Verification Continuum™

VC Verification IP DisplayPort UVM User Guide

Version Q-2020.06, June 2020

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Preface

About This Guide

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

Guide Organization

The chapters of this guide are organized as follows:

- Chapter 1, "Introduction", introduces the DisplayPort VIP and its features.
- Chapter 2, "Download and Installation", describes system requirements and provides instructions on how to install, configure, and begin using the DisplayPort VIP.
- Chapter 3, "Design Directory Maintenance", leads you through setting up the VIP.
- Chapter 4, "Overview of the DisplayPort VIP", provides detail for each UVM-derived class.
- Chapter 5, "DP Environment", describes how the VIP can be connected in various testbench scenarios.
- Chapter 6, "VIP Tools", provides useful information that can help you troubleshoot common issues that you may encounter while using the DisplayPort VIP.
- Chapter 7, "Usage Notes", explains the necessary task.
- Appendix A, "Reporting Problems", outlines the process for working through and reporting VC VIP for DisplayPort issues.

Customer Support

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

- Accessing SolvNetPlus

  Access documentation through SolvNetPlus from the following location:

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

- Contacting the Synopsys Technical Support Center
  - Go to https://solvnetplus.synopsys.com and open a case.
    - Enter the information according to your environment and your issue.
    - If applicable, provide the information noted in Appendix A, Reporting Problems.
  - Send an e-mail message to support_center@synopsys.com
    - Include the Product name, Sub Product name, and Product version for which you want to register the problem.
    - If applicable, provide the information noted in Appendix A, Reporting Problems.
  - Call your local support center.
    - North America:

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

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

Contents

1 Introduction

1.1 Overview

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

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

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

This chapter discusses the following topics:

- Product Overview
- Simulator Support
- DisplayPort Feature Support

1.2 Product Overview

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

The source and sink agents support all functionalities associated with active and passive UVM components, including the creation of transactions, checking and reporting the protocol correctness, transaction logging and functional coverage.

1.3 Simulator Support

The DisplayPort VIP suite supports the following languages and simulators:

- Languages
  - SystemVerilog
- Simulators
  - VCS
  - IUS
  - MTI
- Methodology
  - UVM 1.1d
  - OVM

1.4 DisplayPort Feature Support

The following subsections provide an overview of the supported DisplayPort features for verification:

1.4.1 Supported Features

- Specification version
- DP v1.4 Errata 5
- Embedded Display Port (eDP) 1.4b
- High Bandwidth Digital Content Protection (HDCP) v1.3, v2.2.
- Display Port Configuration Data (DPCD) 1.0, 1.1, 1.2, 1.3, and 1.4
- Extended Display ID (EDID) v1.4
- Monitor Control command Set/Display Data Channel (MCCS/DDC)
- CEA-861-F, DMT v1r13, CVT v1.2 video timing standard support
- DSC v1.2a Compression
- Fast link training (no aux transfer needed)
- Bypass link training (no aux transfer needed)

1.4.2 Verification Features

- DP VIP as Source (Active/Passive)
- DP VIP as Sink (Active/Passive)
- Random Stimulus generation using Sequences

- Analysis ports for connecting Source, Sink agents to scoreboard etc.
- Callbacks for Source Sink driver and monitors
- Constraints: Reasonable, valid and user-defined
- Functional coverage and verification plans
- Configurable coverage and protocol checks
- Scoreboard (Audio/Video/Control)
- Error Injection using exceptions
- Trace files
- Transcript (log) file
- Verbosity control (all UVM features)
- Debug ports (internal states and transaction-level debug signals)
- Generic command for Source DUT programming
- Command-line argument controls to run a specific configuration
- Video and Audio input file support
- Verdi Protocol Analyzer

1.4.3 DP 1.4 Protocol Feature Support

1.4.3.1 Main Link

- One to four lanes
- RGB444/YCbCr444, YCbCr422 with 6bpp RGB444 and 8/10/12/16BPP for both
- VIC 1~107 and custom frame size support
- Symbol Stuffing and Transfer Unit (TU)
- Audio data Secondary Data Packets (SDPs)
- SDP (Extension, AudioStream, AudioTimestamp, AudioCopyManagement, ISRC, Camera Generic, Audio Infoframe, VSC_EXT_XXX_PPS)
- ECC for SDP, Nibble interleaving
- Insertion/Detection of Main Stream Attribute (MSA)
- Inter-lane skewing/de-skewing
- Stream clock regeneration
- Framing mode
- Scrambling and de-scrambling
- Serialization/de-serialization
- 8b/10b encoding/decoding
- Video data generation (random data, image data, and pattern data)
- RBR,HBR,HBR2,HBR3 Rate support
- 2-8 audio channel

- Video only stream (without audio)
- Audio only stream (without video)
- Majority voting error correction in DPRX
- Enhanced Framing Mode, FEC, and ACT Trigger

1.4.3.2 AUX Link

- I2C over Auxilary (AUX)
- EDID
- DPCD
- Manchester-II encoding/decoding
- Full link training (TPS1, TPS2, TPS3, TPS4)
- Bypass link training for faster simulation
- Clock recovery
- Channel equalization
- Pre-emphasis levels
- Post link training adjust request
- Link maintenance after loss of synchronization
- Audio data generation (random)
- AUX channel states
- Link layer arbitration
- Stream transport initiation
- Aux messaging layer

1.4.3.3 HPD

- Hot Plug Detect (HPD) IRQ
- HPD plug/unplug detect
- HPD initiated link training

1.4.3.4 HDCP 1.3

- First part of authentication protocol
- Second part of authentication protocol
- Data encryption and decryption
- Encryption status signaling in SST mode

1.4.3.5 HDCP 2.2

- Authentication with non-stored KM, pairing, and locality checks.
- Authentication with stored KM, pairing, and locality checks.
- Authentication with repeater and non-stored KM, pairing, and locality checks.

1.4.3.6 eDP v1.4b

- Single SST mode
- eDP DPCD registers
- Fast link training without AUX handshake
- Reduced Aux timing
- Custom link rates (R216, R243, R324, R432)
- ASSR protection
- Automatic minimum required lanes calculation
- DP/eDP revision registers
- Panel Self Refresh (eDP)
- Aux frame Sync (eDP)
- GUID Register
- Global Time Code

1.4.4 DP 2.0 Protocol Feature Support

1.4.4.1 Main Link

- Single and multi-stream
- 1,2 and 4 physical lanes
- All 3 link rates (20Gbps, 13.5Gbps and 10Gbps)
- FEC is supported
- Link layer to PHY logical ML lane count conversion
- Intra super symbol shifting
- Scrambling
- CDI field insertion,
- Pre-coding and de-coding
- Serial and 32 bit parallel interface supported
- Link training (without AUX) supported
- Bypass link training custom feature supported
- Supported HDCP 2.x encryption
- DSC
- Panel Replay (PR)
- 128b/132b encoding/decoding
- RGB444/YCbCr444, YCbCr422 with 6bpp RGB444 and 8/10/12/16BPP for both
- VIC 1~234 and custom frame size support
- Symbol Stuffing
- Audio data Secondary Data Packets (SDPs)

- SDP with CRC 16 is supported
- SDP (Extension, AudioStream, AudioTimestamp, AudioCopyManagement, ISRC, Camera Generic, Audio Infoframe, VSC_EXT_XXX_PPS)
- Insertion/Detection of Main Stream Attribute (MSA)
- 2-8 audio channel
- Video only stream (without audio)
- Audio only stream (without video)
- Majority voting error correction in DPRX

1.4.4.2 HPD

- Hot Plug Detect (HPD) IRQ
- HPD plug/unplug detect
- HPD initiated link training

1.4.4.3 HDCP 1.3

- First part of authentication protocol
- Second part of authentication protocol
- Data encryption and decryption
- Encryption status signaling in SST mode

1.4.4.4 HDCP 2.2

- Authentication with non-stored KM, pairing, and locality checks.
- Authentication with stored KM, pairing, and locality checks.
- Authentication with repeater and non-stored KM, pairing, and locality checks.

1.4.5 Unsupported Features

1.4.5.1 DP 1.4 Protocol Features

- Branch Device
- Clock jitter and voltage swing levels
- Link quality test patterns (Nyquist, PRBS7, Custom, and so on.)
- Backlight (eDP)
- MultiTouch (eDP)
- OUI (eDP)
- Renewability (HDCP 1.3)
- CEC power management

1.4.5.2 DP 2.0 Protocol Features

- Panel Replay Selective Update
- Link Training through Aux
- Branch Device

- Clock jitter and voltage swing levels
- Link quality test patterns (Nyquist, PRBS7, Custom, and so on.)
- Backlight (eDP)
- MultiTouch (eDP)
- OUI (eDP)
- Renewability (HDCP 1.3)
- CEC power management
- SDP Splitting

1.4.5.3 Verification features

- Verdi Protocol Analyzer for DP 2.0

2
Download and Installation

This section leads you through installing and setting up the DisplayPort VIP. When you complete this checklist, the provided example testbench will be operational and the DisplayPort VIP will be ready to use.

The quickstart consists of the following major steps:

1. "Verifying the Hardware Requirements"
2. "Verifying Software Requirements"

Note If you encounter any problems with installing the DisplayPort VIP, see.Customer Support.

2.1 Verifying the Hardware Requirements

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

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

2.2 Verifying Software Requirements

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

2.2.1 Platform/OS and Simulator Software

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

  Support Matrix for SVT-Based DisplayPort VIP is in:

  DisplayPort VIP Release Notes

2.2.2 Synopsys Common Licensing (SCL) Software

- The SCL software provides the licensing function for the DisplayPort VIP. Acquiring the SCL software is covered here in the installation instructions in Licensing Information.

2.2.3 Other Third Party Software

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

2.3 Preparing for Installation

1. Set DESIGNWARE_HOME to the following absolute path where DisplayPort VIP is installed:

   `setenv DESIGNWARE_HOME absolute_path_to_designware_home`

2. 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 Synopsys 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

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

Follow the instructions below for downloading the software from Synopsys. You can download from the Download Center using either HTTPS or FTP, or with a command-line FTP session. If your Synopsys SolvNet password is unknown or forgotten, go to http://solvnet.synopsys.com.

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

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

2.4.1 Downloading From the Electronic Software Transfer (EST) System (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

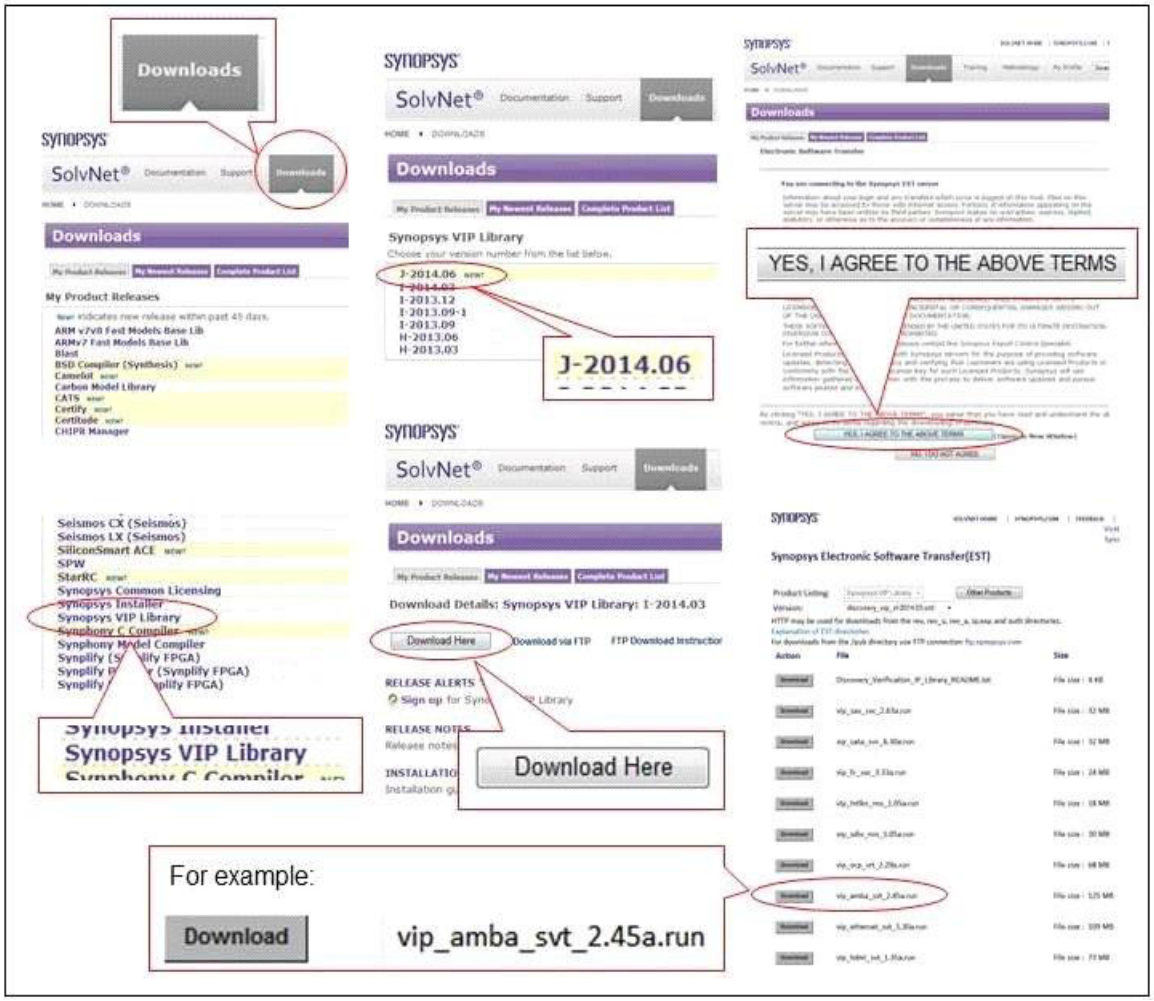
   iii. <release_version>

   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

1. Follow the above instructions through the product version selection step
2. Click the "Download via FTP" link instead of the "Download Here" button
3. Click the "Click Here To Download" button
4. Select the file(s) that you want to download
5. Follow browser prompts to select a destination location
6. Execute the run file:

   % <*vip run file name*>.run

Answer the prompts that the.run script generates until the install is complete.

2.5 What's Next?

The remainder of this chapter describes the details of the different steps you performed during installation and setup, and consists of the following sections:

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

2.5.1 Licensing Information

The DisplayPort VIP uses the Synopsys Common Licensing (SCL) software to control its usage. You can find general SCL information at:

http://www.synopsys.com/keys

VC VIP for DisplayPort uses a licensing mechanism that is enabled by one of several license features:

- `VIP-DISPORT14-SVT`: Enables DisplayPort feature upto 1.4a.
- `VIP-EDISPORT14-SVT`: Enables eDisplayPort feature upto 1.4b.
- `VIP-DP-SVT`: Enable DisplayPort feature upto 1.4a alongwith eDisplayPort feature upto 1.4b.

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

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

2.5.1.1 If Licensing Fails

By default, simulations exit with an error when a Synopsys VIP license cannot be secured. Alternatively, the DW_NOAUTH_CONTINUE environment variable can be set to allow simulations to continue when one or more VIP models fail to authorize. Unauthorized Synopsys VIP models essentially become disabled when DW_NOAUTH_CONTINUE is set to any value.

```
% setenv DW_NOAUTH_CONTINUE
```

Also, some simulation environments allow *license polling*, which pauses the simulation until a license is available. License polling is described next.

If you encounter problems with licensing, see Customer Support.

2.5.1.2 License Polling

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

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

2.5.1.3 Simulation License Suspension

All Synopsys Verification IP products support license suspension. Simulators that support license suspension allow a model to check in its license token while the simulator is suspended, then check the license token back out when the simulation is resumed.

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

2.5.2 Environment Variable and Path Settings

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

- DESIGNWARE_HOME – The absolute path to where the Synopsys VIP is installed.
- 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.
- SNPSLMD_LICENSE_FILE – The absolute path to a file that contains the license keys for Synopsys Common Licensing software or the *port@host* reference to this file.
- LM_LICENSE_FILE – The absolute path to a file that contains the license keys for your third-party tools. Also, include the absolute path to the third party executable in your PATH variable.

2.5.2.1 Simulator-Specific Settings

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

2.6 Determining Your Model Version

The version of the DisplayPort VIP at time of publication is Q-2020.06. The following steps tell you how to check the version of the models you are using.

Note Verification IP products are released and versioned by the suite and not by individual model. The version number of a model indicates the suite version.

