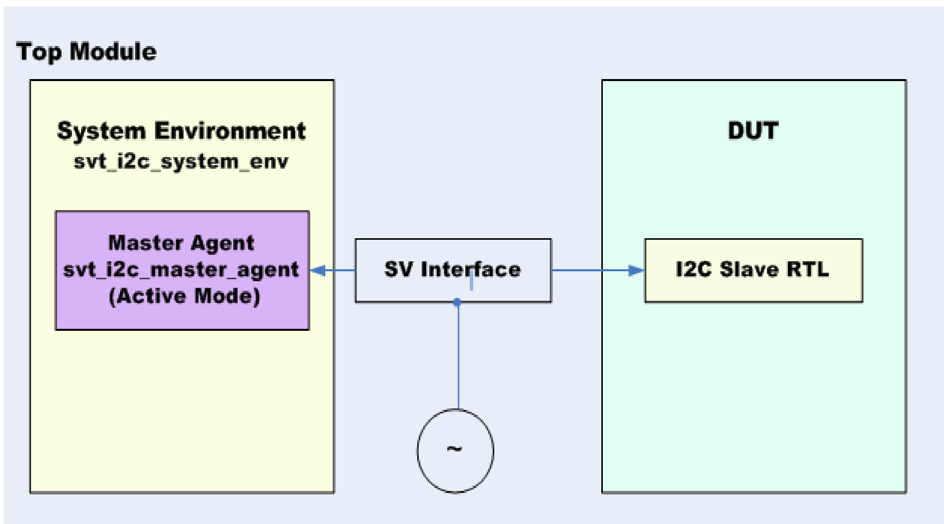
```
sys_cfg.num_masters = 1;
sys_cfg.num_slaves = 1;
```

Agent Configuration Settings:

```
sys_cfg.master_cfg[0].is_active = 1;
sys_cfg.slave_cfg[0].is_active = 0;
```

Figures 4-2 shows Slave DUT and Master VIP: Usage with the system environment.

Figure 4-2Slave DUT and Master VIP - Usage With the System Environment



4.3 System DUT and Passive VIP

Scenario: The DUT is an I2C system with multiple I2C masters, slaves, and interconnect. The VIP is required to monitor the DUT.

Testbench Setup: Assuming that the I2C System has M masters and S slaves, configure the I2C System environment to have M master agents and S slave agents, in the passive mode. The passive master and slave agents 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 system configuration is `sys_cfg`,

Assuming the number of master ports on interconnect = 2,

Assuming the number of slave ports on interconnect = 2)

System Configuration Settings:

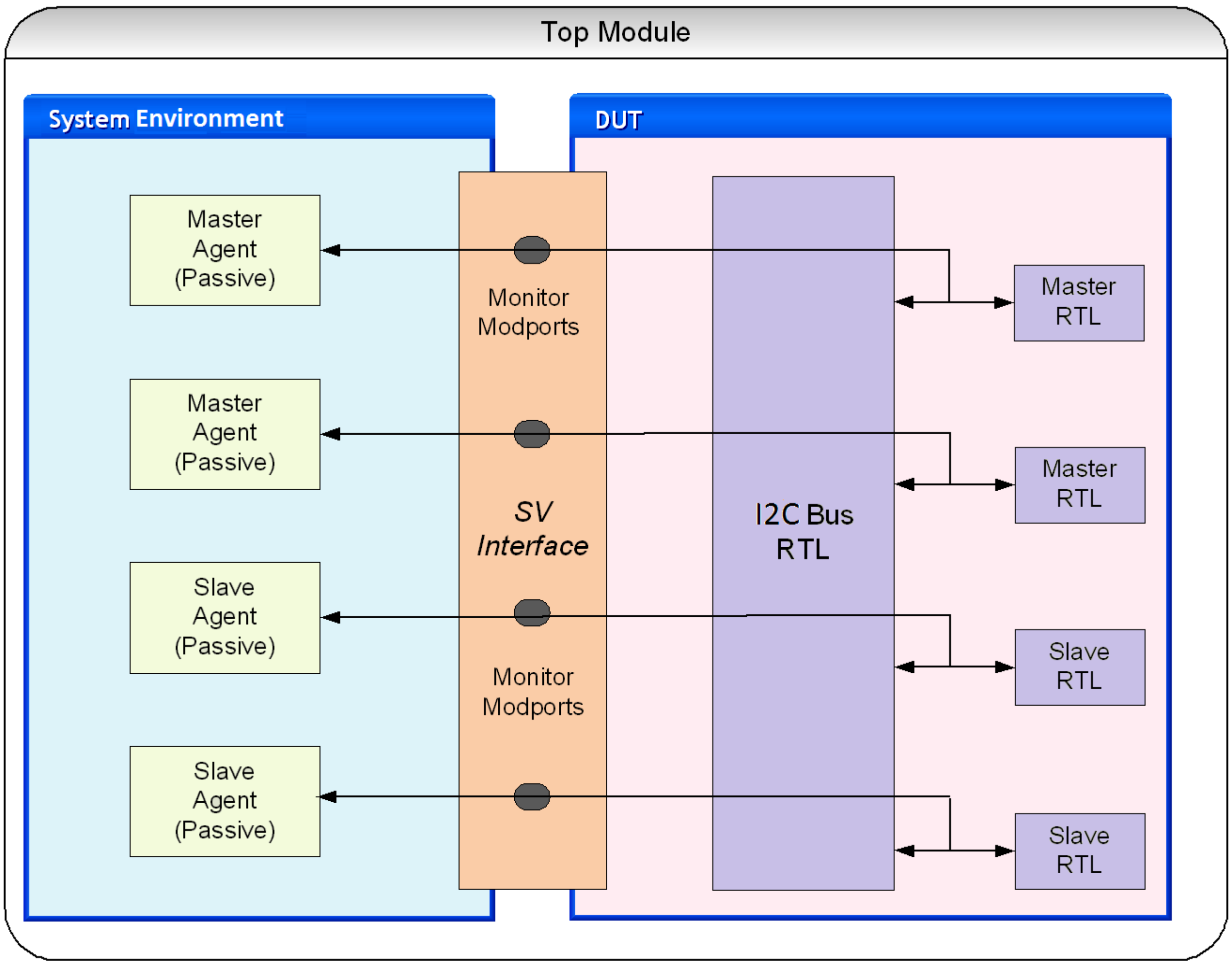
```
sys_cfg.num_masters = 2;
sys_cfg.num_slaves = 2;
```

Agent Configuration Settings:

```
sys_cfg.master_cfg[0].is_active = 0;
sys_cfg.master_cfg[1].is_active = 0;
sys_cfg.slave_cfg[0].is_active = 0;
sys_cfg.slave_cfg[1].is_active = 0;\
```

Figures 4-3 show System DUT with Passive VIP.

Figure 4-3System DUT With Passive



4.4 System DUT With a Mix of Active and Passive VIP

In this scenario, the DUT is a system with multiple I2C masters, slaves, and interconnect. The VIP is required to provide the background traffic on some ports, and to monitor on ports.

Testbench Setup: Assuming that the I2C System DUT has two master ports and two slave ports. The VIP is required to provide background traffic to ports S0 and M0. All the ports need to be monitored. Configure the I2C System environment to have two master agents and two slave agents. Configure the master agent connected to port S0, and the slave agent connected to port M0 as active. Configure the master agent connected to port M1 and the slave agent connected to port M1 as passive. All the agents continue to perform passive functions, such as protocol checking and coverage.

Figures 4-4 show System DUT with a mix of Active and Passive VIP.

Figure 4-4System DUT With a Mix of Active and Passive VIP

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

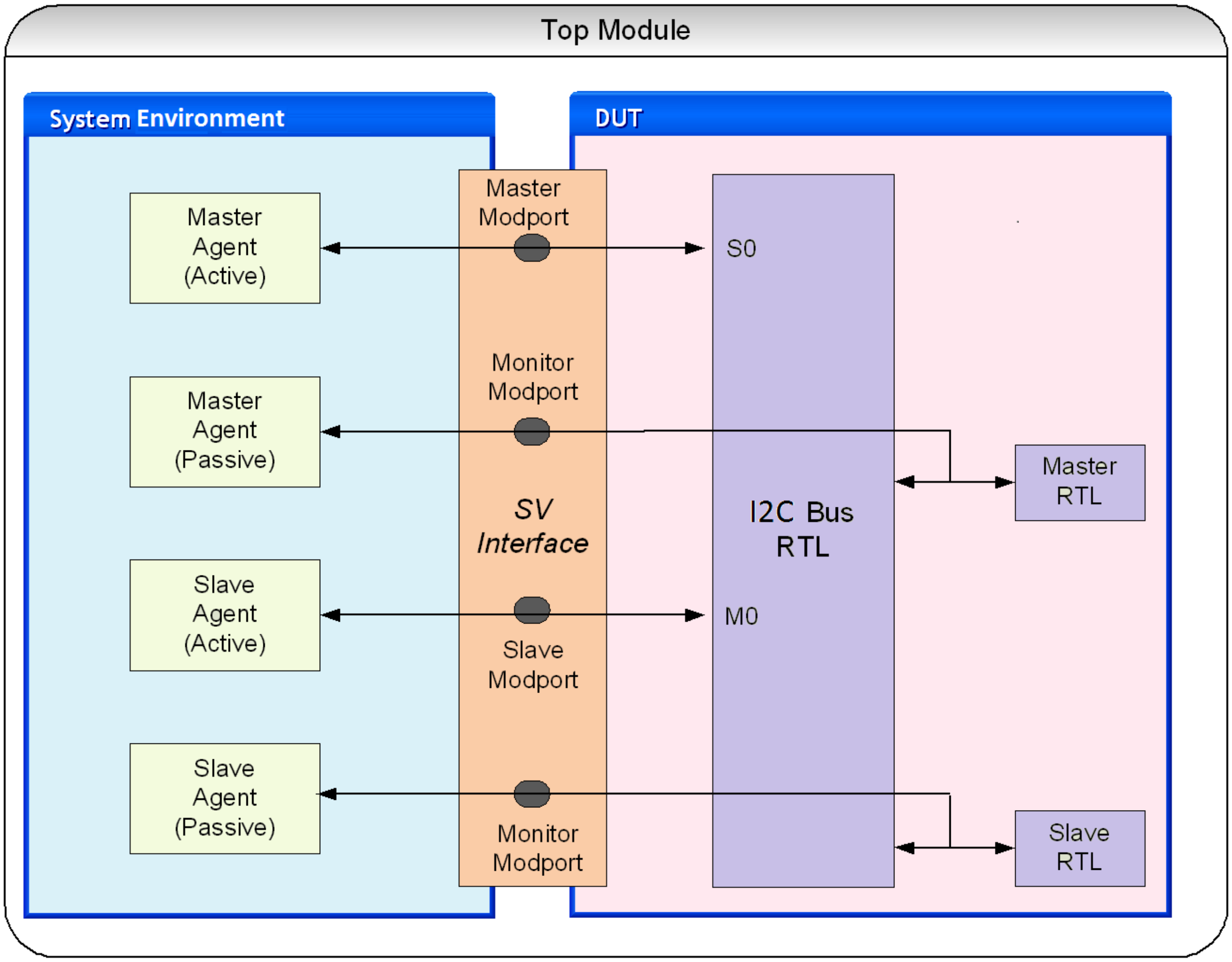
(Assuming the instance name of system configuration is `sys_cfg`)

System Configuration Settings:

```
sys_cfg.num_masters = 2;
sys_cfg.num_slaves = 2;
```

Agent Configuration Settings:

```
sys_cfg.master_cfg[0].is_active = 1;
sys_cfg.master_cfg[1].is_active = 0;
sys_cfg.slave_cfg[0].is_active = 1;
sys_cfg.slave_cfg[1].is_active = 0;
```


Top Module
System Environment
DUT
Master
Agent
(Active)
Master
Agent
(Passive)
Slave
Agent
(Active)
Slave
Agent
(Passive)
Master
Modport
Monitor
Modport
SV
Interface
Slave
Modport
Monitor
Modport
S0
M0
I2C Bus
RTL
Master
RTL
Slave
RTL

5
Using I2C Verification IP

This section describes SystemVerilog UVM example testbenches that show the general usage for various applications. Tables 5-1 lists the summary of the examples.

Table 5-1 SystemVerilog Example Summary

Example Name	Level	Description
tb_i2c_svt_uvm_basic_sys	Basic	The example consists of the following: • A top-level testbench in SystemVerilog • A dummy DUT in the testbench, which has two I2C interfaces • A UVM verification environment having an I2C system environment configured with single master and slave agent • Two tests illustrating the directed and random transaction generation from master and slave agents A quickstart for this example is available at the following location: `$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/examples/sverilog/tb_i2c_svt_uvm_basic_sys/doc/tb_i2c_svt_uvm_basic_sys/index_basic.html`
tb_i2c_svt_uvm_intermediate_sys	Intermediate	The example consists of the following: • A top-level testbench in SystemVerilog • A dummy DUT in the testbench, which has two I2C interfaces • A UVM verification environment having an I2C system environment configured with single master and slave agent • Tests illustrating the usage of verification features, such as sequence collection, PA, exceptions, callbacks, coverage, and scoreboard • User-specified data generation from the slave, EEPROM-mode slave

Table 5-1 SystemVerilog Example Summary

Example Name	Level	Description
		A quickstart for this example is available at the following location: `$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/examples/sverilog/tb_i2c_svt_uvm_intermediate_sys/doc/tb_i2c_svt_uvm_intermediate_sys/index_intermediate.html`

You can perform the following steps to install and run the `tb_i2c_svt_uvm_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
i2c_svt/tb_i2c_svt_uvm_basic_sys -svtb
```

The example gets installed in the following location:
`<design_dir>/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_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_i2c_svt_uvm_basic_sys -w directed_test vcsvlog-svlog
```

Invoke `./run_i2c_svt_uvm_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/i2c_svt/latest/examples/sverilog/tb_i2c_svt_uvm_basic_sys/README`

or

`<design_dir>/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/README`

Note You can use the above steps to install and run the `tb_i2c_svt_uvm_intermediate_sys` example also.

6
Usage Notes

To model different I2C protocol scenarios, the properties of master and slave transactions can be used to create different stimulus from the master agent. This section covers the code snippets of Master and Slave sequences to cover the common I2C verification scenarios.

This chapter consists of the following sections:

- ❖ "Changing Driving strength of SDA Line from VIP" on page 101
- ❖ "Essential Requirements" on page 101
- ❖ "I2C UVM Scenario Reference Guide" on page 103

6.1 Master Transaction Properties

The Master transaction class has different properties, which are used to create the different stimulus from the master agent.

For details on the properties of the transaction class, refer to the following I2C VIP class reference HTML documentation:

```
$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/class_svt_i
2c_master_transaction.html
```

Examples for Master Sequences

This section covers the examples for the following:

- 7-bit addressing, Write command, and 4-byte user-defined data
- 10-bit addressing, Write-read command, Repeated start, and Random data
- 7-bit addressing, write command, retry enable
- General Call

Note To apply a constraint, you may have to override the reasonable constraints, such as `reasonable_addr_10bit, reasonable_sr_or_p_gen`, and `reasonable_send_start_byte`, etc. in the `cust_svt_i2c_master_transaction` class and the `cust_svt_i2c_slave_transaction` class.

Figures 6-1 shows 7-bit addressing, Write command, and 4-byte user-defined data.

Figure 6-1. 7-Bit Addressing, Write command, and 4-byte User-Defined Data

```
class svt_i2c_7bit_write_seq extends uvm_sequence #(svt_i2c_master_transaction);

`uvm_object_utils(svt_i2c_7bit_write_seq)
.................
.................
virtual task body();
   `uvm_info("body", "Entering...", UVM_DEBUG)
   `uvm_do_with( req,
               { req.addr              == `I2C_SLAVE0_ADDRESS  ; // Slave Address
                 req.cmd               == `I2C_WRITE           ; // command
                 req.data.size()       ==  4                   ; // no. of data bytes to be sent by Master
                 req.data[0]           ==  8'b10101010         ; // user defined data byte 0
                 req.data[1]           ==  8'b10010011         ; // user defined data byte 1
                 req.data[2]           ==  8'b11110000         ; // user defined data byte 2
                 req.data[3]           ==  8'b10000001         ; // user defined data byte 3
                 req.sr_or_p_gen       ==  0                   ; // STOP(P) condition,for repeated start make it to 1
                 req.send_start_byte   ==  0                   ; // START Byte disabled, to enable make it to 1
                 req.addr_10bit        ==  0                   ; // to enable 10 bit addressing,: make it to 1
                 })
endtask : body

endclass : svt_i2c_7bit_write_seq
```

Figures 6-2 shows 10-bit addressing, Write-read command, Repeated start, and Random data. Note that you need to set the value of the slave configuration property, that is `enable_10bit_addr` to 1 to configure the slave for 10-bit addressing.

Figure 6-2 10-Bit Addressing, Write-Read Command, Repeated Start, and Random Data

```
class svt_i2c_10bit_write_seq_with_sr extends uvm_sequence #(svt_i2c_master_transaction);

`uvm_object_utils(svt_i2c_10bit_write_seq_with_sr)
.................
.................
virtual task body();
   `uvm_info("body", "Entering...", UVM_DEBUG)

    `uvm_do_with( req,
                { req.addr           == `I2C_SLAVE0_ADDRESS  ;
                 req.cmd            ==  I2C_WRITE           ;
                 req.data.size()    ==  4                   ; // 4-byte data to be transfered
                 req.sr_or_p_gen    ==  1                   ; // repeated start enabled
                 req.send_start_byte ==  0                  ; // start byte disabled
                 req.addr_10bit     ==  1                   ; // 10-bit addressing enabled
                })
    `uvm_do_with( req,
                {req.addr           == `I2C_SLAVE0_ADDRESS  ;
                 req.cmd            ==  I2C_READ            ;
                 req.data.size()    ==  4                   ; // 4-byte data to be transfered
                 req.sr_or_p_gen    ==  0                   ; // repeated start disabled
                 req.send_start_byte ==  0                  ; // start byte disabled
                 req.addr_10bit     ==  1                   ; // 10-bit addressing enabled
                 })

endtask : body

endclass : svt_i2c_10bit_write_seq_with_sr
```

Figures 6-3 shows 7-bit addressing, write command, and retry enable 2 times retry.

Figure 6-3 7-Bit Addressing, Write Command, and Retry Enable 2 Times Retry

```
class svt_i2c_mst_retry_for_nack extends uvm_sequence #(svt_i2c_master_transaction);

    `uvm_object_utils(svt_i2c_mst_retry_for_nack)
    ..............
    ..............
   virtual task body();
           `uvm_info("body", "Entering...", UVM_DEBUG)
          `uvm_do_with( req,
                  { req.addr           == `I2C_SLAVE0_ADDRESS ;
                    req.cmd            == `I2C_WRITE         ; // for read command it will be `I2C_READ
                    req.data.size()    == 4                  ; // 4-byte of data to be transferred
                    req.sr_or_p_gen    == 0                  ; // stop condition disabled
                    req.send_start_byte == 0                 ; // start byte disabled
                    req.addr_10bit     == 0                  ; // 7-bit addressing enabled
                    req.num_of_retry   == 2                  ; // it is the count that "how many times master will retry
                    req.retry_if_nack  == `I2C_TRUE          ; // it should be true only when slave is configured for NA
                                                               // it is to enable the master for retry on receiving NACK
                 })

  endtask : body

endclass : svt_i2c_mst_retry_for_nack
```

Figure 6-4 shows General Call.

Figure 6-4 General Call

```
class svt_i2c_gen_call_addr extends uvm_sequence #(svt_i2c_master_transaction);
    `uvm_object_utils(svt_i2c_gen_call_addr)
     .............
     .............

     virtual task body();
             `uvm_info("body", "Entering...", UVM_DEBUG)
             `uvm_do_with( req,
                      { req.addr            == `I2C_SLAVE0_ADDRESS ;
                        req.cmd             == `I2C_GEN_CALL       ;
                        req.data.size()     == 4                   ;
                        req.sr_or_p_gen     == 0                   ;
                        req.send_start_byte == 0                   ;
                        req.addr_10bit      == 0                   ;
                        req.sec_byte_gen_call == 8'h04             ;
                       })

             `uvm_do_with( req,
                        { req.addr            == `I2C_SLAVE0_ADDRESS ;
                          req.cmd             == `I2C_GEN_CALL       ;
                          req.data.size()     == 4                   ;
                          req.sr_or_p_gen     == 0                   ;
                          req.send_start_byte == 0                   ;
                          req.addr_10bit      == 0                   ;
                          req.sec_byte_gen_call == 8'h06             ;
                         })
     endtask : body

  endclass : svt_i2c_gen_call_addr
```

6.2 Slave Transaction Properties

The Slave transaction class has different properties, which are used for configuring the different responses from the slave agent.

For details on the classes, which are available with the source-code visibility feature, refer to the I2C following VIP class reference HTML documentation:

```
$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/class_svt_i
2c_slave_transaction.html
```

Examples for Slave Sequences

This section covers the examples for the following:

- ✦ Byte stretching
- ✦ Slave configuration for sending the user-defined data

Figures 6-5 shows Byte stretching.