- To determine the versions of Synopsys VIP models installed in your $DESIGNWARE_HOME tree, use the setup utility as follows:

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

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

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

2.7 Integrating a Synopsys VIP into Your Testbench

After installing a Synopsys VIP, follow these procedures to set up the VIP for use in testbenches:

- "Creating a Testbench Design Directory"
- "The dw_vip_setup Utility"
- Using Synopsys Verification IP in Your Testbench

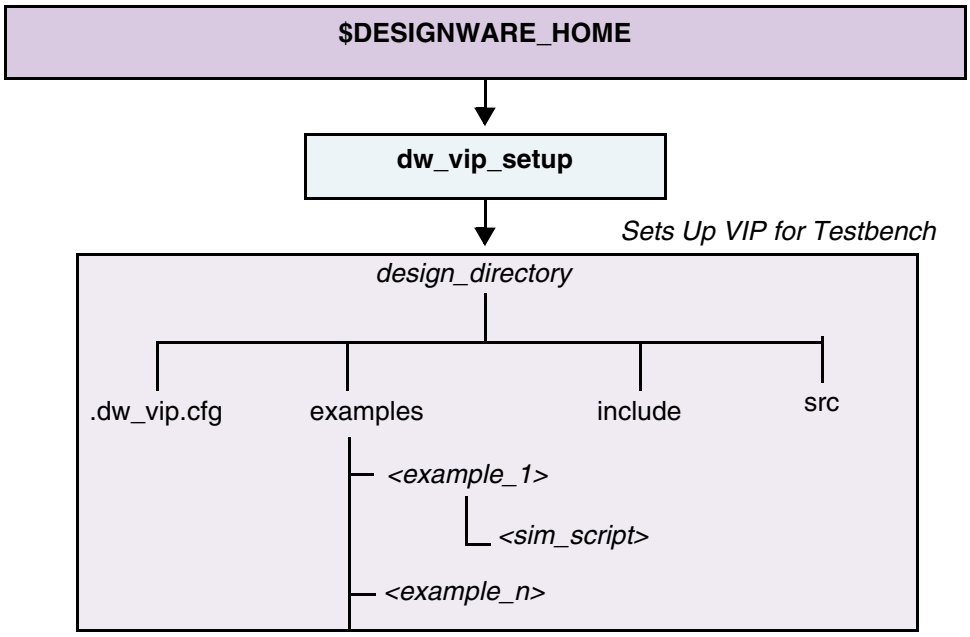
2.7.1 Creating a Testbench Design Directory

A *design directory* contains a version of the Synopsys VIP that is set up and ready for use in a testbench. You use the dw_vip_setup utility to create design directories. For the full description of dw_vip_setup, refer to The dw_vip_setup Utility.

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

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

Figure 2-2 Design Directory Created by dw_vip_setup



A design directory contains:

examples Each Synopsys VIP includes example testbenches. The dw_vip_setup utility adds them in this directory, along with a script for simulation. If an example testbench is specified on the command line, this directory contains all files required for model, suite, and system testbenches.

include Language-specific include files that contain critical information for Synopsys VIP models. This directory is specified in simulator command lines.

src Synopsys VIP-specific include files (not used by all Synopsys VIP). This directory may be specified in simulator command lines.

.dw_vip.cfg A database of all Synopsys VIP models being used in the testbench. The dw_vip_setup program reads this file to rebuild or recreate a design setup.

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

This section contains three examples that show common usage scenarios.

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

2.7.1.1 Adding or Updating Synopsys VIP Models In a Design Directory

DisplayPort VIP models include:

- dp_aux_agent_svt
- dp_aux_svt
- dp_env_svt
- dp_main_link_agent_svt
- dp_main_link_svt
- dp_hdcp_svt
- dp_aux_group_svt
- dp_main_link_group_svt
- dp_aux_msg_client_svt

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

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

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

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

In these examples, the dw_vip_setup utility does the following:

1. Creates an include directory under the current directory and copies:
   - All files in the dp_source_env_svt model include directory
   - All include files in the Synopsys VIP suite
   - The latest SVT library include files into the include directory
2. Creates the DisplayPort VIP suite libraries and SVT libraries.

2.7.1.2 Removing Synopsys VIP Models from a Design Directory

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

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

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

2.7.1.3 Reporting Information About DESIGNWARE_HOME or a Design Directory

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

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

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

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

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

2.7.2 The dw_vip_setup Utility

The dw_vip_setup utility:

- Adds, removes, or updates Synopsys VIP models in a design directory
- Adds example testbenches to a design directory, the Synopsys 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

2.7.2.1 Setting Environment Variables

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

`DESIGNWARE_HOME - Points to where the Synopsys VIP is installed`

2.7.2.2 The dw_vip_setup Command

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

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

or

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

where

[**-p**[ath] *directory*]	The optional -path argument specifies the path to your design directory. When omitted, dw_vip_setup uses the current working directory.
switch	The *switch* argument defines dw_vip_setup operation. Table 2-1 lists the switches and their applicable sub-switches.

Table 2-1 Setup Program Switch Descriptions

Switch	Description
-a[dd] (*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: • dp_aux_agent_svt • dp_aux_svt • dp_env_svt • dp_main_link_agent_svt • dp_main_link_svt • dp_hdcp_svt • dp_aux_group_svt • dp_main_link_group_svt • dp_aux_msg_client_svt The -add switch causes dw_vip_setup to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-r[emove] *model*	Removes ***all versions*** of the specified model or models from the design. The dw_vip_setup program does not attempt to remove any include files used solely by the specified model or models. The model names are: • dp_aux_agent_svt • dp_aux_svt • dp_env_svt • dp_main_link_agent_svt • dp_main_link_svt • dp_hdcp_svt • dp_aux_group_svt • dp_main_link_group_svt • dp_aux_msg_client_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: • dp_aux_agent_svt • dp_aux_svt • dp_env_svt • dp_main_link_agent_svt • dp_main_link_svt • dp_hdcp_svt • dp_aux_group_svt • dp_main_link_group_svt • dp_aux_msg_client_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.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-e[xample] {*scenario* | *model*/*scenario*} [**-v**[ersion] *version*]	The dw_vip_setup script configures a testbench example for a single model or a system testbench for a group of models. The program creates a simulator run program for all 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 available system examples.
-ntb	Not supported.
-svtb	Use this switch to set up models and example testbenches for SystemVerilog VMM. 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.
-i[nfo] *design* | *home* [:<product>[:<version>[:<methodology>]]]	Generates an informational report on a design directory or VIP installation. design: If the -info design switch is specified, the tool displays product and version content within the specified design directory to standard output. This output can be captured and used as a model list file, as an input to this tool to create another design directory with the same content. home: If the -info home switch is specified, the tool displays product, version and example content within the VIP installation to standard output. Optional filter fields can also be specified such as <product>, <version> and <methodology> delimited by colons (:). An error will be reported if a nonexistent or invalid filter field is specified. Valid methodology names include: OVM, RVM, UVM, VMM and VLOG.
-h[elp]	Returns a list of valid dw_vip_setup switches and the correct syntax for each.
model	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.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-m[odel_list] *filename*	The model names are: • dp_aux_agent_svt • dp_aux_svt • dp_env_svt • dp_main_link_agent_svt • dp_main_link_svt • dp_hdcp_svt • dp_aux_group_svt • dp_main_link_group_svt • dp_aux_msg_client_svt Specifies a file name which contains a list of model names to be added, updated or removed in the design directory. This switch is only valid when following an operation switch such as -add, -update or -remove. Only one model name per line and each model may include a version selector. The default version is 'latest'. This switch is optional, but if given the filename argument is required. Lines in the file starting with the pound symbol (#) will be ignored.
-s[uite_list] *filename*	Specifies a file name which contains a list of suite names to be added, updated or removed in the design directory. This switch is only valid when following an operation switch such as -add, -update or -remove. Only one suite name per line and each suite may include a version selector. The default version is 'latest'. This switch is optional, but if given the filename argument is required. Lines in the file starting with the pound symbol (#) will be ignored.
-b/ridge	Update the specified design directory to reference the current DESIGNWARE_HOME installation. All product versions contained in the design directory must also exist in the current DESIGNWARE_HOME installation.
-pa	Enables the run scripts and Makefiles generated by dw_vip_setup to support PA. If this switch is enabled, and the testbench example produces XML files, PA will be launched and the XML files will be read at the end of the example execution. For run scripts, specify `-pa`. For Makefiles, specify `-pa = 1`.
-waves	Enables the run scripts and Makefiles generated by dw_vip_setup to support the `fsdb` waves option . To support this capability, the testbench example must generate an FSDB file when compiled with the WAVES Verilog macro set to `fsdb`, that is, `+define+WAVES=\"fsdb\"`. If a .fsdb file is generated by the example, the Verdi nWave viewer will be launched. For run scripts, specify `-waves fsdb`. For Makefiles, specify `WAVES=fsdb`.
-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.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-methodology <name>	When specified with -doc, only documents associated with the specified methodology name are added to the design directory. Valid methodology names include: OVM, RVM, UVM, VMM and VLOG.
-copy	When specified with -doc, documents are copied into the design directory, not linked.
`-simulator <vendor>`	When used with the `-example` switch, only simulator flows associated with the specified vendor are supported with the generated run script and Makefile. **Note:** Currently the vendors VCS, MTI, and NCV are supported.

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

3 Design Directory Maintenance

The Design Directory Maintenance chapter discusses the following topics:

- Setting Up a New VIP
- Including and Importing Model Files into Your Testbench
- Compile-Time and Runtime Options

Hint This section and the next describe setting up and running example testbenches to prove that your Synopsys VIP is installed correctly.
However, if you are confident with your installation and you want to add or update the Synopsys VIP in an existing design, see Integrating a Synopsys VIP into Your Testbench.

Note For a walk-through example that demonstrates basic UVM concepts, see “QuickStart UVM Examples”.

3.1 Setting Up a New VIP

Once you have installed the VIP, you must set up the VIP for use. All VIP suites contain various components such as transceivers, masters, slaves, and monitors depending on the protocol. The setup process gathers all the required component files you need to incorporate into your testbench and simulation runs.

You have the choice to set up all of them or only specific components. For example, the DisplayPort VIP contains the following components:

- `dp_aux_agent_svt`: Defines an aux agent, which can be Source or Sink, active or passive. It contains aux Sequencer, Driver, Monitor, shared status, reporter, coverage collector and PA callbacks.
- `dp_aux_svt`: It contains methodology independent protocol implementation of aux interface. In addition, it contains infrastructure for aux callbacks, exceptions, error check and coverage.
- `dp_env_svt`: This is VIP top-level environment which contains main agent, aux agent, virtual sequencer and shared status instances. It work as a top-level enclosure for VIP agents.
- `dp_main_link_agent_svt`: Defines a main link agent, which can be Source or Sink, active or passive. It contains main link Sequencer, Driver, Monitor, shared status, reporter, coverage collector and PA callbacks.

- dp_main_link_svt: It contains methodology independent protocol implementation of main link interface. In addition, it contains infrastructure for aux callbacks, exceptions, error check and coverage.
- dp_hdcp_svt: It contains methodology independent HDCP implementation—that is, aes_cipher, rsa, sha256, hdcp configuration and hdcp 1.3 & 2.2 core.

You can set up either an individual component or the entire set of components within one protocol suite. Use the Synopsys tool, namely dw_vip_setup, for these tasks. It resides in $DESIGNWARE_HOME/bin. To get help on dw_vip_setup, use the following command:

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

The following command adds a model, svt_dp, to the design_dir directory:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/design_dir -add svt_dp
-svlog
```

This command sets up all the required files in /tmp/design_dir. The dw_vip_setup utility creates three directories in design_dir, which contain all the necessary model files. Files for every VIP are included in the following three directories:

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

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

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

3.1.1 Installing and Setting Up More than One VIP Protocol Suite

All VIPs for a project must be set up in a single common directory once you execute the *.run file. You may have different projects. In this case, the projects can use their own VIP setup directory. However, all the VIPs used by that specific project must reside in a common directory.

The examples in this chapter call that directory as design_dir, but you can use any name. In this example, assume you have the AXI-suite setup in the design_dir directory. In addition to the AXI VIP, you require the Ethernet and USB VIP suites.

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

Once installed, you must setup and locate the Ethernet and USB suites in the same design_dir location as AMBA. Use the following commands:

```
// First install AXI.
```

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

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

// Add USB to the same design_dir directory as AMBA and Ethernet
%unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path  /tmp/design_dir
-add usb_system_env_svt -svlog
```

To specify other model names, consult the VIP documentation.

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

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

- All installed VIP models will use the EA SVT after the EA VIP is installed.
- Synopsys attempts to maintain backward compatibility of new EA SVT releases with the latest LCA/GA VIPs.
- In the case where the EA SVT changes are not backward incompatible, you can use the '-svt' option of dw_vip_setup to use a specific version of SVT.
- Synopsys does not recommend you use the '-svt' option as you must remember to remove this when all VIPs move to a compatible version of SVT. Please only use '-svt' as a work-around.
- Following is an example of using the -svt switch:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/vip_model_design_dir
-add amba_system_env_svt svlog -svt Q-2020.06
```

3.2 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/dp_svt/tb_dp_svt_uvm_basic_sys/

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

./run_dp_svt_uvm_basic_sys -32 -incdir adaptive_sync_test vcsvlog
```

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

```
vcs -l ./logs/compile.log -q -Mdir=./output/csrc
+define+DESIGNWARE_INCDIR=<DESIGNWARE_HOME> \
```

```
+define+SVT_LOADER_UTIL_ENABLE_DWHOME_INCDIRS
+incdir+<DESIGNWARE_HOME>/vip/svt/dp_svt/O-2018.09-2/sverilog/include \
+incdir+<DESIGNWARE_HOME>/vip/svt/hdcp_svt/latest/sverilog/include \
-ntb_opts uvm -sverilog +define+UVM_PACKER_MAX_BYTES=1500000
+define+UVM_DISABLE_AUTO_ITEM_RECORDING \
-timescale=1ns/1ps +define+SVT_DP_UI_MAN_SCALEDOWN_FACTOR=100
+define+SVT_DP_HPD_SCALEDOWN_FACTOR=100 \
+define+SVT_DP_LT_SCALEDOWN_FACTOR=100
+define+SVT_DP_HPD_DISCONNECT_DURATION_MAX_NS=20000000 \
+define+SVT_DP_HPD_DEBOUNCE_PERIOD_NS=10000000
+define+SVT_DP_HPD_UNPLUG_DURATION_MAX_NS=10000000 \
+define+LOADFILE_PPM_DIR=<testbench_dir>/examples/sverilog/dp_svt/tb_dp_svt_uvm_basic_sy
s \
+define+SVT_DP_EXT_RESET_FROM_TB +define+SVT_UVM_TECHNOLOGY +define+SYNOPSYS_SV
+incdir+<testbench_dir>/examples/sverilog/dp_svt/tb_dp_svt_uvm_basic_sys/. \
+incdir+<testbench_dir>/examples/sverilog/dp_svt/tb_dp_svt_uvm_basic_sys/../../env \
+incdir+<testbench_dir>/examples/sverilog/dp_svt/tb_dp_svt_uvm_basic_sys/../env \
+incdir+<testbench_dir>/examples/sverilog/dp_svt/tb_dp_svt_uvm_basic_sys/env \
+incdir+<testbench_dir>/examples/sverilog/dp_svt/tb_dp_svt_uvm_basic_sys/dut \
+incdir+<testbench_dir>/examples/sverilog/dp_svt/tb_dp_svt_uvm_basic_sys/hdl_interconnec
t \
+incdir+<testbench_dir>/examples/sverilog/dp_svt/tb_dp_svt_uvm_basic_sys/lib \
+incdir+<testbench_dir>/examples/sverilog/dp_svt/tb_dp_svt_uvm_basic_sys/tests \
-o ./output/simvcssvlog -f top_files -f hdl_files
```

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

3.2.1 Supported Methodologies with Simulators

Table 3-1 lists the methodologies supported with simulators.

Table 3-1 Supported Methodologies with Simulators

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

3.2.2 Getting Help on Example Run/make Scripts

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

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

```
run_dp_svt_uvm_basic_sys
usage: run_dp_svt_uvm_basic_sys [-32] [-incdir] [-verbose] [-debug] [-waves] [-clean]
[-nobuild] [-norun] <scenario> <simulator>
  where   <scenario> is one of:   all adaptive_sync_test audio_v1_4_with_video_test
aux_native_rd_wr_test compressed_frame_with_decomp_data_test custom_frame_type_A
custom_frame_VIC_1_60Hz edid_wr_rd_test ext_reset_from_tb_test
```

```
hdcp_1_3_first_stage_auth_test hdcp_2_2_auth_repeater_test
main_link_clk_adjustment_test mst_basic_test_with_enumeration mst_fec_no_aux_test
mst_hdcp_2_2_test multilane_fec_video_basic_test multi_sst_2_lanes_2_sst_test
multi_sst_4_lanes_2_sst_test multi_sst_4_lanes_4_sst_test
parse_ext_file_for_compression_test parse_ext_file_test video_basic_test
wrong_aux_stop_exception
<simulator> is one of:  vcsvlog vcsmxvlog 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    enable debug mode for SVT simulations
         -waves    [fsdb|verdi|dve|dump] enables waves dump and optionally opens viewer
(VCS only)
          -clean    clean simulator generated files
          -nobuild  skip simulator compilation
          -norun    exit after simulator compilation
```

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|1|] [NOBUILD=1] [<scenario> ...]
Valid simulators are: vcsvlog vcsmxvlog mtivlog ncvlog
Valid scenarios are:  all adaptive_sync_test audio_v1_4_with_video_test
aux_native_rd_wr_test compressed_frame_with_decomp_data_test custom_frame_type_A
custom_frame_VIC_1_60Hz edid_wr_rd_test ext_reset_from_tb_test
hdcp_1_3_first_stage_auth_test hdcp_2_2_auth_repeater_test
main_link_clk_adjustment_test mst_basic_test_with_enumeration mst_fec_no_aux_test
mst_hdcp_2_2_test multilane_fec_video_basic_test multi_sst_2_lanes_2_sst_test
multi_sst_4_lanes_2_sst_test multi_sst_4_lanes_4_sst_test
parse_ext_file_for_compression_test parse_ext_file_test video_basic_test
wrong_aux_stop_exception
```

3.2.3 Updating VIP Components

To update an existing model, perform the following steps:

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

```
%unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path  /tmp/design_dir
-add dp_svt -svlog
```

You can also update your `design_dir` by specifying the version number of the model.

```
%unix> dw_vip_setup -path design_dir -add dp_svt -v Q-2020.06
```

3.3 Including and Importing Model Files into Your Testbench

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

```
/* include uvm_package before VIP includes, if not included elsewhere */
`include "uvm_pkg.sv"

/* include DisplayPort VIP interface */
```

```
`include "svt_dp_if.svi"

/** Include the DP SVT UVM package */
`include "svt_dp.uvm.pkg"

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

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

  /** Import the DisplayPort VIP */
  import svt_dp_uvm_pkg::*;
```

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

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

Supported vendors are vcs, mti, and ncv. For example:

```
+incdir+<design_dir>/src/sverilog/vcs
```

Using the previous examples, the <design_dir> directory would be /tmp/design_dir.

3.4 Compile-Time and Runtime Options

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

```
$DESIGNWARE_HOME/vip/svt/dp_svt/latest/examples/sverilog/<test_name>
```

The following files contain the options:

- For compile-time options:

  sim_build_options (also vcs_build_options)

- For runtime options:

  sim_run_options (also vcs_run_options)

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

vcs_build_options

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

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

vcs_run_options

Required: `+UVM_TESTNAME=$scenario`

Note `scenario` is the UVM test name you pass to VCS.

4 Overview of the DisplayPort VIP

4.1 Introduction to UVM

UVM is an object-oriented approach. It provides a blueprint for building testbenches using the constrained random verification. In addition, the resulting structure supports Directed testing. This chapter describes the data objects that support higher structures which comprise DisplayPort VIP. Refer to the Class Reference HTML for a description of attributes and properties of the objects mentioned in this chapter. This chapter assumes that you are familiar with SystemVerilog and UVM.

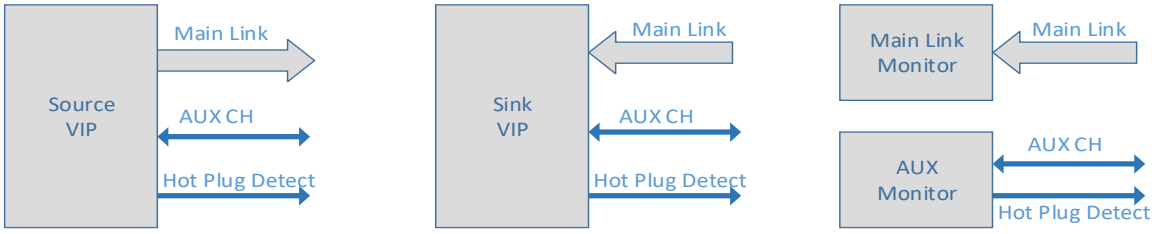
This section describes the interface details. The Main-Link consists of one, two, or four AC-coupled, doubly terminated differential pairs (called lanes). The Source Main Link physical interface consists of one, two, or four AC-coupled, doubly terminated differential pairs (called lanes) and those will be input at Sink Main Link physical interface.

AUX CH consists of an AC-coupled, doubly-terminated differential pair. It has a half-duplex, bidirectional PHY Layer. The Source device is the master and the Sink device the slave. A Sink device may toggle the HPD signal to prompt the Source device to initiate an AUX request transaction to read DPCD Link/Sink status register.

4.2 Interfaces

Interface signals to Source, Sink and Monitor for Main Link and AUX link are defined as below:

Figure 4-1 Display Port Interface



All lanes of Main link interface and AUX CH carry data. There is no dedicated clock channel. The clock is extracted from the data stream itself. Source and Sink devices are allowed to support the

minimum number of lanes required for their needs. Four link rates are supported, that is 8.1, 5.4, 2.7, and 1.62Gbps/lane. All enabled lanes must be operating at the same link rate.

AUX CH provides a data rate of 1Mbps.

4.2.1 Interface Declaration

The section describes a pseudo code for the interfaces. The sample code for interfaces is shown below:

```
interface svt_dp_main_link_if ();

   /** Indicates reset pulse */
   logic rst;
   /**
    * Main Link lane 0
   */
   logic lane_0;
   /**
   * Main Link lane 1
   */
   logic lane_1;

   /**
    * Main Link differential line.
    */
   logic lane_2;
..............
..............
clocking dp_main_link_monitor_cb @(posedge ls_clk);
.......
endclocking : dp_main_link_monitor_cb

modport svt_dp_main_link_modport (clocking dp_main_link_cb, input ls_clk);
..............
..............

endinterface: svt_dp_main_link_if
```

Note Preliminary signals are mentioned for interface and it may change during implementation and will be updated in the documentation.

4.2.1.1 DP Main Link Interface

```
interface svt_dp_main_link_if();
```

Note For more details on interface pins, refer to the following location:
$DESIGNWARE_HOME/vip/svt/dp_svt/latest/doc/dp_svt_uvm_class_reference/html/interfaces.html

4.2.1.2 DP AUX Interface

```
interface svt_dp_aux_if (
  input logic aux_clk);
```

For more details on interface pins, refer to the following location:
$DESIGNWARE_HOME/vip/svt/dp_svt/latest/doc/dp_svt_uvm_class_reference/html/interfaces.html

4.2.1.2.1 Link Training Debug States

Following states from Table 4-1 are encoded on the link_training_state debug bus, you can see these states in simulation waveform in ASCII form:

Table 4-1 Link training states

State Name	Description
CR_training_start	Link training Starts
CR_training_lanex_set_v_lev_pe_lev	Updating TRAINING_LANEx_SET registers with voltage level and pre-emphasis level, Traning start with 0 0
CR_wait_rd_training_interval	Transmitter wait at least the period of time specified by TRAINING_AUX_RD_INTERVAL register
CR_check_cr_done	Read LANE*x*_CR_DONE register
CR_check adj_req lanex	Read ADJUST_REQUEST_LANE*x_y* both registers
CR_max_level_or_same_five_time	Check if either maximum Voltage Swing reached or same Voltage five times
CR_reduce_link_rate	Reducing link rate
CR_reduce_lane_count	Reducing lane count
CR_training_failed	
CE_training_lanex_set_v_lev_pe_lev	Updating TRAINING_LANEx_SET registers with voltage level and pre-emphasis level, Channel Equalization starts with current values, determined by Clock Recovery sequence
CE_ wait_rd_training_interval	
CE_ check_cr_done	Read LANE*x*_CR_DONE register
CE_check adj_req_lanex	Read ADJUST_REQUEST_LANE*x_y* both registers
CE_check ch_eq_done	Read LANE*x*_CHANNEL_EQ_DONE bit in the LANE*x_y*_STATUS register(s)
CE_check_lanex_symbol_locked	Read LANE*x*_SYMBOL_LOCKED bit in the LANE*x_y*_STATUS register(s)
CE_reduce_link_rate	Reducing link rate
CE_training_failed	
CR: Clock Recovery, CE: Channel Equalization	

4.3 DP Main Link Configuration

4.3.1 DP Configuration Class

This class will define configuration variables related to DP Main Link, Aux and HPD operation. The Configuration class includes some random and non-random attributes. For all the random attributes, appropriate constraints will be added. This Configuration holds array instance of DP Stream configuration and creates array depth based on MST/SST mode and number of streams supported. This configuration will also hold the virtual interface pointer to their DP main link interface and DP Aux interface.

Class name: `svt_dp_configuration` extends `svt_configuration`;

For more details on Data members of main link configuration class, see the following location:

$DESIGNWARE_HOME/vip/svt/dp_svt/latest/doc/dp_svt_uvm_class_reference/html/

class_svt_dp_configuration.html

4.3.2 DP Stream Configuration Class

This class will define configuration variables related to DP Streams, it is useful in setting different configurations for different stream in MST operation. DP configuration class has an array instance of this class which will be created based on MST/SST and number of supported streams. In case of SST it is tied to 0 and creates only single stream configuration. The Configuration class includes some random and non-random attributes. For all the random attributes, appropriate constraints will be added. This configuration will also have array instance of Audio configuration as each stream can hold multiple audio streams.

Class name: `svt_dp_stream_configuration` extends `svt_configuration`;

For more details on Data members of DP Stream configuration class, see the following location:

$DESIGNWARE_HOME/vip/svt/dp_svt/latest/doc/dp_svt_uvm_class_reference/html/

class_svt_dp_stream_configuration.html

4.3.3 DP Audio Configuration Class

This class will define configuration variables related to DP Audio Streams, it is useful in setting different configurations for different Audio streams in SST/MST operation. DP Stream configuration class has an array instance of this class which will be created based on number of supported audio streams. In case of single audio stream, it is tied to 0 and creates only single audio stream configuration. The Configuration class includes some random and non-random attributes. For all the random attributes, appropriate constraints will be added.

Class name: `svt_dp_audio_configuration` extends `svt_configuration`;

For more details on Data members of DP Audio configuration class, see the following location:

$DESIGNWARE_HOME/vip/svt/dp_svt/latest/doc/dp_svt_uvm_class_reference/html/

class_svt_dp_audio_configuration.html

4.3.4 DP Env Configuration Class

This class extends from `svt_dp_configuration` and also defines non-rand control variables which will control various verification aspects of the DP main link and Aux agents.

For more details on Data members of *env* configuration class, see the following location:

$DESIGNWARE_HOME/vip/svt/dp_svt/latest/doc/dp_svt_uvm_class_reference/html/

class_svt_dp_env_configuration.html

4.4 DP Status Class

The DP status class is used to define values for a Main Link/AUX. It is common for both types of agents.

For more details on Status Class members of AUX configuration class, refer to the following location:

```
$DESIGNWARE_HOME/vip/svt/dp_svt/latest/doc/dp_svt_uvm_class_reference/html/class_svt_dp_status.html
```

4.5 DP Agents

4.5.1 Overview

Agents are UVM components that are composed of basic components such as drivers, monitors and sequencers. The DisplayPort VIP defines an agent that can be customized to support a variety of verification scenarios. This section describes the DisplayPort VIP agent component. Refer to the HTML Class Reference for description of objects, classes, and attributes mentioned in this section.

The DisplayPort VIP should be built upon Synopsys' SVT technology and in accordance to Synopsys' guidelines. It needs to support different SV verification methodology flavors: VMM, OVM and UVM.

In a protocol sense DisplayPort VIP needs to support:

- "DP 1.3 protocol version
- "Source and Sink devices
- "All CEA Video formats

The DisplayPort VIP will support Source and Sink devices. Each device is implemented in one environment that is comprised of Main Link agent and AUX agent. Agents' and Environment's architectures are defined in UVM/OVM manner. Same architecture concepts should be followed in VMM implementation according to recommendations from SVT Developers Guide document.

Figure 4-2 DP Environment

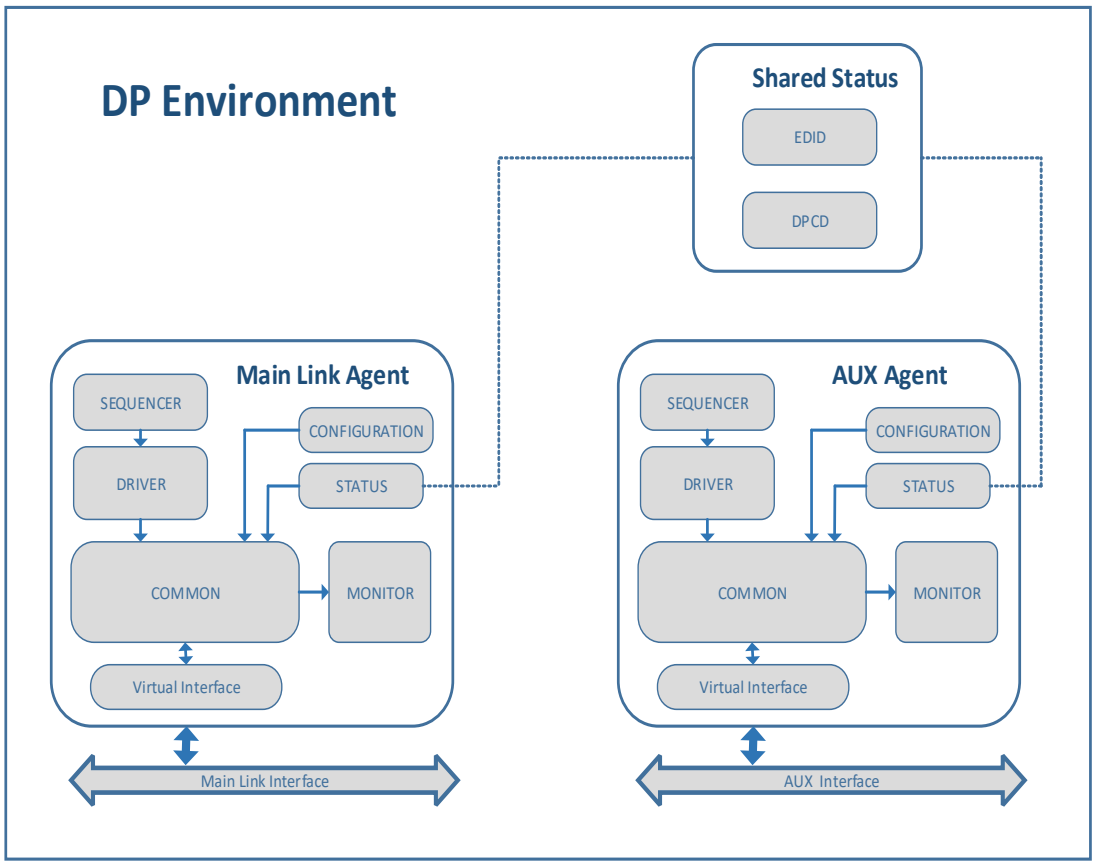


Figure 4-2 shows a DP Environment with active Main Link agent and active AUX agent. Both components share status object which contains all DPCD and EDID structures. Main Link agent reads register values while AUX agent can read and write. In case when there is no AUX agent, you have to provide register values through configurations and status will be initialized with those values. Detailed description for both components is provided in the following sections.

4.5.2 Main Link Agent architecture

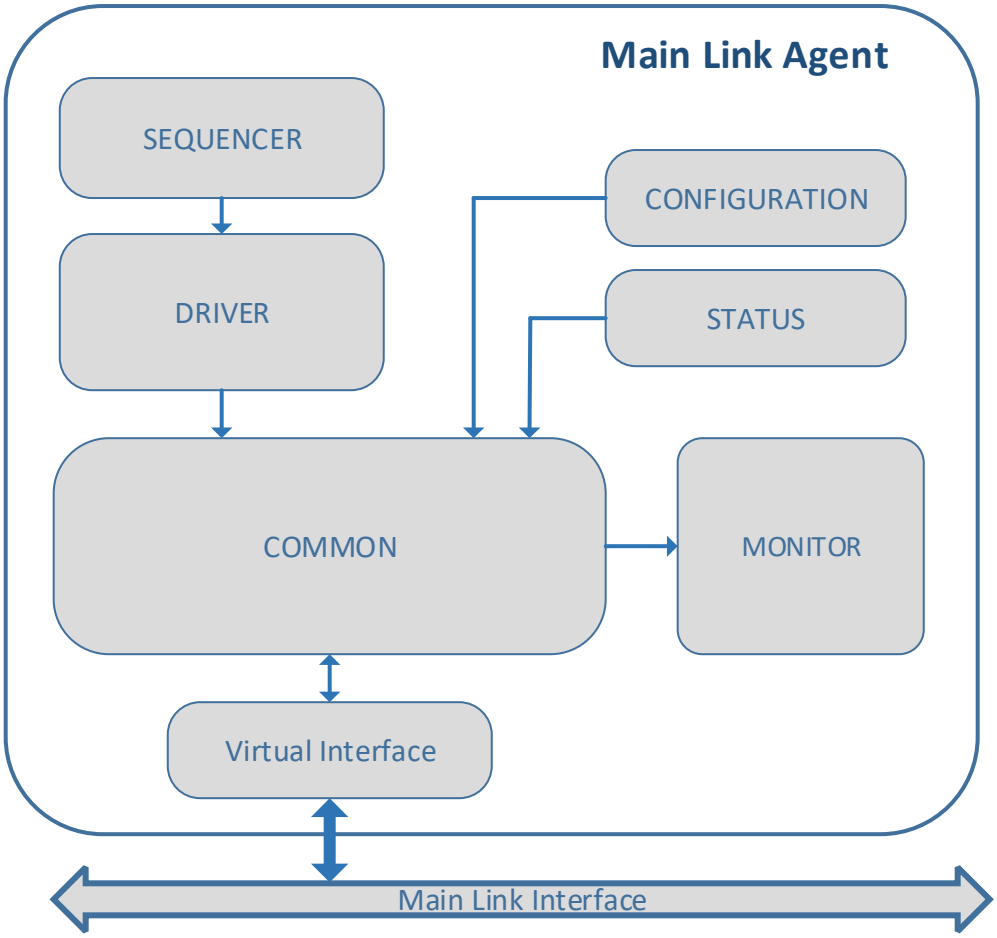
Figure 4-3 shows the basic architecture of a Main Link agent. This agent will have dedicated set of components (sequencer, driver & monitor) based upon active/passive setting (is_active field in the agent configuration class).

The active Main Link agent is responsible for transporting video and audio data. It is required that this agent instantiates driver and sequencer. A sequencer will get its respective transaction from upper layer or testbench. The transactions will be passed to the related DP Main Link driver. On every get request the related sequencer will provide the transaction which will be encoded considering DP Main Link specification and transmitted on corresponding link.

Monitor is instantiated in both active and passive agent. It will be responsible to collect the bit level information from link and decode the received bits and make transaction from the same. A monitor will also be used for protocol checking and coverage. Each agent will use the status class object to keep track of the device registers' current values.

The configuration class instance contains information for configuring video and audio formats and transport and it will exist in each agent.

Figure 4-3 Main Link Agent architecture



4.5.2.1 Driver

The driver is methodology dependent. In order to enable code sharing between the different methodologies, all protocol implementation resides in the Common class. Therefore, the driver will be very generic and it will be top level methodology class that instantiates and utilizes Common class object.

The driver responsibility is only to fetch a transaction from the sequencer and to pass it to the Common class object by calling its send_frame_line_transaction() task which will handle further transaction processing and driving. This task is blocking and it will return only when driver starts to drive last pixel in the line. By returning at the beginning of the last pixel instead of the end we are avoiding potential race condition between sequencer/driver at one side and common driving process at the other side. After this task is completed, driver calls `item_done` to indicate to the sequencer that the request is completed so it can send another item (video line).

4.5.2.2 Sequencer

A sequencer is an advanced stimulus generator that controls the items that are provided to the driver for execution. Requests for new sequence items are initiated by the driver. Upon such requests, the sequencer selects a sequence from a list of available sequences to produce and deliver the next item to execute. Main Link basic sequence represent one complete video frame with auxiliary data (audio data, MSA, SDPs, etc.) while compound sequence represent video stream.

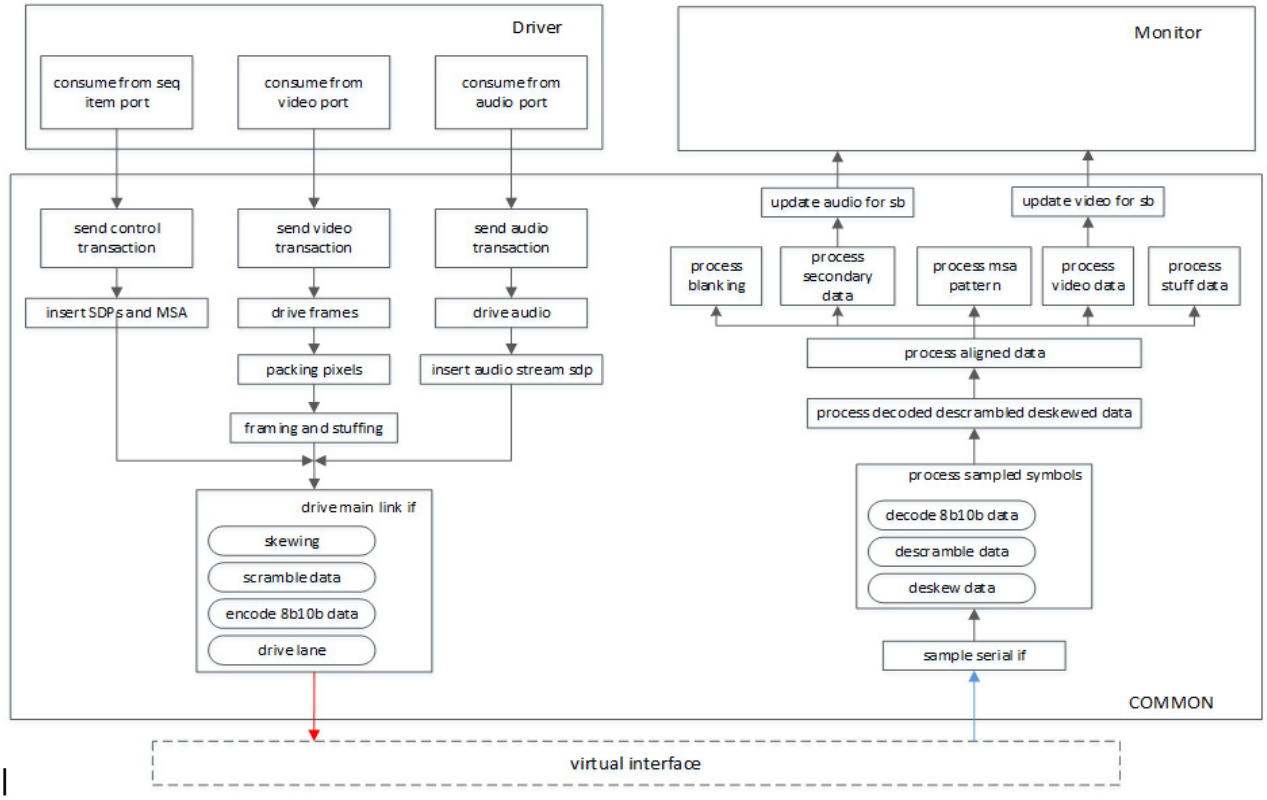
4.5.2.3 Monitor

Because all protocol implementation is done in the Common class, the monitor is just a shell which is accessible by the user. Transactions collected by monitor methods within Common class object will be passed to the monitor through callbacks. It is monitor's responsibility to provide callbacks. Detailed callbacks support has to be defined. Checkers will also be implemented in the Common class.

4.5.2.4 Common class

The Common class contains a complete protocol implementation. Implementation is methodology independent so that the same Common class is used for all three methodologies. For VIP performances, a single instance of Common class is instantiated in the agent and is used by both the driver and the monitor. The Common class basic elements are shown in Figure 4-4.

Figure 4-4 Main Link common class structure



4.5.2.4.1 Description of driving part

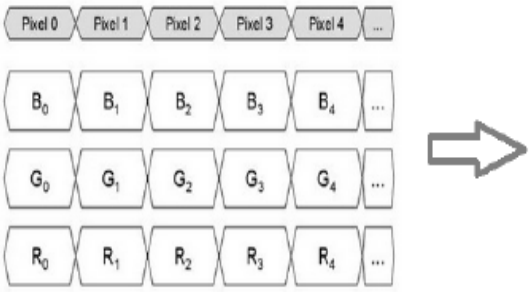
- This section describes the functionality of these logical blocks and their implementation through Common class methods.
- The main_stream_data_packing() function is responsible for steering pixel data in a pixel-within-lane manner, as shown in .

Table 4-2 Pixel steering inot Main Link lanes

Number of Lanes	Pixel Steering[a]
One	All pixels to Lane 0
Two	Pixel 2N to Lane 0 Pixel 2N+1 to Lane 1
Four	Pixel 4N to Lane 0 Pixel 4N+1 to Lane 1 Pixel 4N+2 to Lane 2 Pixel 4N+3 to Lane 3
a. *N is 0 or positive integer*	

These rules apply regardless of the color space/pixel bit depth of the video stream. During the last symbol time for a line of pixel data, there may be insufficient pixel data to provide data on all lanes of the link. The Source device must send 0s for those bits (zero-padded bits). The Sink device with the number of active pixels per horizontal line (from the main stream attribute), must discard zero-padded bits as "don't care." This contains functionality related to fragmentation of pixel data in active video line. It will put the video data to be transmitted on each active lane in an appropriate array. These are arrays of active video data symbols (8 bits data). Fragmentation of pixel data to data symbols is based on the pixel encoding format and component/pixel bit depth. Fore more detailes on fragmentation, refer to the VESA Display Port Standard, Version 1.3, chapters 2.2.1.3.1 - 2.2.1.3.24. One example fragmentation pixel data and mapping into Main-Link lanes and is given in Figure 4-5. It illustrates 8bpc RGB stream mapping into a 4-lane Main-Link. Bit 7 of each color is mapped to bit 7 of each lane, while bit 0 of each color is mapped to bit 0 of each lane.

Figure 4-5 8bpc RGB stream mapping into a 4-lane Main-Link



Lane 0	Lane 1	Lane 2	Lane 3
R0-7:0	R1-7:0	R2-7:0	R3-7:0
G0-7:0	G1-7:0	G2-7:0	G3-7:0
B0-7:0	B1-7:0	B2-7:0	B3-7:0
R4-7:0	R5-7:0	R6-7:0	R7-7:0
G4-7:0	G5-7:0	G6-7:0	G7-7:0
B4-7:0	B5-7:0	B6-7:0	B7-7:0
R8-7:0	R9-7:0	R10-7:0	R11-7:0
G8-7:0	G9-7:0	G10-7:0	G11-7:0
B8-7:0	B9-7:0	B10-7:0	B11-7:0

The `symbol_stuffing()` function is responsible for stuffing dummy symbols to keep the link up. The packed data rate must be equal to or lower than the link symbol rate in order to avoid the over subscription of the link bandwidth. When the packed data rate is lower than the link symbol rate, the symbol stuffing must be performed. The function calculates the number of symbols to be stuffed.

The steered pixel data is packed into a Micro-Packet which is called the TU (or Transfer Unit). The TU is 32-64 symbols long per lane. The length of the TU is fixed to limit the size of buffers needed. As shown in Figure 4-7, stuffing symbols are inserted when the Display Port transmitter does not have enough valid symbols to transmit within the TU. The FS and FE are the framing symbols indicating the start and end of stuffing symbols respectively.

Figure 4-6 Transfer unit

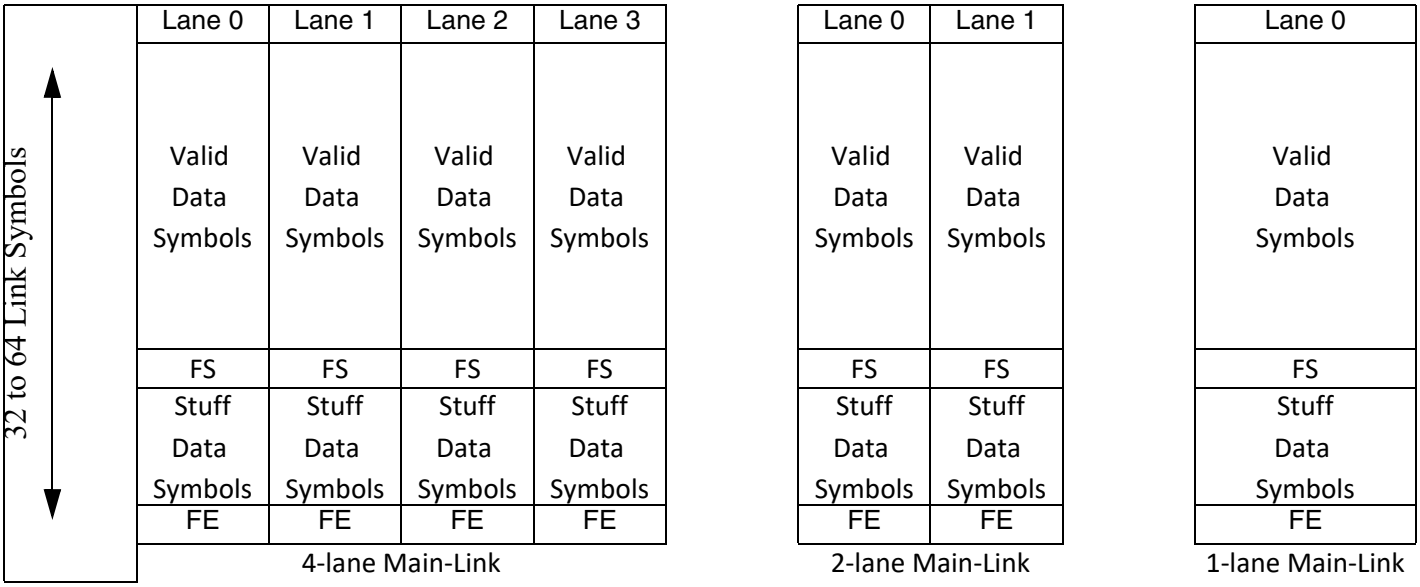
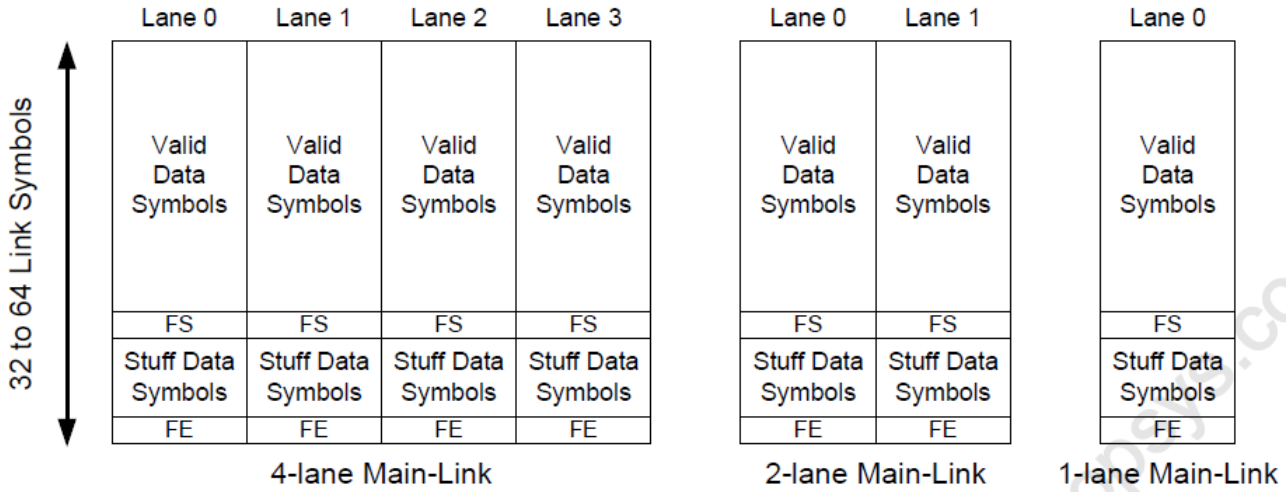


Figure 4-7



The framing() function is responsible for adding framing mode control symbols (BS, BE, SS, SE, FS, FE, SR etc.) as well as other control symbols (VB-ID, Mvid7:0, Maud7:0) which forms the basic

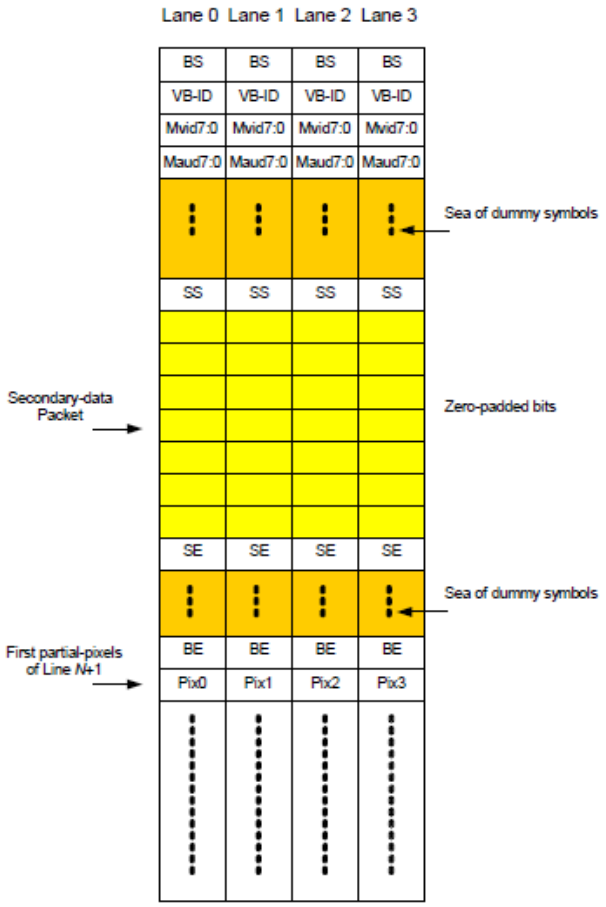
structure that will be driven on each lane. The VB-ID, Mvid7:0, and Maud7:0 must be transported four times, regardless of the number of lanes in the Main-Link.

The function also sends Idle Pattern during Link training and generates the scrambler reset pulse for every 512th BS symbol.

The msa_insertion() function is responsible for substituting the dummy stuffing data symbols during the video blanking periods with main stream attribute (MSA) data if transferring_msa transaction field is set to 1. The MSA data must be framed with two consecutive SS and one SE control symbols per lane.

The sdp_insertion() function is responsible for substituting the dummy stuffing data symbols during the video blanking periods with secondary data packet (SDP). The function will be called for each SDP type which is indicated by an appropriate transaction flag (audio_copy_management, audio_time_stamp, audio_stream, isrc, vsc, camera_generic, extension, avi_info_frame, audio_info_frame, vendor_info_frame) for sending in this line. The SDP data must be framed SS and SE control symbols per lane and must be protected through ECC. Insertion of secondary data packets is illustrated in Figure 4-8.

Figure 4-8 Secondary-Data insertion



The SDP must consist of a 4-byte header protected by four bytes of parity, followed by a payload consisting of a multiple of 8 bytes of data protected by the same multiple of 2 bytes of parity (that is, 8 packet body bytes have 2 bytes of parity, 16 packet body bytes have 4 bytes of parity, and so on).

The SDP must end with a parity byte. SDPs constructed with fewer than 8 bytes of data must use zero padding to fill the remaining data positions.

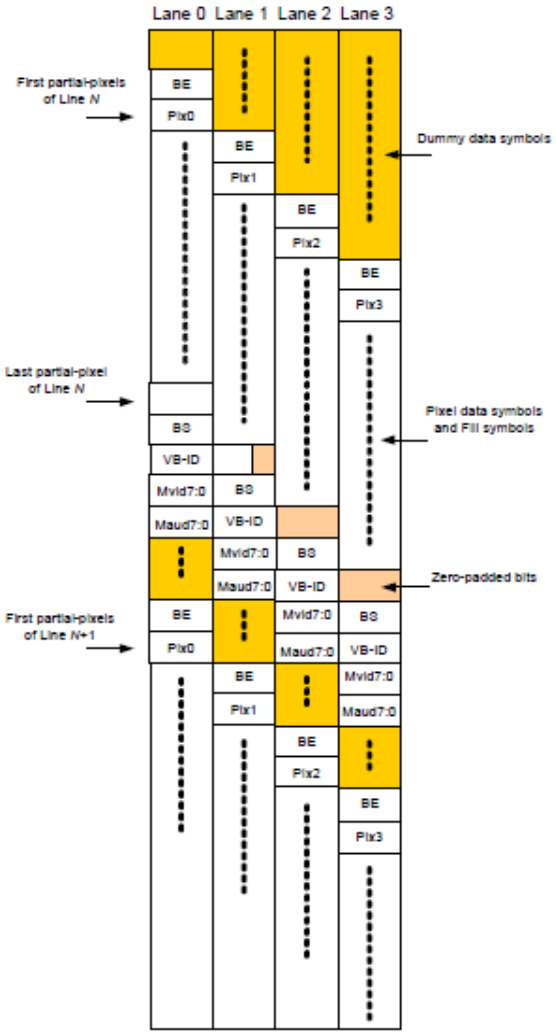
Secondary Data Packets are used for transmitting a variety of data types which are shown in Table 4-3. Therefore, this function will call a different number of auxiliary functions responsible for packing SDPs in an appropriate form, deciding the place within the line, ECC encoding.

Table 4-3 Secondary Data Packets types

SDP Types	Description
Audio Time Stamp	Sent once per video frame for audio-audio and audio-video synchronization.
Audio Stream	Inserted within video stream blanking period. An Audio Stream SDP includes audio stream itself and some attribute information such as audio coding type and channel count. Depending on the coding type of Audio Stream SDP, it may contain status information about parameters of the audio stream.
Extension	An application- or vendor-specific option.
Audio Copy Management	Content protection for audio.
ISRC	Transport of an International Standard Recording Code (ISRC) SDP is an application-specific option. When an audio stream is transported and it has specific ISRC and/or UPC/EAN requirement from the higher level application, the ISRC SDP describing the UPC_EAN_ISRC information of the audio stream must also be transported. For general audio stream that has no ISRC or UPC/EAN requirement, no ISRC SDP will be sent.
VSC (Video Stream Configuration)	Contains additional 3D format information not declarable in the MSA field.
Camera Generic 0 through 7	Reserved for vendor-specific camera application use.
CEA-861-F InfoFrames	Sent once per video frame for each InfoFrame packet type.

The skewing() function is responsible for inserting a skew of two LS_Clk cycles between adjacent lanes. The inter-lane skewing is done after inserting the Main-Link attributes data (and optionally, SDP and it applies to all the symbols, both those transmitted during video display period and those transmitted during video blanking period. Figure 4-9 shows how the symbols must be transported after this inter-lane skewing. The Symbol trace file of VIP will show similar data (encoded, decoded, scrambles) for each lane.

Figure 4-9 Inter-lane skewing



The scrambling() function contains functionality related to scrambling the data. Scrambling of the Main-Link data is performed for EMI reduction prior to ANSI 8b/10b encoding on the transmitter. Each of the Main-Link lanes is scrambled independently, each with a 16-bit internal LFSR, as follows:

G(X) = X16 + X5 + X4 + X3 + 1

Each byte of data is scrambled with the most significant 8 bits of the LFSR in reverse bit order:

```
{D'[7], D'[6], D'[5], D'[4], D'[3], D'[2], D'[1], D'[0]}
= {D[7], D[6], D[5], D[4], D[3], D[2], D[1], D[0]} ^ {LFSR[8]. LFSR[9], LFSR[10],
LFSR[11], LFSR[12], LFSR[13], LFSR[14, LFSR[15]}
```

In SST mode, every 512th BS BF BF BS or BS CP CP BS symbol sequence must be replaced with an SR BF BF SR or SR CP CP SR symbol sequence. The SR symbol or SR BF BF SR or SR CP CP SR symbol sequence is used to reset the LFSR to, so that the first byte of data following the scrambler reset is scrambled/de-scrambled with FFh. The scrambler is then advanced to contain E817h.

The data scrambling rules are as follows:

- FSR advances on all symbols, both data symbols (D), and special symbols (K).

- Special symbols (K) are not scrambled.
- Data symbols, including "fill data" are scrambled. Fill data is normally zero before scrambling.
- Multi-stream indexed control symbols are scrambled before being encoded as special symbols (K).

Scrambling must be disabled during Link Training and Recovered Link Clock Quality Measurement. Receivers should implement appropriate robustness to ensure that bit errors that generate a false SR symbol do not result in the de-scrambler LFSR being reset.

The 8b_10_encoding process maps 8-bit symbols to 10-bit symbols to achieve DC balance and bounded disparity, and yet provide enough state changes to allow reasonable clock recovery. Therefore, Main Link agent will use the following SVT function:

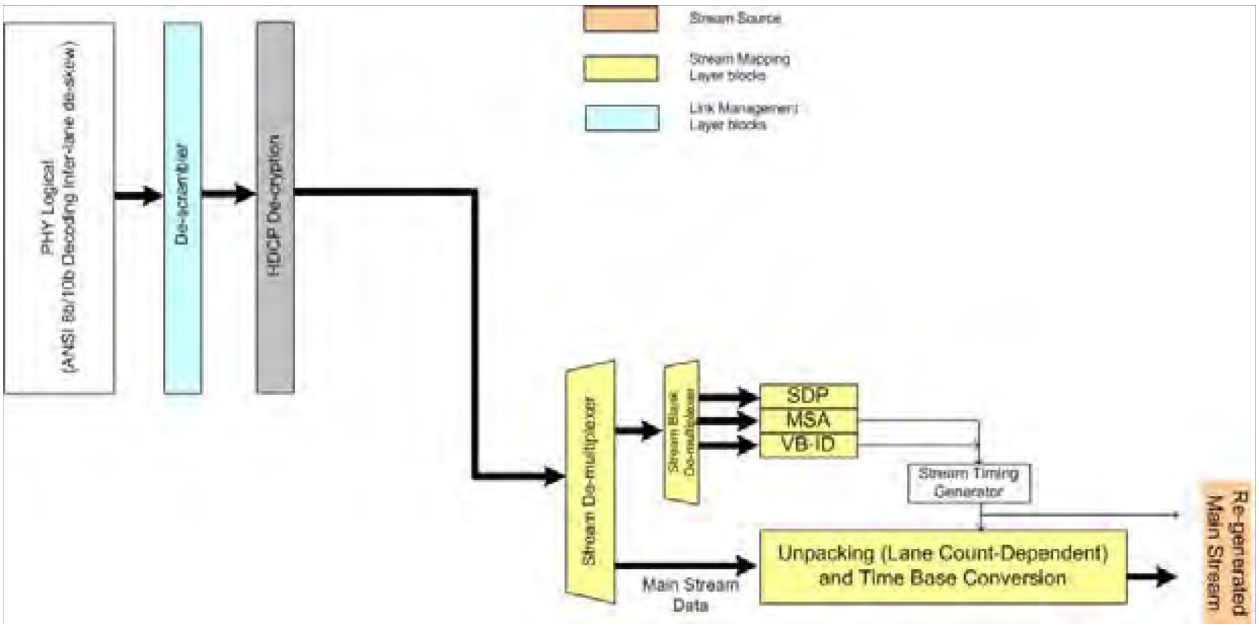
```
function bit
svt_data_converter :: encode_8b10b_data (input bit[7:0] data_in,  input
bit data_k, ref bit running_disparity,out-put  bit[9:0] data_out)
```

The serial_driving() task drives on the bus serialized 10-bit symbols. The least significant bit (lsb) is transported first and the most significant bit (msb) last.

4.5.2.5 Description of monitor part

Figure 4-10 illustrates the logical block diagram of monitor part of a Main Link agent.

Figure 4-10 Monitor part of SST DP Main Link agent - logical block diagram



This section describes how the functionality of these logical blocks will be implemented through Common class processes.

The sampling_serial_if() task captures data from the interface.

The 8b_10_decoding process decodes a 10-bit data value into its 8-bit representation. Therefore, Main Link agent will use the following SVT function:

```
function bit
svt_data_converter :: decode_8b10b_data (bit[7:0] data_in, ref bit running_disparity,
out-put bit data_k, out-put bit[7:0] data_out )
```

Following ANSI 8b/10b decoding descrambling() function is performed at the receiver side. Each of the Main-Link lanes is descrambled independently, each with a 16-bit internal LFSR. Each byte of data is descrambled with the most significant 8 bits of the LFSR in reverse bit order.

The deskewing() function is responsible for removing the two LS_Clk skewing among ad-jacent lanes.

The error_correction() function is filtering out any intermittent data corruption by comparing values of MSA fields with the previous values. All the values of Display Port main stream attributes except for time stamp value Mvid must stay constant. As for the time stamp values Mvid/Maud and VB-ID, "majority voting" must be used to determine the value.

The sdp_demultiplexing() function is responsible for de-multiplexing secondary data packets using SS and SE as the separator. Upon extracting the SDP Reed-Solomon (15:13) (RS (15:13)) decoding must be performed.

The unstuffing() function is responsible for removing stuffing symbols.

The unpacking() function contains functionality related to reconstruction of pixel data from data characters transported over the Main-Link. Unpacking is dependent on the pixel data color depth and format.

The send_collected_transaction() function contains functionality related to passing transactions to the monitor once a transaction is collected.

Protocol Checks are also implemented by the common class to provide a complete check of protocol requirements. They are implemented as a set of independent, concurrent and continuous processes. A unique identifier is associated with each check performed. That identifier also provides a cross-reference to a requirement in the DP protocol specification that is not being met if the check fails.

4.6 AUX Agent architecture

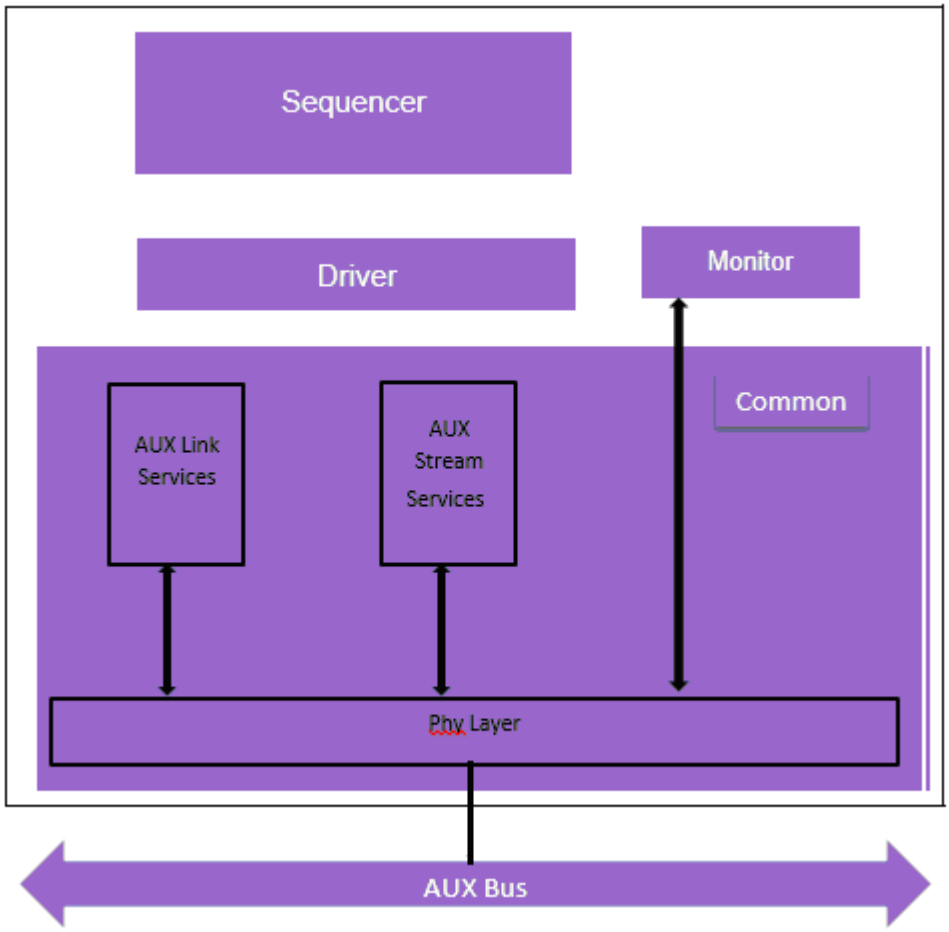
Link Layer of the DP protocol contains two Policy Makers, Link Policy maker and Stream Policy Maker with its own responsibility. These makers use AUX services that must be available whenever HPD signal is active. Therefore, AUX Channel provides two types of services, AUX Device Services and AUX Link Services. Stream Policy Maker manages stream. It is responsible for stream transport initialization and maintenance. It uses AUX Device Services. Optionally this maker gets link information from Link Policy Maker. Link Policy Maker manages link. Its responsibility is link discovery, initialization and maintenance. It uses AUX Link Services.

Both functionalities are implemented inside single AUX agent.

- It supports Link Transport Layer and Link Policy Maker and Stream Policy Maker
- It supports both kind of AUX commands, Native AUX and I2C_over_AUX with all their types
- Link Layer implements both AUX Device Service and AUX Link Service functionality as two independent processes.

Figure 4-11 shows the basic architecture of an AUX agent.

Figure 4-11 AUX agent architecture



As shown in Figure 4-11, environment is comprised of 1 agent. Main functionality is placed in the Common object of the AUX agent. It is divided on the three main processes: AUX Link Services functionality, AUX Device Services functionality and PHY layer functionality. AUX Device Services are responsible for stream transport initialization, and maintenance. AUX Link Services' responsibility is link discovery, initialization and maintenance. PHY Layer functionality is responsible for physical signaling, link discovery, flow control, bit timings, command timings, and all other operations on physical bus.

AUX agent's data item is svt_dp_aux_transaction.

All AUX agent subcomponents and connections between them are described in the following subsections.

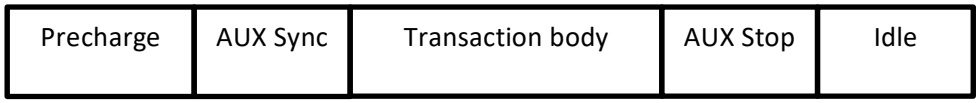
4.6.1 AUX commands

Within a transaction byte, bit 0 is the least significant bit while bit 7 is the most significant bit. For a field with more than one bit, the most significant bit of the field is transmitted first across AUX bus, with each consecutive bit following, down to the least significant bit.

In burst write/read operations over the AUX CH, the address is increased by one after each data byte. For the DPCD fields that have multiple bytes, the least significant byte is stored at the lowest address, unless otherwise explicitly stated in the DPCD Address Mapping table. During the burst operation of an AUX transaction, therefore, the least significant byte is transported first.

Figure 4-12 shows common form of the requester(DPTX) and the replier(DPRX) command.

Figure 4-12 AUX command



Requester and replier transaction differs in the transaction body. Figure 4-13 and Figure 4-14 show example of AUX request transaction body and AUX replay transaction body respectively.

Figure 4-13 Request transaction body

COMM3:0|ADDR19:16 ► ADDR15:8 ►

Figure 4-14 AUX Replay transaction body

COMM3:0 | 0000◄ (DATA0_7:0◄

Table 4-4 shows planned methods for svt_dp_aux_transaction class.

Table 4-4 AUX Transaction Methods

Method Prototype	Description
get_rx_out_aux_xact	Retrieve the RX outgoing DP AUX Transaction
get_rx_observed_aux_xact	Retrieve the RX observed DP AUX Transaction
get_tx_observed_aux_xact	Retrieve the TX observed DP AUX Transaction
sink_rx_aux_transaction	Capture the next RX DP AUX Transaction
sample_sync_condition	Sample rPe-charge, pre-amble and SYNC END pattern
sample_stop_condition	Sample Stop Pattern
sample_request_command	Sample Request command
sample_reply_command	Sample reply command
sample_data_bytes	Sample data bytes
manchester_II_decode	Decode logical value from a PHY value using a Machester II decoding
clock_regeneration	Task for clock regeneration
link_training_fsm	Task for tracking link training states
check_edid	Task for tracking EDID read
check_receiver_cap	Task for tracking Receiver Capability read
check_clock_recovery	Task for tracking Clock Recovery sequence
check_channel_equalization	Task for tracking Channel Equalization sequence

Method Prototype	Description
check_post_lt_adj_req	Task for tracking Post-Link Training Adjust Request sequence
check_rd_training_interval	Task for checking delay for AUX Read Training Interval
update_status	Writes appropriate register in status class

4.6.2 Driver

The driver is methodology dependent. In order to retain minimum code rewrite for the driver all protocol implementation, including driving, resides in the "Common" object. Therefore, the driver will be generic.

After the driver receives a transaction from the sequencer it calls drive_xact() task in "Common" which will handle further driving. Class svt_dp_aux_driver implements driver func-tionality. Its task consume_from_seq_item_port gets svt_dp_aux_transaction sequence item from sequencer and then it forwards it to the common object by calling drive_xact() task in common object. This task is blocking and it will return only when driver completes transaction driving. After this task is completed, driver calls item_done to indicate to the sequencer that the request is completed so that it can send another item.

4.6.3 Sequencer

Requests for new sequence items are initiated by the driver. Upon such requests, the sequencer selects a sequence from a list of available sequences to produce and deliver the next item to execute. If device type is Source, the agent has an instance of proactive sequencer. If device type is Sink, agent has instance of reactive sequencer.

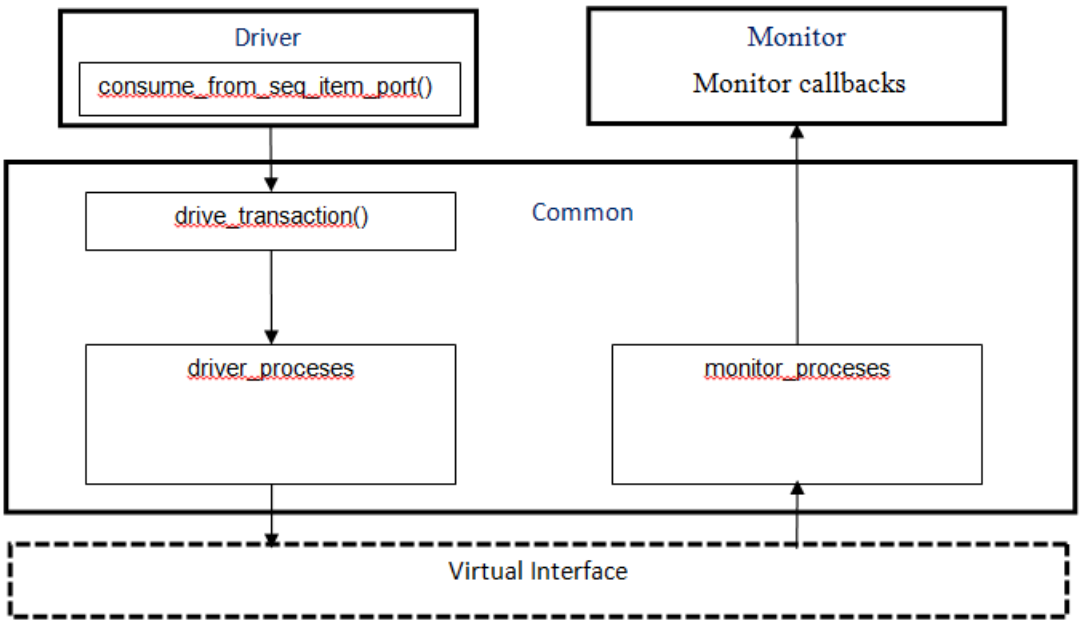
4.6.4 Monitor

Because all protocol implementation is done in the "common", monitor is just a shell which is accessible by the user, and provides callbacks. Checkers will also be implemented in the "common".

4.6.5 Common class

The "Common" object contains a complete protocol implementation. Implementation is methodology neutral so the same "Common" object is used for all three methodologies. A single instance of Common class is instantiated in the agent and is used by both the driver and the monitor. The Common class basic elements are shown in Figure 4-15.

Figure 4-15 AUX Common class basic elements



4.6.5.1 Driving Processes

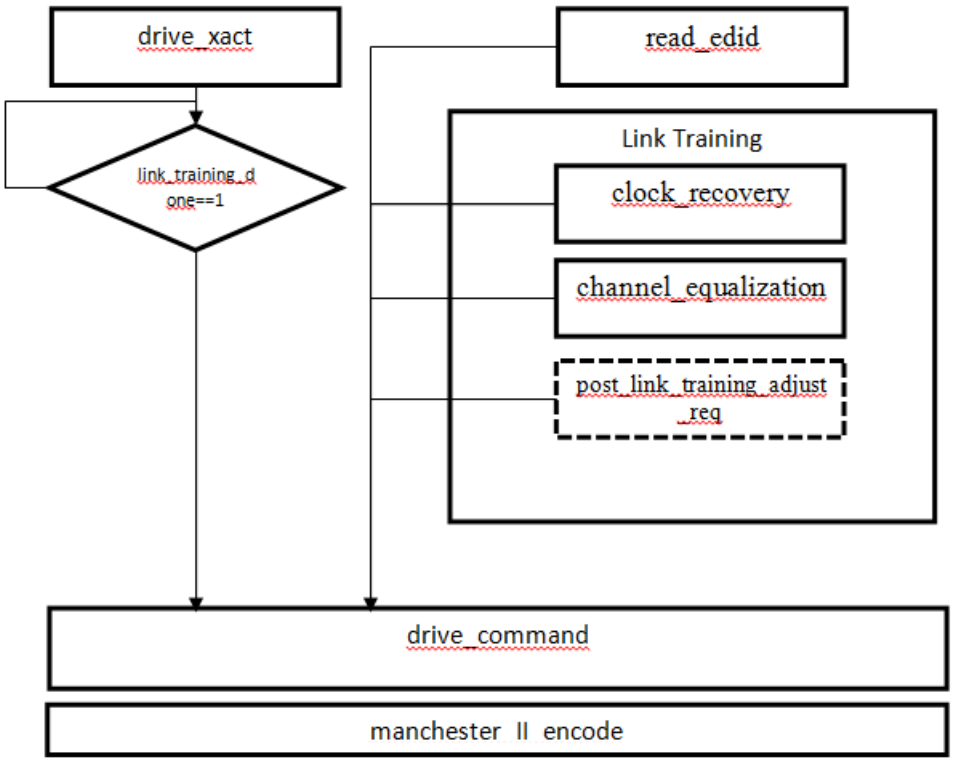
Driving methods are listed in the Table 4-5.

Table 4-5 Common driving methods

Method Prototype	Description
drive_command()	Drive AUX command that is received from sequencer. Also can be used by automatic common processes like read_edid(), clock_recovery()…
drive_hpd()	Drives HPD signal, relevant for Sink de-vice
read_edid()	Group of the AUX read command, initi-ated by Source devide in order to get information about Sinc EDID structure
clock_recovery()	First phase of the Link Training process, starts if has_link_training cfg field is set
channel_equalization()	Second phase of the Link Training process
post_link_training_adjust_req()	Third phase of the Link Training process (optional)
manchester_II_encode()	Encoding single AUX transaction and driving to AUX BUS

Common driving organization for Source and Sink devices is shown in Figure 4-16:

Figure 4-16 Main Common driving functionality



4.6.5.1.1 Link Training sequences

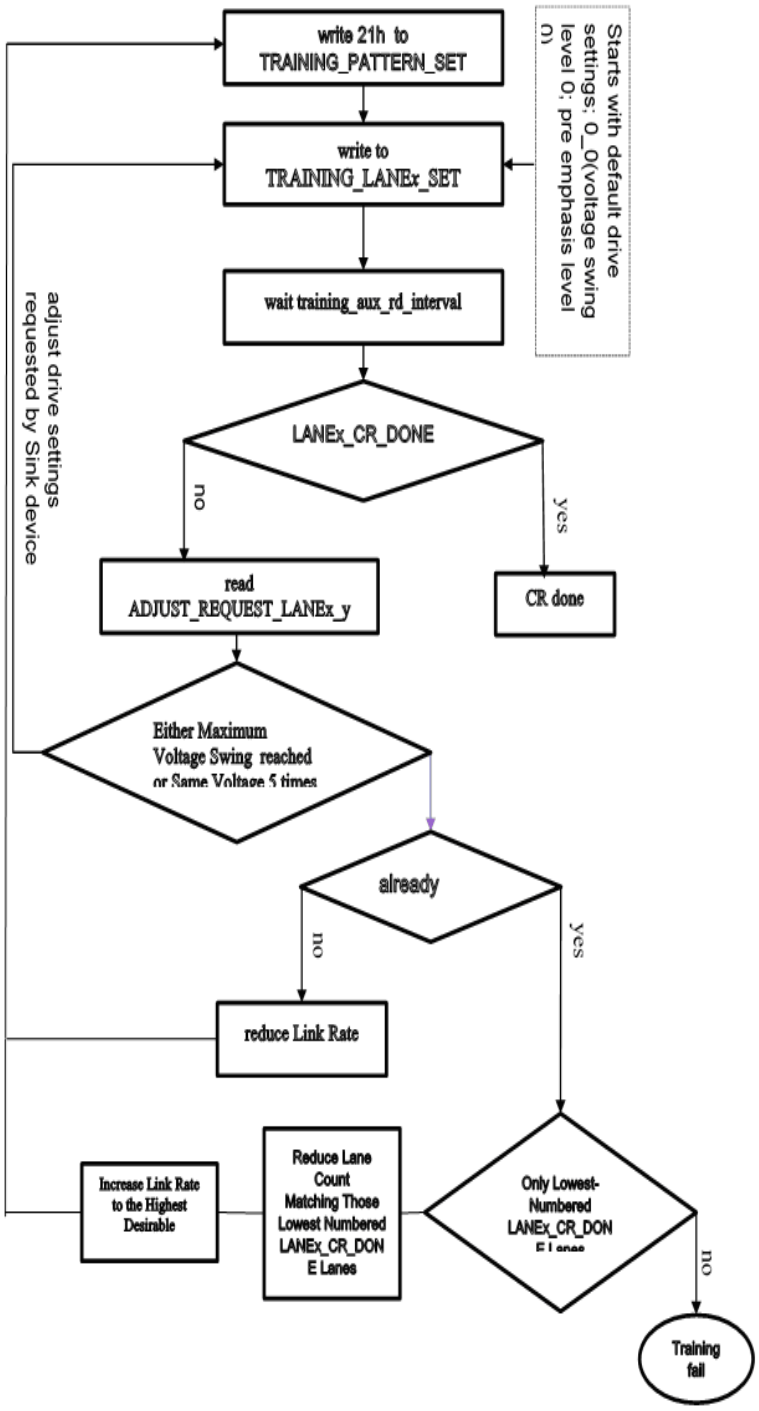
Link training consists of two distinct tasks which must be completed successfully in sequence to establish the link:

- Clock Recovery: Locks the DPRX CR (clock recovery) PLL to the repetition of D10.2 data symbols.
- Channel Equalization/Symbol-Lock/Inter-lane Alignment - When successful, the Symbol-Lock and Inter-lane alignment must be achieved by the end of this sequence
- Upon detects HPD signal is asserted high, Source reads EDID from Sink device, then reads Receiver Capability registers or Extended Receiver Capability registers (if the downstream device supports the Extended Receiver Capability field). Link Policy Maker must then determine the link configuration based on the DPRX's capability and its own needs, and then write the following Link Configuration parameter.
- LINK_BW_SET register (DPCD Address 00100h)
- LANE_COUNT_SET field in the LANE_COUNT_SET register (DPCD Address 00101h, bits 4:0)
- DOWNSPREAD_CTRL register (DPCD Address 00107h)
- MAIN_LINK_CHANNEL_CODING_SET register (DPCD Address 00108h)

Now Link Training can start with its first process, Clock Recovery sequence.

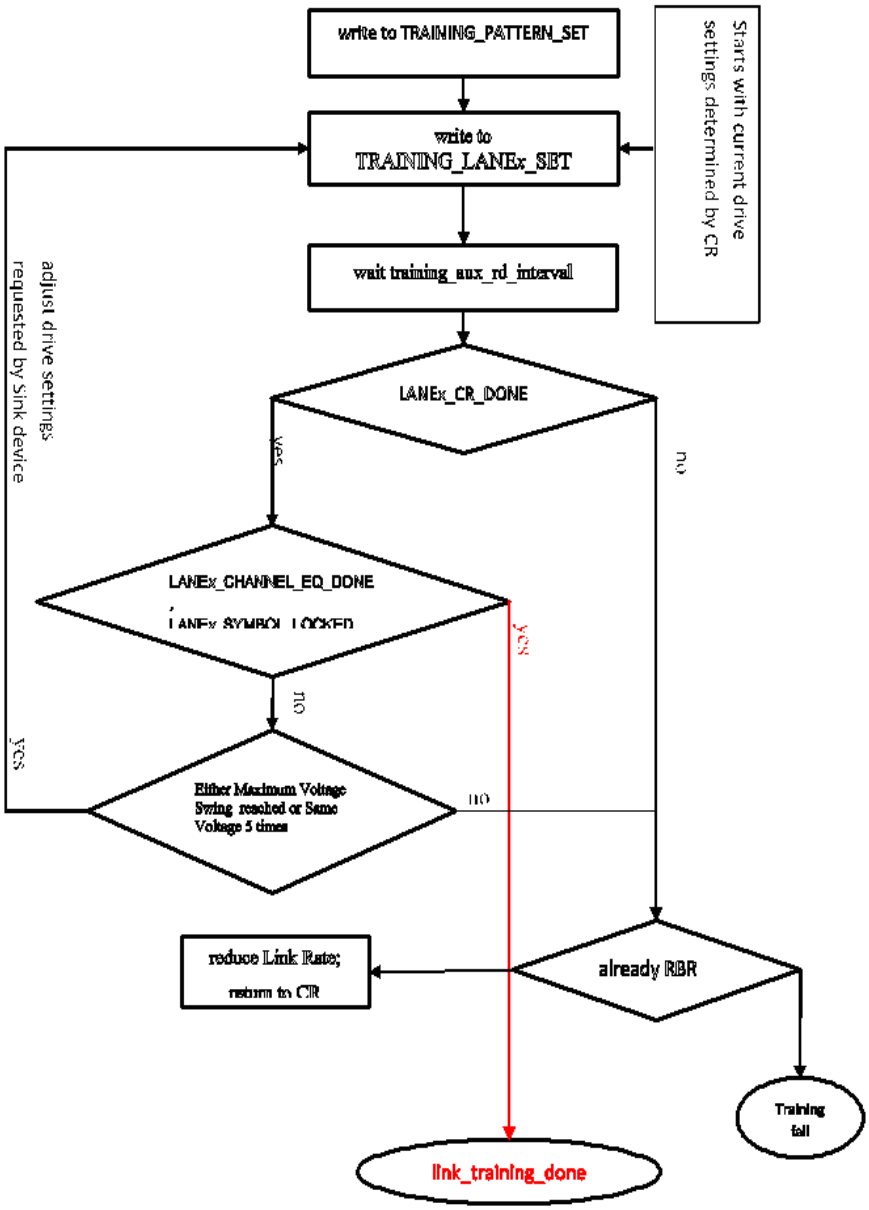
Clock Recovery diagram is shown in .Figure 4-17

Figure 4-17CR Sequence



Channel Equalization diagram is shown on the Figure 4-18

Figure 4-18 CE Sequence



Both read and write initiated by DPTX will use common's drive_command() method.

Once Link Training is done, link_training_done event will be switched on to indicate Main Link to finish with a training pattern and proceed with normal operation.

drive_command()

AUX transaction is an atomic one. That means that new transaction cannot be initiated until the previous one is finished. Requester cannot start new request if replier is not finished with replay transaction or response time is not out. This is managed by arbitration process inside drive_command() task.

drive_command() calls sub-methods for encoding its parts, shown in Figure 4-11, to Manchester_II code and driving to AUX channel differential pair, with nominal rate 1 Mbps.

Between transactions, the AUX Channel is in electrical idle state. In the electrical idle state, neither device is driving the channel and, thus, both AUX-CH+ and AUX-CH- are at the termination voltage.

4.6.5.2 Monitor Processes

Table 4-6 Common monitor methods

Method Prototype	Description
sample_command()	Sample AUX command from the bus.
hpd_detect()	Detecs HPD signal changes
manchester_II_decode()	Decoding AUX Manchester transaction from AUX BUS

manchester_II_decode() perform decoding Manchester transaction from AUX BUS. It also calls sample_command() to pack decoded field to AUX transaction. Upon sampling transaction is ready for scoreboard and Status class should be updated.

hpd_detect() detecs any change of HPD signal. It emits hpd_event and hpd_irq events. Once HPD is asserted high AUX bus should be available for its services.

4.6.6 Coverage

4.6.6.1 AUX (enable_aux_cov)

The functional coverage approach of DisplayPort VIP is bottom-up—that is, analysis starts at the signal level and goes up to the transaction level.

The `svt_dp_aux_monitor_def_cov_callback` class contains all covergroup definitions of Main Link in it. The `svt_dp_aux_err_check` class contains coverage for all Protocol Checks present in the VIP.

4.6.6.1.1 Enabling Default Coverage

To enable the default functional coverage, set the `enable_aux_cov` attribute in the

`svt_dp_aux_agent_configuration` class to 1, which is the default. To disable coverage, set the attribute to '0'.

4.6.6.1.2 Enabling Protocol Checks Coverage

To enable or disable Protocol checks coverage, use the following agent class attributes:

- The `enable_aux_chk_pass_cov` attribute controls Protocol Checks pass coverage (0=OFF, 1=ON).
- The `enable_aux_chk_fail_cov` attribute controls Protocol Checks fail coverage (0=OFF, 1=ON).

4.6.6.2 Main Link (enable_main_link_cov)

The functional coverage approach of DisplayPort VIP is bottom-up—that is, analysis starts at the signal level and goes up to the transaction level.

The `svt_dp_main_link_monitor_def_cov_callback` class contains all covergroup definitions of Main Link in it. The `svt_dp_main_link_err_check` class contains coverage for all protocol checks present in the VIP.

4.6.6.2.1 Enabling Default Coverage

To enable the default functional coverage, set the `enable_main_link_cov` attribute in the `svt_dp_main_link_agent_configuration` class to 1, which is the default. To disable coverage, set the attribute to '0'.

4.6.6.2.2 Enabling Protocol Checks Coverage

To enable or disable Protocol checks coverage, use the following agent class attributes:

- The `enable_main_link_chk_pass_cov` attribute controls Protocol Checks pass coverage (0=OFF, 1=ON).