Figure 6-5. Byte Stretching

```
class i2c_slv_directed_sequence extends uvm_sequence #(svt_i2c_slave_transaction);

        svt_i2c_slave_transaction  tx_xacts_s;
        .............
        .............



        virtual task body();
                `uvm_info("body", "Entering...", UVM_DEBUG)

                `uvm_create(tx_xacts_s)
                 tx_xacts_s.nack_addr                  =  0      ;
                 tx_xacts_s.clk_stretch_time_after_byte  =  0      ;   // it will introduce byte stretching after each byte
                                                                       // irrespective of Address/Data
                 tx_xacts_s..clk_stretch_time_addr_byte  =  10000  ;   // it will introduce bit stretching during address byte
                 tx_xacts_s.clk_stretch_time_data_byte   =  0      ;   // it will introduce bit stretching during data byte
                 tx_xacts_s.nack_addr_count             =  0      ;   // count shows how many times slave will
                                                                       // send nack for it's address on bus


                `uvm_send(tx_xacts_s)

                `uvm_info("body", "Exiting...", UVM_DEBUG)
        endtask: body

 endclass: i2c_slv_directed_sequence
```

Figures 6-6 shows the slave configuration for sending the user-defined data.

Figure 6-6. Slave Configuration for User-Defined Data

```
class svt_i2c_slv_read_predef_data extends uvm_sequence #(svt_i2c_slave_transaction);

    `uvm_object_utils(svt_i2c_slv_read_predef_data)
     .................
     .................

    virtual task body();
          `uvm_info("body", "Entering...", UVM_DEBUG)

          `uvm_do_with(req,
                      {req.data.size() == 4;
                       req.data[0] == 8'b00000001;
                       req.data[1] == 8'b01010101;
                       req.data[2] == 8'b00110011;
                       req.data[3] == 8'b00001111;
                      })
    endtask : body

endclass : svt_i2c_slv_read_predef_data
```

Note

If you want to create a scenario in which Slave has to respond with certain specified responses then before running the master sequence, the slave sequence with the desired values of attributes should get executed first.

Figures 6-7 shows the execution order of slave and master sequences. The slave sequence is executed first to configure the slave response before starting the execution of the master sequence.

Figure 6-7. Execution Order of Slave and Master Sequences

```
class i2c_default_virtual_sequence extends uvm_sequence;
    `uvm_object_utils(i2c_default_virtual_sequence)
            ............
            ...........
      virtual task body();
            ..........
            .........
    /** Instance of default sequence, to be started on the virtual sequencer. */
       svt_i2c_7bit_read_seq                  :       mst_7bit_rd;
       svt_i2c_read_seq_predefined_data        slv_rd_predef_data;

    /** to run read command with predefined data from slave. */
     `uvm_do_on(slv_rd_predef_data, p_sequencer.slv_sequencer)
     `uvm_do_on(mst_7bit_rd, p_sequencer.mst_sequencer)

     endtask : body

endclass : i2c_default_virtual_sequence
```

6.3 Configuring the I2C VIP With Different Frequencies

This section consists of the following sub-sections:

- "Operating Frequency" on page 64
- "Variables and Defines" on page 64
- "Example of the Fast Speed Mode, 300kHz Frequencies With 0 Offset Value" on page 65

6.3.1 Operating Frequency

The operating frequency is the clock frequency, which appears on the SCL line. You can calculate the operating frequency using the following formula:

```
1/{(scl_high_time_<speed-mode>+ scl_high_time_offset_<speed-mode>) +
(scl_low_time_<speed-mode> + scl_low_time_offset_<speed-mode>)}
```

where, the values of `<speed-mode>` are `ss` or `fs`.

6.3.2 Variables and Defines

All the variables related to the frequency calculation are present in the `svt_i2c_configuration.sv` file. The default initial values and valid ranges for variables can be specified using macros. You can specify default initial values and valid ranges for the variables using macros. Macro definitions are present in the `svt_i2c_common_define.h` file.

The details of macros definitions are as follows:

```
//-----------------------------------------------------------------------
// Timing Parameter For Standard Speed Mode (assuming clock period of 1ns)
//-----------------------------------------------------------------------
`define SVT_I2C_CLK_HIGH_SS                                         4000
`define SVT_I2C_CLK_LOW_SS                                          4700
`define SVT_I2C_MAX_CLK_HIGH_OFFSET_SS                              1000
`define SVT_I2C_MAX_CLK_LOW_OFFSET_SS                                300
`define SVT_I2C_MIN_CLK_HIGH_OFFSET_SS                                 0
`define SVT_I2C_MIN_CLK_LOW_OFFSET_SS                                  0
//----------------------------------------------------------------------
// Timing Parameter For Fast Speed Mode (assuming clock period of 1ns)
//----------------------------------------------------------------------
`define SVT_I2C_CLK_HIGH_FS                                          600
`define SVT_I2C_CLK_LOW_FS                                          1300
`define SVT_I2C_MAX_CLK_HIGH_OFFSET_FS                               300
`define SVT_I2C_MAX_CLK_LOW_OFFSET_FS                                300
`define SVT_I2C_MIN_CLK_HIGH_OFFSET_FS                                20
`define SVT_I2C_MIN_CLK_LOW_OFFSET_FS                                 20
```

Note: You can set any value in the possible range of the above variables. You can also change the range of the variables by changing the value of defines in the `svt_i2c_common_define.h` file.

For details on the classes, which are available with the source code visibility feature, refer to the following I2C VIP class reference HTML documentation:

```
$DESIGNWARE_HOME/vip/svt/i2c_svt/latest/doc/i2c_svt_uvm_class_reference/html/class_svt_i
2c_configuration.html
```

6.3.3 Example of the Fast Speed Mode, 300kHz Frequencies With 0 Offset Value

To get 0 offset, you need to set the value of the following two defines to 0 in the `svt_i2c_configuration.sv` file:

```
`define SVT_I2C_MIN_CLK_HIGH_OFFSET_FS          0
`define SVT_I2C_MIN_CLK_LOW_OFFSET_FS           0
```

Set the variables of master and slave configurations to the values given in Figures 6-8 to obtain 300Khz frequency using a test case.

Figure 6-8. Fast Speed Mode

```
class i2c_base_test extends uvm_test;

   -------------
   -------------
   /** Instantiate the configuration for Master*/
  cust_svt_i2c_system_configuration cfg;

    /** build() - Method to build various component */
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    -------------------
    -------------------
    /** Create the configuration object for Master agent */
    cfg = cust_svt_i2c_system_configuration::type_id::create("cfg");

    /** Configure Master and Slave configurations */
    cfg.set_bus_speed(FAST_MODE);                                  // Set Bus-Speed
    cfg.master_cfg[0].master_code       = 3'b101             ;     // Set Master Code
    cfg.slave_cfg[0].slave_address      = `SVT_I2C_SLAVE0_ADDRESS; // Set Slave Address
    cfg.slave_cfg[0].enable_10bit_addr  =  0                 ;     // disable 10-bit Addressing
    cfg.slave_cfg[0].slave_type         = `SVT_I2C_GENERIC       ; // Set Slave as Generic

    cfg.master_cfg[0].scl_high_time_offset_fs  = 32'd0;            // Set Master high_time_offset
    cfg.master_cfg[0].scl_low_time_offset_fs   = 32'd0;            // Set Master low_time_offset
    cfg.slave_cfg[0].scl_high_time_offset_fs   = 32'd0;            // Set slave high_time_offset
    cfg.slave_cfg[0].scl_low_time_offset_fs    = 32'd0;            // Set slave low_time_offset

    cfg.master_cfg[0].scl_high_time_fs = 32'd2000;                 // Set Master high_time
    cfg.master_cfg[0].scl_low_time_fs = 32'd1333;                  // Set Master low_time
    cfg.slave_cfg[0].scl_high_time_fs = 32'd2000;                  // Set slave high_time
    cfg.slave_cfg[0].scl_low_time_fs = 32'd1333;                   // Set slave low_time

    ------------------------
    ------------------------

    endfunction : build_phase

   --------------------
   --------------------

endclass : i2c_base_test
```

6.4 Timing Parameters

This section consists of timing parameters for the following modes:

✦ "Standard-Speed Mode" on page 66
✦ "Fast-Speed Mode" on page 68
✦ "Fast-Mode Plus" on page 70
✦ "High-Speed Mode" on page 72

6.4.1 Standard-Speed Mode

The standard-speed mode provides a bit rate up to 100 Kbit/s. Tables 6-1 shows the timing parameters with a minimum value for the standard-speed mode (assuming the clock period of 1ns).

Table 6-1 Standard-Speed Mode: Timing Parameters (Minimum Value)

Purpose	Symbol	Macro Name	Value (In Spec)	VIP Implementation (In ns)
High period of SCL clock	tLOW	SVT_I2C_CLK_HIGH_SS	4.0 µs	4000
Low period of SCL clock	tHIGH	SVT_I2C_CLK_LOW_SS	4.7 µs	4700
Setup time for the repeated START condition	tSU;STA	SVT_I2C_MIN_SU_STA_SS	4.7 µs	4700
Setup time for the STOP condition	tSU;STO	SVT_I2C_MIN_SU_STO_SS	4.0 µs	4000
Data-setup time	tSU; DAT	SVT_I2C_MIN_SU_DAT_SS	250 ns	250
Hold time for the repeated START condition. After this period, the first clock pulse is generated	tHD; STA	SVT_I2C_MIN_HD_STA_SS	4.0 µs	4000
Data-hold time: for bus devices	tHD; DAT	SVT_I2C_MIN_HD_DAT_SS SVT_I2C_MIN_HD_DAT_MAX_SS	0* µs	300 1000
Bus free time between a STOP and START condition	tBUF	SVT_I2C_TBUF_TIME_SS	4.7 µs	4700
Rise time of both SDA and SCL signals	tr	SVT_I2C_MIN_CLK_HIGH_OFFSET_SS	-	0
Fall time of both SDA and SCL signals	tf	SVT_I2C_MIN_CLK_LOW_OFFSET_SS	-	0

Note

- All values referred to V_{IHmin} and V_{ILmax} levels.
- * indicates that a device must internally provide a hold time of at least 300 ns for the SDA signal (referred to V_{IHmin} of the SCL signal) to bridge the undefined region of the falling edge of SCL.

Tables 6-2 shows the timing parameters with a maximum value for the standard-speed mode (assuming the clock period of 1ns).

Table 6-2 Standard-Speed Mode: Timing Parameters (Maximum Value)

Purpose	Symbol	Macro Name	Value (In Spec)	VIP Implementation (In ns)
High period of SCL clock	tLOW	SVT_I2C_MAX_CLK_HIGH_SS	-	10000
Low period of SCL clock	tHIGH	SVT_I2C_MAX_CLK_LOW_SS	-	10000
Setup time for the repeated START condition	tSU;STA	SVT_I2C_MAX_SU_STA_SS	-	10000
Setup time for the STOP condition	tSU;STO	SVT_I2C_Max_SU_STO_SS	-	10000
Data-setup time	tSU; DAT	SVT_I2C_MAX_SU_DAT_SS	-	1000
Hold time for the repeated START condition. After this period, the first clock pulse is generated	tHD; STA	SVT_I2C_MAX_HD_STA_SS	-	6000
Data-hold time: for bus devices	tHD; DAT	SVT_I2C_MAX_HD_DAT_SS SVT_I2C_MAX_HD_DAT_MIN_SS	3.45** µs	3450 2000
Bus free time between a STOP and START condition	tBUF	SVT_I2C_MAX_TBUF_TIME_SS	-	10000
Rise time of both SDA and SCL signals	tr	SVT_I2C_MAX_CLK_HIGH_OFFSET_SS	1000 ns	1000
Fall time of both SDA and SCL signals	tf	SVT_I2C_MAX_CLK_LOW_OFFSET_SS	300ns	300

Note

- All values referred to V_{IHmin} and V_{ILmax} levels.
- ** indicates that the maximum tHD; DAT should meet only if the device does not stretch the low period (tLOW) of the SCL signal

6.4.2 Fast-Speed Mode

The fast-speed mode provides a bit rate up to 400 Kbit/s. Tables 6-3 shows the timing parameters with a minimum value for the fast-speed mode (assuming the clock period of 1ns).

Table 6-3 Fast-Speed Mode: Timing Parameters (Minimum Value)

Purpose	Symbol	Macro Name	Value (In Spec)	VIP Implementation (In ns)
High period of SCL clock	tLOW	SVT_I2C_CLK_HIGH_FS	0.6 µs	600
Low period of SCL clock	tHIGH	SVT_I2C_CLK_LOW_FS	1.3 µs	1300
Setup time for the repeated START condition	tSU;STA	SVT_I2C_MIN_SU_STA_FS	0.6 µs	600
Setup time for the STOP condition	tSU;STO	SVT_I2C_MIN_SU_STO_FS	0.6 µs	600
Data-setup time	tSU; DAT	SVT_I2C_MIN_SU_DAT_FS	100* ns	100
Hold time for the repeated START condition. After this period, the first clock pulse is generated	tHD; STA	SVT_I2C_MIN_HD_STA_FS	0.6 µs	600
Data-hold time: for bus devices	tHD; DAT	SVT_I2C_MIN_HD_DAT_FS SVT_I2C_MIN_HD_DAT_MAX_FS	0** µs	300 500
Bus free time between a STOP and START condition	tBUF	SVT_I2C_TBUF_TIME_FS	1.3 µs	4700
Rise time of both SDA and SCL signals	tr	SVT_I2C_MIN_CLK_HIGH_OFFSET_FS	$20 + 0.1C_b^{\#}$	20
Fall time of both SDA and SCL signals	tf	SVT_I2C_MIN_CLK_LOW_OFFSET_FS	$20 + 0.1C_b^{\#}$	20

Note

- All values referred to V_{IHmin} and V_{ILmax} levels.
- * indicates that a fast-mode I2C-bus device can be used in a standard-mode I2C-bus system, but the requirement tSU;DAT >= 250 ns should meet. This is automatically the case if the device does not stretch the low period of the SCL signal. If such a device does stretch the low period of the SCL signal, it must output the next data bit to the SDA line: tr max + tSU;DAT = 1000 + 250 = 1250 ns (as per the standard-mode I2C-bus specification) before the SCL line is released.
- ** indicates that a device must internally provide a hold time of at least 300 ns for the SDA signal (referred to V_{IHmin} of the SCL signal) to bridge the undefined region of the falling edge of SCL.
- # C_b = Total capacitance of one bus line in pF. If mixed with high-speed-mode devices, it allows faster fall-times according to Table 6.

Tables 6-4 shows the timing parameters with a maximum value for the fast-speed mode (assuming the clock period of 1ns).

Table 6-4 Fast-Speed Mode: Timing Parameters (Maximum Value)

Purpose	Symbol	Macro Name	Value (In Spec)	VIP Implementation (In ns)
High period of SCL clock	tLOW	SVT_I2C_MAX_CLK_HIGH_FS	-	10000
Low period of SCL clock	tHIGH	SVT_I2C_MAX_CLK_LOW_FS	-	10000
Setup time for the repeated START condition	tSU;STA	SVT_I2C_MAX_SU_STA_FS	-	1500
Setup time for the STOP condition	tSU;STO	SVT_I2C_MAX_SU_STO_FS	-	1500
Data-setup time	tSU; DAT	SVT_I2C_MAX_SU_DAT_FS	-	500
Hold time for the repeated START condition. After this period, the first clock pulse is generated	tHD; STA	SVT_I2C_MAX_HD_STA_FS	-	1500
Data-hold time: for bus devices	tHD; DAT	SVT_I2C_MAX_HD_DAT_FS SVT_I2C_MAX_HD_DAT_MIN_FS	0.9* µs -	900 800
Bus free time between a STOP and START condition	tBUF	SVT_I2C_TBUF_TIME_FS	-	2500
Rise time of both SDA and SCL signals	tr	SVT_I2C_MAX_CLK_HIGH_OFFSET_FS	300 ns	300
Fall time of both SDA and SCL signals	tf	SVT_I2C_MAX_CLK_LOW_OFFSET_FS	300 ns	300

Note

- All values referred to V_{IHmin} and V_{ILmax} levels.
- * indicates that the maximum tHD;DAT sould meet if the device does not stretch the low period (tLOW) of the SCL signal.

6.4.3 Fast-Mode Plus

The fast-mode plus provides a bit rate up to 1 Mbit/s. Tables 6-5 shows the timing parameters with a minimum value for the fast-mode plus (assuming the clock period of 1ns).

Table 6-5 Fast-Mode Plus: Timing Parameters (Minimum Value)

Purpose	Symbol	Macro Name	Value (In Spec)	VIP Implementation (In ns)
High period of SCL clock	tLOW	SVT_I2C_CLK_HIGH_FM_PLUS	0.5 µs	500
Low period of SCL clock	tHIGH	SVT_I2C_CLK_LOW_FM_PLUS	0.26 µs	260
Setup time for the repeated START condition	tSU;STA	SVT_I2C_MIN_SU_STA_FM_PLUS	0.26 µs	260
Setup time for the STOP condition	tSU;STO	SVT_I2C_MIN_SU_STO_FM_PLUS	0.26 µs	260
Data-setup time	tSU; DAT	SVT_I2C_MIN_SU_DAT_FM_PLUS	50* ns	50
Hold time for the repeated START condition. After this period, the first clock pulse is generated	tHD; STA	SVT_I2C_MIN_HD_STA_FM_PLUS	0.26 µs	260
Data-hold time: for bus devices	tHD; DAT	SVT_I2C_MIN_HD_DAT_FM_PLUS SVT_I2C_MIN_HD_DAT_FM_MAX_PLUS	0 µs	300 500
Bus free time between a STOP and START condition	tBUF	SVT_I2C_TBUF_TIME_FM_PLUS	0.5 µs	500
Rise time of both SDA and SCL signals	tr	SVT_I2C_MIN_CLK_HIGH_OFFSET_FM_PLUS	-	0
Fall time of both SDA and SCL signals	tf	SVT_I2C_MIN_CLK_LOW_OFFSET_FM_PLUS	-	0

Tables 6-6 shows the timing parameters with a maximum value for the fast-mode plus (assuming the clock period of 1ns).

Table 6-6 Fast-Mode Plus: Timing Parameters (Maximum Value)

Purpose	Symbol	Macro Name	Value (In Spec)	VIP Implementation (In ns)
High period of SCL clock	tLOW	SVT_I2C_MAX_CLK_HIGH_FM_PLUS	-	10000
Low period of SCL clock	tHIGH	SVT_I2C_MAX_CLK_LOW_FM_PLUS	-	10000
Setup time for the repeated START condition	tSU;STA	SVT_I2C_MAX_SU_STA_FM_PLUS	-	1500
Setup time for the STOP condition	tSU;STO	SVT_I2C_MAX_SU_STO_FM_PLUS	-	1500
Data-setup time	tSU; DAT	SVT_I2C_MAX_SU_DAT_FM_PLUS	-	500
Hold time for the repeated START condition. After this period, the first clock pulse is generated	tHD; STA	SVT_I2C_MAX_HD_STA_FM_PLUS	-	1500
Data-hold time: for bus devices	tHD; DAT	SVT_I2C_MAX_HD_DAT_FM_PLUS SVT_I2C_MAX_HD_DAT_FM_MIN_PLUS	-	900 800
Bus free time between a STOP and START condition	tBUF	SVT_I2C_MAX_TBUF_TIME_FM_PLUS	-	1000
Rise time of both SDA and SCL signals	tr	SVT_I2C_MAX_CLK_HIGH_OFFSET_FM_PLUS	120 ns	120
Fall time of both SDA and SCL signals	tf	SVT_I2C_MAX_CLK_LOW_OFFSET_FM_PLUS	120* ns	120

Note

- * In the fast-mode plus, the fall time is same for both the output stage and the bus timing. If designers use series resistors, they should allow this while considering the bus timing.
- Currently, the minimum value of macros for the data-hold time is between 300ns to 500ns. And, its maximum value is between 800ns to 900ns.

6.4.4 High-Speed Mode

The high-speed mode provides a bit rate up to 3.4 Mbit/s. Tables 6-7 shows the timing parameters with a minimum value for the high-speed mode (assuming the clock period of 1ns and C_b = 100 pF MAX).

Table 6-7 High-Speed Mode: Timing Parameters (Minimum Value)

Purpose	Symbol	Macro Name	Value (In Spec)	VIP Implementation (In ns)
High period of SCL clock	tLOW	SVT_I2C_CLK_HIGH_HS	60 ns	60
Low period of SCL clock	tHIGH	SVT_I2C_CLK_LOW_HS	160 ns	160
Setup time for the repeated START condition	tSU;STA	SVT_I2C_MIN_SU_STA_HS	160 ns	160
Setup time for the STOP condition	tSU;STO	SVT_I2C_MIN_SU_STO_HS	160 ns	160
Data-setup time	tSU; DAT	SVT_I2C_MIN_SU_DAT_HS	10 ns	10
Hold time for the repeated START condition. After this period, the first clock pulse is generated	tHD; STA	SVT_I2C_MIN_HD_STA_HS	160 ns	160
Data-hold time	tHD; DAT	SVT_I2C_MIN_HD_DAT_HS SVT_I2C_MIN_HD_DAT_MAX_HS	0* ns -	40 50
Rise time of both SDA and SCL signals	tr	SVT_I2C_MIN_CLK_HIGH_OFFSET_HS	10 ns	10
Fall time of both SDA and SCL signals	tf	SVT_I2C_MIN_CLK_LOW_OFFSET_HS	10 ns	10

Note

- All values referred to V_{IHmin} and V_{ILmax} levels.
- For bus line loads C_b between 100 and 400 pF, the timing parameters must be linearly interpolated.
- * indicates that a device must internally provide a data-hold time to bridge the un-defined part between VIH and VIL of the falling edge of the SCLH signal. An input circuit with a threshold as low as possible, for the falling edge, minimizes this hold time.

Tables 6-8 shows the timing parameters with a maximum value for the high-speed mode (assuming the clock period of 1ns and C_b = 100 pF MAX).

Table 6-8 High-Speed Mode: Timing Parameters (Maximum Value)

Purpose	Symbol	Macro Name	Value (In Spec)	VIP Implementation (In ns)
High period of SCL clock	tLOW	SVT_I2C_MAX_CLK_HIGH_HS	-	1000
Low period of SCL clock	tHIGH	SVT_I2C_MAX_CLK_LOW_HS	-	1000
Setup time for the repeated START condition	tSU;STA	SVT_I2C_MAX_SU_STA_HS	-	1000
Setup time for the STOP condition	tSU;STO	SVT_I2C_MAX_SU_STO_HS	-	1000
Data-setup time	tSU; DAT	SVT_I2C_MAX_SU_DAT_HS	-	50
Hold time for the repeated START condition. After this period, the first clock pulse is generated	tHD; STA	SVT_I2C_MAX_HD_STA_HS	-	400
Data-hold time	tHD; DAT	SVT_I2C_MAX_HD_DAT_HS SVT_I2C_MAX_HD_DAT_MIN_HS	70 ns	70 60
Rise time of both SDA and SCL signals	tr	SVT_I2C_MAX_CLK_HIGH_OFFSET_HS	40 ns	40
Fall time of both SDA and SCL signals	tf	SVT_I2C_MAX_CLK_LOW_OFFSET_HS	40 ns	40

6.5 Clock Stretching

I2C SVT supports the following three types of clock stretching:

- Byte-level clock stretching with a random or user-defined value
- Bit-level clock stretching in the address phase with a random or user-defined value
- Bit-level clock stretching in the data phase with a random or user-defined value

You can detect Clock Stretching through print messages as shown:

a. When Clock is stretched by Master/Slave VIP, there are UVM_INFO prints in the simulation log which state the same.

In this case, the following messages are displayed:

   i. For clock stretched by Master: 'Clock stretching started from Master'
   ii. For clock stretched by Slave: 'Clock stretching started from Slave'

b. When clock is stretched by Master/Slave VIP stops, the VIP Checker samples the stretched clock and provides UVM_INFO prints in the simulation log.

In this case, the message 'Clock stretching completed' is displayed.

This section consists of the following sub-sections:

- "Clock Stretching With a Random Value" on page 74
- "Clock Stretching With a User-Defined Value" on page 76
- "Configurable Clock Stretching" on page 77

6.5.1 Clock Stretching With a Random Value

You can generate the stretch-time value via randomization. The `svt_i2c_slave_transaction` as well as svt_i2c_master_transaction class provide the following three variables to support the clock stretching with a random value:

- `enable_random_clk_stretch_time_after_byte`: Enables the byte-level stretching with a random value after each byte irrespective of an address or a data byte.
- `enable_random_clk_stretch_time_addr_byte`: Enables the bit-level stretching in the address phase with a random value.
- `enable_random_clk_stretch_time_data_byte`: Enables the bit-level stretching in the data phase with a random value.

Note

- You should not enable all the above three variables in the same transaction of the slave.
- You can enable one or both variables for the bit-level stretching in the same transaction.

To control the range of randomization as per your requirement, use the following macros:

```
`define SVT_I2C_RAND_CLOCK_STRETCH_MAX  10000 // maximum value
`define SVT_I2C_RAND_CLOCK_STRETCH_MIN  5000  // minimum value
```

These macros are present in the `svt_i2c_common_defines.h` file.

To enable the random-clock stretching, set the variables of the `svt_i2c_slave_transaction` class in the slave sequence as follows :

- To enable the byte-level stretching with a random value, refer the following code snippet:

```
svt_i2c_slave_transaction  tx_xacts_s;//handle of theslavetransaction
...
...
virtual task body();
   `uvm_create(tx_xacts_s)
      tx_xacts_s.enable_random_clk_stretch_time_after_byte = 1;
   `uvm_send(tx_xacts_s)
endtask: body
```

- To enable the bit-level stretching in the address phase with a random value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s; //handle of the slave transaction
...
...
virtual task body();
   `uvm_create(tx_xacts_s)
      tx_xacts_s.enable_random_clk_stretch_time_addr_byte= 1;
   `uvm_send(tx_xacts_s)
 endtask: body
```

- To enable the bit-level stretching in the data phase with a random value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s; //handle of the slave transaction
...
...
virtual task body();
   `uvm_create(tx_xacts_s)
      tx_xacts_s.enable_random_clk_stretch_time_data_byte= 1;
   `uvm_send(tx_xacts_s)
 endtask: body
```

- To enable the bit-level stretching in the address phase as well as the data phase with a random value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s; //handle of the slave transaction
...
...
virtual task body();
   `uvm_create(tx_xacts_s)
      tx_xacts_s.enable_random_clk_stretch_time_addr_byte= 1;
     tx_xacts_s.enable_random_clk_stretch_time_data_byte= 1;
   `uvm_send(tx_xacts_s)
endtask: body
```

6.5.2 Clock Stretching With a User-Defined Value

You can generate the stretch time using a user-defined value. The `svt_i2c_slave_transaction` class as well as svt_i2c_master_transaction provide the following three variables to support the clock stretching with a user-defined value:

- `clk_stretch_time_after_byte`: Enables the byte-level stretching with a user-defined value after each byte irrespective of an address or a data byte.
- `clk_stretch_time_addr_byte`: Enables the bit-level stretching in the address phase with a user-defined value.
- `clk_stretch_time_data_byte`: Enables the bit-level stretching in the data phase with a user-defined value.

Note

- All the above three variables should not have a non-zero value in the same transaction of the slave.
- You can set a non-zero value to one or both variables for the bit-level stretching in the same transaction.

To enable the byte-level stretching with a user-defined value, set a non-zero value to the variables of the `svt_i2c_slave_transaction` and `svt_i2c_master_transaction` class in the slave sequence as follows:

- To enable the byte-level stretching with a user-defined value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s; //handle of the slave transaction
...
...
virtual task body();
   `uvm_create(tx_xacts_s)
      tx_xacts_s.clk_stretch_time_after_byte = 1000;
   `uvm_send(tx_xacts_s)
 endtask: body
```

- To enable the bit-level stretching in the address phase with a user-defined value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s; //handle of the slave transaction
...
...
virtual task body();
   `uvm_create(tx_xacts_s)
      tx_xacts_s.clk_stretch_time_addr_byte= 1500;
   `uvm_send(tx_xacts_s)
 endtask: body
```

- To enable the bit-level stretching in the data phase with a user-defined value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s; //handle of the slave transaction
...
...
virtual task body();
   `uvm_create(tx_xacts_s)
      tx_xacts_s.clk_stretch_time_data_byte= 1000; // user-defined value
   `uvm_send(tx_xacts_s)
 endtask: body
```

- To enable the bit-level stretching in the address phase as well as the data phase with a user-defined value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s; //handle of the slave transaction .
..
...
virtual task body();
   `uvm_create(tx_xacts_s)
      tx_xacts_s.clk_stretch_time_addr_byte= 1;// user-defined value
     tx_xacts_s.clk_stretch_time_data_byte= 1; // user-defined value
   `uvm_send(tx_xacts_s)
 endtask: body
```

6.5.3 Configurable Clock Stretching

You can configure the position of the clock stretching with a random or user-defined value. The `svt_i2c_slave_transaction` as well as svt_i2c_master_transaction class provide the following three variables to support the configurable position of the clock stretching:

- `clk_stretch_bit_level_addr_pos`: Specifies the bit position that you want to stretch in the address phase.
- `clk_stretch_bit_level_data_pos`: Specifies the bit position that you want to stretch in the data phase.
- `clk_stretch_byte_level_pos`: Specifies the byte position that you want to stretch in a complete transaction.

Note

- You should not enable the bit-level stretching and byte-level stretching in the same transaction of the slave.
- You can enable the bit-level stretching in the same transaction in the address phase and the data phase.

There are the following scenarios with the configurable clock stretching:

- "Bit-Level Stretching (1 to 8) in the Address Phase" on page 77
- "Bit-Level Stretching (1 to 8) in the Data Phase" on page 78
- "Byte-Level Stretching [1 to data.size()]" on page 80

6.5.3.1 Bit-Level Stretching (1 to 8) in the Address Phase

You can perform the bit-level stretching in the address phase with the following scenarios:

- With a randomized-stretching position
  - With a hard-coded stretching value
  - With a random-stretching value
- With a user-defined stretching position
  - With a hard-coded stretching value
  - With a random-stretching value

For example:

- To stretch the random bit of the address phase with a user-defined stretched-time value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;       // handle of slave transaction
...
...
virtual task body();
     `uvm_create(tx_xacts_s)
        tx_xacts_s. clk_stretch_time_addr_byte= 10000;
     `uvm_rand_send(tx_xacts_s)
endtask: body
```

- To stretch the random bit of the address phase with a random stretched-time value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;       // handle of slave transaction
...
...
virtual task body();
     `uvm_create(tx_xacts_s)
        tx_xacts_s.enable_random_clk_stretch_time_addr_byte = 1'b1;
     `uvm_rand_send(tx_xacts_s)
endtask: body
```

- To stretch the third bit of the address phase with a random stretched time, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;       // handle of slave transaction
...
...
virtual task body();
     `uvm_create(tx_xacts_s)
        tx_xacts_s.enable_random_clk_stretch_time_addr_byte = 1'b1;
        tx_xacts_s.clk_stretch_bit_level_addr_pos  = 4'b0011;
     `uvm_send(tx_xacts_s)
endtask: body
```

- To stretch the third bit of the address phase with a user defined stretched time, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;  // handle of slave transaction
...
...
virtual task body();
     `uvm_create(tx_xacts_s)
        tx_xacts_s.clk_stretch_bit_level_addr_pos= 4'b0011;
        tx_xacts_s.clk_stretch_time_addr_byte= 10000;
     `uvm_send(tx_xacts_s)
endtask: body
```

Note

In 7-bit addressing mode:

- Eight bit, that is the read or write bit, also gets stretched.
- Start byte also gets stretched.

6.5.3.2 Bit-Level Stretching (1 to 8) in the Data Phase

You can perform the bit-level stretching in the data phase with the following scenarios:

- With a randomized-stretching position

    - With a hard-coded stretching value
    - With a random-stretching value
- With a user-defined stretching position
    - With a hard-coded stretching value
    - With a random-stretching value

For example:

- To stretch the random bit of each data byte with a user-defined stretched-time value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;       // handle of slave transaction
...
...
virtual task body();
    `uvm_create(tx_xacts_s)
        tx_xacts_s. clk_stretch_time_data_byte= 10000;
    `uvm_rand_send(tx_xacts_s)
endtask: body
```

- To stretch the random bit of each data byte with a random stretched-time value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;       // handle of slave transaction
...
...
virtual task body();
    `uvm_create(tx_xacts_s)
        tx_xacts_s.enable_random_clk_stretch_time_data_byte = 1'b1;
    `uvm_rand_send(tx_xacts_s)
endtask: body
```

- To stretch the seventh bit of each data byte with a random stretched-time value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;       // handle of slave transaction
...
...
virtual task body();
    `uvm_create(tx_xacts_s)
        tx_xacts_s.enable_random_clk_stretch_time_data_byte = 1'b1;
        tx_xacts_s.clk_stretch_bit_level_data_pos   = 4'b0111;
    `uvm_send(tx_xacts_s)
endtask: body
```

- To stretch the seventh bit of each data byte with a user-defined stretched-time value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;       // handle of slave transaction
...
...
virtual task body();
    `uvm_create(tx_xacts_s)
       tx_xacts_s.clk_stretch_bit_level_data_pos= 4'b0111;
                                                   // data bit position
       tx_xacts_s.clk_stretch_time_data_byte= 5000; // stretching time
```

```
     `uvm_send(tx_xacts_s)
endtask: body
```

6.5.3.3 Byte-Level Stretching [1 to data.size()]

You can perform the byte-level stretching with the following scenarios:

- With a randomized-stretching position
  - With a hard-coded stretching value
  - With a random-stretching value
- With a user-defined stretching position
  - With a hard-coded stretching value
  - With a random-stretching value

Note The byte-level stretching is applicable for the write command only.

For example:

- To stretch the random byte of a complete transaction with a user-defined stretched-time value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;       // handle of slave transaction
...
...
virtual task body();
     `uvm_create(tx_xacts_s)
         tx_xacts_s. clk_stretch_time_after_byte= 10000;
     `uvm_rand_send(tx_xacts_s)
endtask: body
```

- To stretch the random byte of a complete transaction with a random stretched-time value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;       // handle of slave transaction
...
...
virtual task body();
     `uvm_create(tx_xacts_s)
         tx_xacts_s.enable_random_clk_stretch_time_after_byte = 1'b1;
     `uvm_rand_send(tx_xacts_s)
endtask: body
```

- To stretch the fourth byte of a complete transaction with a random stretched-time value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;  // handle of slave transaction
.........................
 virtual task body();
     `uvm_create(tx_xacts_s)
        tx_xacts_s.enable_random_clk_stretch_time_after_byte = 1'b1;
        tx_xacts_s.clk_stretch_byte_level_pos  = 32'h0004;
     `uvm_send(tx_xacts_s)
endtask: body
```

✦ To stretch the fourth bit of a complete transaction with a user-defined stretched-time value, refer the following code snippet:

```
svt_i2c_slave_transaction tx_xacts_s;       // handle of slave transaction
...
...
virtual task body();
    `uvm_create(tx_xacts_s)
        tx_xacts_s. clk_stretch_byte_level_pos = 32'h0004;
        tx_xacts_s. clk_stretch_time_after_byte= 5000;   // stretching time
    `uvm_send(tx_xacts_s)
endtask: body
```

6.6 Blocking Slave

You can configure the slave as a blocking slave to generate blocking slave transactions. To generate multiple transactions with different configurations in the same sequence, that is back-to-back, configure the slave with different configurations for each transaction of the master in the same sequence. For this, use any of the following two ways in the same sequence:

✦ Enable the `enable_slave_blocking` variable of the `svt_i2c_agent_configuration.sv` file. By default, this variable is disabled.

✦ Call the `get_response()` method in a sequence

Note If you want to configure the slave as a blocking slave, enable the `enable_slave_blocking` variable. If you need a response from the driver, call the `get_response()` method.

Perform the following steps to configure the slave as a blocking slave using the `enable_slave_blocking` variable:

1. Enable the `enable_slave_blocking` variable in the `i2c_base_test.sv` file.

```
cfg.slave_cfg[0].enable_slave_blocking = 1 ;  // Set slave blocking
```

2. Send multiple transactions from the master. For this, refer the following code snippet:

```
virtual task body();
    `uvm_info("body", "Entering...", UVM_LOW)

    // first transaction
      `uvm_do_with(tx_xacts_m,
         {
          tx_xacts_m.cmd                      == I2C_WRITE ;
          tx_xacts_m.addr                      ==`SVT_I2C_SLAVE0_ADDRESS ;
          tx_xacts_m.data.size()          == 5;
          tx_xacts_m.sr_or_p_gen       == 0 ;
          tx_xacts_m.send_start_byte == 0 ;
         })

    // second transaction
      `uvm_do_with(tx_xacts_m,
```

```
            {
             tx_xacts_m.cmd                      == I2C_WRITE ;
             tx_xacts_m.addr                      ==`SVT_I2C_SLAVE0_ADDRESS ;
             tx_xacts_m.data.size()          == 3;
             tx_xacts_m.sr_or_p_gen       == 0 ;
             tx_xacts_m.send_start_byte == 0 ;
            })

       `uvm_info("body", "Exiting ...", UVM_LOW)
     endtask: body
```

3. Configure the slave using sequences for the corresponding transactions of master. For this, refer the following code snippet:

```
virtual task body();
      `uvm_info("body", "Entering...", UVM_LOW)
// first transaction to configure the slave for first transaction of master
       `uvm_create(tx_xacts_s)
                 tx_xacts_s. clk_stretch_time_addr_byte = 1000;
       `uvm_send(tx_xacts_s)

// Second transaction to configure the slave for second transaction of master
    `uvm_create(tx_xacts_s)
        tx_xacts_s. clk_stretch_time_data_byte = 2000;
    `uvm_send(tx_xacts_s)

    `uvm_info("body", "Exiting...", UVM_LOW)
endtask: body
```

Perform the following step to configure the slave as a blocking slave using the `get_response()` method:

After each transaction in master and slave sequences, call the `get_response()`method to get response from the driver.

For master sequences, refer the following code snippet:

```
virtual task body();
    `uvm_info("body","Entering...",UVM_LOW)
    // SVT configuration handle
    svt_configuration cfg;
    // Get the SVT configuration
    p_sequencer.get_cfg(cfg);
    // Cast the SVT configuration handle on the local I2C configuration handle
    if (!$cast(i2c_cfg, cfg)) begin
      `uvm_fatal("body", "Unable to cast the configuration to a
                 svt_i2c_configuration class");
    end
`uvm_do_with( req,
                { req.addr                == `SVT_I2C_SLAVE0_ADDRESS;
                  req.cmd                 == I2C_WRITE;
                  req.data.size()         == 4;
                  req.sr_or_p_gen         == 0;
                  req.send_start_byte     == 0;
                })
   // Call get_response only if configuration attribute, enable_put_response
   // is set 1.
   if(i2c_cfg.enable_put_response == 1)
      get_response(rsp);
```

```
        `uvm_info("body","Exiting...",UVM_LOW)
    endtask: body
```

For slave sequences, refer the following code snippet:

```
virtual task body();
    // SVT configuration handle
    svt_configuration cfg;
    // Get the SVT configuration
    p_sequencer.get_cfg(cfg);
    // Cast the SVT configuration handle on the local I2C configuration handle
    if (!$cast(i2c_cfg, cfg)) begin
      `uvm_fatal("body", "Unable to cast the configuration to a
      svt_i2c_configuration class");
    end
    `uvm_do(req)
    // Call get_response only if configuration attribute, enable_put_response
    // is set 1.
    if(i2c_cfg.enable_put_response == 1)
      get_response(rsp);
endtask: body
```

6.7 User-Defined Directed Data Generation

You can configure the slave for user-specified directed data generation in a generic mode. For this, perform the following steps:

1. Slave configuration settings: Set enable_slave_blocking to 1:

```
cfg.slave_cfg[0].enable_slave_blocking = 1 ;
```

2. Specify the following data value in the `svt_i2c_slave_transaction` class:

```
      data.size() = 5;
      data[0] = 9;
      data[1] = 11;
      data[2] = 3; and so on.
```

Note You need to set the data value in an individual data-array element.

For example, to enable master to read 5 bytes of data from pre-defined data-array elements, refer the following code snippet:

For master sequences:

```
virtual task body();
    `uvm_do_with(tx_xacts_m, {
            tx_xacts_m.cmd              == I2C_READ ;
            tx_xacts_m.addr             ==`SVT_I2C_SLAVE0_ADDRESS ;
            tx_xacts_m.data.size()      == 5;
            tx_xacts_m.sr_or_p_gen      == 0 ;
            tx_xacts_m.send_start_byte == 0 ;
            })
endtask: body
```

For slave sequences:

```
virtual task body();
    `uvm_create(tx_xacts_s)
```

```
            // User Defined data value
            tx_xacts_s.data.size() = 5;
            tx_xacts_s.data[0]     = 9;
            tx_xacts_s.data[1]     = 8;
            tx_xacts_s.data[2]     = 7;
            tx_xacts_s.data[3]     = 6;
            tx_xacts_s.data[4]     = 5;
        `uvm_send(tx_xacts_s)
endtask: body
```

The master gets 5 data bytes as the data specified in the slave data-array elements is 5. If you specify only 4 data bytes in slave data-array elements and read 5 data bytes from the salve, then 5^{th} data byte will be a random-data value.

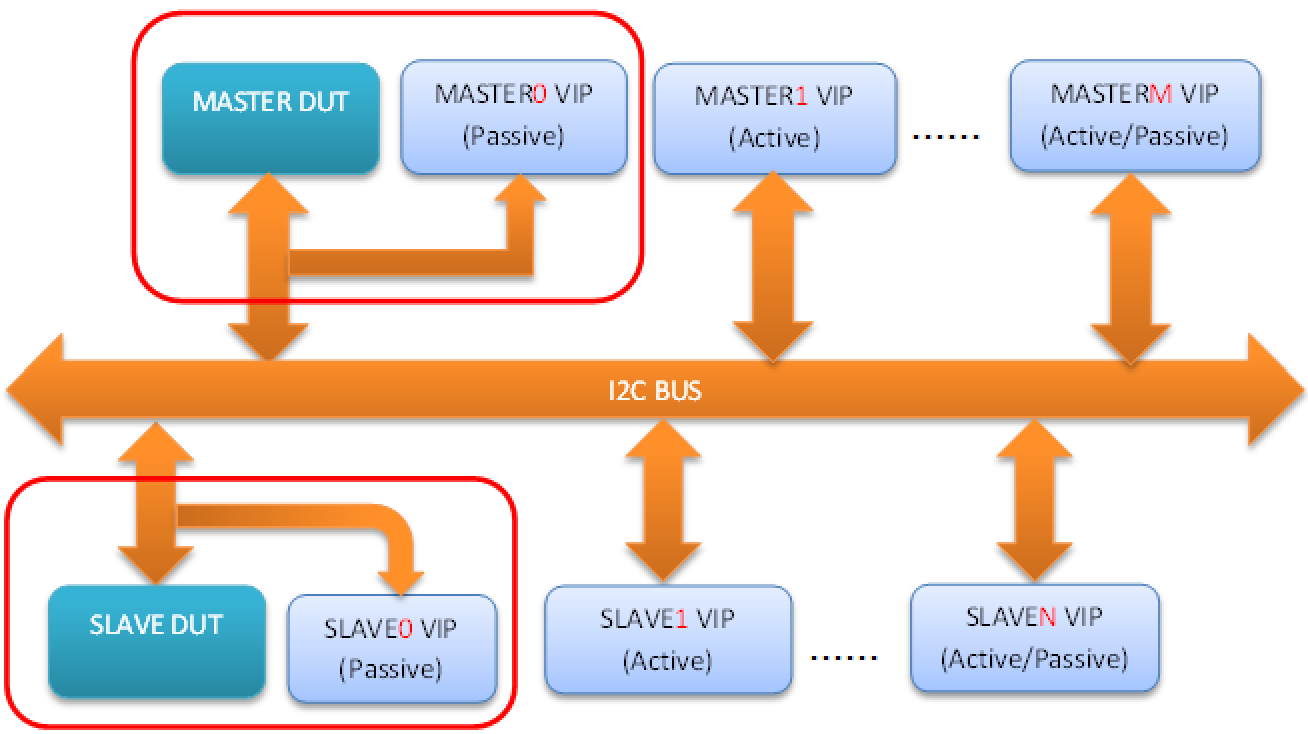
6.8 Verification Topology

You can configure the instances of master agents or slave agents as `ACTIVE` or `PASSIVE` depending on your requirements.