- The `enable_main_link_chk_fail_cov` attribute controls Protocol Checks fail coverage (0=OFF, 1=ON).

4.7 DisplayPort VIP Ports and Transactions

4.7.1 Overview

This section describes the transactor objects that the DisplayPort VIP supports. Refer to the Class Reference HTML for description of objects, classes, and attributes mentioned in this section.

4.7.2 Transaction Objects

Transaction objects, which are extended from the uvm_sequence_item base class, define a unit of DP 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. The transaction object can represent the desired activity to be simulated on the bus, or the actual bus activity that was monitored.

DP transaction objects store the data content information for DP transactions. These data objects extend from the uvm_sequence_item base class and implement all methods specified by UVM for that class.

DP transaction data objects are used to:

- Generate random stimulus
- Report observed transactions
- Collect functional coverage statistics
- Support error injection

Class properties are public and accessed directly to set and read values. Transaction data objects support randomization and provide built-in constraints. Two set of constraints are provided—that is, valid_ranges and reasonable constraints.

- valid_ranges constraints limit generated values to those acceptable to the drivers. These constraints ensure basic VIP operation and should not be disabled.
- reasonable_* constraints, which can be disabled individually or as a block, limit the simulation by:
  - Enforcing the protocol: These constraints are enabled unless errors are being injected into 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 blocks of reasonable_* constraints.

4.7.3 DP Main Link Transaction Class

- DP Control Transaction (svt_control_transaction): Fields in the Control transaction class can be randomized or constrained by the user.
- DP Video Transaction (svt_video_transaction): Fields in the Video transaction class can be randomized or constrained by the user. This class contains pixels for frame.
- DP Audio Transaction (svt_audio_transaction): Fields in the Audio transaction class can be randomized or constrained by the user. This class contains audio samples.

4.7.4 DP Video Frame Sequences

DP Main Link data item represent one video line and you may want to drive some meaningful frames instead of video line. CEA Video Frame will be implemented as a DP sequence and VIP will provide sequence library with all supported frames. DP video mode support requirement is shown in Table 4-7.

Table 4-7 Video Formats

VIC	Format	Picture Aspect Ratio
1	640x480p	4:3
2	720x480p	4:3
3	720x480p	16:9
4	1280x720p	16:9
5	1920x1080i	16:9
69	1280x720p	64:27
76	1920x1080p	64:27
102	4096x2160p	256:135

Beside CEA-defined frames VIP will support shorter versions of those Frames in which user can set num-ber of blanking lines and active lines and to define vsync position and duration while line format will remain the same so instead for example to send 1080 lines for VIC 16 user can select only 10 active lines. Any us-er-defined format can be implemented as a sequence of lines and VIP will provide mechanism (sequence) in which user will be able to set his own parameters that are not defined in CEA.

4.7.5 DP AUX Transaction Class

This class represents sequence item for AUX Source, Sink and Passive Monitor Agents. This class contains attributes of the transaction for AUX Channel commands. There are two types of AUX transaction defined by transaction_type_e:

- Native AUX Transaction
- I2C mapped onto I2C_over_AUX Transaction

Class name - svt_dp_aux_transaction extends svt_transaction;

Table 4-8 Data members in AUX transaction class

Name	Type	Initial Value	Description	Constraint
Random Data Members				
transaction_type_e	rand svt_dp_types::channel_type_enum	svt_dp_types::native_aux	Indicates the transaction type	
cmd_e	rand svt_dp_types::cmd_enum		Indicates the AUX request/replay command type	
addr	Rand bit [19:0]		Targets DisplayPort address space	
data[]	rand bit [7:0]		Contains data	
length	Rand bit [7:0]		Indicates data length	length <= 16
mot	rand bit		Indicates if I2C ends in current AUX transaction	Used only for i2c_over_aux type

5
DP Environment

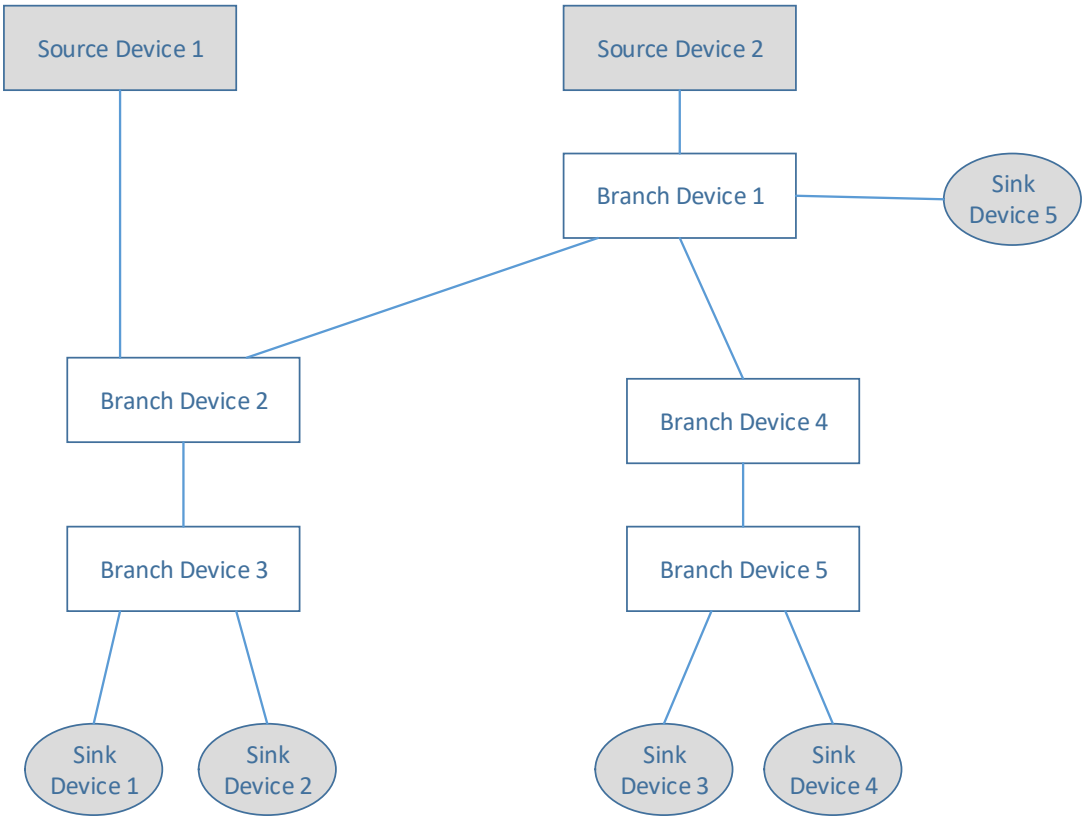
5.1 Overview

UVM system-level verification environments encapsulate the various agents that are designed for flexible reuse. This facilitates the capability of creating custom environments for testing a variety of designs through a common component set. When a VIP defines multiple interacting transactors, implementing an environment subset that comprises basic components can minimize maintenance, simplify instantiation, and reduce the complexity of VIP as viewed from the highest abstraction level. Environments are UVM components that are composed of two or more agents. The DisplayPort VIP defines an environment that can be customized to support a variety of verification scenarios. This chapter describes the DisplayPort VIP environment object. Refer to the Class Reference HTML for a description of objects, classes, and attributes mentioned in this chapter.

The system environment of DisplayPort VIP can be configured as per requirement to implement vari-ous topologies. It might consist of a configurable numbers of DP Source Devices, Branch Devices and DP Sink Devices.

VESA DisplayPort standard supports interconnections of MST and SST DP devices into topologies. Figure 5-1 shows an example of MST topology where all DP Source and Branch devices are MST devices supporting Topology Management and all DP Sink devices are those that have no branching unit.

Figure 5-1 Example MST Topology



The Branch devices of the MST topology must be MST Branch devices. SST devices are allowed in the topology only as the end devices such as SST DP Source devices and SST DP Sink devices. The SST device is identified by the MST Topology Management layer as a peer device connected to a MST Branch device. The maximum number of links between a stream source to a stream sink must be 15 or fewer. Of these, the maximum number of physical links is limited to seven.

The following sections deals with following topologies that can be used for Standalone DP design.

- Source VIP and Sink VIP
- Source VIP and Sink DUT
- Source DUT and Sink VIP
- Source and Sink DUT with Passive Agents

Standalone DP test bench topology can use Source (having active Main Link active agent and AUX active agent) and Sink (having Main Link passive agent and AUX active agent). You can configure the verification environment to implement multiple topologies as mentioned above. Main Link agent will be used to transport isochronous data streams such as uncompressed video and audio and AUX agent will be used for link management and device control transmission. Main Link and AUX agents can work independently.

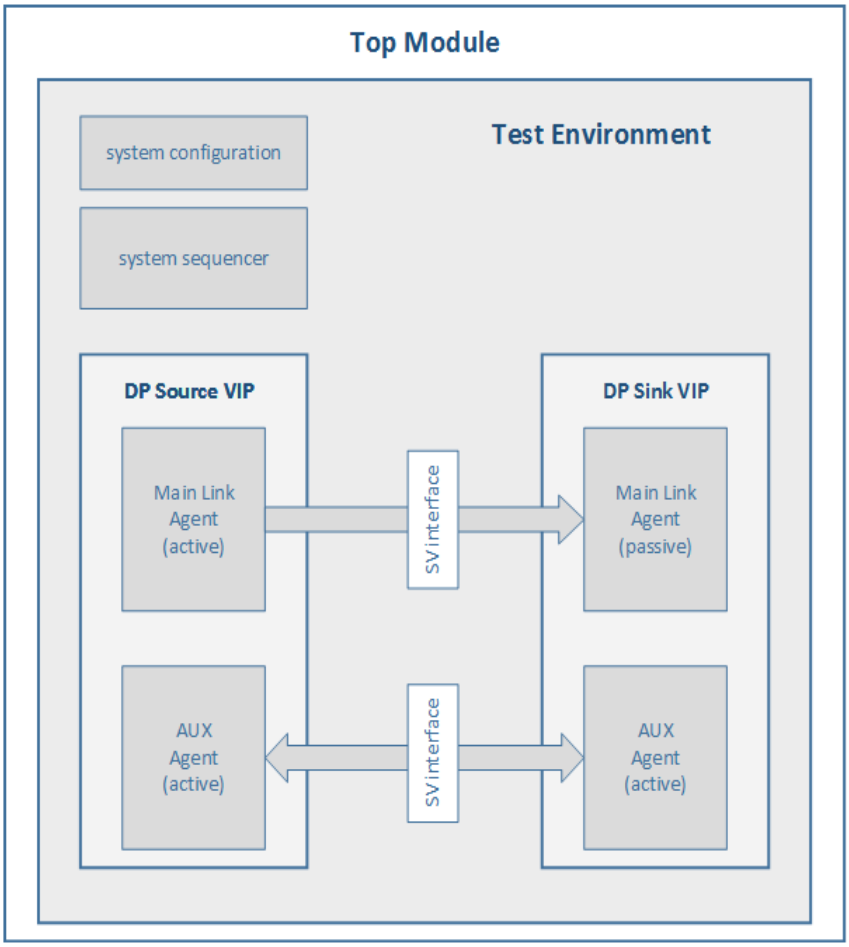
Note MST is not supported in this release.

5.1.1 Source VIP and Sink VIP

Scenario - Both DP Source and Sink VIPs connected back-to-back as example for VIP.

Testbench Setup: In DP environment connect one Source VIP (active Main Link agent and active AUX agent) with one Sink VIP (passive Main Link agent and active AUX agent) in back-to-back topology. The Source generates DP transactions for the Sink. Source and Sink VIPs also perform passive functions such as protocol checking, coverage generation and transaction logging. Data integrity check will be done with expected source side video/audio data collected at sequence level with actual received data from sink monitor.

Figure 5-2 Example Test Environment containing one Source VIP and one Sink VIP

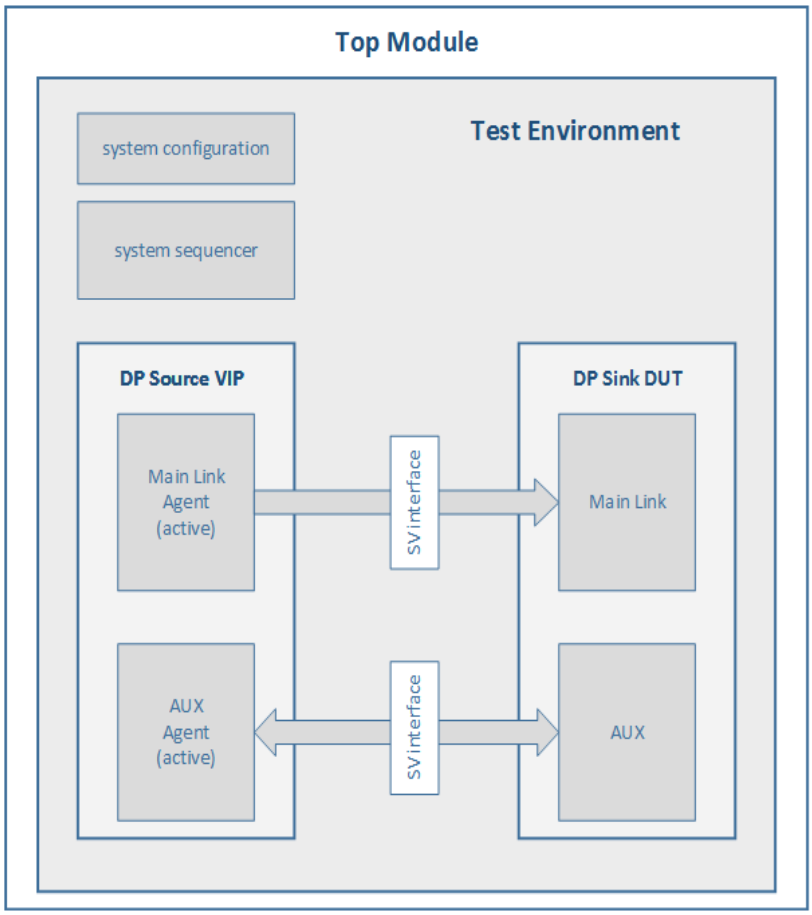


5.1.2 Source VIP and Sink DUT

Scenario - The Source VIP is required to verify the DP Sink DUT.

Testbench Setup: In DP environment, connect one Source VIP (active Main Link agent and active AUX agent) with one Sink DUT (Main Link and AUX). The Source generates DP transactions for the Sink. Source VIP also performs passive functions such as protocol checking, coverage generation and transaction logging. Data integrity check will be done with expected source side video/audio data collected by source monitor with actual received data from sink design at output interface.

Figure 5-3 Example Test Environment containing one Source VIP and one Sink DUT

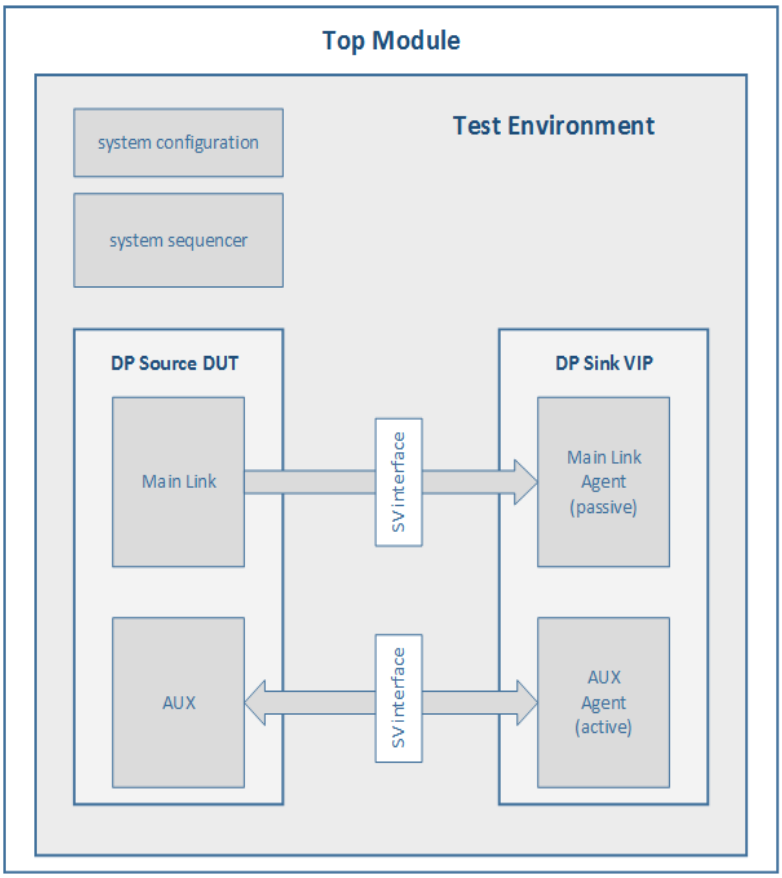


5.1.3 Source DUT and Sink VIP

Scenario - The Sink VIP is required to verify the DP Source DUT.

Testbench Setup: In DP environment connect one Sink VIP (passive Main Link agent and active AUX agent) with one Source DUT (Main Link and AUX). The Source generates DP transactions for the Sink. Sink VIP also performs passive functions such as protocol checking, coverage generation and transaction logging. Data integrity check will be done with expected source side video/audio data collected at sequence with actual received data from sink monitor.

Figure 5-4 Example Test Environment containing one Sink VIP and one Source DUT

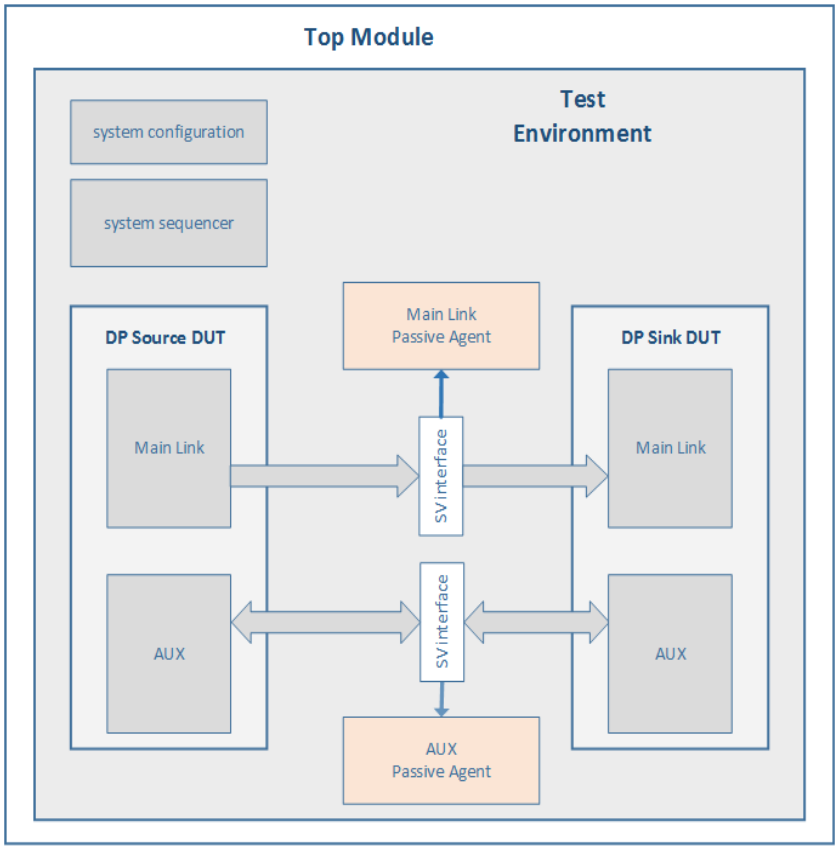


5.1.4 Source and Sink DUT with Passive Agents

Scenario - The Passive agents are used to check both Source and Sink DUTs connected back to back.

Testbench Setup: In DP environment connect one Source DUT (Main Link and AUX) with one Sink DUT (Main Link and AUX) along with Passive Mail Link Agent and Passive AUX Agent. The Source generates DP transactions for the Sink. Passive Agents perform passive functions such as protocol checking, coverage generation and transaction logging. Data integrity check can be done with two approaches. In first approach, data comparison will be between expected source side data collected at sequence level with actual received data from passive agent to verify the Source DUT and in second approach, expected data from passive agent with actual data received from the output of sink design to verify the Sink DUT.

Figure 5-5 Example Test Environment Source and Sink DUTs and Main Link and AUX Passive Agents



5.2 Description

DP environments are component objects in a UVM-compliant verification environment. The environments, svt_dp_env, extend from svt_env, which extends from uvm_env.

DP verification component architecture is completely based on the UVM Manual for System Verilog. Source and Sink environments can be individually controlled using the respective Configuration Classes.

5.3 Using Callback for Error Injection

Test Case Name: *ts. multilane_fec_video_basic_test_with_errors.sv*

The following example demonstrates how to use callbacks in VIP for error injection scenarios. This example demonstrates the error injection in FEC block using a callback. FEC block count is a free running counter during the FEC envelope. One of the callbacks named `fec_block_count[i]` is used to identify and trigger the error injection flag and another callback is used to corrupt the symbol within FEC block which is of 512 code/256 codes overall for single and multi-lane respectively.

This test case injects errors in lane 0 and lane 3 in FEC block. 1 FEC block is a collation of 250 bytes. In this test case, we are trying to corrupt any of these bytes in one of the blocks. Currently, in the test case there are two callbacks added per lane—that is, lane 0 and lane 2 and therefore there are four functions implemented in the class named `svt_dp_fec_block_count` which is extended from `svt_dp_main_link_callback`. In one of the functions `fec_block_lane0`, we are inserting error in particular block on lane 0. In another function named `symbol_error_in_fec_blk_lane0`, we are inserting error in a specific byte. First callback is used to say which FEC block is to be corrupted and another callback indicates which byte in a particular

block is to be corrupted. This callback is mainly used at the transmitter side to inject the errors in 10-bit code group. As a VIP transmitter, we do not highlight our own transmission errors and so we may not see in Verdi as an event in RED.

There is a report catcher to indicate what errors are to be expected and to be demoted/masked. Because we are injecting errors in 10-bit code group, there could be decoding error or running disparity error. These errors will be demoted to `UVM_INFO`. In order to see the errors in the form of `UVM_ERROR` without demoting them, error catcher can be commented in the test case. It is up to you whether to demote the `ERROR` from the test case.

5.3.1 Running a Test Case

To run the test case, use the following command:

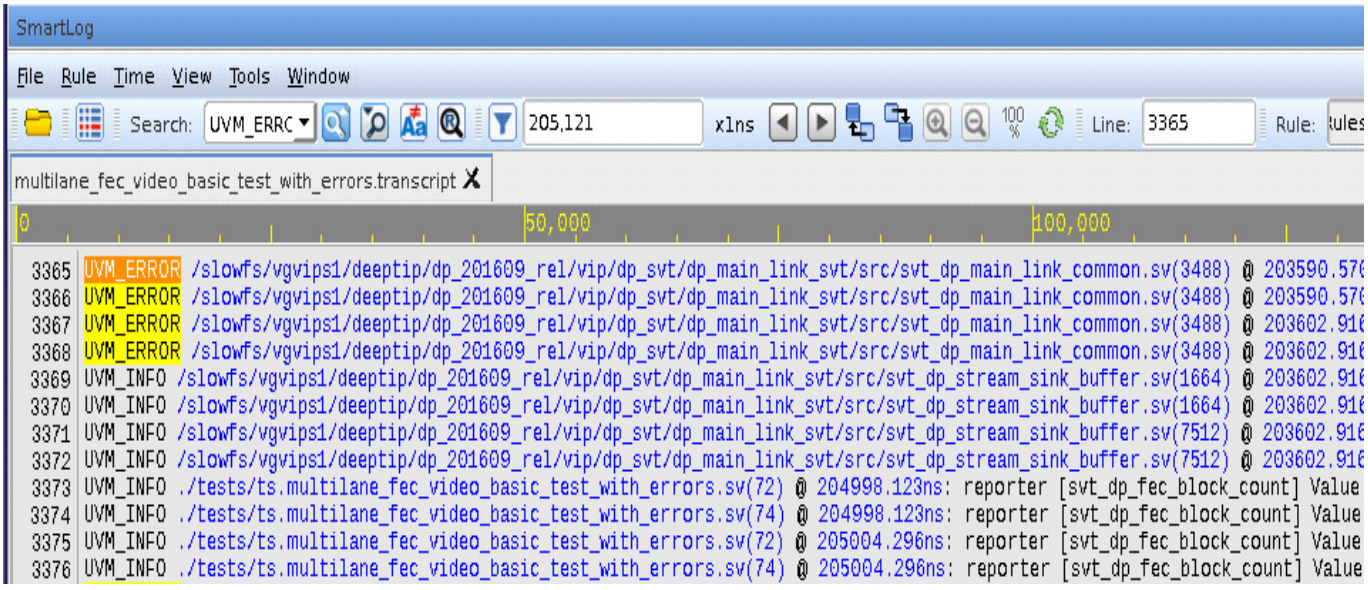
```
gmake clean multilane_fec_video_basic_test_with_errors WAVES=fsdb
```

In the log, you can search the log messages to understand what is the original FEC code and what is the corrupted FEC code. At that time stamp in waveform, `lane0_data` and `lane0_decoded_data` can be plotted. For original and corrupted FEC code, check the following messages in the log file:

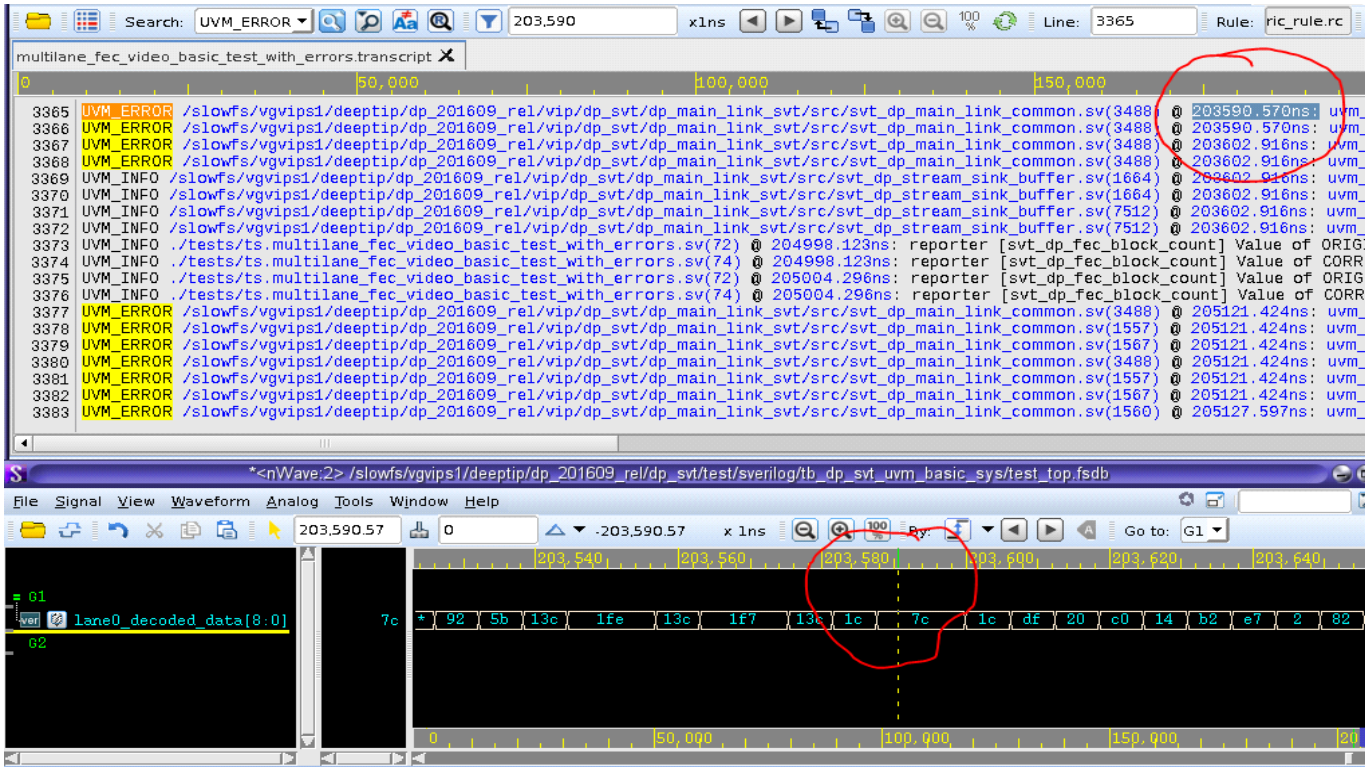
```
Value of ORIGINAL FEC code is
Value of CORRUPTED FEC code is
```

5.3.1.1 Snippet from Verdi

You can use the SmartLog feature of Verdi to synchronize the log with the waveform.



You can use the `Sync Wave cursor` option of the SmartLog to synchronize it with generated *fsdb* as shown in the following figure:



6 VIP Tools

6.1 Using Native Protocol Analyzer for Debugging

6.1.1 Introduction

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

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

6.1.2 Prerequisites

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

Compile-Time Option

- `-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 by setting the `pa_format_type` configuration variable as shown below:

Configuration Variable Setting:

Set the `enable_main_link_xml_gen` a parameter of DP configuration class `svt_dp_env_configuration` to enable generation of XML files.

For example,