Note In every scenario, at least one instance of the master VIP and the slave VIP is a must.

Figures 6-9 shows the verification topology to model the required number of master and slave agents in the testbench environment.

Figure 6-9. Verification Topology



In this topology, consider the following points:

a. If the DUT works as the master (active component) then the instance of Master0 VIP is configured as the passive agent and only monitor is `ACTIVE`. Instances of other agents, such as Master1 VIP,...., MasterM VIP, are configured as `ACTIVE` or `PASSIVE`.

   On the other side, single or multiple instances of the slave VIP is done to model the required number of slave agents. At least, one slave agent is `ACTIVE`. In the above figure, `N` is (num_slaves-1) and `M` is (num_masters-1).

b. If the DUT works as the slave (active component) then the instance of Slaver0 VIP is configured as the passive agent and only monitor is `ACTIVE`. Instances of other agents, such as Slave1 VIP,...., SlaveN VIP, are configured as `ACTIVE` or `PASSIVE`.

   On the other side, single or multiple instances of the master VIP is done to model the required number of master agents. At least, one master agent is `ACTIVE`. In the above figure, `N` is (num_slaves-1) and `M` is (num_masters-1).

   Using this topology, you can model 1X1, 1XN, MX1, or MXN environments.

Note In passive agents, only monitor is `ACTIVE`. The sequencer and driver of agents are `PASSIVE`.

To model the above topology, the following three steps are required:

a. Instantiation of i2c interface and pass it to all the agents and DUT instance also.

```
/** Instantiate SV Interface for Master/Slave and connect the system clock */
svt_i2c_if   i2c_if (SystemClock);
```

b. Instantiation of the master agent and slave agent (as per requirements) in the top-level file. Pass Agent ID and interface to it.

```
/** Instantiate Master Wrapper to model Master0 Agent*/
svt_i2c_master_wrapper   #(.I2C_AGENT_ID(0)) Master0 (i2c_if);

// Instantiate either or both the master DUT and slave DUT , and pass the interface
// to it.
MASTER_DUT   mst_dut0 (interface connection)   ;
SLAVE_DUT  slv_dut0 (interface connection)   ;

/** Instantiate Master Wrapper to model Master1 Agent*/
svt_i2c_master_wrapper   #(.I2C_AGENT_ID(1)) Master1 (i2c_if);

/** Instantiate Master Wrapper to model MasterM Agent*/
svt_i2c_master_wrapper   #(.I2C_AGENT_ID(M)) MasterM (i2c_if);

/** Instantiate Slave Wrapper to model Slave0 Agent*/
svt_i2c_slave_wrapper   #(.I2C_AGENT_ID(0)) Slave0 (i2c_if);

/** Instantiate Slave Wrapper to model Slave1 Agent*/
svt_i2c_slave_wrapper   #(.I2C_AGENT_ID(1)) Slave1 (i2c_if);

/** Instantiate Slave Wrapper to model Slave2 Agent*/
svt_i2c_slave_wrapper   #(.I2C_AGENT_ID(2)) Slave2 (i2c_if);
……
…….

/** Instantiate Slave Wrapper to model SlaveN Agent*/
svt_i2c_slave_wrapper   #(.I2C_AGENT_ID(N)) SlaveN (i2c_if);
```

c. Set the "num_masters", "num_slaves", and "is_active" configuration parameters in the `env/cust_svt_i2c_system_configuration.sv` file.

```
/** Assign necessary configuration parameters.   */
this.num_masters = M; //M is a positive integer value and must be greater than zero
// and equal to the number of master agent instances created in the top-level file.
this.num_slaves  = N; // N is a positive integer value and must be equal to greater
//than zero and the number of slave agent instances created in the top-level file.

/** Create port configurations */
 this.create_sub_cfgs(this.num_masters, this.num_slaves);
/** Set mode */
 this.master_cfg[0].is_active = 0;
 this.master_cfg[1].is_active = 1;
              ………. 
 this.master_cfg[M].is_active = 1/0;

 this.slave_cfg[0].is_active = 0;
 this.slave_cfg[1].is_active = 1;
 this.slave_cfg[2].is_active = 1/0;
             ………
 this.slave_cfg[N].is_active = 1/0;
```

6.9 Glitch Insertion and Rejection

Glitch insertion and rejection has been implemented for both the Serial Clock (SCL) and the Serial Data (SDA) lines.

For the SCL line, glitch insertion is implemented only for the master, where as glitch rejection is implemented for both the master and the slave.

For the SDA line, glitch insertion and rejection is implemented for the master and the slave.

The implementation details are as follows:

- Four configurable parameters have been added in the `svt_i2c_configuration.sv` file. They are:
  - **enable_glitch_insert_sda** : Determines whether glitch insertion on the SDA line is enabled or not. The possible values for this parameter are:
    - 0 : Glitch insertion feature is disabled
    - 1 : Glitch insertion feature is enabled
  - **glitch_size_sda** : Specifies glitch size and delay from the transition edge of SDA in terms of number of reference clock cycles.
  - **enable_glitch_insert_scl** : Determines whether glitch insertion on the SCL line is enabled or not. The possible values for this parameter are:
    - 0 : Glitch insertion feature is disabled
    - 1 : Glitch insertion feature is enabled
  - **glitch_size_scl** : Specifies glitch size and delay from the transition edge of SCL in terms of number of reference clock cycles.
- Glitch insertion can be enabled or rejected on the SDA line as follows:
  - On SDA line , both the master and the slave can insert the glitch while driving, and both can reject the glitch while sampling. To enable glitch insertion and rejection for I2C Master, set the

value of the `enable_glitch_insert_sda` configuration parameter to 1, and provide an integral value to the `glitch_size_sda` parameter as shown below:

```
// Enable glitch insertion and rejection for Master
cfg.master_cfg[0].enable_glitch_insert_sda = 1'b1;

// Specify the glitch size  for Master
cfg.master_cfg[0].glitch_size_sda = 4;
```

   - To enable glitch insertion and rejection for I2C Slave, set the value of the `enable_glitch_insert_sda` configuration parameter to 1, and provide an integral value to the `glitch_size_sda` parameter as shown below:

```
// Enable glitch insertion and rejection for slave
cfg.slave_cfg[0].enable_glitch_insert_sda = 1'b1;

// Specify the glitch size  for slave
cfg.slave_cfg[0].glitch_size_sda = 2;
```

- Glitch insertion and rejection can be enabled on the SCL line as follows:
   - On the SCL line, only the master can insert the glitch, whereas both the master and the slave can reject the glitch. To enable glitch insertion and rejection for I2C Master, set the value of the `enable_glitch_insert_scl` configuration parameter to 1, and provide an integral value to the `glitch_size_scl` parameter as shown below:

```
// Enable glitch insertion and rejection for Master
cfg.master_cfg[0].enable_glitch_insert_scl = 1'b1;

// Specify the glitch size  for Master
cfg.master_cfg[0].glitch_size_scl = 4;
```

   - To enable glitch rejection for I2C Slave, set the value of the `enable_glitch_insert_scl` configuration parameter to 1, and provide an integral value to the `glitch_size_scl` parameter as shown below:

```
// Enable glitch rejection for Slave
fg.slave_cfg[0].enable_glitch_insert_scl = 1'b1;

// Specify the glitch size  for Slave
cfg.slave_cfg[0].glitch_size_scl = 2;
```

- The glitch inserted by the master, only affects the bits derived by the master. Similarly, the glitch inserted by the slave only affects the bits derived by the slave.

  For example , if glitch is enabled for the master, it will be inserted by the master for the address bits and by the slave for the ACK bit.

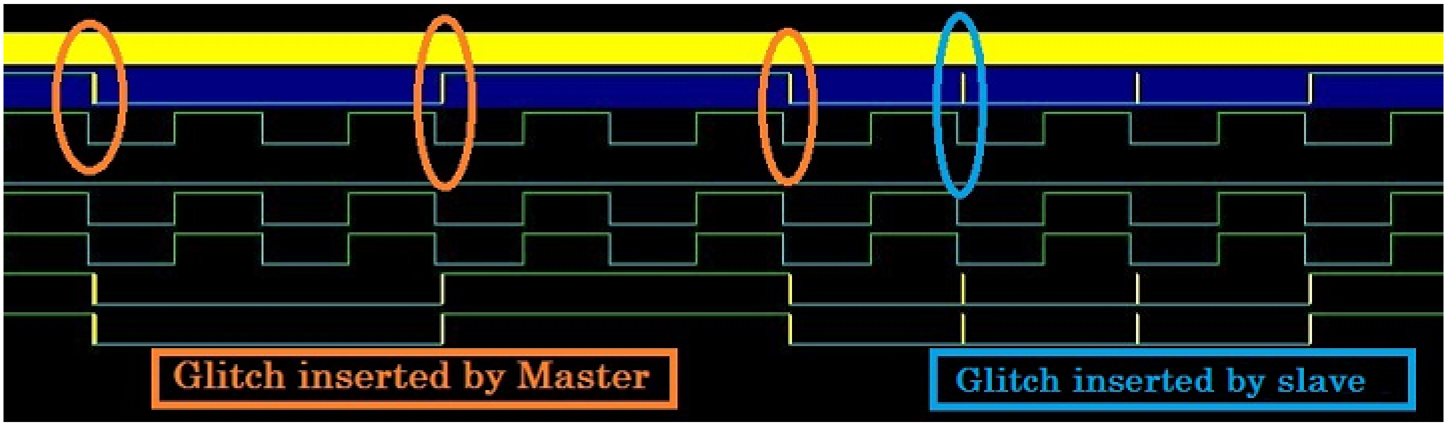
Code to insert glitch on the SDA line from the master and slave, in case of a READ Command is given below:

```
class glitch_insertion_test  extends  i2c_base_test;
  /** UVM component utility macro */
  `uvm_component_utils(glitch_insertion_test)
      /** Class constructor */
      function new(string name = "glitch_insertion_test", uvm_component parent=null);
      super.new(name,parent);
      endfunction: new

      /** build() - Method to build various components */
```

```
        virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info("build_phase","Entered ...",UVM_LOW)

        /** Configure Master and Slave configurations */
        // Enable glitch insertion for Master
            cfg.master_cfg[0].enable_glitch_insert_sda = 1'b1;

        // Enable glitch insertion for slave
            cfg.slave_cfg[0].enable_glitch_insert_sda  = 1'b1;

        // Width of glitch inserted for Master
           cfg.master_cfg[0].glitch_size_sda = 4;

        // Width of glitch inserted for slave
            cfg.slave_cfg[0].glitch_size_sda = 2;

        /** Disable the virtual default sequence on the the virtual sequencer started in
        the i2c_base_test */
            uvm_config_db#(uvm_object_wrapper)::set(this,
        "env.i2c_system_env.sequencer.main_phase", "default_sequence",
        i2c_null_virtual_sequence::type_id::get());

        /** Apply the master random i2c sequence to the i2c master sequencer */
         uvm_config_db#(uvm_object_wrapper)::set(this,
        "env.i2c_system_env.master[0].sequencer.main_phase", "default_sequence",
        i2c_random_master_sequence::type_id::get());

        /** Apply the slave random sequence to the i2c slave sequencer */
         uvm_config_db#(uvm_object_wrapper)::set(this,
        "env.i2c_system_env.slave[0].sequencer.main_phase", "default_sequence",
        i2c_random_slave_sequence::type_id::get());
  `uvm_info("build_phase","Exited ...",UVM_LOW)
endfunction: build_phase
/** This is the main_phase */
  task main_phase(uvm_phase phase);
    `uvm_info("main_phase","Entered ...",UVM_LOW)
    `uvm_info("main_phase", $sformatf("Setting the drain time in the main_phase"),
UVM_NONE)
    phase.phase_done.set_drain_time(this, (5));
    `uvm_info("main_phase","Exited ...",UVM_LOW)
  endtask: main_phase
endclass: glitch_insertion_test
```

Figure 6-10 shows the waveform obtained after glitch is inserted from the master and the slave on the SDA line.

Figure 6-10 Glitch Insertion Waveform on the SDA Line



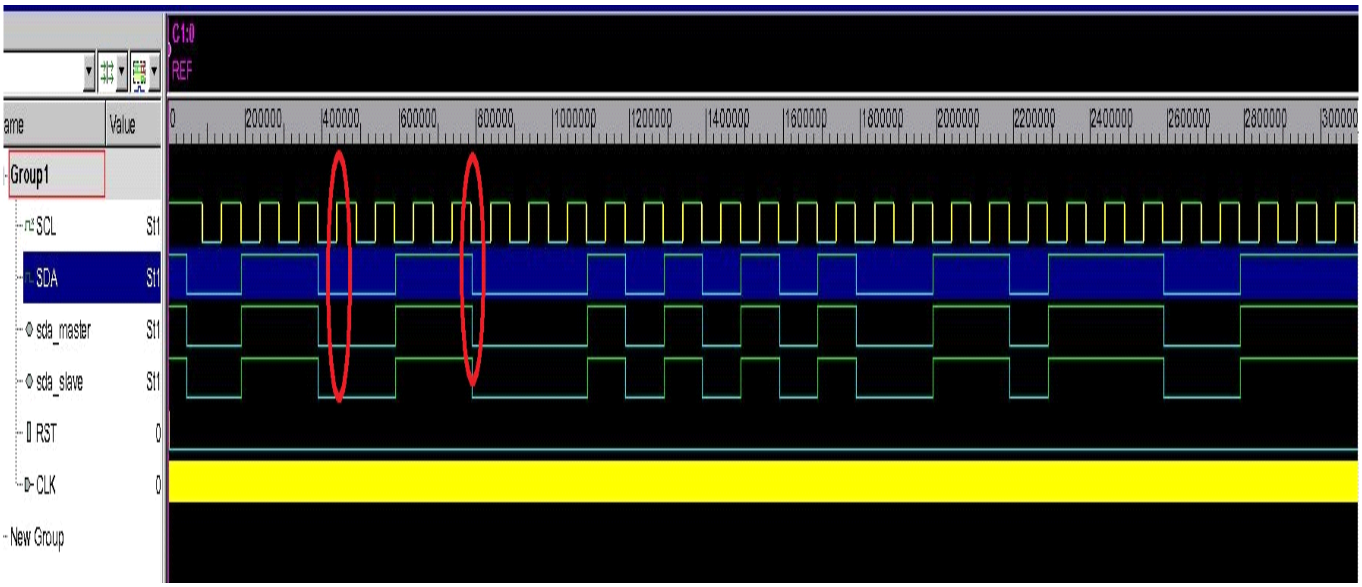
Code to insert glitch on the SCL line from the master and slave, in case of a READ Command is given below:

```
class glitch_insertion_test  extends  i2c_base_test;
  /** UVM component utility macro */
  `uvm_component_utils(glitch_insertion_test)
        /** Class constructor */
        function new(string name = "glitch_insertion_test", uvm_component parent=null);
          super.new(name,parent);
        endfunction: new

        /** build() - Method to build various component */
        virtual function void build_phase(uvm_phase phase);
          super.build_phase(phase);
          `uvm_info("build_phase","Entered ...",UVM_LOW)

      /** Configure master and slave configurations */
      // Enable glitch insertion for master
         cfg.master_cfg[0].enable_glitch_insert_scl = 1'b1;

      // Enable glitch rejection for slave
         cfg.slave_cfg[0].enable_glitch_insert_scl  = 1'b1;

      // Width of glitch inserted for master
         cfg.master_cfg[0].glitch_size_scl = 4;
```

```
      // Width of glitch rejected for slave
      cfg.slave_cfg[0].glitch_size_scl = 2;

      /** Disable the virtual default sequence on the the virtual sequencer started in
      the i2c_base_test */

      uvm_config_db#(uvm_object_wrapper)::set(this,
      "env.i2c_system_env.sequencer.main_phase", "default_sequence",
      i2c_null_virtual_sequence::type_id::get());

      /** Apply the master random i2c sequence to the i2c master sequencer */
       uvm_config_db#(uvm_object_wrapper)::set(this,
      "env.i2c_system_env.master[0].sequencer.main_phase", "default_sequence",
      i2c_random_master_sequence::type_id::get());

      /** Apply the slave random sequence to the i2c slave sequencer */
       uvm_config_db#(uvm_object_wrapper)::set(this,
      "env.i2c_system_env.slave[0].sequencer.main_phase", "default_sequence",
      i2c_random_slave_sequence::type_id::get());
       `uvm_info("build_phase","Exited ...",UVM_LOW)      endfunction: build_phase

      /** This is the main_phase */
      task main_phase(uvm_phase phase);
      `uvm_info("main_phase","Entered ...",UVM_LOW) `uvm_info("main_phase",
      $sformatf("Setting the drain time in the main_phase"), UVM_NONE)
          phase.phase_done.set_drain_time(this, (5));
          `uvm_info("main_phase","Exited ...",UVM_LOW)
  endtask: main_phase
endclass: glitch_insertion_test
```

Figure 6-11 shows the waveform obtained after glitch is inserted from the master and the slave on the SDA line.

Figure 6-11 Glitch Insertion Waveform on the SCL Line



For user defined glitch insertion on SCL and SDA lines, refer the following test cases from intermediate examples:

- ts.glitch_insertion_on_scl_test.sv
- ts.glitch_insertion_on_sda_test.sv

For glitch insertion on SCL and SDA lines with random size and position, refer the following test cases from intermediate examples:

- ts.glitch_insertion_on_scl_with_random_size_and_position_test.sv
- ts.glitch_insertion_on_sda_with_random_size_and_position_test.sv

For glitch rejection from the slave side only, refer the following test case from intermediate examples:

- ts.no_glitch_inserted_from_master_and_glitch_rejection_on_scl_from_slave_test.sv

6.10 EEPROM Mode of Slave

To configure the slave in EEPROM mode, set `slave_type` to SVT_I2C_EEPROM in the `i2c_base_test.sv` file as shown below:

```
cfg.slave_cfg[0].slave_type  = `SVT_I2C_EEPROM ;  // Set Slave as EEPROM
```

Intermediate examples contain an eeprom_test with the file name `ts.eeprom_test.sv`.

In the EEPROM mode, the first two data bytes in every transaction, represent the address of EEPROM from where the Write/Read operation needs to start.

6.10.1 Write Operation

You can write at any address of EEPROM, by sending a transaction with a write command and the address, as the first two bytes of data. For example, the code given below writes at address 16'h0000:

```
/***************************************************************************************/
`uvm_do_with( req,
      {
      req.addr == `SVT_I2C_SLAVE0_ADDRESS ;
      req.cmd == I2C_WRITE ;
      req.data.size() == 6 ;

      // The 2-bytes of data given below represents the EEPROM address from where you
      want to start writing. For example, at the address 16’h0000.

      req.data[0] == 8'h00 ;  // First byte of the starting address 16'h0000
      req.data[1] == 8'h00 ;  // Second byte of the starting address 16'h0000

      // The 4-bytes given below represent the actual data bytes to be written on.

      req.data[2] == 8'hde ;  // Data Byte  0 with EEPROM address = 16’h0000
      req.data[3] == 8'had ;  // Data Byte  1 with EEPROM address = 16’h0001
      req.data[4] == 8'hbe ;  // Data Byte  2 with EEPROM address = 16’h0002
      req.data[5] == 8'haf ;  // Data Byte  3 with EEPROM address = 16’h0003
      req.sr_or_p_gen == 0 ;
      req.send_start_byte  == 0 ;
      })
/***************************************************************************************/
```

6.10.2 Read Operation

You can read from any address of EEPROM by:

- Sending a transaction with a write command and the address from where you want to read, as the first two bytes of data. This sets the EEPROM memory address to the address specified by you. This is also called a dummy write operation.
- Following that, sending a transaction with a read command and the required data size.

For example, the code given below sets the EEPROM address to 16'h0000 using a dummy write:

```
/***************************************************************************************/
    //Setting the EEPROM Memory Address to 16'h0000 (Dummy write by master).
    //The dummy write contains 2 bytes of data, which the represents address of
    //EEPROM from where data bytes needs to be read.


`uvm_do_with( req,
      { req.addr ==`SVT_I2C_SLAVE0_ADDRESS;
      req.cmd == I2C_WRITE ;
      req.data.size() == 2 ;
```

```
        req.data[0] == 8'h00 ;  // First byte of Starting Address 16'h0000
        req.data[1] == 8'h00 ;  // Second byte of Starting Address 16'h0000
        req.sr_or_p_gen == 0 ;
        req.send_start_byte  == 0 ;
        })
```

```
/**************************************************************************************/
```

The pointer is now pointing at EEPROM address 16'h0000. Now, send a transaction with a read command and the required data size to read the data. The code given below reads the data written at address 16'h0000:

```
/**************************************************************************************/
//READ command by Master to read 4 byte of data.
    `uvm_do_with( req,
                { req.addr == `SVT_I2C_SLAVE0_ADDRESS ;
                  req.cmd == I2C_READ ;
                  req.data.size() == 4 ;
                  req.sr_or_p_gen == 0 ;
                  req.send_start_byte == 0 ;
                })
/**************************************************************************************/
```

6.10.3 EEPROM Paging in Slave

Paging mechanism is implemented for 16-bit EEPROM. Various paging schemes can be enabled by using configuration variable:

```
/**
    * This enables paging addressing mode for EEPROM slave
    *  2'b00 - No paging scheme followed
    *  2'b01 - 1 bit paging scheme followed
    *  2'b10 - 2 bit paging scheme followed
    *  2'b11 - 3 bit paging scheme followed
    */
bit [1:0] eeprom_paging;
```

This cfg value needs to be set for VIP slave in the testcase as:

```
cfg.slave_cfg[slave_num] = <required 2bit value>
```

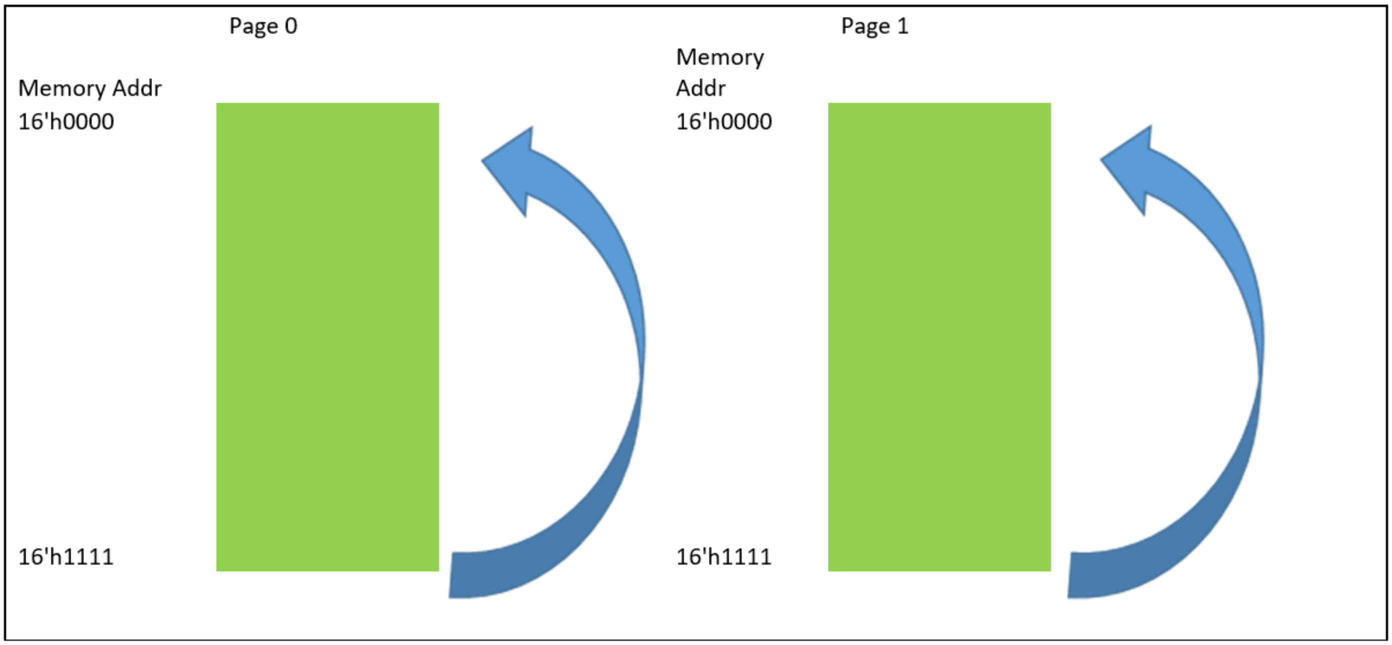
1Mb	1	0	1	0	A2	A1	P0	R/W
	MSB							LSB

The 1-Mb EEPROM requires an 8-bit device address word following a Start condition to enable the chip for a read or write operation. The device address word consists of a mandatory '1010' sequence for the first four most significant bits. This is common to all 2-wire EEPROM devices. The 1-Mb uses the two device address bits, A2 and A1, to allow up to four devices on the same bus. These A2and A1 bits must compare to the corresponding hardwired input pins, A2 and A1. The seventh bit (P0) of the device address is a memory page address bit. This memory page address bit is the most significant bit of the data word address that follows. The eighth bit of the device address is the read/write operation select bit. A read operation is initiated if this bit is high and a write operation is initiated if this bit is low.

On similar lines 2 bit and 3 bit paging schemes have also been implemented.

The read and write operations follow roll-over scenarios for end of page address as follows:

Figure 6-12 EEPROM Paging



Initiating a WRITE operation on EEPROM with 2 bit paging scheme:

```
`uvm_do_with( req,
                { req.addr            == 7'b1010_000; //slave 0 page 0 . Slave 0 will
                                                      // support a total of 4 pages. We
                                                //are referring to page 0 of slave 0 here.
                  req.cmd             == I2C_WRITE;
                  req.data.size()     == 6;
// First two data bytes will constitute 16bit EEPROM memory address
                 req.data[0]         == 8'h00; // First byte of Starting Memory Address
                req.data[1]         == 8'ha8; // Second byte of Starting Memory Address
//Following 4 data bytes represent the data to be written starting at address 16'h00a8.
                  req.data[2]         == 8'haa;  // Data Byte 0
                  req.data[3]         == 8'hbb;  // Data Byte 1
                  req.data[4]         == 8'hcc;  // Data Byte 2
```

```
                  req.data[5]          == 8'hdd;  // Data Byte 3
                  req.sr_or_p_gen      == 0;
                })
```

To initiate a read (to read the 4 byte data written) after above WRITE operation a dummy write needs to be fired to initialize the memory address from which data has to be read.

```
//Dummy WRITE
 `uvm_do_with( req,
                { req.addr             == 7'b1010_000; //slave 0 page 0  ->
eeprom_mem_wr[0][]
                  req.cmd              == I2C_WRITE;
                  req.data.size()      == 2;
// First two data bytes will constitute 16bit EEPROM memory address
                  req.data[0]         == 8'h00; // First byte of Starting Memory Address
                 req.data[1]         == 8'ha8; // Second byte of Starting Memory Address
//Memory address has been initialized to 16'h00a8.
                  req.sr_or_p_gen      == 0;
                })
//ACTUAL READ command
`uvm_do_with( req,
                { req.addr             == 7'b1010_000; //slave 0 page 0  ->
eeprom_mem_wr[0][]
                  req.cmd              == I2C_READ;
//Here the data.size() indicates the number of data bytes that are to be read after
which
//Master will send a NACK to abort the READ operation.
                  req.data.size()      == 4;
                  req.sr_or_p_gen      == 0;
                })
```

Public example for EEPROM mode:

`eeprom_2_bit_paging_wr_rd_test`: This test case runs write and read transactions for 2 bit paging (4 pages) and 2 slaves.

6.11 Bus Clear

In the event where the SDA is stuck LOW, the master sends 9 clock pulses. The device that held the bus LOW should release it sometime within those 9 clocks. If not, you can use the HW reset or cycle power to clear the bus.

This enhancement will be activated on using the define `SVT_I2C_3_0` and is applicable only on the slave side.

You have to add `+define+SVT_I2C_3_0` in the `vcs_build_options` file.

Table 6-9 provides the description of variables to be set for the bus clear feature on the slave side:

Table 6-9 Variables for Bus Clear

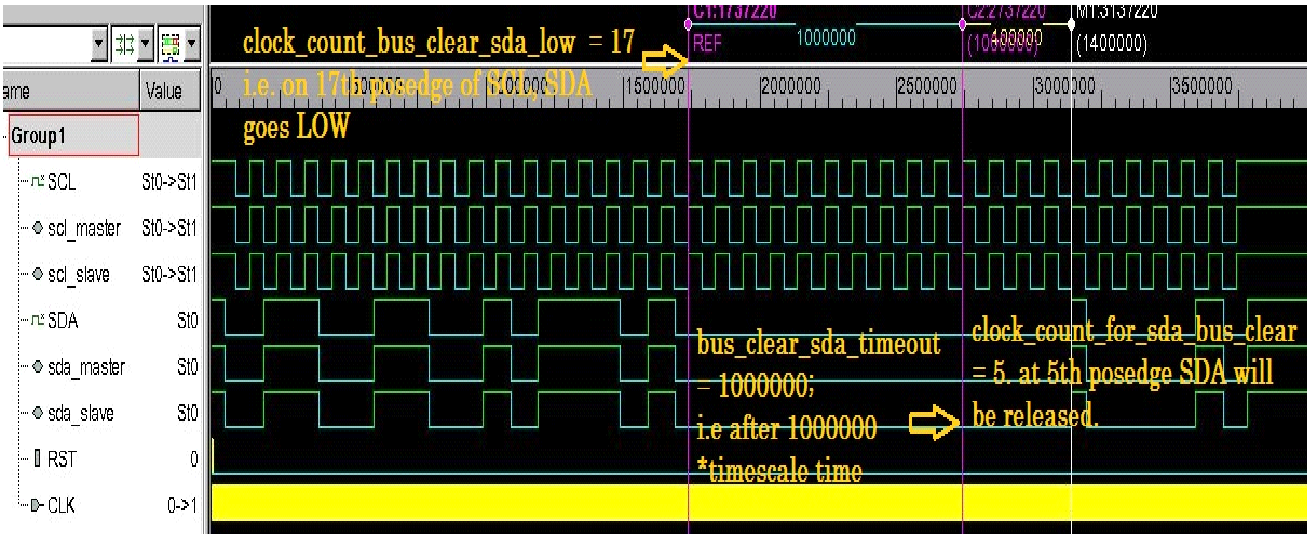
Slave Variable	Description	Valid Range	Remarks
enable_bus_clear_sda	Enables bus clear feature for SDA	0, 1	Configuration Variable (static one-time configuration)
clock_count_bus_clear_sda_low	Specifies the number of SCL posedges on which SDA will be pulled low.	Any integral value	Configuration Variable (static one-time configuration)
bus_clear_sda_timeout	Specifies the delay according to the timescale, after which SDA must remain low to declare that SDA is stuck low	Any integral value	Configuration Variable (static one-time configuration)
clock_count_for_sda_bus_clear	Specifies the number of SCL posedges on which SDA will be released	Any integral value (1-9 as per protocol, but you can set any integral value even outside 1-9)	Configuration Variable (static one-time configuration)

- **To enable SDA bus clear feature** : Pass the define `SVT_I2C_3_0` and set `enable_bus_clear_sda` to 1.
- **To set SDA stuck LOW** : Set value of variable `clock_count_bus_clear_sda_low` such that SDA is pulled down.
- **To detect SDA stuck LOW on bus** : Set configuration variable `bus_clear_sda_timeout` to a number for which time you want the SDA to be held LOW. This will serve as the condition for SDA stuck LOW.
- **To enable Bus Clear** : Set the number of posedges of SCL at which bus clear needs to be done, using the variable `clock_count_for_sda_bus_clear`.

For reference, see the test file `ts.bus_clear_test.sv`.

Figure 6-13 shows the bus clear waveform:

Figure 6-13 Bus Clear Waveform



Note
According to the values of the variables set, protocol violations may be observed from checker (setup/hold time violations or unintended slave behavior) while using the bus clear feature. You need to either set the values accordingly or demote the errors.

Note
Limitations with bus clear:
When Master VIP is used against bus clear and bus clear from Slave is enabled at rising edge of SCL (means Slave SDA pulls LOW at rising edge of SCL), then bus clear must always be used only during Low (Value-0) driving from Master at that SCL clock.
This limitation is because of the fact that when Master is driving 1 and Slave transitions from 1 to 0 at rising edge of SCL, it is interpreted as Start condition in VIP which leads to errors.

When using Master VIP, these are the phases where Bus clear is recommended to be done:

- **Address phase**: In this phase, bus clear can be done only on those address bits which are driven as 0. If address bit is set to 1, then above limitation will follow.
- **Data phase**: Same limitation as #1, when bus clear is enabled, Master should drive a 0 at that particular point.
- **Ack Phase**: When bus clear happens during slave ack phase (where slave has to ack), there is no limitation, because slave to provide ACK, slave already pulled SDA line to 0. But during Slave NACK phase, the scenario remains stuck.

6.12 FM Plus for Master Code in HS Mode

I2C SVT VIP supports Fast mode plus speed for Master code in HS mode. To enable this feature, set the `start_hs_in_fm_plus` configuration parameter for the slave.

For more information, see `ts.fm_mode_plus_in_hs_mode_test.sv` file.

6.13 Mixed Speed Support

I2C SVT VIP supports mixed speed device support in the following scenarios:

Single master and single slave:

Table 6-10 Mixed Speed Support for Single Master and Single Slave

Master (Tx)	Slave (Rx)
SS	FM
FM	SS

Single master and multiple slaves on the same bus:

Table 6-11 Mixed Speed Support for Single Master and Multiple Slaves

Master (Tx)	Slave 0 (Rx)	Slave 1 (Rx)
SS	SS	FM
FM	SS	FM

I2C SVT VIP contains a dedicated example environment for mixed speed mode support, named as `tb_i2c_svt_uvm_single_master_multi_mix_speed_slave`. By default it has a single master and two slaves with different speeds as stated in Table 6-11.

For the master (SS), slave0 (SS) and slave1 (FM) scenario, the test is `ts.mst_std_s0_std_s1_fast_test.sv`.

For the master (FM), slave0 (SS) and slave1 (FM) scenario, the test is `ts.mst_fast_s0_std_s1_fast_test.sv`.

Note

When the master is configured in the fast mode and the slowest slave is in the standard mode, the master should communicate at the speed of the standard mode, which is at the speed of the slowest slave. To configure the master in a speed compatible with the slowest slave, configure the timing parameters of the master as per the slowest slave's timing parameters, that is to the standard mode.

The example environment contains the `svt_i2c_user_defines.svi` file, which contains the macro definitions of all the speed modes. For the master (FM), slave0 (SS) and slave1 (FM) condition, modify the following macros as per the standard speed mode, by specifying the range of the timing parameters:

```
SVT_I2C_MAX_HD_DAT_FS               2000 //changed from 900 to 2000
SVT_I2C_MAX_CLK_HIGH_OFFSET_FS      1000 //changed from 300 to 1000
SVT_I2C_MIN_CLK_HIGH_OFFSET_FS      0    //changed from 20 to 0
SVT_I2C_MIN_CLK_LOW_OFFSET_FS       0    //changed from 20 to 0
SVT_I2C_MAX_SU_STA_FS               10000   //changed from 1500 to 10000
SVT_I2C_MAX_SU_STO_FS               10000   //changed from 1500 to 10000
SVT_I2C_MAX_SU_DAT_FS               1000    //changed from 500 to 1000
SVT_I2C_MAX_HD_STA_FS               6000    //changed from 1500 to 6000
SVT_I2C_MIN_HD_DAT_MAX_FS           1000    //changed from 500 to 1000
SVT_I2C_MAX_HD_DAT_MIN_FS           900     //changed from 800 to 900
SVT_I2C_MAX_TBUF_TIME_FS            10000   //changed from 2500 to 10000
```

By default the `svt_i2c_user_defines.svi` file in the `tb_i2c_svt_uvm_single_master_multi_mix_speed_slave` example environment, has fast mode macro definitions according to the scenario master (FM), slave0 (SS) and slave1 (FM).

Directed value in the range specified in `svt_i2c_user_defines.svi` file will be set in `i2s_base_test.sv` file as follows:

```
/** Setting fast speed mode timing parameters value of master, to communicate in
standard speed mode**/
    cfg.master_cfg[0].scl_high_time_fs     =  4300;
    cfg.master_cfg[0].scl_low_time_fs      =  4700;
    cfg.master_cfg[0].min_su_sta_time_fs   =  10000;
    cfg.master_cfg[0].min_su_sto_time_fs   =  10000;
    cfg.master_cfg[0].min_su_dat_time_fs   =  1000;
    cfg.master_cfg[0].min_hd_sta_time_fs   =  6000;
    cfg.master_cfg[0].min_hd_dat_time_fs   =  1000;
    cfg.master_cfg[0].max_hd_dat_time_fs   =  2000;
    cfg.master_cfg[0].tbuf_time_fs         =  10000;
    cfg.master_cfg[0].scl_high_time_offset_fs =  700;
    cfg.master_cfg[0].scl_low_time_offset_fs  =  300;
```

6.14 Event Pool

I2C SVT VIP has implemented the event pool concept to access the events inside VIP components.

Table 6-12 provides the list of the events.

Table 6-12 List of Events

Name of events in Master's agent Monitor	Name of events in Master's agent Driver	Name of events in Slave's agent Monitor	Name of events in Slave's agent Driver
EVENT_TX_XACT_ENDED	EVENT_START_GENERATED	EVENT_TX_XACT_ENDED	EVENT_START_DETECTED
EVENT_START_CONDITION	EVENT_STOP_GENERATED	EVENT_START_CONDITION	EVENT_STOP_DETECTED
EVENT_STOP_CONDITION	EVENT_ACK_GENERATED	EVENT_STOP_CONDITION	EVENT_ACK_RECEIVED
EVENT_ACK_RECEIVED	EVENT_NACK_GENERATED	EVENT_ACK_RECEIVED	EVENT_NACK_RECEIVED
EVENT_NACK_RECEIVED	EVENT_ACK_RECEIVED	EVENT_NACK_RECEIVED	EVENT_ACK_GENERATED
EVENT_START_BYTE	EVENT_NACK_RECEIVED	EVENT_START_BYTE	EVENT_NACK_GENERATED

Table 6-12 List of Events

Name of events in Master's agent Monitor	Name of events in Master's agent Driver	Name of events in Slave's agent Monitor	Name of events in Slave's agent Driver
EVENT_GEN_CALL	EVENT_REPEATED_START_GENERATED	EVENT_GEN_CALL	
EVENT_SLAVE_ADD_7_B	EVENT_START_BYTE_TRANSMITED	EVENT_SLAVE_ADD_7_B	
EVENT_SLAVE_ADD_10_B	EVENT_GENERAL_CALL_ADDR_SENT	EVENT_SLAVE_ADD_10_B	
EVENT_GET_PACKET	EVENT_GENERAL_CALL_SEC_BYTE_SENT	EVENT_GET_PACKET	
EVENT_REPEATED_START_CONDITION	EVENT_ARBITRATION_LOSS_DETECTED	EVENT_REPEATED_START_CONDITION	

The code given below illustrates the method to access the events of an event pool from a top level scope:

```
// Instance of a top level environment
i2c_basic_env env1;
// Instances of uvm_event to access individual events
uvm_event e1, e2, e3, e4;
// Four instances of event pool to access event pools associated with the four
components - master's monitor, master's driver, slave's monitor and slave's driver
uvm_event_pool ep1, ep2, ep3, ep4;
      // Start of initial block
      initial
        begin
      // Casting the environment
      #1 $cast(env1,uvm_top.find("*.env"));

      // Mapping the master's monitor event pool to uvm_event_pool handle ep1
      ep1 = env1.i2c_system_env.master[0].monitor.event_pool ;

      // Mapping the master's monitor event pool to uvm_event_pool handle ep2
      ep2 = env1.i2c_system_env.master[0].driver.event_pool  ;

      // Mapping the master's monitor event pool to uvm_event_pool handle ep3
      ep3 = env1.i2c_system_env.slave[0].monitor.event_pool ;

      // Mapping the master's monitor event pool to uvm_event_pool handle ep4
      ep4 = env1.i2c_system_env.slave[0].driver.event_pool  ;

      // Printing the events of event pools
      ep1.print();
      ep2.print();
      ep3.print();
      ep4.print();

      // Mapping individual events of event pool to uvm_event handles i.e e1,e2,e3,e4
      etc
      e1 = ep1.get("EVENT_START_CONDITION");
```

```
        e2 = ep2.get("EVENT_START_GENERATED");
        e3 = ep3.get("EVENT_START_CONDITION");
        e4 = ep4.get("EVENT_START_DETECTED");

        // Waiting for triggering of event e4 i.e "EVENT_START_DETECTED"
        e4.wait_trigger();
        // Any method can be placed over here
        $display(" event EVENT_START_DETECTED  has been accessed ");
    end
```

6.15 Analysis Port for Byte Level Data Transmission

Byte level data transmission has been implemented in the master and slave monitors, at the analysis port `data_observed_port`. For byte-by-byte data comparison, connect the analysis port `data_observed_port` with `item_collected_port` of scoreboard in a top level environment.

For example, in the intermediate examples of I2C SVT VIP, `i2c_intermediate_env` is a top level environment in which scoreboard has been instantiated. It's `item_collected_<master/slave>` port is connected to the `data_observed_port` analysis port of the master and the slave monitor as shown below:

```
i2c_system_env.master[0].monitor.data_observed_port.connect(sb.item_collected_master);
i2c_system_env.slave[0].monitor.data_observed_port.connect(sb.item_collected_slave);
```

6.16 Changing Driving Strength of SCL Line from VIP

To change the driving strength of SCL on the I2C interface from the VIP, use the following options in the compile time script (`vcs_build_options`):

```
+define+SVT_I2C_SLV_SCL_STRENGTH=weak0,strong1
+define+SVT_I2C_MST_SCL_STRENGTH=weak0,strong1
```

Any strength can be passed according to the requirement as arguments of macro.