```
cfg.enable_main_link_xml_gen = 1;
cfg.enable_aux_xml_gen = 1;
```

Runtime Option

```
+svt_enable_pa=fsdb
```

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

6.1.3 Invoking Protocol Analyzer

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

Post-processing Mode

- Add the following options:

  ```
  WAVES=fsdb

  PA=1
  ```

  For more details, see *Makefile* in the installation example.
- Invoke Verdi with `ssf` option.

  ```
  verdi -ssf waves=fsdb
  ```
- Load FSDB signals and invoke Transaction and Protocol Analyzer by navigating to Tools ->Transaction Debug -> Transaction and Protocol Analyzer

Note FSDB dump includes signal groups of the interface, you can right-click on a transaction object in the Verdi Protocol Analyzer and add related debug signals to the waveform.

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.

6.1.4 Documentation

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

$VERDI_HOME/doc/Verdi_Transaction_and_Protocol_Debug.pdf

7 Usage Notes

This chapter provides some useful information that can help you troubleshoot common issues that you may encounter while using the DisplayPort VIP. This chapter discusses the following topics.

7.1 Setting Verbosity Levels

You can set VIP debug verbosity levels either in the testbench or as an option during run-time. To set the verbosity level in the testbench, use the UVM-specified log-levels in the code as shown in the following code snippet:

```
/**
 * Here the verbosity level, of source agent instance, is set to the
 * UVM_FULL.
 */
 source_agent.set_report_verbosity_level(UVM_FULL);
```

To set the verbosity level during run-time, you can use one of the following methods:

- Method 1: To enable the specified severity in the VIP, DUT, and testbench
- Method 2: To enable the specified severity to specific sub-classes of VIP

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

In this method, set it at the command line.

```
<Simulator specific options> +UVM_VERBOSITY=UVM_DEBUG
```

Or:

```
<Simulator specific options> +VERBOSITY=DEBUG
```

7.1.2 Method 2: To Enable the Specified Severity to Specific Sub-classes of VIP

Use the following:

```
"+vip_verbosity=<vip_subunit_1_name>:<verbosity_level_1>,<vip_subunit_2_name>:
<verbosity_level_2>…."
```

Use Case Example:

```
<Simulator specific options>
+vip_verbosity=svt_video_transaction:verbose,svt_audio_transaction:debug,
svt_dp_control_transaction:debug,svt_dp_aux_transaction:debug
```

7.2 Disabling Specific In-line Checking

All VIP error checking is enabled by default. Refer to chk_cov_mgr in the Class Reference HTML for valid checks defined.

The following code snippet illustrates the enabled default VIP error checking:

```
svt_err_check_stats temp_check;
temp_check =
   source_device.driver.chk_cov_mgr.find("aux_response_timer_timeout_check");
temp_check.set_is_enabled(0);
```

7.3 DSC Compression Setup Requirements

Perform the following steps to setup DSC compression:

1. You must set the path of the downloaded VESA DSC *C* code to `VESA_DSC_SRC_PATH` environment variable. This variable is used in the compilation of the VESA C code. You can set the path by providing the absolute path of the VESA DSC source code as shown in the following example:

   ```
   setenv VESA_DSC_SRC_PATH {ABSOLUTE_VESA_DSC_MODEL_PATH}/DSC_model_20151013/source
   ```

2. Recommended GCC compiler is gcc-4.2.2
   a. For 32-bit OS, load the 32-bit gcc

      For example, `module load gcc/4.2.2`

   b. For 64-bit OS, load the 64-bit gcc

      For example, `module load gcc/4.2.2-64`

   c. To run DSC test cases with MTI, you must set the GCC_BIN environment variable to the *gcc* version supported by MTI.

      For example, `setenv GCC_BIN /global/apps/mti_10.4a/modeltech/gcc-4.7.4-linux/bin`

   sim_build_options:

   ```
   +define+SVT_DP_DSC_ENABLE
   ${DESIGNWARE_HOME}/vip/svt/dp_svt/latest/sverilog/src/c/src/svt_dp_dsc_api.c
   ${VESA_DSC_SRC_PATH}/dsc_codec.c ${VESA_DSC_SRC_PATH}/dsc_utils.c
   ${VESA_DSC_SRC_PATH}/fifo.c ${VESA_DSC_SRC_PATH}/logging.c
   ${VESA_DSC_SRC_PATH}/multiplex.c ${VESA_DSC_SRC_PATH}/psnr.c ${VESA_DSC_SRC_PATH}/utl.c
   ```

   vcs_build_options:

   ```
   -CFLAGS -DVCS -timescale=1ns/1ps +define+UVM_PACKER_MAX_BYTES=1500000
   +define+UVM_DISABLE_AUTO_ITEM_RECORDING -CFLAGS -std=gnu99 -CFLAGS -
   I${DESIGNWARE_HOME}/vip/svt/dp_svt/latest/sverilog/src/c/include -CFLAGS -
   I${VESA_DSC_SRC_PATH}
   ```

ncv_build_options:

```
-CFLAGS -std=gnu99 -incdir ${VESA_DSC_SRC_PATH} -
I${DESIGNWARE_HOME}/vip/svt/dp_svt/latest/sverilog/src/c/include -I${VESA_DSC_SRC_PATH}
```

mti_build_options:

```
-ccflags -std=gnu99 +incdir+${VESA_DSC_SRC_PATH} -ccflags "-
I${DESIGNWARE_HOME}/vip/svt/dp_svt/latest/sverilog/src/c/include”
```

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>,verbosity:<string>
```

The following table explains each control string:

Table A-1 Control Strings for Debug Automation plusarg

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

Examples:

Enable on all VIP instances with default options:

```
+svt_debug_opts
```

Enable on all instances:

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

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

Enable on all instances:

- starting at time 1000 until time 1500:

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

Enable debug feature on all instances using default options:

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

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

Note

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

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

A.4 Debug Automation Outputs

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

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

The following are the output files generated:

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

A.5 FSDB File Generation

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

A.5.1 VCS

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

```
-debug_access
```

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

A.5.2 Questa

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

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

A.5.3 Incisive

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

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

A.6 Initial Customer Information

Follow these steps when you call the Synopsys Support Center:

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

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

A.7 Sending Debug Information to Synopsys

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

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

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

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

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

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

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

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

A.8 Limitations

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

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