6.17 Changing Driving strength of SDA Line from VIP

To change the driving strength of SDA on the I2C interface from the VIP, use following options in the compile time script (`vcs_build_options`):

```
+define+SVT_I2C_SLV_SDA_STRENGTH=weak1,strong0
+define+SVT_I2C_MST_SDA_STRENGTH=weak1,strong0
```

6.18 Essential Requirements

- In the I2C VIP, the RST pin is active HIGH, and it requires a toggle from 0 -> 1 -> 0 before the START condition. It should be in LOW state during simulation.
- At least one master and one slave agent instantiation is essential for VIP usage. If you are using a single master and a single slave environment, and one component is your DUT, then the corresponding VIP AGENT should be in PASSIVE mode.
- By default, VIP supports data size up to 4KBytes. If you want to increase the data size, then reconfigure the value of `I2C_NVS_MAX_DATA_ARR_SIZE`. Data size has been verified till 8KBytes.

  For example,

  For setting data bytes equal to 8 KBytes (8192 bytes), `I2C_NVS_MAX_DATA_ARR_SIZE` needs to be set to 49152 (49152 *6).

Way to access: `defparam <wrapper instance name>.Master.Master.I2C_NVS_MAX_DATA_ARR_SIZE = <Number of bytes *6>`

```
defparam Master.Master.Master.I2C_NVS_MAX_DATA_ARR_SIZE = 614400;
defparam Slave.Slave.Slave.I2C_NVS_MAX_DATA_ARR_SIZE = 614400;
```

- By default all timing parameters have value with respect to 1 GHz reference clock. For example, 1ns reference Clock period. It is recommended to pass 1 GHz reference clock to VIP, whereas rest of the components in the User's system level environment can operate on their own frequency.
- When using the High speed mode the configuration parameters `start_hs_in_ss` and start_hs_in_fm_plus should be configured identical for each VIP component (active/passive) even if the BUS speed for all components are different.

Example:

Case 1: If master code has to be sent in Slow speed mode

```
foreach(cfg.master_cfg[i]) begin
cfg.master_cfg[i].start_hs_in_ss = 1;
cfg.master_cfg[i].start_hs_in_fm_plus = 0;
end
foreach(cfg.slave_cfg[i]) begin
cfg.slave_cfg[i].start_hs_in_ss = 1;
cfg.slave_cfg[i].start_hs_in_fm_plus = 0;
end
```

Case 2: If master code has to be sent in Fast speed mode

```
foreach(cfg.master_cfg[i]) begin
cfg.master_cfg[i].start_hs_in_ss = 0;
cfg.master_cfg[i].start_hs_in_fm_plus = 0;
end
foreach(cfg.slave_cfg[i]) begin
cfg.slave_cfg[i].start_hs_in_ss = 0;
cfg.slave_cfg[i].start_hs_in_fm_plus = 0;
end
```

Case 3: If master code has to be sent in Fast speed plus mode

```
foreach(cfg.master_cfg[i]) begin
cfg.master_cfg[i].start_hs_in_ss = 0;
cfg.master_cfg[i].start_hs_in_fm_plus = 1;
end
foreach(cfg.slave_cfg[i]) begin
cfg.slave_cfg[i].start_hs_in_ss = 0;
cfg.slave_cfg[i].start_hs_in_fm_plus = 1;
end
```

Note
It is recommended to manipulate all the timing parameters value accordingly in-order to update the reference frequency.

For more information, see Frequently Asked Questions.

6.19 I2C UVM Scenario Reference Guide

Table 6-13 UVM Scenario Reference Guide

Scenario	Configuration Attributes Used	Reference Tests
To configure Slave Address	slave_address	<intermediate_example>/env/i2c_base_test.sv
To configure type of Slave	slave_type	<intermediate_example>/env/i2c_base_test.sv <intermediate_example>/tests/ts.eeprom_test.sv
To configure Bus Speed	bus_speed	<intermediate_example>/env/i2c_base_test.sv
To configure 10-bit addressing	enable_10bit_addr	<intermediate_example>/env/i2c_base_test.sv
To Enable bus clear	enable_bus_clear_sda bus_clear_sda_timeout	<intermediate_example>/tests/ts.bus_clear_test.sv
To enable Glitch Insertion at SDA	enable_glitch_insert_sda glitch_size_sda	ts.glitch_insertion_on_sda_test.sv ts.glitch_insertion_on_sda_with_random_size_and_position_test.sv
To enable Glitch Insertion at SCL	enable_glitch_insert_scl glitch_size_scl	ts.glitch_insertion_on_scl_test.sv ts.glitch_insertion_on_scl_with_random_size_test.sv

Table 6-14 UVMUVM Scenario Reference Guide (For Master Only)

Scenario	Reference Tests	Transction Attributes	Stimulus
Set Master to send READ Command	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.send_multi_transaction_with_start_byte_and_repeated_start_test.sv	cmd = `I2C_READ	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_multi_transaction_with_repeated_start_and_start_byte_sequence.sv
Set Master to send WRITE Command	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.send_multi_transaction_with_start_byte_and_repeated_start_test.sv	cmd = `I2C_WRITE	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_multi_transaction_with_repeated_start_and_start_byte_sequence.sv
Set Master to send GENERAL_CALL Command	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.general_call_test.sv	cmd = `I2C_GEN_CALL	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_gen_call_sequence.sv

Scenario	Reference Tests	Transction Attributes	Stimulus
Set Master to generate particular Slave Address	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.send_multi_transaction_with_start_byte_and_repeated_start_test.sv	addr	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_multi_transaction_with_repeated_start_and_start_byte_sequence.sv
Set Master to send 10-bit Address	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.send_multi_transaction_with_start_byte_and_repeated_start_test.sv	addr_10bit = 0	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_multi_transaction_with_repeated_start_and_start_byte_sequence.sv
Set Master to send 10-bit Address	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.directed_test_for_10_bit_addressing_test.sv	addr_10bit = 1	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_10_bit_addressing_sequence.sv
Set Master to generate N number of data bytes	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.send_multi_transaction_with_start_byte_and_repeated_start_test.sv	data.size()	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_multi_transaction_with_repeated_start_and_start_byte_sequence.sv
Set Master to generate User defined data value in case of WRITE command	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.send_multi_transaction_with_start_byte_and_repeated_start_test.sv	data[]	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_multi_transaction_with_repeated_start_and_start_byte_sequence.sv
Set Master to generate repeated Start condition	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.send_multi_transaction_with_start_byte_and_repeated_start_test.sv	sr_or_p_gen = 1	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_multi_transaction_with_repeated_start_and_start_byte_sequence.sv
Set Master to generate Stop condition	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.send_multi_transaction_with_start_byte_and_repeated_start_test.sv	sr_or_p_gen = 0	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_multi_transaction_with_repeated_start_and_start_byte_sequence.sv
Set master to send Start Byte	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.send_multi_transaction_with_start_byte_and_repeated_start_test.sv	send_start_byte = 1	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_multi_transaction_with_repeated_start_and_start_byte_sequence.sv
Set Master to send second byte general call	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.general_call_test.sv	sec_byte_gen_call	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_gen_call_sequence.sv

Scenario	Reference Tests	Transction Attributes	Stimulus
Set Master to abort it's transaction if it loss the Arbitration	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_multi_master_multi_slave/tests/ts.base_test.sv	abort_if_arb_lost	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_UVM_multi_master_multi_slave/env/i2c_default_mst_sequence.sv
Enable Arbitration for Master	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_multi_master_multi_slave/tests/ts.base_test.sv	arbitrate	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_UVM_multi_master_multi_slave/env/i2c_default_mst_sequence.sv
Set Master to retry if it receive NACK from Slave	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.directed_test_for_retry_if_nack_test.sv	retry_if_nack	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_mst_retry_if_nack_sequence.sv

Table 6-15 UVM Scenario Reference Guide (For Slave Only)

Scenario	Reference Tests	Transction Attributes	Stimulus
Set Slave to send User Defined data for READ command	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.user_conf_data_test.sv	data[]	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/env/i2c_slave_user_conf_sequence.sv
Set Slave to send NACK for Particular Data byte	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.directed_test_for_nack_for_data_test.sv	nack_data**	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_slv_nack_data_sequence.sv
Set Slave to send NACK for it's Address	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.directed_test_for_retry_if_nack_test.sv	nack_addr = 1	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_slv_nack_address_sequence.sv
Set Slave to send NACK N number of times when Master is retrying	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/tests/ts.directed_test_for_retry_if_nack_test.sv	nack_addr_count	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_basic_sys/env/i2c_slv_nack_address_sequence.sv
Enable clock stretching after each byte	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/tests/ts.directed_clock_stretching_after_each_byte_test.sv	clk_stretch_time_after_byte	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/env/i2c_slave_directed_clock_stretch_after_each_byte_sequence.sv

Scenario	Reference Tests	Transction Attributes	Stimulus
Enable clock stretching after each address bit	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/tests/ts.directed_clock_stretching_after_each_addr_bit_test.sv	clk_stretch_time_addr_byte	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/env/i2c_slave_directed_clock_stretch_after_each_addr_bit_sequence.sv
Enable clock stretching after each data bit	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/tests/ts.directed_clock_stretching_after_each_data_bit_test.sv	clk_stretch_time_data_byte	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/env/i2c_slave_directed_clock_stretch_after_each_data_bit_sequence.sv
Enable clock stretching with random stretching time after each byte	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/tests/ts.random_clock_stretching_after_each_byte_test.sv	enable_random_clk_stretch_time_after_byte	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/env/i2c_slave_random_clock_stretch_after_each_byte_sequence.sv
Enable clock stretching with random stretching time after each address bit	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/tests/ts.random_clock_stretching_after_each_address_bit_test.sv	enable_random_clk_stretch_time_addr_byte	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/env/i2c_slave_random_clock_stretch_after_each_address_bit_sequence.sv
Enable clock stretching with random stretching time after each data bit	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/tests/ts.random_clock_stretching_after_each_data_bit_test.sv	enable_random_clk_stretch_time_data_byte	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/env/i2c_slave_random_clock_stretch_after_each_data_bit_sequence.sv
Enable clock stretching with random bit level position in address	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/tests/ts.random_clock_stretching_at_specified_address_bit_test.sv	clk_stretch_bit_level_addr_pos	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/env/i2c_slave_random_clock_stretch_at_specified_address_bit_sequence.sv
Enable clock stretching with random bit level position in data byte	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/tests/ts.random_clock_stretching_at_specified_byte_test.sv	clk_stretch_bit_level_data_pos	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/env/i2c_slave_random_clock_stretch_at_specified_data_bit_sequence.sv
Enable clock stretching with random byte level position of data bytes	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/tests/ts.random_clock_stretching_at_specified_data_bit_test.sv	clk_stretch_byte_level_pos	$DESIGNWARE_HOME/examples/sverilog/i2c_svt/tb_i2c_svt_uvm_intermediate_sys/env/i2c_slave_random_clock_stretch_at_specified_byte_sequence.sv

Note
** Nack response to data is supported and verified till only 8192 data bytes. Possible values for nack_data = 1 to 8192.

6.20 SMBUS Configuration

Table 6-16 SMBUS Configuration List

Variable	Enum Value
bus_type (svt_i2c_configuration)	SMBUS_BUS (Default Value = I2C_BUS)
smbus_unique_id	128 bit smbus udid
t_reset_detect_time	`SVT_SMBUS_RESET_DETECT_TIME_VALUE (Defined value = 2500000000 (2.5s))
t_timeout_min	`SVT_SMBUS_TTIMEOUT_MIN_VALUE (Defined Value = 25000000 (25ms))
t_timeout_max	`SVT_SMBUS_TTIMEOUT_MAX_VALUE (Defined Value = 35000000 (30ms))
t_low_sext	`SVT_SMBUS_TLOW_SEXT_VALUE (Defined Value = 25000000 (25ms))
t_low_mext	`SVT_SMBUS_TLOW_MEXT_VALUE (Defined Value = 10000000 (30ms))
t_low_min_100khz	`SVT_SMBUS_TLOW_MIN_100KHZ (Defined Value = 470 (4.7us))
t_low_min_400khz	`SVT_SMBUS_TLOW_MIN_400KHZ (Defined Value = 130 (1.3us))
t_low_min_1mhz	`SVT_SMBUS_TLOW_MIN_1MHZ (Defined Value = 50 (0.5us))
t_setup_data_100khz	`SVT_SMBUS_DATA_SETUP_100KHZ (Defined Value = 250 (250ns))
t_setup_data_400khz	`SVT_SMBUS_DATA_SETUP_400KHZ (Defined Value = 100 (100ns))
t_setup_data_1mhz	`SVT_SMBUS_DATA_SETUP_1MHZ (Defined Value = 50(50ns))
smbus_pec_seed_value	8'hFF
smbus_pec_support	1
smbus_send_byte_cmd_code	8'ha0
smbus_receive_byte_cmd_code	8'ha1
smbus_write_byte_cmd_code	8'ha2
smbus_read_byte_cmd_code	8'ha4
smbus_process_call_cmd_code	8'ha6

Table 6-16 SMBUS Configuration List (Continued)

Variable	Enum Value
smbus_write_word_cmd_code	8'ha3
smbus_read_word_cmd_code	8'ha5
smbus_block_write_cmd_code	8'ha7
smbus_block_read_cmd_code	8'ha8
smbus_block_process_cmd_code	8'ha9
smbus_write_32_cmd_code	8'hab
smbus_write_64_cmd_code	8'had
smbus_read_32_cmd_code	8'hac
smbus_read_64_cmd_code	8'hae
smbus_send_byte_pec_cmd_code	8'hb0
smbus_receive_byte_pec_cmd_code	8'hb1
smbus_write_byte_pec_cmd_code	8'hb2
smbus_read_byte_pec_cmd_code	8'hb4
smbus_process_call_pec_cmd_code	8'hb6
smbus_write_word_pec_cmd_code	8'hb3
smbus_read_word_pec_cmd_code	8'hb5
smbus_block_write_pec_cmd_code	8'hb7
smbus_block_read_pec_cmd_code	8'hb8
smbus_block_process_pec_cmd_code	8'hb9
smbus_write_32_pec_cmd_code	8'hbb
smbus_write_64_pec_cmd_code	8'hbd
smbus_read_32_pec_cmd_code	8'hbc
smbus_read_64_pec_cmd_code	8'hbe
smbus_host_notify_cmd_code	8'haa
smbus_general_arp_arg	8'h00
smbus_directed_arp_arg	8'h01
enable_tlow_100khz_check	0
enable_tlow_400khz_check	0
enable_tlow_1mhz_check	0

Table 6-16 SMBUS Configuration List (Continued)

Variable	Enum Value
is_smbus_host	0

Table 6-17 SMBUS Command List

Name of the Command	Enum Command	PEC (Supported/Not Supported)	Command Code Value
Send Byte	SMBUS_SND_BYTE_CMD	Not Supported	8'ha0
Receive Byte	SMBUS_RCV_BYTE_CMD	Not Supported	8'ha1
Write Byte	SMBUS_WR_BYTE_CMD	Not Supported	8'ha2
Write Word	SMBUS_WR_WORD_CMD	Not Supported	8'ha3
Read Byte	SMBUS_RD_BYTE_CMD	Not Supported	8'ha4
Read Word	SMBUS_RD_WORD_CMD	Not Supported	8'ha5
Process Call	SMBUS_PROCESS_CMD	Not Supported	8'ha6
Write Block	SMBUS_WR_BLOCK_CMD	Not Supported	8'ha7
Read Block	SMBUS_RD_BLOCK_CMD	Not Supported	8'ha8
Block Write Read Process call	SMBUS_PROCESS_BLOCK_CMD	Not Supported	8'ha9
Host notify command	SMBUS_HOST_NOTIFY_CMD	Not Supported	8'haa
Write 32	SMBUS_WR32_CMD	Not Supported	8'hab
Read 32	SMBUS_RD32_CMD	Not Supported	8'hac
Write 64	SMBUS_WR64_CMD	Not Supported	8'had
Read 64	SMBUS_RD64_CMD	Not Supported	8'hae
Send Byte with PEC	SMBUS_SND_BYTE_CMD_PEC	Supported	8'hb0
Receive Byte with PEC	SMBUS_RCV_BYTE_CMD_PEC	Supported	8'hb1
Write Byte with PEC	SMBUS_WR_BYTE_CMD_PEC	Supported	8'hb2
Write Word with PEC	SMBUS_WR_WORD_CMD_PEC	Supported	8'hb3
Read Byte with PEC	SMBUS_RD_BYTE_CMD_PEC	Supported	8'hb4
Read Word with PEC	SMBUS_RD_WORD_CMD_PEC	Supported	8'hb5
Process Call with PEC	SMBUS_PROCESS_CMD_PEC	Supported	8'hb6
Write Block with PEC	SMBUS_WR_BLOCK_CMD_PEC	Supported	8'hb7

Table 6-17 SMBUS Command List (Continued)

Name of the Command	Enum Command	PEC (Supported/Not Supported)	Command Code Value
Read Block with PEC	SMBUS_RD_BLOCK_CMD_PEC	Supported	8'hb8
Block Write Read Process call with PEC	SMBUS_PROCESS_BLOCK_CMD_PEC	Supported	8'hb9
Write 32 with PEC	SMBUS_WR32_CMD_PEC	Supported	8'hbb
Read 32 with PEC	SMBUS_RD32_CMD_PEC	Supported	8'hbc
Write 64 with PEC	SMBUS_WR64_CMD_PEC	Supported	8'hbd
Read 64 with PEC	SMBUS_RD64_CMD_PEC	Supported	8'hbe
Quick command	SMBUS_QUICK_CMD	Not Supported	NA
SMB Alert	SMBUS_SMBALRT_CMD	Not Supported	NA
Notify ARP Master	SMBUS_NOTIFY_ARP_MASTER_CMD	Supported	8'hbf

Note

- The command code values mentioned in the table are fixed for the current release, but will be configurable in future.
- The command code is not applicable for 'Quick Command' and 'SMBUS Alert Command'.
- Host Notify commands are supported.

6.20.1 PEC Error Insertion Example

Pec Error could be inserted through Exception Implemented. Name of Exception is `svt_i2c_slave_transaction_exception::PEC_ERROR`, Example for this could be find in sequence `smbus_pec_error_sequence` provided in `cust_smbus_sequence_collection.sv`.

6.20.2 Analysis Port Usage

For analysis Port usage example test is the `smbus_scoreboard_dir_test`.

Where a scoreboard (`env/i2c_scoreboard.sv`) is connected between a master and a slave. In transaction object provided by these ports contain a data variable. This variable contains byte by byte whole transaction starting for Slave address, first byte (location 0) to end byte that is PEC.

6.20.3 ARP Process

For ARP process, 'Prepare to ARP' must be sent first followed by other ARP commands.

Once all the slaves have valid addresses, 'Prepare to ARP' can come only after General RESET or if all the allocated addresses have been RESET.

6.20.3.1 ARP Addressing

Bit [7:6] of Device Capabilities	Device Type		Supported ARP Commands	Response to Gen GET UDID	Response to Assign address	Response to Directed Get UDID	Response to RESET (Directed/General)
00b	Fixed Address Device	Fixed and discoverable	PREPARE TO ARP GENERAL GET UDID GENERAL RESET ASSIGN ADDRESS (reassignment not permitted) DIRECTED GET UDID DIRECTED RESET	1. Always returns build time configured address along with UDID. 2. After Assign Address, device will not respond to Gen. GET UDID unless RESET happens.	RETURNS Address allocated through Assign Address cmd	1. ACK only if address allocation through ASSIGN ADDRESS is done. 2. Always returns build time configured address along with UDID	1.Always ACKs. 2. No Change in address value.
		Fixed, not discoverable	Prep to ARP DIRECTED GET UDID	NACK	NACK	1. Always ACKs 2. Always returns build time configured address along with UDID	NACK
		Non ARP Capable	No ARP Command supported	NACK	NACK	NACK	NACK

Bit [7:6] of Device Capabilities	Device Type		Supported ARP Commands	Response to Gen GET UDID	Response to Assign address	Response to Directed Get UDID	Response to RESET (Directed/General)
01b	Dynamic and Persistent Address Device		PREPARE TO ARP GENERAL GET UDID GENERAL RESET ASSIGN ADDRESS (reassignment possible) DIRECTED GET UDID DIRECTED RESET	1. Returns 'hFF for address along with UDID, if Address not allocated through ASSIGN ADDRESS. 2. If address allocation through ASSIGN ADDRESS has already happened, Returns address allocated through ASSIGN ADDRESS along with UDID(even after RESET , since PSA). 3. After Assign Address, device will not respond to Gen. GET UDID unless RESET happens.	1.always Acks Assign address. 2.Always updates the address	1. ACK only if address allocation through ASSIGN ADDRESS is done. 2. Returns address allocated through ASSIGN ADDRESS along with UDID	1.Always ACKs. 2.No change in address value

Bit [7:6] of Device Capabilities	Device Type	Supported ARP Commands	Response to Gen GET UDID	Response to Assign address	Response to Directed Get UDID	Response to RESET (Directed/General)
10b	Dynamic and volatile	PREPARE TO ARP GENERAL GET UDID GENERAL RESET ASSIGN ADDRESS (reassignment possible) DIRECTED GET UDID DIRECTED RESET	1. Returns 'hFF for address along with UDID, if Address not allocated through ASSIGN ADDRESS or RESET has been done. 2. After Assign Address, device will not respond to Gen. GET UDID unless RESET happens.	1.always Acks Assign address. 2.Always updates the address	1. ACK only if address allocation through ASSIGN ADDRESS is done. 2. Returns address allocated through ASSIGN ADDRESS along with UDID	1.Always ACKs. 2.Resets address to 'Hff

Bit [7:6] of Device Capabilities	Device Type	Supported ARP Commands	Response to Gen GET UDID	Response to Assign address	Response to Directed Get UDID	Response to RESET (Directed/General)
11b	Random number	PREPARE TO ARP GENERAL GET UDID GENERAL RESET ASSIGN ADDRESS (reassignment possible) DIRECTED GET UDID DIRECTED RESET	1. Returns 'hFF for address along with UDID, if Address not allocated through ASSIGN ADDRESS. 2. After Assign Address, device will not respond to Gen. GET UDID unless RESET happens. 3. After RESET, Returns 'hFF for address along with updated UDID(last 16 bits randomized)	1.always Acks Assign address. 2.Always updates the address	1. Returns address allocated through ASSIGN ADDRESS along with UDID	1.Always ACKs. 2.Resets address to 'Hff 3. UDID is updated to a random value for UDID bits [31:0].

Note

For configuring the fixed address device, an agent configuration `smbus_fixed_addr_device_type` needs to be set to a valid value. For reference, see above table. For ARP flow, refer the SMBus Specification Fig. 62.

See Sequence smbus_prepare_to_arp_sequence.

Table 6-18 Example Sequence of Events and Response of All Types of Slave Devices

Event	Dynamic Persistent	Dynamic Volatile	Random Number	Fixed, discoverable	Fixed, not discoverable
General RESET	ACK	ACK	ACK, UDID[31:0] randomized	ACK	NACK

Event	Dynamic Persistent	Dynamic Volatile	Random Number	Fixed, discoverable	Fixed, not discoverable
Gen. GET UDID	ACK, Returns address 'Hff	ACK, Returns address 'Hff	ACK, Returns address 'Hff	ACK, Returns address Address configured from test ('h55)	NACK
ASSIGN ADDR	ACK, Address updated to 'h55	ACK, Address updated to 'h55	ACK, Address updated to `h55	ACK, Address not updated (still 'h55)	NACK
Gen. RESET	ACK, Address not RESET (still 'h55)	ACK, Address RESET to `hFF	ACK, Address RESET to `hFF , UDID[31:0] randomized	ACK, Address not RESET (still 'h55)	NACK
Gen. GET UDID	ACK, Returns Address 'h55	ACK, Returns address 'Hff	ACK, Returns address 'Hff UDID[31:0] randomized	ACK, Address not RESET (still 'h55)	NACK
ASSIGN ADDRESS	ACK, Address updated to 'h51	ACK, Address updated to 'h51	ACK, Address updated to `h51	ACK, Address not updated (still 'h55)	NACK
Directed GET UDID	ACK, Returns Address 'h51	ACK, Returns Address 'h51	ACK, Returns Address 'h51	ACK, Returns Address 'h51	ACK, Returns address Address configured from test ('h55)
Directed RESET	ACK, Address not RESET (still 'h55)	ACK, Address RESET to `hFF	ACK, Address RESET to `hFF, UDID[31:0] randomized	ACK, Address not RESET (still 'h55)	NACK

Table 6-19 SMBUS Checker List

Rule	Description	Section
SVT_SMBUS_PEC_MISMATCH	PEC received is incorrect	6.4
SVT_SMBUS_DATA_SIZE_MISMATCH	The number of data bytes don't match the block size in block read/write commands	6.5
SVT_SMBUS_WR32_DATA_SIZE_MISMATCH	Data payload is not 4 bytes for write 32 command	6.5
SVT_SMBUS_WR64_DATA_SIZE_MISMATCH	Data payload is not 8 bytes in write 64 command	6.5

Table 6-19 SMBUS Checker List (Continued)

Rule	Description	Section
SVT_SMBUS_RD64_DATA_SIZE_MISMATCH	Data size is not 4 bytes in read 32 command	6.5
SVT_SMBUS_RD32_DATA_SIZE_MISMATCH	Data size is not 8 bytes in read 64 command	6.5
SVT_SMBUS_BLOCK_PROCESS_MAX_DATA_COUNT	Number of data bytes in block process read and write exceed 255	6.5
SVT_SMBUS_REPEATED_START_NOT_DETECTED	Repeated start is not detected when a read command is received	6.5
SVT_SMBUS_GET_UDID_DATA_COUNT_MISMATCH	Block count or number if data bytes is not 17 in get udid command	6.6.3
SVT_SMBUS_DYNAMIC_ADDRESS_ALREADY_ASSIGNED	The dynamic address being assigned in assign address is already assigned to a different slave	6.6.3
SVT_SMBUS_RESERVED_ARP_CMD	The command in ARP process is a reserved ARP command	6.6.3
SVT_SMBUS_ARP_NOT_ACKED	UDID of ARP Devices is validated but ARP Slave Device doesn't ACK the "Prepare to ARP" command..	6.6
SVT_SMBUS_CLK_LOW_SEXT_VIOLATION	Cumulative Clock (tLOW:SEXT) low extend time within one message (start to stop) crossed its Limit of 25 ms; observed max low period is %d	4.2
SVT_SMBUS_CLK_LOW_MEXT_VIOLATION	Cumulative Clock (tLOW:MEXT) low extend time within message (start to ack, ack to ack, ack to stop) crossed its Limit of 10 ms; observed max low period is%d	4.2
SVT_SMBUS_INVALID_tLOW_MIN_1mhz	Multiple masters are trying to place their clk into Bus then their first high-to-low transition is not less than (tLOW:MIN - tSU:DAT) for 1 MHz	5.3
SVT_SMBUS_INVALID_tLOW_MIN_400khz	Multiple masters are trying to place their clk into Bus then their first high-to-low transition is not less than (tLOW:MIN - tSU:DAT) for 400 KHz	5.3
SVT_SMBUS_INVALID_tLOW_MIN_100khz	Multiple masters are trying to place their clk into Bus and their first high-to-low transition is not less than (tLOW:MIN - tSU:DAT) for 100 KHz	5.3

Table 6-19 SMBUS Checker List (Continued)

Rule	Description	Section
SVT_SMBUS_DATA_TRANSITION_DURING_CLK_HIGH	Data on SMBDAT is not stable during the high period of the clock.	5.1.1
SVT_SMBUS_INVALID_CLK_LOW_TIME	CLK is held low for more than ttimeout:min	4.2
SVT_SMBUS_HOST_NOTIFY_DATA_SIZE_MISMATCH	DATA PAYLOAD should be 2 bytes for HOST NOTIFY protocol	6.5.9
SVT_SMBUS_READ_BYTE_DATA_SIZE_MISMATCH	DATA PAYLOAD should be 1 bytes for READ BYTE protocol	6.5.5
SVT_SMBUS_READ_WORD_DATA_SIZE_MISMATCH	DATA PAYLOAD should be 2 bytes for READ WORD protocol	6.5.5
SVT_SMBUS_WRITE_WORD_DATA_SIZE_MISMATCH	DATA PAYLOAD should be 2 bytes for WRITE WORD protocol	6.5.4
SVT_SMBUS_WRITE_BYTE_DATA_SIZE_MISMATCH	DATA PAYLOAD should be 1 bytes for WRITE BYTE protocol	6.5.4
SMBUS_WR_CMD_INCORRECT_PEC_ACKED	Incorrect Value of PEC driven by Master for any write command (with PEC) shall be NACKed by Slave.	6.4.1
SMBUS_WR_CMD_END_WITH_NACK	SMBus Write command shall always end with an ACK.	6.5.4
SMBUS_RD_CMD_END_WITH_ACK	SMBus Read command shall always end with a NACK.	6.5
SVT_SMBUS_INVALID_ADDRESS_ACKED	A SMBus device shall not ACK an unallocated address or reserved SMBus address.	6.2.2.1
SVT_SMBUS_NOTIFY_ARP_MASTER_DATA_SIZE_ MISMATCH	The number of data bytes in notify ARP Master command is not 16 bits	6.6.3.8
SVT_SMBUS_NOTIFY_ARP_RSRVD_DATA_BYTES	For Notify ARP Master command, values other than 0000h in data bytes field are reserved for future use.	6.6.3.8

Table 6-20 SMBUS Coverage Plan

Coverage Feature	Cover point	Cover Bin
cg_smbus_master_packet	cp_slave_address	b_slave_address_range
		b_read_command
	cp_read_write_field	b_write_command
	cp_command_field	b_command_field_range
		b_write_word_command
		b_write_32byte_command
		b_write_64byte_command
		b_write_block_command
		b_read_byte_command
		b_read_word_command
		b_read_32byte_command
		b_read_64byte_command
		b_read_block_command
		b_process_block_command
		b_process_call_command
		b_write_byte_pec_command
		b_write_word_pec_command
		b_write_32byte_pec_command
		b_write_64byte_pec_command
		b_write_block_pec_command
		b_host_notify_command
		b_notify_arp_master_command

Table 6-20 SMBUS Coverage Plan (Continued)

Coverage Feature	Cover point	Cover Bin
		b_read_byte_pec_command
		b_read_word_pec_command
		b_read_32byte_pec_command
		b_read_64byte_pec_command
		b_read_block_pec_command
		b_process_block_pec_command
		b_process_call_pec_command
	cp_byte_count	b_byte_count_255
		b_byte_count_0
		b_byte_count_range
cg_smbus_master_timeout_values	cp_t_buf	b_t_buf_min
		b_t_buf_range
	cp_t_low	b_t_low_min
		b_t_low_range
	cp_t_high	b_t_high_min
		b_t_high_range
		b_t_high_max

Table 6-20 SMBUS Coverage Plan (Continued)

Coverage Feature	Cover point	Cover Bin
cg_master_arp_state	cp_arp_command	b_prepare_to_arp_command
		b_gen_reset_device_command
		b_gen_get_udid_command
		b_assign_address_command
	cp_address_type	b_fixed_address
		b_dynamic_psa
		b_dynamic_and_volatile
		b_randome_device_number
	cp_supported_protocols	b_zone
		b_ipmi
		b_asf
		b_oem
	cp_pec_support	b_pec_supported
		b_pec_not_supported
cg_slave_udid_fields	cp_smbus_revision	b_smbus_v1_0
		b_smbus_v1_1
		b_smbus_v2_0
		b_smbus_v_3_0
cg_smbus_slave_packet	cp_slave_address	b_slave_address_range
	cp_read_write_field	b_read_command
		b_write_command
	cp_byte_count	b_byte_count_255
		b_byte_count_0
		b_byte_count_range
	cp_command_field	b_write_byte_command
		b_write_word_command
		b_write_32byte_command
		b_write_64byte_command
		b_write_block_command

Table 6-20 SMBUS Coverage Plan (Continued)

Coverage Feature	Cover point	Cover Bin
		b_host_notify_command
		b_notify_arp_master_command
		b_read_byte_command
		b_read_word_command
		b_read_32byte_command
		b_read_64byte_command
		b_read_block_command
		b_process_block_command
		b_process_call_command
		b_write_byte_pec_command
		b_write_word_pec_command
		b_write_32byte_pec_command
		b_write_64byte_pec_command
		b_write_block_pec_command
		b_read_byte_pec_command
		b_read_word_pec_command
		b_read_32byte_pec_command
		b_read_64byte_pec_command
		b_read_block_pec_command
		b_process_block_pec_command
		b_process_call_pec_command
cg_smbus_slave_timeout_values	cp_t_buf	b_t_buf_min
		b_t_buf_range
	cp_t_low	b_t_low_min
		b_t_low_range
	cp_t_high	b_t_high_min
		b_t_high_range
		b_t_high_max

Table 6-20 SMBUS Coverage Plan (Continued)

Coverage Feature	Cover point	Cover Bin
cg_slave_arp_state	cp_arp_command	b_prepare_to_arp_command
		b_gen_reset_device_command
		b_gen_get_udid_command
		b_assign_address_command

6.20.4 Host Notify Protocol

Host Notify Protocol is supported for Master VIP only because the host device needs to have mastership capabilities. For Host Notify Protocol, one master must be configured as host device using agent configuration `is_smbus_host` being set to 1.

There are 3 transaction class variables added for nack from SMBus Master:

a. smbus_host_nack_addr

b. smbus_host_nack_addr_count

c. smbus_host_nack_data

For usage example, refer to the following tests:

- smbus_host_notify_protocol_test
- smbus_host_notify_nack_addr_test
- smbus_host_notify_nack_data_test

6.20.5 SMBALERT

The SMBALERT is a wired-AND signal. Slave-only devices can signal the host using SMBAlert pin. Then the host sends Alert response Address. Only the slaves which have sent SMBAlert acknowledge the request and start sending their address. If multiple slaves have sent SMBAlert signal, then they arbitrate and the one with the lowest address wins.

Use case model:

The slave VIP can pull the SMBLAERT low through sequence. Multiple slave VIPs can send SMBAlert simultaneously depending on the scenario.

The Master-VIP or Host sees the net result of SMBALERT signal. Then it automatically starts driving the SMBAlert frame [start -> ARA -> Rd]. The slaves which have pulled the SMBAlert pin low will ACK and send their addresses for arbitration.

Note

The delay of at least 3 system clocks is required after SMBAlert transaction from slave. If any ARP or other command is sent through the sequence on Master, then no delay is required.

For example, refer ts.smbus_smbalert_basic_test.sv in the testbench tb_smbus_svt_uvm_basic_sys.

6.20.6 SMBUS EXCEPTIONS

These are the exceptions for master transactions:

- PEC ERROR

- SMBUS_ABORT_TRANS_RANDOM_ERROR
- MST_CORRUPT_BYTE_COUNT
- ACK_ON_LAST_READ_ERROR

These are the exceptions for slave transactions:

- SMBUS_BLOCK_PROCESS_MAX_COUNT_ERROR
- PEC ERROR
- SLV_NACK_PREP_TO_ARP

For more details on exceptions, refer the I2C SVT VIP class reference HTML document.

6.20.7 Notify ARP Master

Notify ARP Master command is supported for master VIP only because the host needs to have mastership capabilities. For Notify ARP Master, one master must be configured as HOST device using agent configuration '`is_smbus_host`' being set to 1.

Transaction class variables (refer Section 6.20.4) added for Host Notify Protocol are applicable for Notify ARP Master also.

Notify ARP Master shall have data values 'h0000 based on the specification. You need to set these values from the sequence itself. Setting data fields to any value other than 'h0000 results into checker failure.

Two checker Rules have been added for this command in Table 6-19.

For usage example, refer to public example ts.smbus_notify_arp_master_test.sv.

6.21 Exceptions Use Model

The following set of variables have been added to the respective classes:

svt_i2c_transaction

```
/** Enables/Disables error insertion for current transaction */
rand bit do_insert_error = 1;
```

Any exception will only be inserted when the "do_insert_error" is set to 1 and exception callback is added in the exception list. If anyone is missing, then exception will not be inserted.

svt_i2c_master_transaction_exception

```
  /** Configures the number of back-to-back START (S -> S -> S -> S …) or back-to-back
STOP (P -> P -> P -> P …)  conditions that are to be inserted.   */
  rand int no_of_tags = 1;


  /** Configures whether to retry or flush the current transaction after inserting the
error */
  rand bit retry_txn = 0;


  /** Configures the STOP condition insertion either in Address byte or data byte
   * - 0 : Insert STOP in Address byte<br/>
   * - 1 : Insert STOP in Data byte<br/>   */
```

```
  rand bit p_at_cmd_data = 1;

  /** Configures the position of the tag (Start/Stop) in write data bytes. If 1, the tag
will be
   * inserted in 1st write data byte  */
  rand int data_byte_pos = 1;

  /** Configures the bit level position of tag in address byte or data byte of write
command.
   *     If 1, the tag will be inserted after 1st bit of the byte.   */
  rand bit [3:0] byte_bit_pos = 4;

  /**Configures the number of START->STOP pattern to be inserted   */
  rand int no_of_s_p_pattrn = 1;
```

The following rules have been modified or added along with their test cases:

1. START_STOP_ERROR - The rule already existed but only generated a single START-> STOP pattern. The rule has been modified and can be configured as shown in the following examples:
   a. start_stop_exception_1_test (i2c_master_start_stop_error_1_callback)
      i. Scenario generated will be, (1st txn) S ->P (1st txn flushed) -> Normal txn
      - exception.error_kind = svt_i2c_master_transaction_exception::START_STOP_ERROR;
      - exception.no_of_tags = 1;
      - exception.no_of_s_p_pattrn = 1;
      - exception.retry_txn = 0;
   b. start_stop_exception_2_test (i2c_master_start_stop_error_2_callback)
      i. Scenario, (1st txn) S -> S->P->S->P->S->P->S->P ->(1st txn retried and run normally) -> 2nd txn
      - exception.error_kind = svt_i2c_master_transaction_exception::START_STOP_ERROR;
      - exception.no_of_tags = 2;
      - exception.no_of_s_p_pattrn = 4;
      - exception.retry_txn = 1;
   c. start_stop_exception_3_test (i2c_master_start_stop_error_3_callback)
      i. Scenario, (1st txn) S->S->S->S->P-S->P (1st txn flushed) -> 2nd txn
      - exception.error_kind = svt_i2c_master_transaction_exception::START_STOP_ERROR;
      - exception.no_of_tags = 4;
      - exception.no_of_s_p_pattrn = 2;
      - exception.retry_txn = 0;
   d. start_stop_exception_4_test (i2c_master_start_stop_error_4_callback)
      i. Scenario, (1st txn) S->S->S->S->P>S->P>S->P>S->P->(1st txn retried and run normally) -> 2nd txn
      - exception.error_kind = svt_i2c_master_transaction_exception::START_STOP_ERROR;

- ✧ exception.no_of_tags = 4;
- ✧ exception.no_of_s_p_pattrn = 4;
- ✧ exception.retry_txn = 1;

2. STOP_IN_BYTE - New rule added to generate STOP condition in command or write data byte.
   a. Please Note that the exception is as of now only tested for 7 bit address, in Standard , Fast and Fast Mode Plus, with only basic Read/Write transactions
   b. stop_in_cmd_exception_test (i2c_master_stop_in_cmd_error_callback)
      i. Scenario, (1st txn) S-> P (after 4th bit of command byte)->(1st txn retried and run normally) -> 2nd txn
      - ✧ exception.error_kind = svt_i2c_master_transaction_exception::STOP_IN_BYTE;
      - ✧ exception.p_at_cmd_data = 0;
      - ✧ exception.byte_bit_pos = 4;
      - ✧ exception.retry_txn = 1;
   c. stop_in_data_byte_exception_test (i2c_master_stop_in_data_byte_error_callback)
      i. Scenario, (1st txn) S->Cmd Byte-> data1 -> P (after 4th bit of 2nd data byte)->(1st txn retried and run normally) -> 2nd txn
      - ✧ exception.error_kind = svt_i2c_master_transaction_exception::STOP_IN_BYTE;
      - ✧ exception.p_at_cmd_data = 1;
      - ✧ exception.byte_bit_pos = 4;
      - ✧ exception.data_byte_pos = 2;
      - ✧ exception.retry_txn = 1;
3. FALSE_STOP_BEFORE_START - New rule added to generate multiple STOP conditions before starting a transaction
   a. false_stop_before_start_exception_test (i2c_master_false_stop_before_start_error_callback)
      i. Scenario, (1st txn) P->P->P->P (1st txn flushed) -> normal txn
      - ✧ exception.error_kind = svt_i2c_master_transaction_exception::FALSE_STOP_BEFORE_START;
      - ✧ exception.no_of_tags = 4;
      - ✧ exception.retry_txn = 0;
4. START_FOLLOWED_BY_START - New rule added to generate multiple START conditions before starting a transaction. In this case, the transaction cannot be flushed
   a. start_followed_by_start_exception_test (i2c_master_start_followed_by_start_error_callback)
      i. Scenario, (1st txn) ->S->S->S->S->(normal txn) -> 2nd txn
      - ✧ exception.error_kind = svt_i2c_master_transaction_exception::START_FOLLOWED_BY_START;
      - ✧ exception.no_of_tags = 4;
5. START_IN_BYTE - New rule added to generate START condition in command byte or write data byte. The transaction cannot be flushed
   a. Please Note that the exception is as of now only tested for 7 bit address, in Standard , Fast and Fast Mode Plus, with only basic Read/Write transactions

b. start_in_cmd_exception_test (i2c_master_start_in_cmd_error_callback)

   i. Scenario, (1st txn) S-> Command byte 4th bit -> S-> (1st txn runs normally) -> 2nd txn

   ✧ exception.error_kind = svt_i2c_master_transaction_exception::START_IN_BYTE;

   ✧ exception.p_at_cmd_data = 0;

   ✧ exception.byte_bit_pos = 4;

c. start_in_data_byte_exception_test (i2c_master_start_in_data_byte_error_callback)

   i. scenario, (1st txn) S->cmd byte-> data1->data2->data3->data4, 7th bit -> S ->(1st txn runs normally) -> 2nd txn

   ✧ exception.error_kind = svt_i2c_master_transaction_exception::START_IN_BYTE;

   ✧ exception.p_at_cmd_data = 1;

   ✧ exception.byte_bit_pos = 7;

   ✧ exception.data_byte_pos = 4;

6. CMD_CORRUPT_FOR_10BIT_READ - New error injection capability added in VIP. This is used to corrupt first frame of 10bit addressing READ command.

   a. After START when READ command is fired , it is mandatory to send (11110) appended with two MSB bits of 10 bit address with WRITE command. By default master VIP drives WRITE in the first frame. But, by using this error injection, READ can be driven instead of WRITE.

   b. This EI does not work when 10 bit READ is attempted after Sr given a 10 bit WRITE has already been sent to it. This is because:

      i. It is not compulsory to send the Master code pad with WRITE bit for READS after Sr.

      ii. If VIP master is configured to send the Master CODE + WRITE after SR and this EI is used, slave still acknowledges the frame as it was not aware whether master is sending the first frame or not. This might hang the test case.

   c. Testcase: `ts.corrupt_10b_rd_test.sv` added in `tb_i2c_svt_uvm_basic_sys` to demonstrate a use case for this EI.

   d. This EI can be used for hitting fail coverage for checker rule: `i2c_10b_slv_addr_fst_rd_cmd`

7. SEND_ACK_FOR_RESERVED_ADDR_ERROR - New error injection capability added to the VIP. This EI is used to send an ACK after RESERVED address is sent on the bus.

   a. Testcase: ts.`reserved_address_drive_by_master_and_ack_by_slv_error_test.sv` added in `tb_i2c_svt_uvm_intermediate_sys` to demonstrate the use case. Callback class: `i2c_slave_ack_for_reserved_address_callback.sv.`

6.22 Configuration Based Timing Parameters Check

VIP supports configuration based check for each speed modes under macro `I2C_SVT_CFG_VAL_CHK`. To enable it user has to pass `+define+ I2C_SVT_CFG_VAL_CHK` as compile time argument.

In this check, monitor compare the configuration settings of different timing parameters and match it with the value coming on interface. If it matches then no error else it will throw an error message.

For low clock period, consider clock stretching as well. When clock stretching will appear on interface, no error will come for clock low period.

Supported timing parameter checks as part of configuration based checks are mentioned as follows:

❖ Setup time for STOP condition

❖ Low clock duration

❖ High clock duration

1. Exact checking for timing parameters.

   Added exact checking to cross check the value appeared on interface with respect to the configuration settings.

   The Checker rules added for above checking are as follows:

   - SVT_I2C_SETUP_TIME_STOP_CFG_VALUE_STANDARD_MODE
   - SVT_I2C_SETUP_TIME_STOP_CFG_VALUE_FAST_MODE
   - SVT_I2C_SETUP_TIME_STOP_CFG_VALUE_HS_MODE
   - SVT_I2C_SETUP_TIME_STOP_CFG_VALUE_FAST_MODE_PLUS
   - SVT_I2C_HOLD_TIME_START_CFG_VALUE_STANDARD_MODE
   - SVT_I2C_HOLD_TIME_START_CFG_VALUE_FAST_MODE
   - SVT_I2C_HOLD_TIME_START_CFG_VALUE_HS_MODE
   - SVT_I2C_HOLD_TIME_START_CFG_VALUE_FAST_MODE_PLUS
   - SVT_I2C_LOW_CLK_DURATION_CFG_VALUE_STANDARD_MODE
   - SVT_I2C_LOW_CLK_DURATION_CFG_VALUE_FAST_MODE
   - SVT_I2C_LOW_CLK_DURATION_CFG_VALUE_FAST_MODE_PLUS
   - SVT_I2C_HIGH_CLK_DURATION_CFG_VALUE_STANDARD_MODE
   - SVT_I2C_HIGH_CLK_DURATION_CFG_VALUE_FAST_MODE
   - SVT_I2C_HIGH_CLK_DURATION_CFG_VALUE_FAST_MODE_PLUS

   Configuration variables added to enable/disable the above checks are as follows:

   - enable_chk_min_su_sto_time_ss
   - enable_chk_min_su_sto_time_fs
   - enable_chk_min_su_sto_time_hs
   - enable_chk_min_su_sto_time_fm_plus
   - enable_chk_min_hd_sta_time_ss
   - enable_chk_min_hd_sta_time_fs
   - enable_chk_min_hd_sta_time_hs
   - enable_chk_min_hd_sta_time_fm_plus
   - enable_chk_scl_low_time_ss
   - enable_chk_scl_low_time_fs
   - enable_chk_scl_low_time_fm_plus
   - enable_chk_scl_high_time_ss
   - enable_chk_scl_high_time_fs
   - enable_chk_scl_high_time_fm_plus

     1 -> To enable check

0 -> To disable check

Default Value -> 0

6.23 Tolerance Value Control

The “tolerance_limit” configuration variable added to configure the tolerance limit applicable on all the timing parameters.

For example, take “scl_low_time_ss” timing parameter as an example.

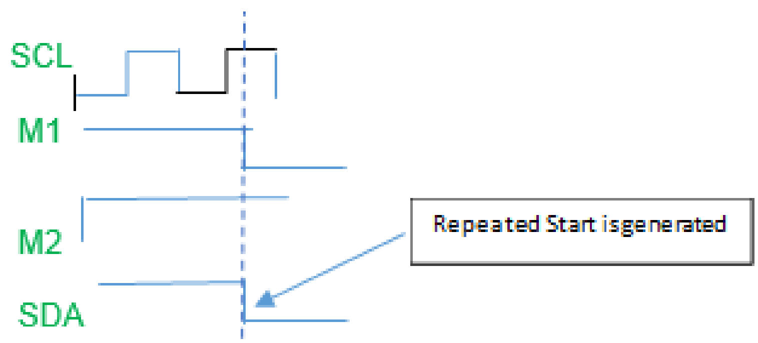
If configured value of scl_low_time_ss = 100 and tolerance_limit = 3,

Then on the interface scl_low_time_ss must be within [97:103] -> {97,98, 99, 100, 101, 102,103}

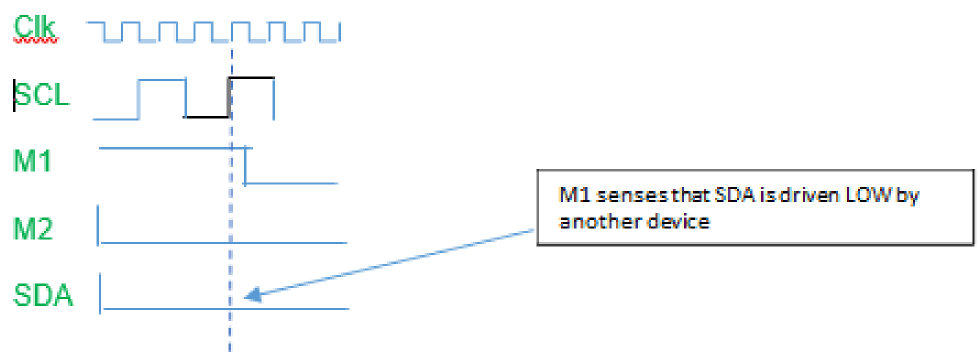
6.24 Arbitration

Arbitration is broadly categorized into following two categorize:

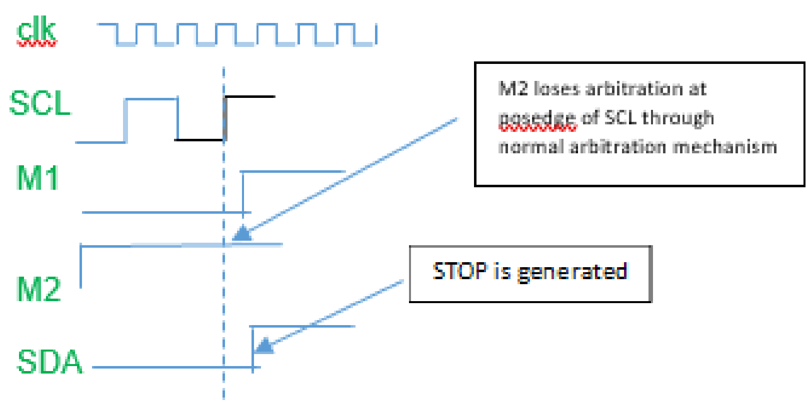
1. Combination resulting in Undefined condition.
   a. When Master 1 send Repeated START, Master 2 sends data bit 1’b1, M1 will successfully generate Repeated Start. M2 will detect that Repeated Start is generated while it was transmitting data. Following this M2 will assume arbitration is lost and wait for STOP on Bus. M2 can then retry its previous transaction. M2 can retry its incomplete transaction after STOP condition.



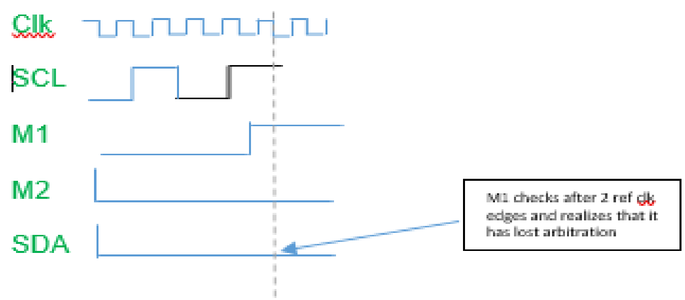
   b. When Master 1 send Repeated START, Master 2 sends data bit 1’b0, M2 successfully drives data bit 1’b0 and complete its transaction. M1 senses on Rising edge of SCL that SDA is LOW, so it assumes that Repeated Start cannot be generated and loses arbitration and waits for STOP condition on Bus. As the data transmission by M1 is complete, it will start new transaction (not retry previous transaction) after detection of STOP.



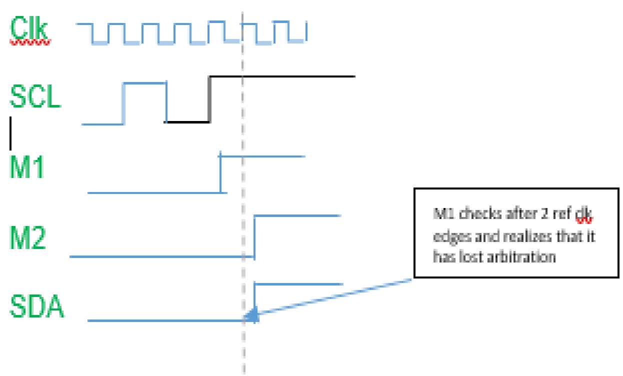
c. When Master 1 send STOP, Master 2 sends data bit 1'b1, M1 drives SDA LOW during LOW cycle of SCL to generate STOP during SCL HIGH cycle M2 drives data 1'b1 during SCL LOW cycle and checks for arbitration loss at posedge of SCL. In this scenario M2 loses arbitration as SDA is driven LOW by M1, M2 waits for STOP condition on bus. M2 can then retry its previous transaction. M1 successfully generates STOP condition on the bus.



d. When Master 1 sends STOP, Master 2 sends data bit 1'b0, M2 drives SDA LOW during SCL LOW cycle and continues driving throughout the SCL HIGH cycle M1 attempts generating STOP by driving SDA HIGH during SCL HIGH cycle. M1 checks after n (eg:2) ref clk edges if the SDA is driven HIGH on bus. As the SDA is LOW, M1 assumes loss of arbitration and waits for STOP on Bus. As the transaction is completed M1 will start new transaction (not retry previous transaction) after detection of STOP. M2 completes its transaction.

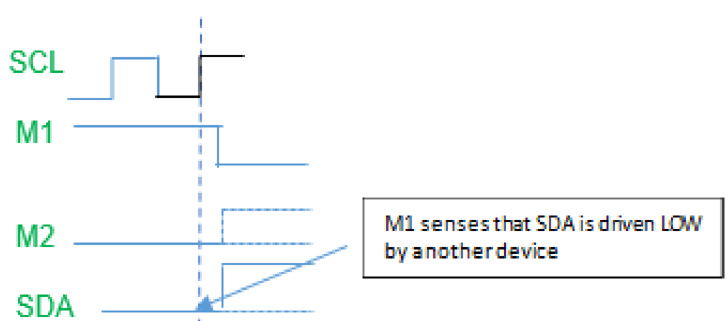


e. Master 1 sends STOP earlier than the other Master 2 which will also put STOP condition. M1 and M2 drive SDA LOW during SCL LOW cycle. M1 attempts STOP generation by driving SDA HIGH during SCL HIGH cycle. M1 checks after n (example, 2) ref clk edges if the SDA is driven HIGH on bus. As SDA is driven LOW by M2, M1 assumes loss of arbitration and waits for STOP on bus. As the transaction is completed M1 will start new transaction (not retry previous transaction) after detection of STOP. M2 successfully attempts STOP generation on the bus by driving SDA HIGH.

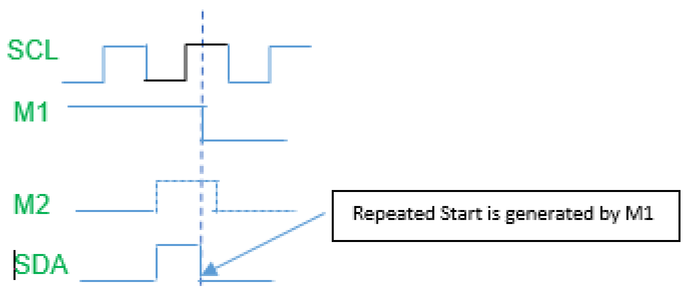


f. Combination resulting in defined condition.

i. When Master 1 send Repeated START, Master 2 drives SDA LOW on SCL LOW CYCLE. M1 drives SDA HIGH during SCL LOW cycle to generate Repeated Start during SCL HIGH cycle. M2 drives SDA LOW during SCL LOW cycle to generate STOP or to drive a Data bit. M1 senses on Rising edge of SCL that SDA is LOW, so it assumes that Repeated Start cannot be generated and loses arbitration and waits for STOP condition on Bus. As the data transmission by M1 is complete, it will start new transaction (not retry previous transaction) after detection of STOP. M2 successfully drive SDA bus.



ii. When Master 1 send Repeated START, Master 2 drives SDA HIGH on SCL LOW CYCLE M1 and M2 drive SDA HIGH during SCL LOW cycle to generate Repeated Start during SCL HIGH cycle. M2 attempts to generate Repeated Start after 'delta' time. M1 pulls SDA LOW earlier than M2 to generate Repeated Start and a result Repeated Start is generated on the bus by M1. Since SDA was HIGH after/at Rising Edge of SCL, M2 assumes it has not lost the arbitration and can generate Repeated Start, but does not sample early pull down of SDA line. However, M2 satisfies its configured Setup time for Sr and pulls SDA LOW and successfully observes that SDA is LOW while SCL is still HIGH, it safely assumes that Repeated Start has been generated on the bus. Both devices then attempt to maintain Hold time for Sr on the bus. Since there is multiple device driving, clock synchronization will be in effect and Hold time for Sr will be observed according to whichever device pulls the SCL LOW first.



6.25 Clock Synchronization Feature in Accordance With the I2C Spec Version 3.0

6.26 Configuration Based run Time Checks

- Timing Parameters observed on the interface is checked with the configured value. This feature can be enabled by passing macro "`ENABLE_SVT_I2C_RUNTIME_CFG_CHECKS".

6.27 Macro Based Delay Between Back To Back Transaction With Repeated Start

- Delay between two back to back transaction can be configured can be set using macro "`SVT_I2C_WAIT_FOR_TXN_TIMEOUT". The default value of the macro is 0.

7 VIP Tools

7.1 Using Native Protocol Analyzer for Debugging

7.1.1 Introduction

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

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

7.1.2 Prerequisites

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

Compile Time Options

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

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

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

  Configuration Variable Setting:

  Set `pa_xml_generation_enable` parameter of I2C configuration class `svt_i2c_configuration` to enable the generation of XML files.

  For Example:

```
<i2c_xmtr_agent_configuration>.i2c_cfg.pa_xml_generation_enable = 1
```

  Similarly for DSI receiver:

```
<i2c_rcvr_agent_configuration>.i2c_cfg.pa_xml_generation_enable = 1
```

7.1.3 Invoking Protocol Analyzer

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

Post-processing Mode

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

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

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

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.

7.1.4 Documentation

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

`$VERDI_HOME/doc/Verdi_Transaction_and_Protocol_Debug.pdf.`

7.1.5 Limitations

Interactive support is available only for VCS.

8
Troubleshooting

This chapter provides some useful information that can help you to troubleshoot common issues that you may encounter while using the I2C VIP.

The chapter consists of the following section:

8.1 Enabling Traffic logs

You can enable or disable traffic logs using the `enable_traffic_log` variable, which is defined in the `svt_i2c_agent_configuration` class. The default value of the variable is 1, which enables traffic logs, by default.

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

The following code setting illustrate how you can enable the traffic log assuming the `master_cfg` handle is of the `cust_svt_i2c_agent_configuration` class:

```
master_cfg.enable_traffic_log = 1;
```

To disable traffic logs, set the value of the `enable_traffic_log` variable to 1 in the `cust_svt_i2c_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 8-1 shows the traffic log generated by the above code setting.

Figure 8-1. Traffic Log

```
================+===========+========================
                |   Byte    |
 TIME (100 ps)  | MSB    LSB|        REMARKS
                |           |
================+===========+========================
          47255 | ----------| Start Condition
         937250 |    0110011| 7 Bit Slave ID
         937250 | ----------| Write Command
         937250 | ----------| ACK Received
        1837250 |   11000011| Data Byte
        1837250 | ----------| ACK Received
        2737250 |   00010010| Data Byte
        2737250 | ----------| ACK Received
        3637250 |   01101101| Data Byte
        3637250 | ----------| ACK Received
        4537250 |   01111110| Data Byte
        4537250 | ----------| ACK Received
        4677255 | ----------| Stop Condition
```

8.2 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 89
- "Setting Verbosity During Runtime" on page 89

8.2.1 Setting Verbosity in the Testbench

To set the verbosity level in the testbench, use the UVM-specified log levels in the code. The components extend from the `uvm_report_object` method. You can use the `uvm_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:

- `UVM_NONE`: Prints the report always. The setting of the verbosity level cannot disable it.
- `UVM_LOW`: Prints the report if the configured verbosity is set to `UVM_LOW` or above.
- `UVM_MEDIUM`: Prints the report if the configured verbosity is set to `UVM_MEDIUM` or above.
- `UVM_HIGH`: Prints the report if the configured verbosity is set to `UVM_HIGH` or above.
- `UVM_FULL`: Prints the report if the configured verbosity is set to `UVM_FULL` or above.

8.2.2 Setting Verbosity During Runtime

To set the verbosity level during runtime, you can use one of the following methods:

- "Method 1: To Enable the Specified Severity in the VIP, DUT, and Testbench" on page 90
- "Method 2: To Enable the Specified Severity to the Specific Sub-Classes of VIP" on page 90

8.2.2.1 Method 1: To Enable the Specified Severity in the VIP, DUT, and Testbench

The example for VCS is as follows:

```
vcs <other runtime options> +UVM_VERBOSITY=UVM_MEDIUM
```

8.2.2.2 Method 2: To Enable the Specified Severity to the Specific Sub-Classes of VIP

The example for this method is as follows:

- `+uvm_set_verbosity=component_name,id,verbosity,phase_name,optional_time`

8.3 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, see the Chapter 7 “VIP Tools”.

A
Reporting Problems

A.1 Introduction

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

A.2 Debug Automation

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

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

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

A.3 Enabling and Specifying Debug Automation Features

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

Note the following about the plusarg:

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

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

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

The following table explains each control string:

Table A-1 Control Strings for Debug Automation plusarg

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

Examples:

Enable on all VIP instances with default options:

```
+svt_debug_opts
```

Enable on all instances:

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

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

Enable on all instances:

- starting at time 1000 until time 1500:

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

Enable debug feature on all instances using default options:

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

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

Note

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

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

A.4 Debug Automation Outputs

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

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

The following are the output files generated:

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

A.5 FSDB File Generation

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

A.5.1 VCS

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

```
-debug_access
```

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

A.5.2 Questa

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

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

A.5.3 Incisive

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

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

A.6 Initial Customer Information

Follow these steps when you call the Synopsys Support Center:

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

Enable the Debug Opts feature. For more information, see the "Debug Automation" on page 91.

A.7 Sending Debug Information to Synopsys

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

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

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

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

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

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

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

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

A.8 Limitations

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

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

A.9 Disabling I2C SVT VIP

1. By default, when the run time command argument `- svt_i2c_disable_vip` is not used, the I2C VIPs remain enabled. In general, you must use the VIP normally. In the scenario, where the AGENT_ID parameters in Verilog top and num_of_masters/num_of_slaves in sys_cfg are not used in the recommended manner, you will see a fatal error.

   If you want the VIP disabled, when not using command arg - svt_i2c_disable_vip, you must ensure to:

   a. Set the Verilog wrapper instance parameter `SVT_I2C_ENABLE_WRAPPER_CONNECTION_CHECK` to 0 in the Verilog top.

   b. Do not create the `uvm_agents` in the environment file.
2. When you choose to exercise the argument `- svt_i2c_disable_vip`, the following are the possibilities:

a. When `svt_i2c_disable_vip`=0, the VIPs remain enabled. In this case, if the AGENT_ID parameters in Verilog top and num_of_masters/num_of_slaves in sys_cfg are used in the recommended manner, the simulation proceeds without issues. If you deviate from the recommended usage of the above params/sys-cfg-vars, you will see the fatal error mentioned in #1.

b. When `svt_i2c_disable_vip`=1, the VIPs get disabled. In such a scenario, the user environment must ensure not to create the uvm_agents for I2C VIP. A public example is already available for that (tb_i2c_svt_uvm_disable_vip).